Title	Deep Reinforcement Learning Based Optimal Control of Nonlinear Systems
Author(s)	池本, 隼也
Citation	大阪大学, 2023, 博士論文
Version Type	VoR
URL	https://doi.org/10.18910/92211
rights	
Note	

Osaka University Knowledge Archive : OUKA

https://ir.library.osaka-u.ac.jp/

Osaka University

Deep Reinforcement Learning Based Optimal Control of Nonlinear Systems

Junya Ikemoto

March 2023

Deep Reinforcement Learning Based Optimal Control of Nonlinear Systems

A dissertation submitted to

The Graduate School of Engineering Science

Osaka University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Engineering

By

Junya Ikemoto

March 2023

Abstract

Reinforcement learning (RL) is a machine learning approach for sequential decision making. Recently, in order to solve complicated decision making problems, RL with DNNs, which is called deep RL (DRL), has attracted much attention thanks to development of deep neural network (DNN) technology. DRL is also useful to design optimal controllers of nonlinear systems in the case where it is difficult to identify models of systems accurately and to design the controllers analytically since we can obtain controllers through interactions with systems automatically. However, in the real world, the application of DRL is restricted due to some problems other than learning performances of DRL algorithms. In this dissertation, we tackle the following four problems to extend the application range of DRL for optimal nonlinear control problems.

Firstly, we propose a practical DRL algorithm with a simulator to mitigate high sample complexity. In DRL, the agent needs many interactions with a system to learn its policy. Then, using the simulator, we can collect many experiences more efficiently than through interactions with the real system. In general, however, there is a gap between the real system and the system modeled in the simulator, which may degrade the performance of the learned policy. Thus, we propose the two-stage deep Q-learning algorithm that takes the identification error into consideration.

Secondly, we apply DRL to design of networked controllers to stabilize a nonlinear system at an equilibrium point. In a networked control system (NCS), we must design a controller considering the characteristic of the network. Particularly, it is important to take effects of network delays into consideration. Thus, in order to learn the control policy considering network delays, we propose an extended state consisting of the current system's state and some previously determined control actions. Additionally, we consider the case where a sensor cannot observe some state variables of a system.

Thirdly, we apply DRL to design of a networked controller to complete a given temporal control task with time bounds. We describe the temporal control task as a signal temporal logic (STL) formula. In order to learn a policy to complete the task considering the effect of network delays, we propose an extended Markov decision process (MDP), which is called by a τd-MDP. In the τd-MDP, we regard not only a current system's state but also previous control actions and previous system's states as an environment's state.

Finally, we propose a DRL algorithm to obtain an optimal policy with respect to a given control performance index under a constraint described by an STL formula. We formulate the control problem as a constrained MDP (CMDP), which has two reward functions: one is the reward function for the given control performance index and the other is the reward function for the given STL formula. We use the Lagrangian relaxation method for solving the CMDP problem. By this method, the CMDP problem is transformed into an unconstrained problem with a Lagrangian

multiplier so that the standard DRL algorithm is utilized. Moreover, we introduce a two-phase learning algorithm to collect experiences satisfying the given STL formula efficiently.

Acronyms

CMDP	Constrained Markov decision process
CPO	Constrained policy optimization
DDPG	Deep deterministic policy gradient
DNN	Deep neural network
DP	Dynamic programming
DRL	Deep reinforcement learning
DQN	Deep Q-network
HJB	Hamilton Jacobi Bellman
iLQG	iterative linear quadratic gaussian
iLQR	iterative linear quadratic regulator
LTL	Linear temporal logic
LQR	Linear quadratic regulator
MC	Monte Carlo
MDP	Markov decision process
MITL	Metric interval temporal logic
ML	Machine learning
NAF	Normalized advantage function
NCS	Networked control system
PILCO	Probabilistic inference for learning control
PPO	Proximal policy optimization
ReLU	Rectified linear unit
RL	Reinforcement learning
RNN	Recurrent neural network
SAC	Soft actor critic
STL	Signal temporal logic
TD	Temporal difference
TD3	Twin delayed deep deterministic policy gradient
TRPO	Trust region policy optimization

Contents

Abstract **i**

1 Introduction **1**
1.1 Background 1
1.2 Outline and Contribution 3
1.3 Related Works 7
1.4 Notation 10

2 Preliminaries **11**
2.1 Reinforcement Learning 11
2.1.1 Value Based Algorithm and Policy Based Algorithm 13
2.1.2 On-Policy and Off-Policy Algorithm 14
2.2 Q-Learning 14
2.3 Deep Q-Network (DQN) 16
2.4 Continuous Deep Q-Learning with Normalized Advantage Function (NAF) 18
2.5 Deep Deterministic Policy Gradient (DDPG) 20
2.6 Soft Actor Critic (SAC) 21

3 Deep Q-Learning with a Simulator for Stabilization **24**
3.1 Problem Formulation 24
3.2 Practical Q-Learning with Pre-Trained Multiple Deep Q-Networks . . 25
3.2.1 The Q-Function for the Real System 26
3.2.2 Q-Learning for the Real System with Deep Q-Networks Learned for Multiple Virtual Systems 28
3.2.3 Proposed Algorithm 30
3.3 Example 31
3.3.1 Choice of Basis Functions 37
3.3.2 Adaptivity under Variation of the System Parameter Vector . . 43
3.3.3 Effects of Pre-Training 46

4 Application of DRL to NCSs with Uncertain Network Delays **49**
4.1 Problem Formulation 49

4.2 Design of Networked Controller Using Deep Reinforcement Learning 51
4.2.1 State-Based Learning 51
4.2.2 Output-Based Learning 52
4.2.3 Learning Algorithm 54
4.3 Example 57
4.3.1 State-Based Learning 58
4.3.2 Effect of τ 65
4.3.3 Output-Based Learning 67

5 DRL for STL Tasks under Uncertain Network Delays 73
5.1 Signal Temporal Logic (STL) 73
5.2 Problem Formulation 75
5.3 Q-Learning for Satisfying an STL Formula Using a τ-MDP 76
5.4 DRL for Satisfying an STL Formula under Network Delays 79
5.4.1 τd-Markov Decision Process (τd-MDP) 82
5.4.2 Pre-Process 83
5.4.3 Proposed Algorithm 88
5.5 Example 89
5.5.1 Result 93
5.5.2 Ablation Study for Pre-Processing 96
5.6 Application to a Problem with Random Delays 98

6 DRL under STL Constraints Using Lagrangian Relaxation 100
6.1 Problem Formulation 100
6.2 Constrained Markov Decision Process (CMDP) 101
6.3 τ-Constrained Markov Decision Process (τ-CMDP) 102
6.4 Deep Reinforcement Learning under an STL Constraint 103
6.4.1 DDPG-Lagrangian 104
6.4.2 SAC-Lagrangian 106
6.4.3 Pre-Training and Fine-Tuning Method 108
6.4.4 Pre-Process 109
6.4.5 Algorithm 109
6.5 Example 110
6.5.1 Evaluation 113
6.5.2 Ablation Studies for Pre-Processing 119
6.5.3 Comparison of Based Algorithms 119

7 Conclusions and Future Works 122

References 124

Appendix 132
A Reconstruction of State of Linear Dynamical System 132

B Runge-Kutta Method 136
C NCSs Simulations 139

Acknowledgment **141**

Publication List **142**

Chapter 1

Introduction

1.1 Background

Reinforcement learning (RL) [1, 2] is a *machine learning* (ML) approach for sequential decision making problems. ML is classified into the following three approaches [3].

- Supervised learning.
- Unsupervised learning.
- Reinforcement learning.

In supervised learning, we aim to learn a general rule that maps an input $\mathbf{x}$ to an output y based on a training dataset $\{(\mathbf{x}_i, y_i)\}_{i=1}^{n}$. On the other hand, in unsupervised learning, we consider a set of inputs $\{\mathbf{x}_i\}_{i=1}^{n}$ without corresponding target values $\{y_i\}_{i=1}^{n}$ as the training dataset. For example, we aim to discover groups of similar examples, which is called *clustering*, to estimate the distribution generating $\mathbf{x}$, which is called *probability density estimation*, or to project high-dimensional data to a low-dimensional data space, which is called *dimensionality reduction*. In RL, a learner, which is called an *agent*, actually interacts with the outside of the learner, which is called an *environment*, and learns its policy for sequential decision making based on the interaction data. The interaction between the agent and the environment is often formulated by a *Markov decision process* (MDP). At each time step $k \in \{0, 1, 2, ...\}$, the agent observes the current state of the environment x_k and then determines an action a_k based on its policy. As a result of the determination, the agent observes the next state of the environment x_{k+1} and receives an immediate reward r_k. The goal of RL is to obtain the optimal policy that maximizes sum of immediate rewards. Note that, in RL, the agent must collect training data by actually taking actions unlike supervised learning and unsupervised learning. Then, the *exploration-exploitation tradeoff* is an important problem, where exploration means increasing training data by taking actions in order to make better decision making in the future and exploitation means determining an action based on the learned policy, respectively. Classically, the tradeoff problem has been tackled in the *multi-armed bandit* problem [4].

In [1], Sutton *et al.* classified RL algorithms into the following three approaches.

- Dynamic programming,
- Monte Carlo method,
- Temporal difference learning.

Dynamic programing (DP) was proposed by Bellman [5], which is utilized to solve optimal control problems in the control system community. The framework of DP is breaking a main problem down into several sub-problems to efficiently solve the problem. However, DP requires a full knowledge about a mathematical model of the environment, that is, we need to accurately identify the model beforehand. On the other hand, the *Monte Carlo* (MC) method does not use the knowledge about the environment's model and relies on repeated random sampling. In the MC method, we average the rewards for each state-action pair from different sample sequences. Although the MC method is simple and useful to estimate the obtained rewards, the variance of the estimation value tends to be large, which may lead a poor decision making. Then, *temporal difference* (TD) learning is useful. TD learning is a method including ideas of DP and the MC method and known as one of the significant ideas in RL. *Q-learning*, which is a famous RL algorithm, is one of TD learning algorithms [6].

In classical RL studies, it is assumed that an environment's state space and an agent's action space are discrete spaces, which have at most a finite number of elements. We often solve problems with discrete state-action spaces using a tabular-based algorithm such as the classical Q-learning algorithm [6]. On the other hand, in the case where we consider problems with continuous spaces, we cannot directly apply the tabular-based algorithm. Then, the function approximation is useful. Particular, when the action space is continuous, we often parameterize an agent's policy and make the agent optimize the parameter vector of the policy using a *policy gradient* method [7]. Recently, *deep neural networks* (DNNs) have attracted much attention as function approximations [8]. It is known that DNNs can express complicated models without hand-craft feature engineering. RL with DNNs, which is called *deep RL* (DRL), can solve complicated decision making problems such as *Atari 2600* video games [9,10] and manipulation and locomotion tasks in the physical simulator [11–19]. Furthermore, DRL has been applied in various fields such as autonomous driving [20], robotics [21–24], chaos control [25,26], and finance [27,28].

DRL is also useful for control of nonlinear dynamical systems [29] in the case where it is difficult to accurately identify the models of systems and to design controllers analytically. For example, in optimal control methods, we solve a *Hamilton-Jacobi-Bellman* (HJB) equation derived from a system's model to compute the optimal control inputs. In general, however, it is difficult to solve the equation analytically for a nonlinear system even if we know its accurate model because the HJB equation is a nonlinear equation. Then, DRL is useful since we can obtain an optimal policy through interactions with the system automatically. Using DRL, we can avoid to solve the HJB equation. However, in the real world, the application of DRL is restricted due to some problems other than learning performances of DRL algorithms. Among

the problems, we mainly consider the following three problems. Firstly, in DRL, the agent needs many experiences in order to learn its optimal policy. In addition, we need a large number of trials to heuristically select appropriate hyper-parameters such as learning rates. Secondly, we must properly specify a state space of an environment for a given control problem. The states of the environment need to include sufficient information in order to determine a desired action at each time. Thirdly, we must design a reward function for a given control problem. If we do not design it to evaluate behaviors precisely, the learned policy may not be appropriate for the control problem.

In this dissertation, we tackle the expansion of the application range of DRL for control of nonlinear systems by solving the four problems.

1.2 Outline and Contribution

The remainder of this dissertation is organized as follows. In **Chapter 2**, we review standard frameworks of RL and DRL as preliminaries. In **Chapters 3-6**, we show the main contributions of this dissertation. Finally, in **Chapter 7**, we conclude this dissertation and discuss future works.

Chapter 3

Although DRL can solve complicated decision making problems such as playing video games, applications of DRL to control of a dynamical system in the real world are limited because the agent needs many interactions with the system in order to learn its optimal policy. Furthermore, if we apply DRL to control of a safety-critical physical system, an agent in an early learning stage may determine actions that cause damage to the system during its exploration. Then, we often use a simulator that predicts the behavior of a real system using a mathematical model with a system parameter vector. Using the simulator, we can collect many experiences more efficiently and safely than through interactions with the real system. In general, however, it is difficult to accurately identify the system parameter vector, and in the case where we have an identification error, the experiences obtained by the simulator may degrade the performance of the learned policy.

The contribution of this chapter is that we propose a novel DRL algorithm for stabilization of nonlinear systems using multiple deep Q-functions learned with a simulator. We use the simulator that predicts the behavior of a real system using a mathematical model with a given system parameter vector. We call the system simulated in the simulator a *virtual system*. Our proposed algorithm consists of two stages in order to take the identification error into consideration. In the first stage, multiple system parameter vectors are chosen from a premised set. The multiple virtual systems with the chosen system parameter vectors are prepared in the simulator and an approximated optimal Q-function is learned for each virtual system using

the *continuous deep Q-learning* algorithm [15]. In the second stage, the Q-function for the real system is represented as an approximated linear function whose basis functions are approximated optimal Q-functions pre-trained in the first stage. Thanks to pre-training, we can reduce interactions with the real system and mitigate the high sample complexity of DRL. Additionally, we also apply our proposed algorithm to a system whose system parameter vector varies slowly. This chapter is based on our study [30].

Chapter 4

Network control systems (NCSs) have attracted much attention thanks to the development of network technologies [31–33]. NCSs are systems with loops closed through networks and have many advantages in various control problems such as smart grid, process control, and automated highway systems. In NCS problems, we design remote controllers considering the characteristics of the network. Particularly, it is important to take effects of network delays into consideration. Many controller design methods considering network delays have been studied [31,34,35]. However, in these methods, we need the knowledge of the system's model to predict a future state at a time when a determined control input is inputted to the system. If there exists uncertainty about the system's model or the network characteristic, we must design the networked controller conservatively. In order to solve the problem, RL-based controller designs are useful. In [36,37], Fujita and Ushio proposed the RL-based optimal networked control method for a linear system whose parameters are unknown with uncertain fixed network delays. However, we cannot directly apply the method to nonlinear systems. Thus, we apply the DRL algorithm to design of the networked controller for nonlinear systems.

The main contribution of this chapter is that we apply a DRL algorithm to design of a networked digital controller that stabilizes a nonlinear system at an equilibrium point, where it is assumed that

- The system is nonlinear, where the mathematical model is unknown.
- Network delays fluctuate randomly, where the maximum value is known beforehand as the worst case scenario.
- The sensor may not observe a part of system's state variables, which is called *partial observation*.

In this chapter, we regard a digital controller as an agent for DRL. Note that the agent cannot determine an action based on a true current system's state due to network delays and partial observation. If we regard the latest observed system's output as an environment's state, the agent cannot determine an action based on sufficient information to stabilize the system. Thus, we define an extended state space, whose element consists of not only the latest observed output but also some previously determined actions and some previously observed outputs, as an environment's

state space for DRL. We give the agent sufficient information to determine an action considering network delays and partial observation. This chapter is mainly based on [38] and related to [39, 40].

Chapter 5

In the control system community, control problems completing temporal control tasks such as periodic, sequential, or reactive tasks have been studied [41]. For the problems, *temporal logic* [42], which is a branch of formal methods in the computer science community, is useful to describe the tasks mathematically. Particularly, *linear temporal logic* (LTL) is a standard temporal logic to describe temporal specifications for systems. LTL has also been applied to RL for temporal control tasks [43,44], where we transform a given LTL formula into an ω-automaton that is a finite-state machine and accepts all traces satisfying the LTL formula. On the other hand, LTL cannot express time bounds of temporal control tasks. In the real world, it is often necessary to describe temporal control tasks with time bounds. Then, *metric interval temporal logic* (MITL) and *signal temporal logic* (STL) are useful [45]. MITL is an extension of LTL and has time-constrained temporal operators. Furthermore, STL is an extension of MITL. Although LTL and MITL have predicates over Boolean signals, STL has inequality formed predicates over real-valued signals [46]. The predicates of STL are useful to specify dynamical system's trajectories within bounded time intervals. Additionally, STL has a quantitative semantics called *robustness* that evaluates how well a trajectory satisfies the given STL formula. In the control system community, controller design methods to complete tasks described by STL formulae have been proposed [47, 48], where the control problems are formulated as constrained optimization problems. Furthermore, RL-based controller design methods to complete STL tasks have been proposed [49–52]. In [49], Aksaray *et al.* proposed a Q-learning algorithm for satisfying a given STL formula. The satisfaction of the given STL formula is based on a finite trajectory of the system. Thus, as an environment's state for RL, we use the extended state consisting of the current system's state and the finite-length sequence of the previous system's states. Additionally, we design a reward function using the robustness for the given STL formula. In [50], Venkataraman *et al.* proposed a tractable learning method using a flag state instead of the previous system's states to reduce the dimensionality of the environment's state space. However, these methods cannot be directly applied to problems with continuous state-action spaces because they are based on the tabular-based Q-learning algorithm [6]. For problems with continuous spaces, in [51], Balakrishnan *et al.* introduced a *partial signal* and proposed a DRL-based method to partially satisfy a given STL formula. In [52], Kapoor *et al.* proposed a model-based DRL algorithm. The model of the system is learned using a DNN, and the controller is designed using a *nonlinear model predictive control* method with evolutionary strategies [53]. On the other hand, in these previous studies, the effects of network delays are not considered.

The contribution of this chapter is that we apply a DRL algorithm to design a networked controller for completing a given STL task taking effect of network delays in consideration. Our proposed algorithm is an extension of [49] and has the following advantages in comparison with the previous study.

- We directly design a networked controller using a DRL algorithm instead of discretization of continuous spaces.
- We take the effect of network delays in consideration.

In this chapter, we regard a networked controller as an agent for DRL. We propose an extended MDP, which is called a *τd-MDP*, and a DRL-based networked controller design using the extended MDP. The τd-MDP has an extended state space whose element consists of a latest observed system's state, some previous control actions, and some previous system's states. We regard the extended state space as an environment's state space to learn the policy that completes a given STL task considering the effect of network delays. This chapter is based on our study [54].

Chapter 6

RL-based controller design methods for satisfying a given STL formula have been proposed [49–52, 54]. In these studies, we consider satisfying an STL formula as a main objective. On the other hand, in some control problems, we aim to design a policy that optimizes a given control performance index under a constraint described by an STL formula such as [47,48]. For example, in a practical application, we should operate a system in order to satisfy a given STL formula with minimum fuel costs. In this chapter, we tackle to obtain the optimal policy for a given control performance index among the policies satisfying a given STL formula without a mathematical model of a system.

The main contribution of this chapter is to propose a DRL algorithm to obtain an optimal policy for a given control performance index such as fuel costs under a constraint described by an STL formula. Our proposed algorithm has the following three advantages.

- We directly solve control problems with continuous state-action spaces.
- We aim to obtain the policy that not only satisfies a given STL formula but also is optimal with respect to a given control performance index.
- We introduce a practical two-phase learning algorithm in order to make it easy to learn a policy satisfying a given STL formula.

Our proposed algorithm is based on a DRL algorithm for problems with continuous spaces. We formulate the constrained optimal control problem as a *constrained MDP* (CMDP) [55], which has two reward functions: one is the reward function for the given control performance index and the other is the reward function for the given STL constraint. To solve the CMDP problem, we use the *Lagrangian relaxation*, which

can relax the CMDP problem into an unconstrained problem using a *Lagrangian multiplier* to utilize a standard DRL algorithm. Additionally, in the CMDP problem, it is important to satisfy a given STL constraint. The agent needs many experiences satisfying the given STL formula in order to learn how to satisfy the formula. However, it is difficult to collect the experiences considering both the control performance index and the STL constraint in the early learning stage since the agent may prioritize to optimize its policy with respect to the control performance index. Thus, in the first phase which is called *pre-training*, the agent learns its policy without the control performance index in order to obtain experiences satisfying the STL constraint easily. After obtaining sufficient experiences satisfying the STL constraint, in the second phase which is called *fine-tuning*, the agent learns its optimal policy for the control performance index under the STL constraint. This chapter is based on our study [56].

1.3 Related Works

In this section, we review studies related to this dissertation.

Model-Free DRL

Model-free DRL algorithms can solve complicated decision making problems [57,58]. Particularly, the *deep Q-network* (DQN) algorithm attracted much attention to learn human-level policies for many video games from high dimensional visual inputs [9,10]. Furthermore, the DQN algorithm has been improved using various techniques such as *double Q-learning* technique [59], *prioritized experience replay* [60], *hindsight experience replay* [61], *distributional RL* [62], *noisy networks* [63], and *dueling network* [64].

Unfortunately, the DQN algorithm cannot solve problems with continuous action spaces due to its network architecture. To solve the problem, Gu *et al.* improved the DQN architecture with a quadratic form with respect to actions [15]. In addition, for problems with continuous spaces, we often parameterize a policy using a function approximator and directly optimize the parameter vector of the policy using the policy gradient method [7]. Recently, many policy gradient-based DRL algorithms have been proposed [11–14, 16–19].

Model-Based DRL

Model-based DRL is related to **Chapter 3**. Model-based learning approaches are useful to learn the optimal policy efficiently [65]. In the approaches, the agent learns a model of the system and optimizes its policy based on the learned model in various ways such as DP [5], the *iterative linear quadratic regulation* (iLQR) method [66,67], and the *probabilistic inference for learning control* (PILCO) [68]. For safety critical systems, model-based learning approaches with *Lyapunov functions* are useful [69]. However, model-based approaches heavily depend on the accuracy of the system's model. If the

agent cannot learn an accurate model of the system, the policy learned by the model-based approach may not perform well for a real system. Thus, RL algorithms that integrate model-free and model-based approaches have been proposed [15, 70, 71]. In [15], Gu *et al.* applied *iterative linear quadratic gaussian* (iLQG) [67] based on the model learned by an *iteratively refitted time-varying linear model* [72] to accelerate continuous deep Q-learning. In [70], Nagabandi *et al.* proposed a DRL algorithm using a model predictive controller based on a learned DNN dynamics model to initialize the model-free learner. In [71], Kurutach *et al.* proposed the *model-ensemble* DRL algorithm, which uses an ensemble of DNNs to reduce the effects of model bias. The agent collects experiences through interactions with the learned DNN models and learns its policy using the *trust region policy optimization* (TRPO) algorithm [17].

On the other hand, in **Chapter 3**, it is assumed that a mathematical model of the real system is known, while its accurate system parameter vector is unknown. Thus, we use multiple virtual systems with premised system parameter vectors instead of learning the real system's model.

NCSs

NCSs are considered in **Chapters 4** and **5**. Recently, DRL has been applied to NCSs [73]. Baumann *et al.* proposed a DRL-based event-triggered controller design [74]. Burak *et al.* proposed DRL-based scheduling methods for large-scale networked control problem [75]. However, in these studies, the effects of network delays are not considered. If there exist network delays, it is necessary for the agent to receive sufficient information to learn its policy taking the effect of network delays into consideration. Our proposed methods in **Chapters 4** and **5** are related to [36, 37], which cannot directly deal with nonlinear systems. We extended the methods to nonlinear control systems using DRL algorithms. In the RL community, the decision making problems with delays have also been studied as *delayed MDPs* [76–78].

Partial Observation

Partial observation is considered in **Chapter 4**. For the partial observation problem, DRL algorithms with recurrent neural networks (RNNs) have been proposed. In [79], Hausknecht *et al.* proposed the DQN algorithm with a recurrent long short term memory to solve the Atari video games using incomplete information. In [80], Heess *et al.* proposed the policy gradient based algorithm with RNN for partially observed continuous control problems. On the other hand, in order to deal with both network delays and partial observation, we proposed the method using an extended state consisting of the latest observed output, some previously determined actions, and some previously observed outputs instead of using RNNs.

RL for Temporal Control Tasks

RL for temporal control tasks is considered in **Chapters 5** and **6**. Q-learning for satisfying STL formulae [49] is most related to our proposed methods. Moreover, we also use pre-processing to reduce the dimensionality of the environment's state space [50]. On the other hand, these methods cannot directly apply control problems with continuous state-action spaces, because they are based on the classical Q-learning algorithm [6]. Thus, we extend the previous studies using a DRL algorithm derived from Q-learning and propose a pre-processing for the DRL algorithm.

Learning methods with demonstrations have also been proposed [81, 82]. In the methods, we design a reward function using demonstrate data. On the other hand, in this dissertation, we do not use demonstrations to design a reward function for satisfying a given STL formula. Alternatively, we design it using robustness like [49].

In addition, RL-based methods for temporal control tasks described by LTL formulae have been proposed [43]. Hasanbeig *et al.* proposed a practical Q-learning algorithm with a *product MDP*. In this algorithm, we transform an LTL formula into a *limit deterministic Büchi automaton* [83], which is one of the ω-automaton, and constructed a product MDP based on the original MDP and the automaton. Moreover, the DRL-based method has also been proposed to solve problems with continuous state-action spaces [44]. In [84, 85], Li *et al.* proposed policy search algorithms using *truncated linear temporal logic* that is an extension of LTL with robustness to evaluate the satisfaction quantitatively.

Constrained DRL

Constrained DRL is considered in **Chapter 6**. The CMDP formulation is often used for sequential decision making problems with constraints [55]. To solve the constrained problems, the Lagrangian relaxation is known as a standard approach [86]. In the approach, we relax a CMDP problem into an unconstrained problem. Some algorithms with the Lagrangian relaxation have been proposed [87–90]. Other than algorithms with the Lagrangian relaxation, Achiam *et al.* proposed *constrained policy optimization* (CPO) based on the TRPO algorithm [91] and Chow *et al.* proposed an algorithm using a Lyapunov function [92, 93].

In **Chapter 6**, we consider completing a given STL task as a constraint of the optimal control problem. As a related previous study, Kalagarla *et al.* proposed an RL algorithm for an STL-constrained problem using an online learning method [94]. However, since it is based on the tabular-based RL algorithm, it cannot be directly applied to continuous control tasks.

1.4 Notation

$\mathbb{Z}_{\geq 0}$ is the set of all non-negative integers. $\mathbb{R}$ is the set of the all real numbers. $\mathbb{R}_{\geq 0}$ is the set of the all non-negative real numbers. $\mathbb{R}^n$ is the n-dimensional Euclidean space. 0_n is an n-dimensional zero vector. For a set $A \subseteq \mathbb{R}$, $\max A$ and $\min A$ are the maximum value and the minimum value in A if they exist, respectively. $|\cdot|$ denotes an absolute value. $\|\cdot\|_2$ denotes a Euclidean norm. $\#S$ is the number of elements in a set S. $E_{\epsilon \sim p}[\cdot]$ is an expected value with respect to the distribution p, where ϵ is a random variable generated from p. $\mathcal{N}(\mu, \sigma^2)$ is a normal distribution with a mean parameter μ and a variance parameter σ^2. $U(a, b)$ is a uniform distribution over the range $[a, b]$.

Chapter 2

Preliminaries

This chapter reviews a standard framework of RL and DRL as preliminaries.

2.1 Reinforcement Learning

RL is an ML method for sequential decision making problems [1,2]. In RL, an agent interacts with an environment and learns its policy through the interactions. To model the interactions, we often use an MDP which is defined as a tuple $\langle \mathcal{X}, \mathcal{A}, p_0, p, R \rangle$, where

- $\mathcal{X}$ is an environment's state space,
- $\mathcal{A}$ is an agent's action space,
- p_0 is a probability distribution for an initial state,
- $p(\cdot|x, a)$ is a probability distribution for a transition from a state $x \in \mathcal{X}$ under an action $a \in \mathcal{A}$, and
- $R : \mathcal{X} \times \mathcal{A} \times \mathcal{X} \to \mathbb{R}$ is a reward function.

If $\mathcal{X}$ and $\mathcal{A}$ are continuous spaces, p_0 and p are probability density functions.

As shown in **Fig. 2.1**, at the discrete-time $k \in \mathbb{Z}_{\geq 0}$, the agent observes a state of the environment x_k and determines an action a_k based on a stochastic policy $\pi(\cdot|x_k)$, which is a probability distribution over $\mathcal{A}$, or a deterministic policy $\mu(x_k)$, which is a function of the state. At the next discrete-time $k + 1$, the agent observes a next state $x_{k+1} \sim p(\cdot|x_k, a_k)$ and an immediate reward $r_k = R(x_k, a_k, x_{k+1})$. A tuple (x_k, a_k, x_{k+1}, r_k) obtained by the interaction is called an *experience*. The agent updates its policy using past experiences.

It is assumed that the goal of RL is to obtain the policy that maximizes a *return* $\sum_{k=0}^{\infty} \gamma^k r_k$, where $\gamma \in [0, 1)$ is a discount factor to prevent its divergence. Actually, we

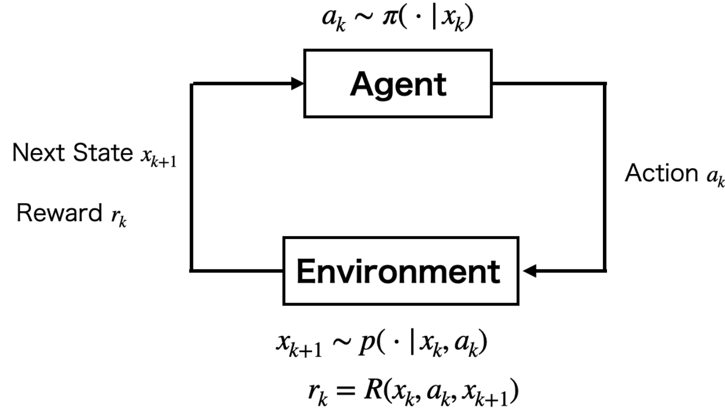


Fig. 2.1: Illustration of an interaction between an agent and an environment.

consider the expected return of the policy π defined by

$$\begin{aligned} J(\pi) &= E_{p_0,p,\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_k\right] \\ &= E_{x_0 \sim p_0, x_{k+1} \sim p(\cdot|x_k,a_k), a_k \sim \pi(\cdot|x_k)}\left[\sum_{k=0}^{\infty} \gamma^k R(x_k, a_k, x_{k+1})\right], \end{aligned} \tag{2.1}$$

where $E_{p_0,p,\pi}[\cdot]$ (or $E_{x_0 \sim p_0, x_{k+1} \sim p(\cdot|x_k,a_k), a_k \sim \pi(\cdot|x_k)}[\cdot]$) is the expected value with respect to the initial state distribution $x_0 \sim p_0$, the transition distribution $x_{k+1} \sim p(\cdot|x_k, a_k),\ k \geq 0$, and the policy distribution $a_k \sim \pi(\cdot|x_k),\ k \geq 0$.

We define a *value function* $V^\pi(x)$ and a *Q-function* $Q^\pi(x, a)$ underlying a policy π as follows:

$$\begin{aligned} V^\pi(x) &= E_{p,\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_k \mid x_0 = x\right] \\ &= E_{x_{k+1} \sim p(\cdot|x_k,a_k), a_k \sim \pi(\cdot|x_k)}\left[\sum_{k=0}^{\infty} \gamma^k R(x_k, a_k, x_{k+1}) \mid x_0 = x\right], \end{aligned} \tag{2.2}$$

$$\begin{aligned} Q^\pi(x, a) &= E_{p,\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_k \mid x_0 = x,\ a_0 = a\right] \\ &= E_{x_{k+1} \sim p(\cdot|x_k,a_k), a_k \sim \pi(\cdot|x_k)}\left[\sum_{k=0}^{\infty} \gamma^k R(x_k, a_k, x_{k+1}) \mid x_0 = x,\ a_0 = a\right], \end{aligned} \tag{2.3}$$

where $E_{p,\pi}[\cdot]$ (or $E_{x_{k+1} \sim p(\cdot|x_k,a_k), a_k \sim \pi(\cdot|x_k)}[\cdot]$) is the expected value with respect to the transition distribution $x_{k+1} \sim p(\cdot|x_k, a_k),\ k \geq 0$ and the policy distribution $a_k \sim \pi(\cdot|x_k),\ k \geq 0$.

In addition, we define an *optimal value function* and an *optimal Q-function* as follows:

$$V^*(x) = \max_\pi V^\pi(x), \ \forall x \in \mathcal{X}, \tag{2.4}$$

$$Q^*(x,a) = \max_\pi Q^\pi(x,a), \ \forall x \in \mathcal{X}, \ \forall a \in \mathcal{A}. \tag{2.5}$$

It is known that there exists at least one policy π^* that obtains Eqs. (2.4) and (2.5), which is called an *optimal policy*.

The value function and the Q-function underlying π satisfy the following equations.

$$V^\pi(x) = E_{x' \sim p(x,a), a \sim \pi(\cdot|x)} \left[R(x,a,x') + \gamma V^\pi(x') \right], \tag{2.6}$$

$$Q^\pi(x,a) = E_{x' \sim p(\cdot|x,a), a' \sim \pi(\cdot|x')} \left[R(x,a,x') + \gamma Q^\pi(x',a') \right], \tag{2.7}$$

which are called *Bellman equations*. Similarly, the optimal value function and the optimal Q-function satisfy the following equations.

$$V^*(x) = E_{x' \sim p(x,a), a \sim \pi^*(\cdot|x)} \left[R(x,a,x') + \gamma V^*(x') \right], \tag{2.8}$$

$$Q^*(x,a) = E_{x' \sim p(\cdot|x,a)} \left[R(x,a,x') + \gamma \max_{a' \in \mathcal{A}} Q^*(x',a') \right], \tag{2.9}$$

which are called *Bellman optimal equations*.

Furthermore, we define an *advantage function* underlying a policy π as follows:

$$A^\pi(x,a) = Q^\pi(x,a) - V^\pi(x). \tag{2.10}$$

Intuitively, it describes how much better it is to determine an action a in the state x than the action according to the given policy π.

2.1.1 Value Based Algorithm and Policy Based Algorithm

RL can be classified into two categories: *value based RL* and *policy based RL*. In a value based RL algorithm, an agent indirectly learns its policy through learning of a value function or a Q-function such as the Q-learning algorithm [6]. The agent learns its value function using a *temporal difference error* (TD-error) which is derived from the Bellman equation. On the other hand, in a policy based RL algorithm, an agent directly learns its policy such as the *REINFORCE* algorithm [95]. In policy based RL, we parameterize the policy using a function approximation such as a neural network. The agent updates its policy using a gradient of a return induced by the parameterized policy, where we often use the MC estimation to estimate the gradient since we cannot compute the gradient analytically. The policy based RL algorithms are known as useful approaches in the case where $\mathcal{X}$ and $\mathcal{A}$ are continuous spaces. Additionally, *actor-critic* is an important approach which has the strong points of value based RL and policy based RL. In the actor-critic approach, an agent has two components: an *actor* and a *critic*. The actor corresponds to the agent's policy and

the critic corresponds to the estimated value function (or Q-function) for the agent's learned policy, respectively. It is known that the variance of an estimated gradient can be reduced by learning the value function instead of the MC estimation. The approach has been also used in the *adaptive dynamic programming* method proposed in the control system community [96].

2.1.2 On-Policy and Off-Policy Algorithm

In terms of how to use experiences for updates of an agent's policy, RL can be classified into two categories [97]: *on-policy RL* and *off-policy RL* as shown in **Fig. 2.2**. To explain the difference between on-policy RL and off-policy RL, we define a *target policy* π_T and a *behavior policy* π_B. The target policy π_T is the policy that an agent is trying to learn using past experiences. The behavior policy π_B is the policy that is used for collecting experiences through interactions with an environment. In an on-policy RL algorithm, the target policy must correspond to the behavior policy $\pi_T = \pi_B$. For example, TRPO [17] and *proximal policy optimization* (PPO) [19] are on-policy algorithms. In general, it is known that on-policy RL algorithms tend to make the learning performance stable. However, on-policy RL algorithms result in poor sample efficiency since the algorithms requires new experiences for every update of the policy. On the other hand, in an off-policy RL algorithm, the target policy does not necessarily correspond to the behavior policy, so that off-policy RL algorithms can reuse past experiences induced by any behavior policy. In particular, sample efficiency is important for RL algorithms with nonlinear function approximations such as neural networks. Mnih *et al*. proposed a practical technique using past experience data for RL algorithms with DNNs [10]. Additionally, RL using previously collected data only without interactions has been studied as a novel approach, which is called *offline RL* [97]. The approach is effective if we have large datasets. Nevertheless, it is known that there exist some difficulties such as *distribution shift* [98].

In this dissertation, we utilize off-policy DRL algorithms to model-free optimal control of nonlinear dynamical systems since these algorithms are more efficient with respect to collecting experiences than on-policy DRL algorithms.

2.2 Q-Learning

Q-learning [6] is an off-policy value based RL algorithm. In a standard Q-learning algorithm, it is assumed that both a state space and an action space are discrete spaces, that is $\#\mathcal{X} < +\infty$ and $\#\mathcal{A} < +\infty$. An agent learns an optimal Q-function using experiences obtained through interactions with an environment. The agent has a $(\#\mathcal{X} \times \#\mathcal{A})$-dimensional *Q-table* that consists of the estimated optimal Q-value $Q_{tab}(x, a)$ for each tuple $(x, a) \in \mathcal{X} \times \mathcal{A}$. The Q-table is updated based on the following

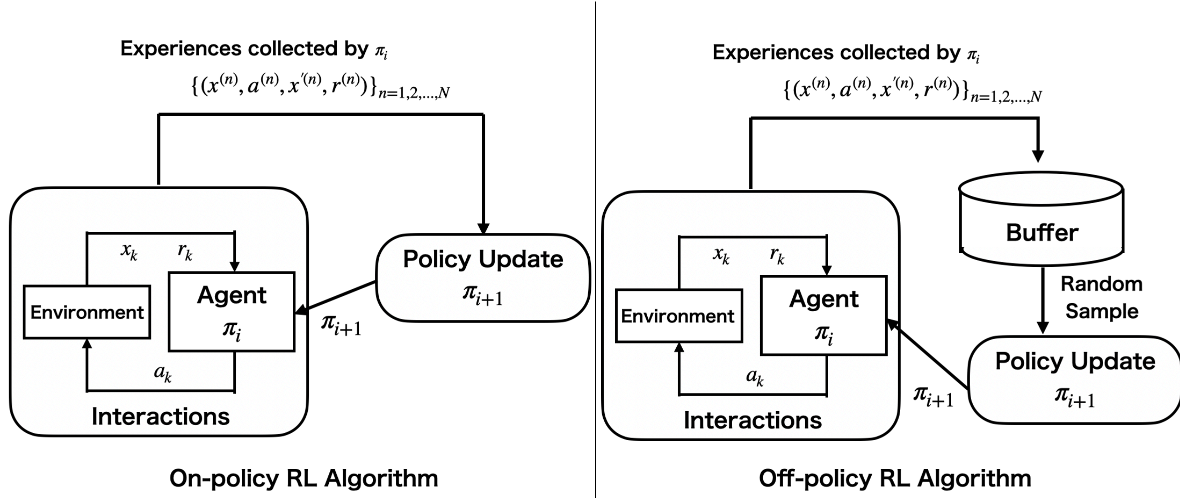


Fig. 2.2: Illustration of frameworks of on-policy RL and off-policy RL. Off-policy RL algorithms are efficient with respect to collecting experiences because they can reuse past experiences induced by any policy.

TD-error with an experience (x, a, x', r).

$$\delta_{TD} = t_{TD} - Q_{tab}(x, a), \tag{2.11}$$

where $t_{TD} = r + \gamma \max_{a' \in \mathcal{A}} Q_{tab}(x', a')$ is a target value. The agent updates the value $Q_{tab}(x, a)$ as follows:

$$Q_{tab}(x, a) \leftarrow Q_{tab}(x, a) + \alpha \delta_{TD}, \tag{2.12}$$

where $\alpha > 0$ is a learning rate. After sufficient learning of the Q-table, the agent greedily determines the action on the state x as follows:

$$a \in \arg\max_{a \in \mathcal{A}} Q_{tab}(x, a). \tag{2.13}$$

In the Q-learning algorithm, the agent often determines an action for exploration based on the *ε-greedy policy* defined by

$$\pi_\varepsilon(a|x) = \begin{cases} 1 - \varepsilon + \dfrac{\varepsilon}{\#\mathcal{A}} & \text{if } a = \arg\max_{a \in \mathcal{A}} Q_{tab}(x, a), \\ \dfrac{\varepsilon}{\#\mathcal{A}} & \text{otherwise}, \end{cases} \tag{2.14}$$

where $\varepsilon \in [0, 1]$ is a constant that is a probability of determining actions randomly.

However, in the case where states and/or actions are continuous values, such as $\mathcal{X} = \mathbb{R}^{n_x}$ and $\mathcal{A} = \mathbb{R}^{n_a}$, we cannot make the Q-table. Thus, an optimal Q-function is often parameterized using a function approximation. The following approximated linear function is one of parameterized Q-functions.

$$Q_{\theta_Q}(x, a) = \theta_Q^\top \zeta(x, a), \tag{2.15}$$

where $\theta_Q \in \mathbb{R}^{n_q}$ is a parameter vector, and $\zeta = [\zeta_1\ \zeta_2\ ...\ \zeta_{n_q}]^\top$ is a vector of the basis functions (or the features) $\zeta_i : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$, $i = 1, 2, ..., n_q$. The agent updates the parameter vector by

$$\begin{aligned}\theta_{Q,k+1} &\leftarrow \theta_{Q,k} + \alpha\delta_{TD,k}\frac{\partial Q_{\theta_Q}(x_k, a_k)}{\partial\theta_Q} \\ &= \theta_{Q,k} + \alpha\delta_{TD,k}\zeta(x_k, a_k),\end{aligned} \tag{2.16}$$

where $\alpha > 0$ is a learning rate and $\delta_{TD,k}$ is the TD error (2.11). The choice of basis functions ζ_i, $i = 1, 2, ..., n_q$ is an important issue in applying the linearized Q-learning algorithm. It is desirable to choose basis functions that can be analytically maximized with respect to the action because we must maximize the Q-function with respect to the action in order to compute the target value t_{TD} and determine the greedy action by (2.13).

2.3 Deep Q-Network (DQN)

A linear approximated Q-function may be lack of function approximation capabilities to solve complicated decision making problems. Then, Q-learning algorithm with DNNs, which is called the DQN algorithm, is useful [10]. The DQN algorithm can deal with problems where $\mathcal{X}$ is a continuous space and $\mathcal{A}$ is a discrete space. We use the DNN Q_{θ_Q} as shown in **Fig. 2.3**. The parameter vector of the DNN is denoted by θ_Q. The DNN outputs the estimated optimal Q-values $Q_{\theta_Q}(x, a)$ for each action $a \in \mathcal{A}$ on the inputted state x.

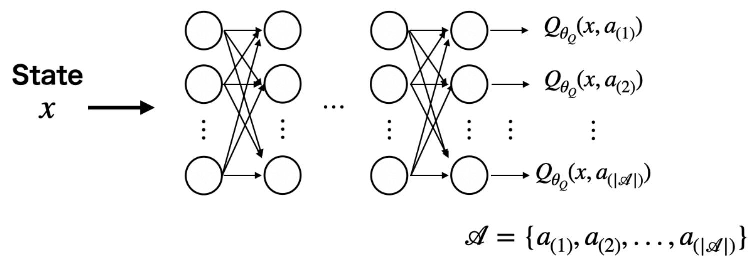


Fig. 2.3: Illustration of a DNN for the DQN algorithm. The DNN outputs an estimated optimal Q-value for each action based on the inputted state x.

The parameter vector is updated using the following two practical techniques: the *experience replay* and the *target network* technique. In the experience replay, an agent obtains experiences through interactions with an environment and stores the experiences to a *replay buffer* $\mathcal{D}$ as shown in **Fig. 2.4**, that is, the agent does not update the parameter vector θ_Q immediately after obtaining an experience. The agent samples past experiences $\{(x^{(n)}, a^{(n)}, x'^{(n)}, r^{(n)})\}_{n=1}^N$ from the replay buffer $\mathcal{D}$ randomly

and updates the parameter vector θ_Q using the past experiences. The technique reduces the correlation among experience data and prevents updates with biased data. In the target network technique, we prepare another DNN, which is called a *target DNN*, to compute a target value t_{TD} for updates of the parameter vector θ_Q as shown in **Fig. 2.5**. The parameter vector of the target DNN is denoted by θ_Q^-. Actually, the parameter vector θ_Q is updated by decreasing the following loss function derived from the TD-error.

$$J(\theta_Q) = E_{(x,a,x',r)\sim\mathcal{D}}\left[\left(r + \gamma \max_{a' \in \mathcal{A}} Q_{\theta_Q^-}(x', a') - Q_{\theta_Q}(x, a)\right)^2\right], \tag{2.17}$$

where $E_{(x,a,x',r)\sim\mathcal{D}}[\cdot]$ is the expected value with respect to selecting past experiences from the replay buffer $\mathcal{D}$ randomly. There are two approaches for updates of θ_Q^-: the *hard update* and the *soft update*. In the hard update, θ_Q^- is periodically synchronized with θ_Q. On the other hand, in the soft update, the parameter vector is slowly updated as follows:

$$\theta_Q^- \leftarrow \xi\theta_Q + (1 - \xi)\theta_Q^-, \tag{2.18}$$

where $\xi > 0$ is a sufficient small positive constant. It is important to update θ_Q^- by tracking θ_Q slowly. If we do not use the target network technique, we need to compute the target value t_{TD} using the current DQN, which is called *bootstrapping*. If we update the parameter vector of the DQN θ_Q substantially, the target value t_{TD} computed by the updated DQN may change largely, which leads to oscillations of the learning performance. It is empirically known that the target network technique can improve the learning stability.

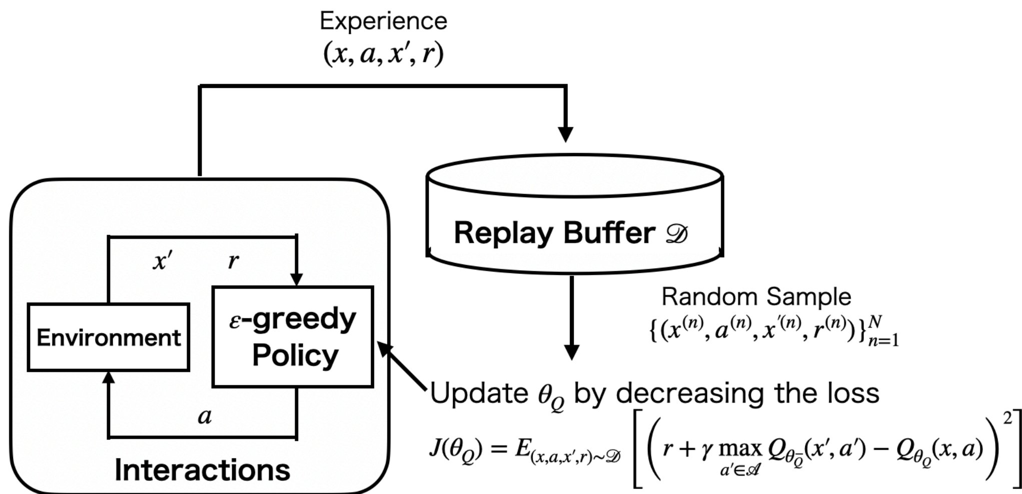


Fig. 2.4: Illustration of the experience replay.

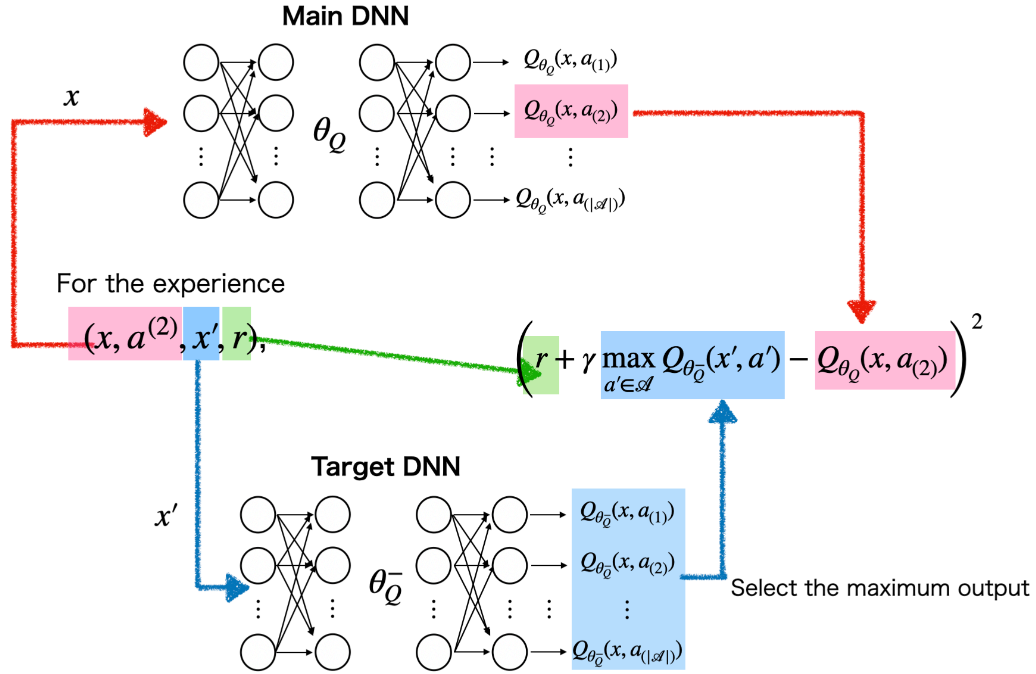


Fig. 2.5: Illustration of the target network.

In the DQN algorithm, the agent determines an exploration action based on the ε-greedy policy (2.14) like the standard Q-learning.

2.4 Continuous Deep Q-Learning with Normalized Advantage Function (NAF)

The DQN algorithm cannot solve decision making problems with continuous action spaces due to its DNN architecture shown in **Fig. 2.3**. The continuous deep Q-learning algorithm is an extension algorithm of the DQN algorithm to solve the problems with continuous spaces [15]. As shown in **Fig. 2.6**, we approximate the optimal Q-function as a DNN. The DNN separately outputs a value function term V and an advantage function term A, which is parameterized as a quadratic function of the action as follows:

$$Q_{\theta_Q}(x, a) = V_{\theta_V}(x) + A_{\theta_A}(x, a), \tag{2.19}$$

$$A_{\theta_A}(x, a) = -\frac{1}{2}(a - \mu_{\theta_\mu}(x))^\top P_{\theta_P}(x)(a - \mu_{\theta_\mu}(x)), \tag{2.20}$$

where $V_{\theta_V}(x)$ is a value function with a parameter vector θ_V, $\mu_{\theta_\mu}(x)$ is an optimal policy with a parameter vector θ_μ, and $P_{\theta_P}(x)$ is a positive definite symmetric matrix with a parameter vector of θ_P. Let $\theta_Q = \{\theta_V, \theta_\mu, \theta_P\}$, and $\theta_A = \{\theta_\mu, \theta_P\}$. $P_{\theta_P}(x)$ is

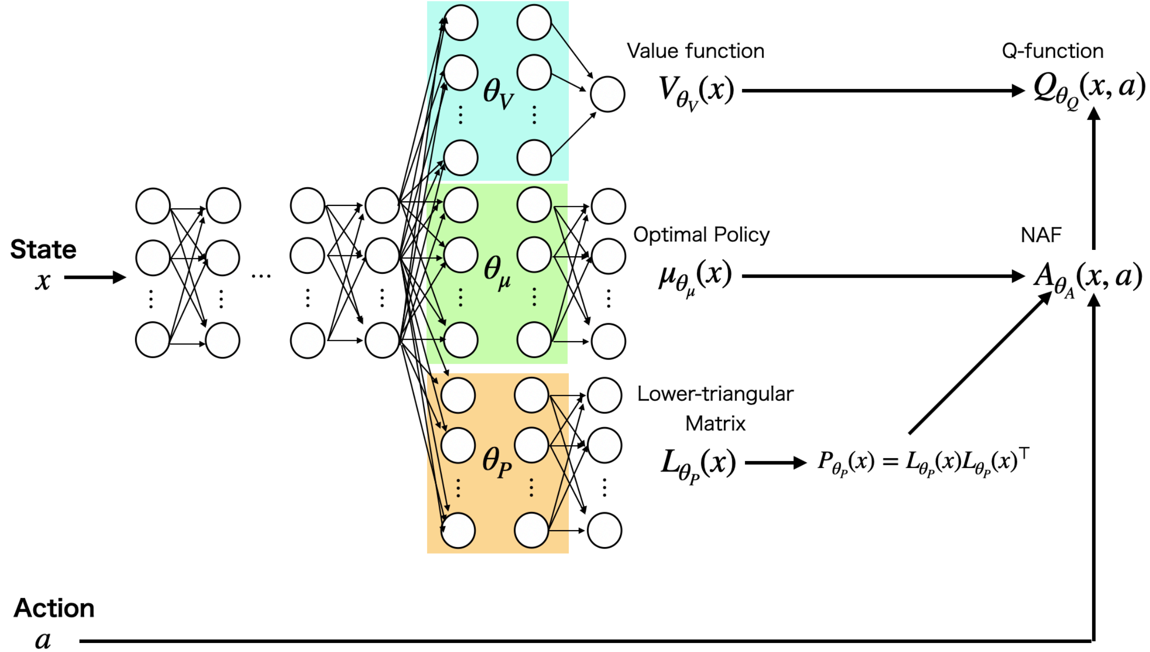


Fig. 2.6: Illustration of a DNN with a normalized advantage function (NAF). The DNN separately outputs a value function term $V_{\theta_V}(x)$ and a quadratic advantage function term $A_{\theta_A}(x,a)$ to estimate an optima Q-value $Q_{\theta_Q}(x,a)$.

a state-dependent matrix given by $P_{\theta_P}(x) = L_{\theta_P}(x)L_{\theta_P}(x)^\top$, where $L_{\theta_P}(x)$ is a lower-triangular matrix whose entries come from an output layer of the DNN. Its diagonal elements are set to be exponential to make the positive definite matrix. The advantage function term (2.20) approximated by a quadratic form with respect to the actions is called a *normalized advantage function* (NAF). Note that the approximated optimal Q-function (2.19) is more restrictive than a general DNN because the function is approximated by the quadratic form with respect to the action a. On the other hand, the quadratic form has an advantage that we can analytically maximize the approximated Q-function with respect to the action a.

The parameter vector θ_Q is updated by the following loss function.

$$
\begin{aligned}
J(\theta_Q) &= E_{(x,a,x',r)\sim\mathcal{D}}\left[\left(r + \gamma \max_{a'\in\mathcal{A}} Q_{\theta_Q^-}(x',a') - Q_{\theta_Q}(x,a)\right)^2\right] \\
&= E_{(x,a,x',r)\sim\mathcal{D}}\left[\left(r + \gamma V_{\theta_V^-}(x') - Q_{\theta_Q}(x,a)\right)^2\right]. \qquad (2.21)
\end{aligned}
$$

The parameter vector of the target DNN θ_Q^- is updated by (2.18).

In the continuous deep Q-learning algorithm, the agent determines an action for an exploration based on the following policy.

$$
\mu'(x) = \mu_{\theta_\mu}(x) + \epsilon_\mu, \qquad (2.22)
$$

where ϵ_μ is an adding noise sampled from a stochastic process such as the (discrete-time) *Ornstein-Uhlenbeck process* [99].

2.5 Deep Deterministic Policy Gradient (DDPG)

In general, if we approximate an optimal Q-function using a nonlinear function approximation such as a DNN, we cannot maximize the approximated optimal Q-function with respect to an action analytically. To solve the problem, Gu *et al.* approximated the optimal Q-function using a quadratic form [15]. On the other hand, the quadratic form may be restrictive for complicated decision making problems. Then, an actor-critic based approach is useful. In the *deep deterministic policy gradient* (DDPG) algorithm [11], Lillicrap *et al.* prepared two DNNs as shown in **Fig. 2.7**: an *actor DNN* and a *critic DNN*. The actor DNN corresponds to an optimal deterministic policy and the critic DNN corresponds to an approximated optimal Q-function. The parameter vectors of the actor DNN and the critic DNN are denoted by θ_μ and θ_Q, respectively.

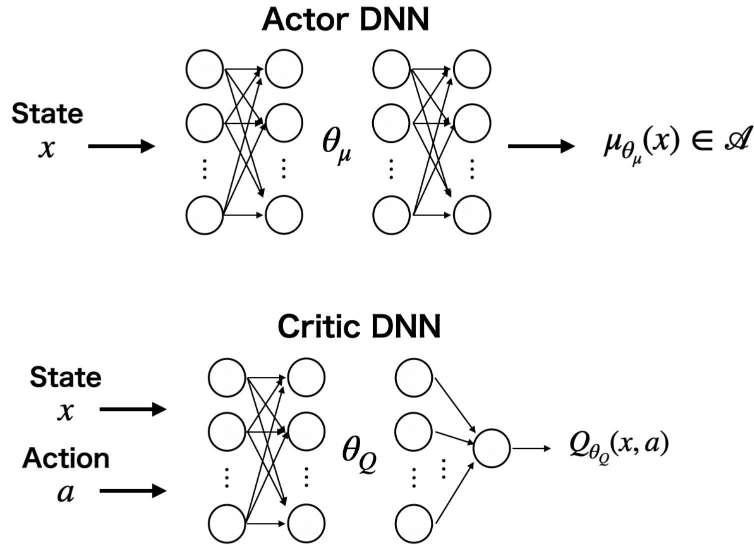


Fig. 2.7: Illustration of an actor DNN and a critic DNN. The actor DNN determines an action under the environment's state x and the critic DNN estimates the optimal Q-value for a tuple (x, a).

The parameter vector of the critic DNN θ_Q is updated by decreasing the following critic loss function.

$$J_c(\theta_Q) = E_{(x,a,x',r)\sim\mathcal{D}}\left[\left(r + \gamma Q_{\theta_Q^-}(x', \mu_{\theta_\mu^-}(x')) - Q_{\theta_Q}(x,a)\right)^2\right], \tag{2.23}$$

where $\mu_{\theta_\mu^-}$ and $Q_{\theta_Q^-}$ are a target actor DNN and a target critic DNN, respectively. The

parameter vectors θ_μ^- and θ_Q^- are updated by (2.18). The parameter vector of the actor DNN θ_μ is updated by decreasing the following actor loss function.

$$J_a(\theta_\mu) = E_{x\sim\mathcal{D}}\left[-Q_{\theta_Q}(x, \mu_{\theta_\mu}(x))\right]. \tag{2.24}$$

In other words, we maximize the optimal Q-function approximated by the critic DNN with respect to the parameter vector θ_μ.

Moreover, Fujimoto *et al.* proposed the improved DDPG algorithm which is called the *twin delayed DDPG* (TD3) algorithm [12]. In the algorithm, we apply the double Q-network technique [59] to reduce overestimation bias for updates of a critic DNN. We prepare two separate critic DNNs and two separate target critic DNN whose parameter vectors are denoted by $\theta_Q^{(1)}$, $\theta_Q^{(2)}$, $\theta_Q^{(1),-}$, and $\theta_Q^{(2),-}$. Then, the parameter vectors of critic DNNs $\theta_Q^{(1)}$ and $\theta_Q^{(2)}$ are updated by decreasing the following loss function instead of (2.23).

$$J_c(\theta_Q^{(i)}) = E_{(x,a,x',r)\sim\mathcal{D}}\left[\left(r + \gamma \min_{j=1,2} Q_{\theta_Q^{(j),-}}(x', \tilde{a}) - Q_{\theta_Q^{(i)}}(x, a)\right)^2\right],\ i = 1, 2, \tag{2.25}$$

where we use the following *target policy smoothing regularization*.

$$\tilde{a} = \mu_{\theta_\mu^-}(x) + \epsilon,\ \epsilon \sim \mathrm{Clip}(\mathcal{N}(0, \sigma^2), -c, c). \tag{2.26}$$

$\mathcal{N}(0, \sigma^2)$ is a normal distribution whose mean and standard deviation are 0 and σ, respectively. The sampled noise is clipped in $[-c, c]$ to keep it close to the original action $\mu_{\theta_\mu^-}(x)$. The parameter vector of the actor DNN θ_μ is updated by decreasing

$$J_a(\theta_\mu) = E_{x\sim\mathcal{D}}\left[-Q_{\theta_Q^{(1)}}(x, \mu_{\theta_\mu}(x))\right]. \tag{2.27}$$

The updates of the target DNNs are same as the standard DDPG algorithm. Additionally, in the TD3 algorithm, we reduce the frequency for updates of the actor DNN and the target DNNs, which is called the *delayed update*.

In both the DDPG algorithm and the TD3 algorithm, an agent determines exploration actions by (2.22) like the continuous deep Q-learning algorithm [15].

2.6 Soft Actor Critic (SAC)

Soft actor critic (SAC) is an off-policy maximum entropy DRL [13, 14]. In maximum entropy RL, the actor aims to simultaneously maximize the expected return and the entropy of the stochastic policy, that is, we aim to maximize

$$J(\pi) = E_{p_0,p,\pi}\left[\sum_{k=0}^{\infty}(\gamma^k R(x_k, a_k, x_{k+1}) + \alpha_{ent}\mathcal{H}(\pi(\cdot|x_k))\right], \tag{2.28}$$

where $\alpha_{ent} > 0$ is an entropy temperature and $\mathcal{H}(\pi(\cdot|x)) = E_{a\sim\pi(\cdot|x)}[-\log \pi(a|x)]$ is the entropy of the stochastic policy π. The entropy temperature determines the relative importance of the entropy term against the sum of rewards. It is known that the approach leads to improvement in both performance and sample efficiency. Additionally, the SAC algorithm is related to *control as inference* [100] from a theoretical point of view.

In the SAC algorithm, we prepare an actor DNN and a critic DNN whose parameter vectors are denoted by θ_π and θ_Q, respectively. The actor DNN corresponds to a gaussian formed policy as shown in **Fig. 2.8**. We represent the policy as follows:

$$a \sim \pi_{\theta_\pi}(\cdot|x) = \mu_{\theta_\pi}(x) + \sigma_{\theta_\pi}(x)\epsilon,$$

where ϵ is sampled from the standard normal distribution $\mathcal{N}(0,1)$, which is called the *reparameterization trick* [101]. The critic DNN corresponds to an estimated *soft*

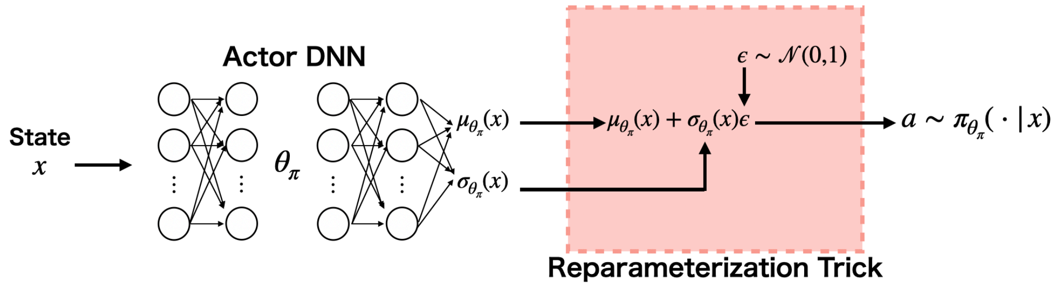


Fig. 2.8: Illustration of an actor DNN with the reparameterization trick.

Q-function for the policy π_{θ_π}. The soft Q-function underlying π is the function that satisfies the following *soft Bellman equation*.

$$Q^\pi_{soft}(x,a) = E_{x'\sim p(\cdot|x,a)}\left[R(x,a,x') + \gamma V^\pi_{soft}(x')\right],$$

where

$$V^\pi_{soft}(x') = E_{a'\sim\pi(\cdot|x')}\left[Q^\pi_{soft}(x',a') - \alpha_{ent}\log \pi(a'|x')\right].$$

The parameter vector θ_Q is updated by decreasing the following critic loss.

$$J_c(\theta_Q) = E_{(x,a,x',r)\sim\mathcal{D}}\left[\left(r + \gamma V_{\theta_Q^-}(x') - Q_{\theta_Q}(x,a)\right)^2\right], \tag{2.29}$$

where

$$V_{\theta_Q^-}(x') = E_{a'\sim\pi_{\theta_\pi}(\cdot|x')}\left[Q_{\theta_Q^-}(x',a') - \alpha_{ent}\log \pi_{\theta_\pi}(a'|x')\right].$$

θ_Q^- is the parameter vector of the target critic DNN, which is updated by (2.18). The parameter vector of the actor DNN θ_π is updated by decreasing the following actor loss function.

$$J_a(\theta_\pi) = E_{x\sim\mathcal{D},\epsilon\sim\mathcal{N}(0,1)}\left[\alpha_{ent}\log\pi_{\theta_\pi}(f_{\theta_\pi}(\epsilon;x)|x) - Q_{\theta_Q}(x, f_{\theta_\pi}(\epsilon;x))\right], \tag{2.30}$$

where $f_{\theta_\pi}(\epsilon;x) = \mu_{\theta_\pi}(x) + \sigma_{\theta_\pi}(x)\epsilon$. Moreover, the entropy temperature $\alpha_{ent} > 0$ is updated by decreasing the following loss function.

$$J_e(\alpha_{ent}) = E_{x\sim\mathcal{D},a\sim\pi_{\theta_\pi}(\cdot|x)}\left[\alpha_{ent}(-\log\pi_{\theta_\pi}(a|x) - \mathcal{H}_0)\right], \tag{2.31}$$

where $\mathcal{H}_0$ is a hyper-parameter. In [14], the hyper-parameter is selected based on the dimensionality of the action space. Additionally, we utilize the double Q-network technique [12].

Finally, we summarize the development of the Q-learning algorithm in **Fig. 2.9**.

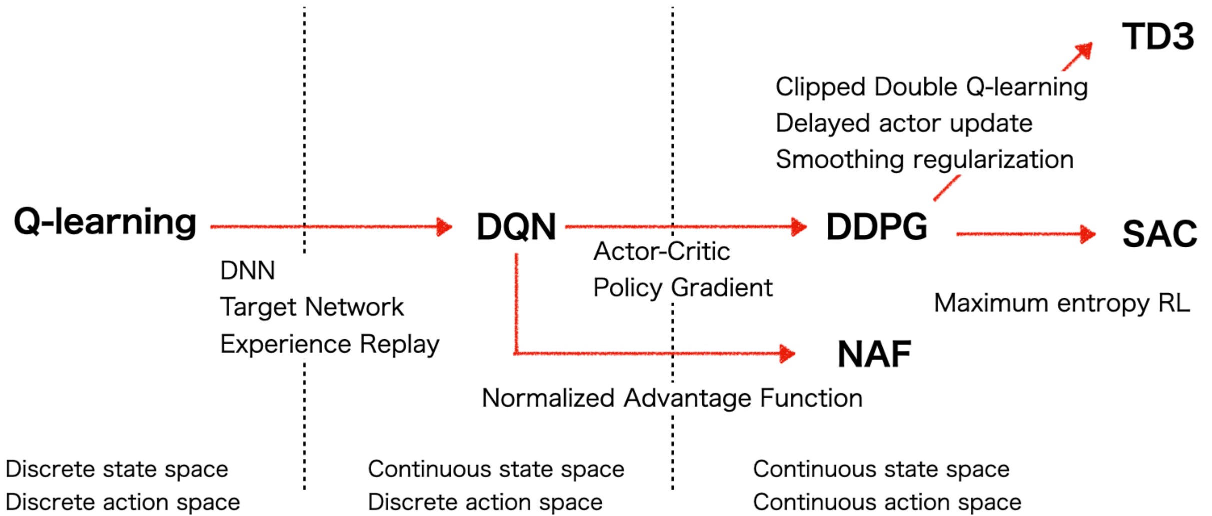


Fig. 2.9: Illustration of development of Q-learning.

Chapter 3

Deep Q-Learning with a Simulator for Stabilization

A simulator that predicts the behavior of a real system is helpful for DRL because we can collect experiences more efficiently than through interactions with the real system. However, we need a mathematical model to use the simulator. In the case where there are identification errors, experiences obtained by the simulator may degrade the performance of the learned policy for the real system. Thus, we propose a two-stage practical DRL algorithm with the simulator. In the first stage, we prepare multiple premised systems in the simulator and obtained approximated optimal Q-functions for these systems with multiple DNNs. In the second stage, we represent a Q-function for the real system as an approximated linear function whose basis functions are the approximated deep Q-functions pre-trained in the first stage. The approximated linear Q-function is learned through interactions with a real system. Additionally, we apply our proposed algorithm to systems whose system parameters vary slowly.

This chapter is based on "Continuous deep Q-learning with a simulator for stabilization of uncertain discrete-time systems" [30] which appeared in *Nonlinear Theory and Its Applications*, © 2021 IEICE.

3.1 Problem Formulation

We consider the following discrete-time nonlinear deterministic dynamical system.

$$x_{k+1} = f(x_k, a_k \,|\xi), \tag{3.1}$$

where $x_k \in \mathcal{X}$ $(\subseteq \mathbb{R}^{n_x})$ and $a_k \in \mathcal{A}$ $(\subseteq \mathbb{R}^{n_a})$ are the state and control input at the discrete-time $k \in \{0, 1, ...\}$, respectively. $f : \mathcal{X} \times \mathcal{A} \to \mathcal{X}$ is a system dynamics. It is assumed that the form of f is known while we cannot accurately identify the *system parameter vector* $\xi = [\xi_1 \ \xi_2 \ ... \ \xi_p]^\top \in \Xi$ $(\subseteq \mathbb{R}^p)$, where Ξ is a premised compact set and known beforehand.

In this chapter, we apply DRL to stabilization of the system (3.1) at the target state $x^* \in \mathcal{X}$ that is one of the fixed points such that

$$x^* = f(x^*, 0_{n_a}|\xi).$$

Then, we regard the system and the controller as the environment and the agent, respectively. To emphasize the DRL-based controller design, control inputs determined by the agent are called control actions. The reward function is defined as follows:

$$R(x,a) = -(x - x^*)^\top R_1 (x - x^*) - a^\top R_2 a, \tag{3.2}$$

where $R_1 \in \mathbb{R}^{n_x \times n_x}$ and $R_2 \in \mathbb{R}^{n_a \times n_a}$ are positive definite matrices. The quadratic reward function is a standard form to stabilize a dynamical system in the modern control theory such as the *linear quadratic regulator* (LQR) control [102]. It takes the maximum value 0 at the target state x^* with $a = 0_{n_a}$.

If we have a mathematical model of a real system, a simulator would be useful for the RL-based controller design. The simulator predicts the behavior of a real system using a mathematical model (3.1) with a given system parameter vector ξ. Many experiences can be collected more efficiently than through interactions with the real system. Thus, we consider a DRL algorithm with the simulator. In general, however, we may have an identification error in the system parameter vector even if the mathematical model is known. The experiences obtained by the simulator degrade the performance of the learned policy for a real system and, in the worst case, the policy may make the real system unstable. On the other hand, the identification error is small, the policy learned in the simulator may stabilize the real system almost as desired. Based on these points of view, we propose a practical DRL algorithm that takes the identification error into account.

3.2 Practical Q-Learning with Pre-Trained Multiple Deep Q-Networks

Although we can easily collect many experiences using a simulator, the experiences obtained by the simulator may degrade the performance of the learned policy due to identification error. To solve the problem, we propose a practical DRL algorithm that consists of two stages, as shown in **Fig. 3.1**. In the first stage, we choose N system parameter vectors $\xi^{(j)}, j = 1, 2, ..., N$ from the premised system parameter set Ξ. Then, for each chosen parameter vector $\xi^{(j)}$, we have the mathematical model $f(x, a|\xi^{(j)})$, which is called a virtual system with $\xi^{(j)}$. Using the simulator, we collect experiences in order to learn an approximated optimal Q-function $Q_j^*(x, a|\theta^{Q_j})$ for each virtual system with $\xi^{(j)}$ using the continuous deep Q-learning algorithm [15]. In the second stage, we represent a Q-function for the real system as an approximated

linear Q-function whose basis functions are approximated Q-functions learned for virtual systems $f(x,a|\xi^{(j)})$, $j = 1,2,...,N$ in the first stage. The agent learns the parameter vector of the approximated linear Q-function through interactions with the real system.

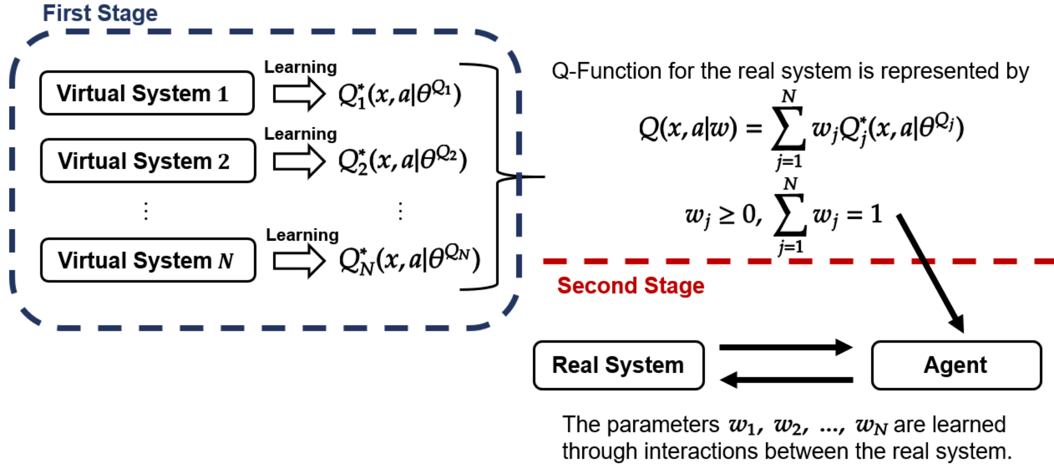


Fig. 3.1: Illustration of our proposed method consisting of two stages. In the first stage, we choose N system parameter vectors $\xi^{(j)}$, $j = 1,2,...,N$ from Ξ and prepare N virtual systems in a simulator. We obtain the approximated optimal Q-functions Q_j^*, $j = 1,2,...,N$, for virtual systems using the continuous deep Q-learning algorithm. In the second stage, we represent the approximated linear Q-function for the real system with approximated optimal Q-functions for virtual systems as basis functions. The agent learns the parameter vector $w = [w_1\ w_2\ ...\ w_N]^\top$ by interacting with the real system.

3.2.1 The Q-Function for the Real System

In the first stage, we obtain the approximated optimal Q-function $Q_j^*(x,a|\theta^{Q_j})$ for each virtual system with $\xi^{(j)}$ using the continuous deep Q-learning. In the second stage, we approximate a Q-function for the real system as follows:

$$Q(x,a|w) = \sum_{j=1}^{N} w_j Q_j^*(x,a|\theta^{Q_j}), \tag{3.3}$$

where $w = [w_1\ w_2\ ...\ w_N]^\top$ is a parameter vector. It is assumed that $\forall j \in \{1,2,...,N\}$, $w_j \geq 0$ and $\sum_{j=1}^{N} w_j = 1$. The basis functions are approximated optimal Q-functions $Q_j^*(x,a|\theta^{Q_j})$, $j = 1,2,...,N$ learned for virtual systems with $\xi^{(j)}$, $j = 1,2,...,N$. The agent learns the parameter vector w through interactions with the real system, as shown in **Fig. 3.1**. In general, we can represent the Q-function for the real system as an arbitrary function of the pre-trained optimal Q-functions.

Nevertheless, the approximated linear form (3.3) has the advantage that we can compute a greedy action analytically.

The greedy action $\mu(x|w)$ maximizes $Q(x, a|w)$, that is,

$$\begin{aligned}\mu(x|w) &\in \arg\max_a Q(x, a|w)\\ &= \arg\max_a \sum_{j=1}^{N} w_j Q_j^*(x, a|\theta^{Q_j})\\ &= \arg\max_a \left(\sum_{j=1}^{N} w_j V_j^*(x|\theta^{V_j}) + \sum_{j=1}^{N} w_j A_j^*(x, a|\theta^{A_j}) \right). \quad (\because (2.19))\end{aligned}$$

Because the term $\sum_{j=1}^{N} w_j V_j^*(x|\theta^{V_j})$ is independent of the action a, we have

$$\mu(x|w) \in \arg\max_a \sum_{j=1}^{N} w_j A_j^*(x, a|\theta^{A_j}). \tag{3.4}$$

To compute the action that maximizes the Q-function (3.3), we solve

$$\frac{\partial}{\partial a} \sum_{j=1}^{N} w_j A_j^*(x, a|\theta^{A_j}) = 0. \tag{3.5}$$

From (3.5), we have

$$\begin{aligned}&\frac{\partial}{\partial a} \sum_{j=1}^{N} w_j \left(-\frac{1}{2}(a - \mu_j^*(x|\theta^{\mu_j}))^\top P_j^*(x|\theta^{P_j})(a - \mu_j^*(x|\theta^{\mu_j})) \right) = 0\\ \Leftrightarrow &-\sum_{j=1}^{N} w_j P_j^*(x|\theta^{P_j})(a - \mu_j^*(x|\theta^{\mu_j})) = 0\\ \Leftrightarrow &\sum_{j=1}^{N} w_j P_j^*(x|\theta^{P_j}) a = \sum_{j=1}^{N} w_j P_j^*(x|\theta^{P_j}) \mu_j^*(x|\theta^{\mu_j}).\end{aligned} \tag{3.6}$$

Then, $\sum_{j=1}^{N} w_j P_j^*(x|\theta^{P_j})$ is a positive definite symmetric matrix because $w_j \geq 0$ (there exists at least one parameter satisfying $w_j > 0$) and the parameter matrices of NAFs P_j^* are positive definite and symmetric. Hence, there is an inverse matrix

$(\sum_{j=1}^{N} w_j P_j^*(x|\theta^{P_j}))^{-1}$ and the stationary solution $\hat{a}$ of (3.5) is as follows:

$$\begin{aligned}\hat{a}(x) &= \left(\sum_{m=1}^{N} P_m^*(x|\theta^{P_m})\right)^{-1} \sum_{j=1}^{N} w_j P_j^*(x|\theta^{P_j}) \mu_j^*(x|\theta^{\mu_j}) \\ &= \sum_{j=1}^{N} \left(\sum_{m=1}^{N} P_m^*(x|\theta^{P_m})\right)^{-1} w_j P_j^*(x|\theta^{P_j}) \mu_j^*(x\theta^{\mu_j}) \\ &= \sum_{j=1}^{N} \tilde{w}_j(x) \mu_j^*(x|\theta^{\mu_j}), \end{aligned} \tag{3.7}$$

where

$$\tilde{w}_j(x) = \left(\sum_{m=1}^{N} w_m P_m^*(x|\theta^{P_m})\right)^{-1} w_j P_j^*(x|\theta^{P_j}).$$

Note that $\sum_{j=1}^{N} w_j P_j^*(x|\theta^{P_j})$ is a positive definite symmetric matrix because the parameter matrices of NAFs $P_j^*(x|\theta^{P_j}),\ j = 1, ..., N$ are positive definite symmetric matrices. Then, because the Hessian matrix

$$\frac{\partial^2}{\partial a^2} \sum_{j=1}^{N} w_j A_j^*(x, a|\theta^{A_j}) = -\sum_{j=1}^{N} w_j P_j^*(x|\theta^{P_j})$$

is negative definite, $\sum_{j=1}^{N} w_j A_j^*(x, a|\theta^{A_j})$ is concave with respect to a. The stationary solution $\hat{a}(x)$ is the global optimal solution, that is $\mu(x|w) = \hat{a}(x)$.

Remark: We consider the parameter vector $\alpha_w w,\ \alpha_w > 0$. Then, from (3.5), we have the following equation to compute a greedy action.

$$\frac{\partial}{\partial a} \sum_{j=1}^{N} \alpha_w w_j A_j^*(x, a|\theta^{A_j}) = 0.$$

The solution of the equation is same as the solution for the parameter vector w shown in (3.7), that is, the greedy action does not depend on the size of the parameter vector. Thus, we assume that $\sum_{j=1}^{N} w_j = 1$.

3.2.2 Q-Learning for the Real System with Deep Q-Networks Learned for Multiple Virtual Systems

The agent learns the parameter vector w so as to reduce the TD-error, where the parameters must satisfy the conditions that, for any $j \in \{1, 2, ..., N\}$, $w_j \geq 0$ and $\sum_{j=1}^{N} w_j = 1$.

At first, we consider the following loss function that evaluates the TD-error for an experience $e = (x, a, x', r)$.

$$\mathcal{L}(w) = \frac{1}{2}\left(t - Q(x, a|w)\right)^2, \tag{3.8}$$

where

$$t = r + \gamma Q(x', a'|w).$$

We regard the target value t as a constant value, where a' is the greedy action determined by (3.7) under x'. From the first-order approximation at w, we obtain

$$\mathcal{L}(w + \Delta w) - \mathcal{L}(w) \simeq \frac{\partial \mathcal{L}(w)}{\partial w}\Delta w,$$

where the Euclidean norm $\|\Delta w\|_2$ is small. We set $\Delta w = -\alpha\frac{\partial \mathcal{L}(w)}{\partial w}$, where α is a sufficiently small positive value such that $\mathcal{L}(w + \Delta w) - \mathcal{L}(w) < 0$. Thus, we update w based on the following rule to minimize the loss function (3.8):

$$w^{(k+1)} \leftarrow w^{(k)} - \alpha \left.\frac{\partial \mathcal{L}(w)}{\partial w}\right|_{w=w^{(k)}}, \tag{3.9}$$

where $w^{(k)}$ and $w^{(k+1)}$ are a current and a next parameter vector, respectively.

Remark: The update vector of the parameter vector w is

$$\alpha\frac{\partial \mathcal{L}(w)}{\partial w} = -\alpha\delta\frac{\partial Q(x, a|w)}{\partial w}, \tag{3.10}$$

where $\delta = r + \gamma \max_{a'} Q(x', a'|w) - Q(x, a|w)$ for the experience (x, a, x', r). The size of the update vector $\alpha\frac{\partial \mathcal{L}(w)}{\partial w}$ depends on the approximated optimal Q-functions for the virtual systems because $\frac{\partial Q(x,a|w)}{\partial w} = [Q_1^*(x, a|\theta^{Q_1}) \ \cdots \ Q_N^*(x, a|\theta^{Q_N})]^\top$. If we can obtain these optimal Q-functions precisely, their outputs are close to zero near the target state because of the reward function (3.2). Thus, the size of the update vector is small near the target state.

Next, we consider an update rule of w_j satisfying for any $j \in \{1, 2, ..., \}$, $w_j \geq 0$. Then, we use the following loss function with a barrier term to keep all parameters nonnegative.

$$\mathcal{L}^B(w) = \mathcal{L}(w) + \eta B(w), \tag{3.11}$$

where $\eta > 0$ is a constant and $B(w)$ is a barrier function. Let $\mathcal{W} = \{w \in \mathbb{R}^N | \ \forall n, \ w_n + \epsilon_w > 0\}$, where $\epsilon_w > 0$ is an arbitrarily small constant. The interior and boundary of

the set $\mathcal{W}$ are denoted by $\mathrm{int}\mathcal{W}$ and $\partial\mathcal{W}$, respectively. The barrier function is given by

$$B(w)\begin{cases} > 0 & w \in \mathrm{int}\mathcal{W}, \\ \to +\infty & w \to \partial\mathcal{W}. \end{cases} \tag{3.12}$$

In our proposed method, we use the following update equation to learn the parameter vector w:

$$\begin{aligned} w^{(k+1)} &\leftarrow w^{(k)} - \alpha \left.\frac{\partial \mathcal{L}^B(w)}{\partial w}\right|_{w=w^{(k)}} \\ &= w^{(k)} - \alpha \left(\left.\frac{\partial \mathcal{L}(w)}{\partial w}\right|_{w=w^{(k)}} + \eta \left.\frac{\partial B(w)}{\partial w}\right|_{w=w^{(k)}} \right). \end{aligned} \tag{3.13}$$

Note that all elements of the parameter vector updated by (3.13) may not always be nonnegative values. Thus, we reduce the update rate α when at least one negative element exists in the updated parameter vector $w^{(k+1)}$. In our proposed algorithm, we continue reducing the learning rate by half until all elements of the updated parameter vector are nonnegative. After updating the parameter vector using (3.13), we normalize it to satisfy $\sum_{j=1}^{N} w_j = 1$ as follows:

$$w^{(k+1)} \leftarrow \frac{1}{\sum_{j=1}^{N} w_j^{(k+1)}} w^{(k+1)}.$$

The agent indirectly learns its policy for the real system by adjusting the parameter vector w.

3.2.3 Proposed Algorithm

Our proposed learning algorithm with multiple virtual systems is shown in **Algorithms 1**. The outline is as follows: In line 1, we choose N system parameter vectors $\xi^{(j)},\ j = 1, 2, ..., N$ from the premised set Ξ. In line 2, we obtain the approximated optimal Q-functions $Q_j^*,\ j = 1, 2, ..., N$, for virtual systems using the continuous deep Q-learning algorithm [15]. In line 3, we initialize the parameter vector w. In line 4, we initialize the state of the real system. From lines 5 to 20, the agent learns the parameter vector w through interactions with the real system online. From lines 12 to 18, if some elements of the updated parameter vector are negative after one update, we reduce the learning rate to keep all elements of the updated parameter vector nonnegative. In line 19, we normalize the parameter vector after the update to ensure that $\sum_{j=1}^{N} w_j = 1$.

Algorithm 1 Q-learning for the real system with multiple virtual systems

1: Choose system parameter vectors $\{\xi^{(j)}\}_{j=1,2,...,N}$ in the premised set Ξ.
2: Obtain approximated optimal Q-functions $\{Q_j^*\}_{j=1,...,N}$ for virtual systems by the continuous deep Q-learning algorithm.
3: Initialize the parameter vector $w^{(0)} = [w_1^{(0)} \ ... \ w_N^{(0)}]^\top$.
4: Initialize the state x_0.
5: **for** Discrete-time $k = 0, ..., K$ **do**
6: Observe the state x_k.
7: Determine the action $a_k = \arg\max_{a \in \mathcal{A}} Q(x_k, a | w^{(k)})$.
8: Add the exploration noise a_k by (3.7).
9: Execute the action a_k to the real system.
10: Receive the next state x_{k+1} and the reward r_k computed by Eq. (3.2).
11: $l \leftarrow 0$.
12: **while** True **do**
13: Compute the next parameter vector $w^{(k+1)}$:
$w^{(k+1)} \leftarrow w^{(k)} - \alpha 2^{-l}\left(\frac{\partial \mathcal{L}(w^{(k)})}{\partial w} + \eta \frac{\partial B(w^{(k)})}{\partial w}\right)$.
14: **if** all elements of $w^{(k+1)}$ are positive **then**
15: **break**
16: **end if**
17: $l \leftarrow l + 1$.
18: **end while**
19: Normalize the parameter vector $w^{(k+1)}$:
$w^{(k+1)} \leftarrow \frac{1}{\sum_{j=1}^{N} w_j^{(k+1)}} w^{(k+1)}$.
20: **end for**

3.3 Example

We consider the following discrete-time system.

$$\begin{bmatrix} x_{1,k+1} \\ x_{2,k+1} \end{bmatrix} = \begin{bmatrix} x_{1,k} + dx_{2,k} \\ x_{2,k} + d(g\sin(x_{1,k}) - \xi_1 x_{2,k} + \xi_2 a_k) \end{bmatrix}, \tag{3.14}$$

where $g = 9.81$ and $d = 2^{-4}$. The state and action spaces are $\mathcal{X} = \mathbb{R}^2$ and $\mathcal{A} = [-1, 1]$, respectively. We assume an uncertain parameter vector of the real system $\xi = [\xi_1 \ \xi_2]^\top$ lies in a region $\Xi = \{(\xi_1, \xi_2) | \ 0 \le \xi_1 \le 1, \ 5 \le \xi_2 \le 50\}$. We prepare the following virtual systems, as shown in **Fig. 3.2**:

virtual system-1 $\xi^{(1)} = (\xi_1^{(1)}, \xi_2^{(1)}) = (0.0, 5.0)$,
virtual system-2 $\xi^{(2)} = (\xi_1^{(2)}, \xi_2^{(2)}) = (1.0, 5.0)$,
virtual system-3 $\xi^{(3)} = (\xi_1^{(3)}, \xi_2^{(3)}) = (0.0, 50.0)$,

virtual system-4 $\xi^{(4)} = (\xi_1^{(4)}, \xi_2^{(4)}) = (1.0, 50.0)$,
virtual system-5 $\xi^{(5)} = (\xi_1^{(5)}, \xi_2^{(5)}) = (0.4, 16.0)$,
virtual system-6 $\xi^{(6)} = (\xi_1^{(6)}, \xi_2^{(6)}) = (0.6, 16.0)$,
virtual system-7 $\xi^{(7)} = (\xi_1^{(7)}, \xi_2^{(7)}) = (0.4, 32.0)$,
virtual system-8 $\xi^{(8)} = (\xi_1^{(8)}, \xi_2^{(8)}) = (0.6, 32.0)$.

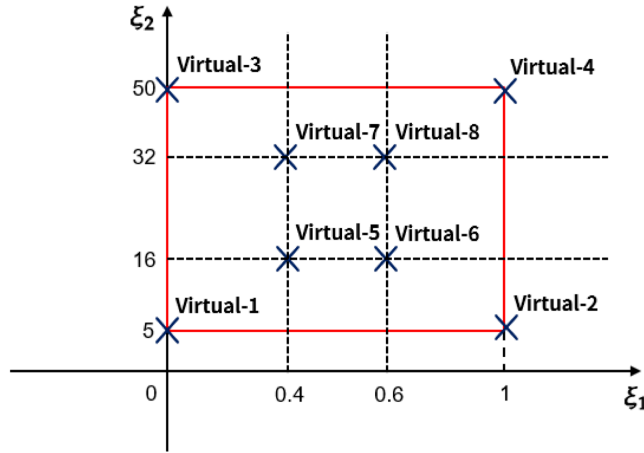


Fig. 3.2: Parameter region Ξ where the real system parameter vector lies. The parameters of the virtual systems are denoted by cross marks.

We use

$$R_1 = \begin{bmatrix} 1.0 & 0 \\ 0 & 0.1 \end{bmatrix}, \quad R_2 = 10.0$$

as the positive definite matrices in the reward function (3.2). Let the target state be the origin $x^* = 0_2$ that is a fixed point of the system (3.14).

We use the same DNN architecture to learn the approximated optimal Q-functions for all the virtual systems. The DNN has four hidden layers, all of which have 128 units, and all layers are fully connected. The activation functions are rectified linear unit (ReLU) functions except for the output layers. Regarding the activation functions of the output layers, we use hyperbolic tangent functions for units of greedy actions $\mu(\cdot|\theta^\mu)$ and linear functions for the other units. The size of the replay buffer $\mathcal{D}$ is 1.0×10^6 and the size of the mini-batch is $I = 128$. The parameter vectors of DNNs are updated by *Adam* [103]. In these simulations, the learning rate for **virtual system-1** is 5.0×10^{-4}, the learning rates for **virtual system-2, 5, 6** are 5.0×10^{-5}, and the learning rates for **virtual system-3, 4, 7, 8** are 1.0×10^{-4}. The update rate of the target network is $\tau = 0.005$. The discounted factor is $\gamma = 0.99$. We use the following

Orenstein-Uhlenbeck process to generate exploration noise ϵ_k^{OU}.

$$\begin{aligned}\epsilon_{k+1}^{\mathrm{OU}} &= \epsilon_k^{\mathrm{OU}} + p_1(p_2 - \epsilon_k^{\mathrm{OU}}) + p_3\epsilon', \\ \epsilon_0^{\mathrm{OU}} &= 0,\end{aligned}$$

where $(p_1, p_2, p_3) = (0.15, 0.0, 0.3)$ and ϵ' is the noise generated by the standard normal distribution $\mathcal{N}(0,1)$. The approximated optimal Q-function and the optimal policy for the virtual system with $\xi^{(j)}$ are denoted by Q_j^* and μ_j^*, respectively. In the first stage, the agent learns its approximated optimal Q-function and policy through $2.0 \times 10^4 \sim 4.0 \times 10^4$ interactions. The policies learned for the virtual systems are shown in **Fig. 3.3**. Although their characteristics are different from each other, all actions determined by the policies are close to zero around the target state $x^* = 0_2$ because it is a fixed point of the system (3.14).

We define the *score*

$$G(\mu|\xi) = \sum_{k=0}^{1000} R(x_k, \mu(x_k)) \tag{3.15}$$

as the performance index of the policy μ for the real system with $\xi \in \Xi$, where

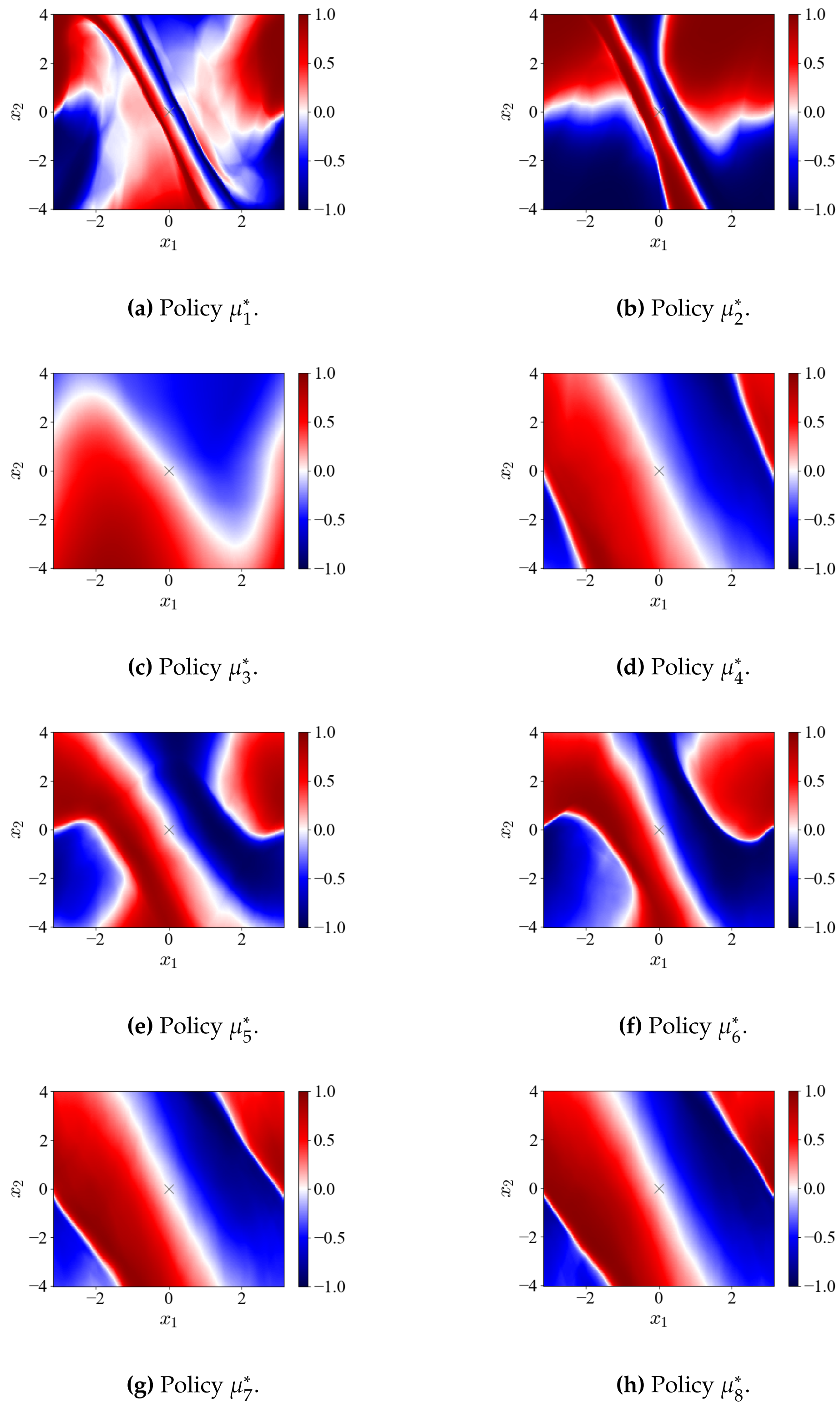
$$x_{k+1} = f(x_k, a_k|\xi),\ x_0 = [\pi\ 0]^\top.$$

In the following simulations, if the agent obtains a score that is smaller than -2000, we consider that the agent's policy μ does not perform well for the real system with ξ. To show the performance of the policy μ_j^*, we plot the scores as shown in **Fig. 3.4**. We show the scores for real systems with $\xi = (\xi_1, \xi_2) \in \Xi_{\mathrm{plot}}$, where $\Xi_{\mathrm{plot}} = \{0.05, 0.15, ..., 0.95\} \times \{5.5, 6.5, ..., 49.5\}$. The policy μ_j^* performs well for real systems whose system parameter vectors are close to $\xi^{(j)}$. However, it is shown that the policy learned with the simulator does not perform well for the real system if there exists a large identification error. Thus, we apply our proposed method. The barrier function (3.12) is given by

$$B(w) = -\sum_{j=1}^{N} \log(w_j + \epsilon_w),\ w \in \mathrm{int}\mathcal{W}, \tag{3.16}$$

where $\mathcal{W} = \{w|\ \forall j \in \{1, 2, ..., N\},\ w_j + \epsilon_w > 0\}$. $B(w)$ diverges as the parameter w approaches the boundary $\partial\mathcal{W}$ as shown in **Fig. 3.5**. It is known as the *log barrier function*. We set $\eta = 1.0 \times 10^{-7}$ and $\epsilon_w = 1.0 \times 10^{-9}$, respectively.

All experiments were run on a computer with an Intel(R) Core(TM) i7-10700 @ 2.9GHz processor and 32GB of memory and were conducted using Python software.



(a) Policy μ_1^*.

(b) Policy μ_2^*.

(c) Policy μ_3^*.

(d) Policy μ_4^*.

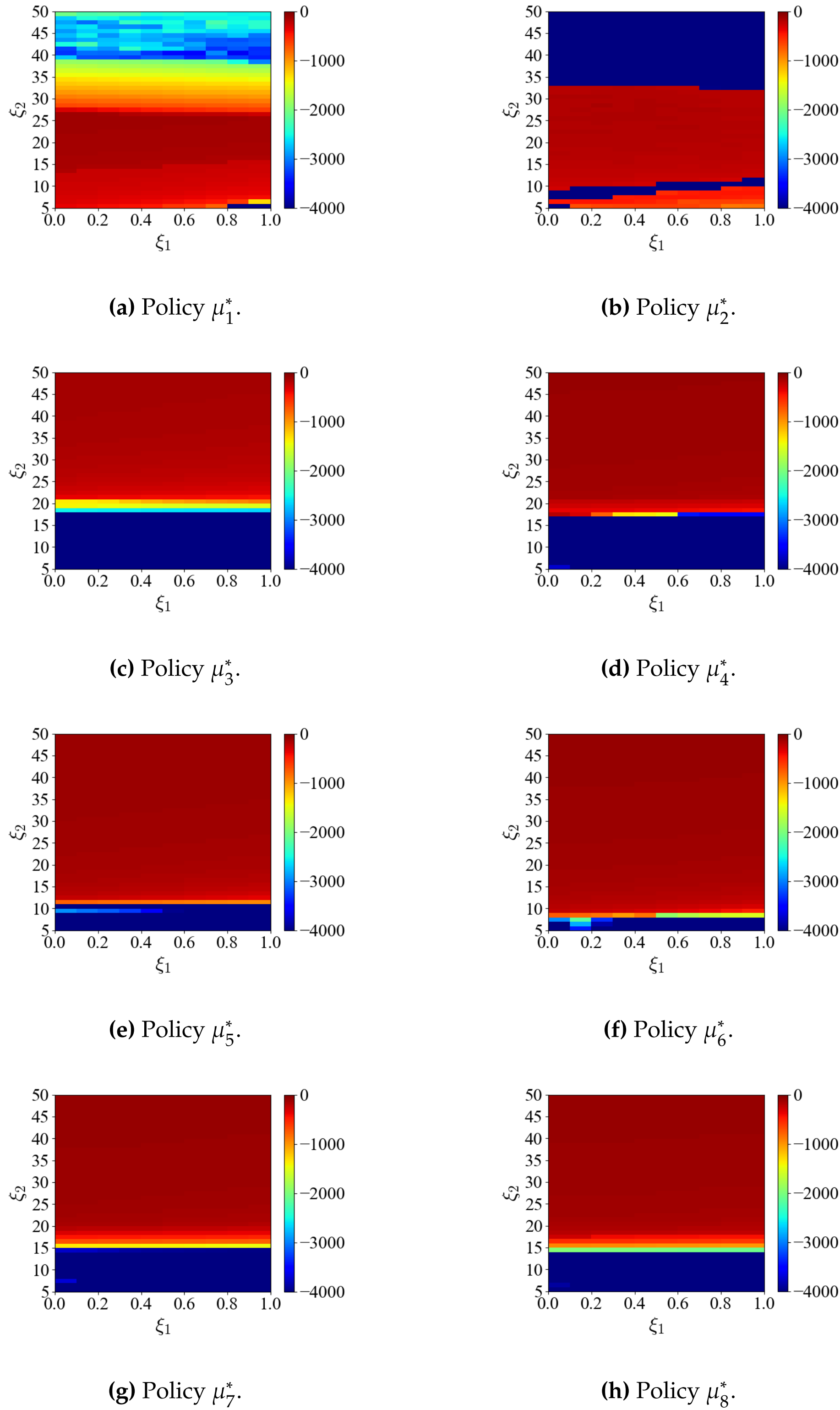
(e) Policy μ_5^*.

(f) Policy μ_6^*.

(g) Policy μ_7^*.

(h) Policy μ_8^*.

Fig. 3.3: Illustrations of policies learned for virtual systems. The fixed points of the system (3.14) are denoted by cross marks.



(a) Policy μ_1^*.

(b) Policy μ_2^*.

(c) Policy μ_3^*.

(d) Policy μ_4^*.

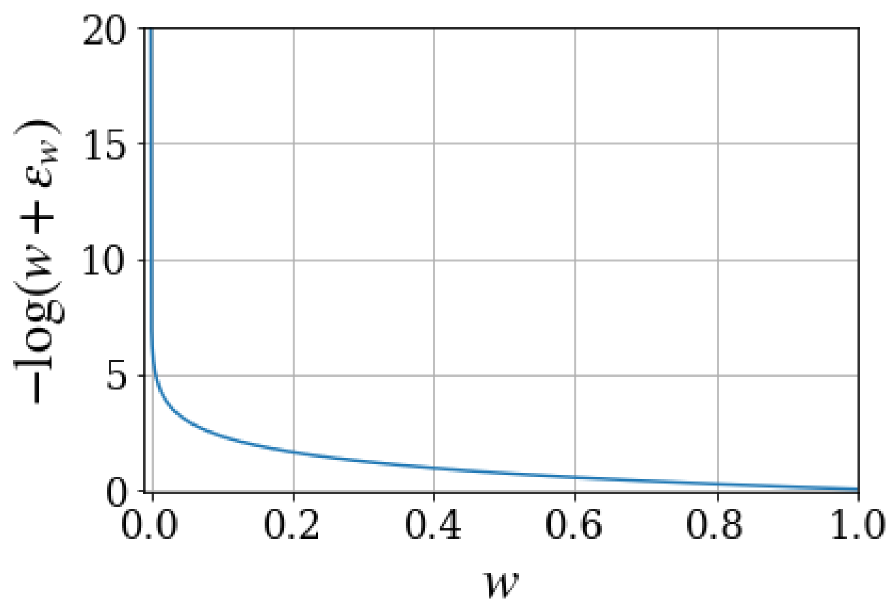
(e) Policy μ_5^*.

(f) Policy μ_6^*.

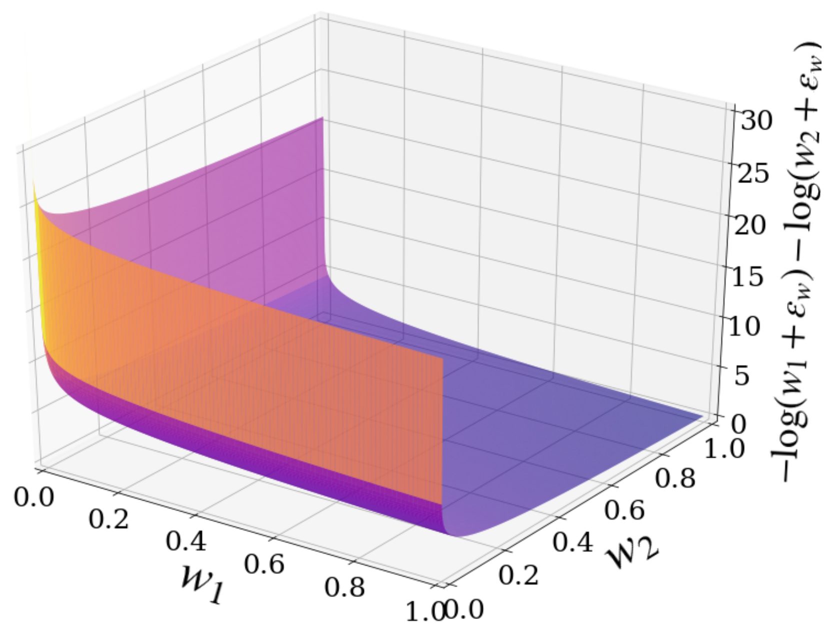
(g) Policy μ_7^*.

(h) Policy μ_8^*.

Fig. 3.4: Scores of pre-trained policies for real systems with $\xi = (\xi_1, \xi_2) \in \Xi_{\text{plot}}$, where $\Xi_{\text{plot}} = \{0.05, 0.15, ..., 0.95\} \times \{5.5, 6.5, ..., 49.5\}$. Each grid shows the score of the pre-trained policy μ_j^*, $j = 1, 2, ..., N$, for the system with ξ.



(a) $N = 1$



(b) $N = 2$

Fig. 3.5: Illustration of log barrier functions.

3.3.1 Choice of Basis Functions

We discuss the relationship between the choice of basis functions and the performance of the learned policy. The exploration noise for the second stage is generated by

$$\epsilon_k = 0.1\frac{\max\{400-k, 0\}}{400}\epsilon', \tag{3.17}$$

where noise ϵ' is generated by the standard normalized distribution $\mathcal{N}(0, 1)$. We do not add exploration noises after the 400th step. The initial state is $[\pi\ 0]^\top$. The learning rate is $\alpha = 5.0 \times 10^{-5}$. The maximum step is $K = 1000$. We set the elements of the initial parameter vector, $w_j = \frac{1}{N},\ j = 1, 2, ..., N$.

Table 3.1: Choice of basis functions for the Q-function.

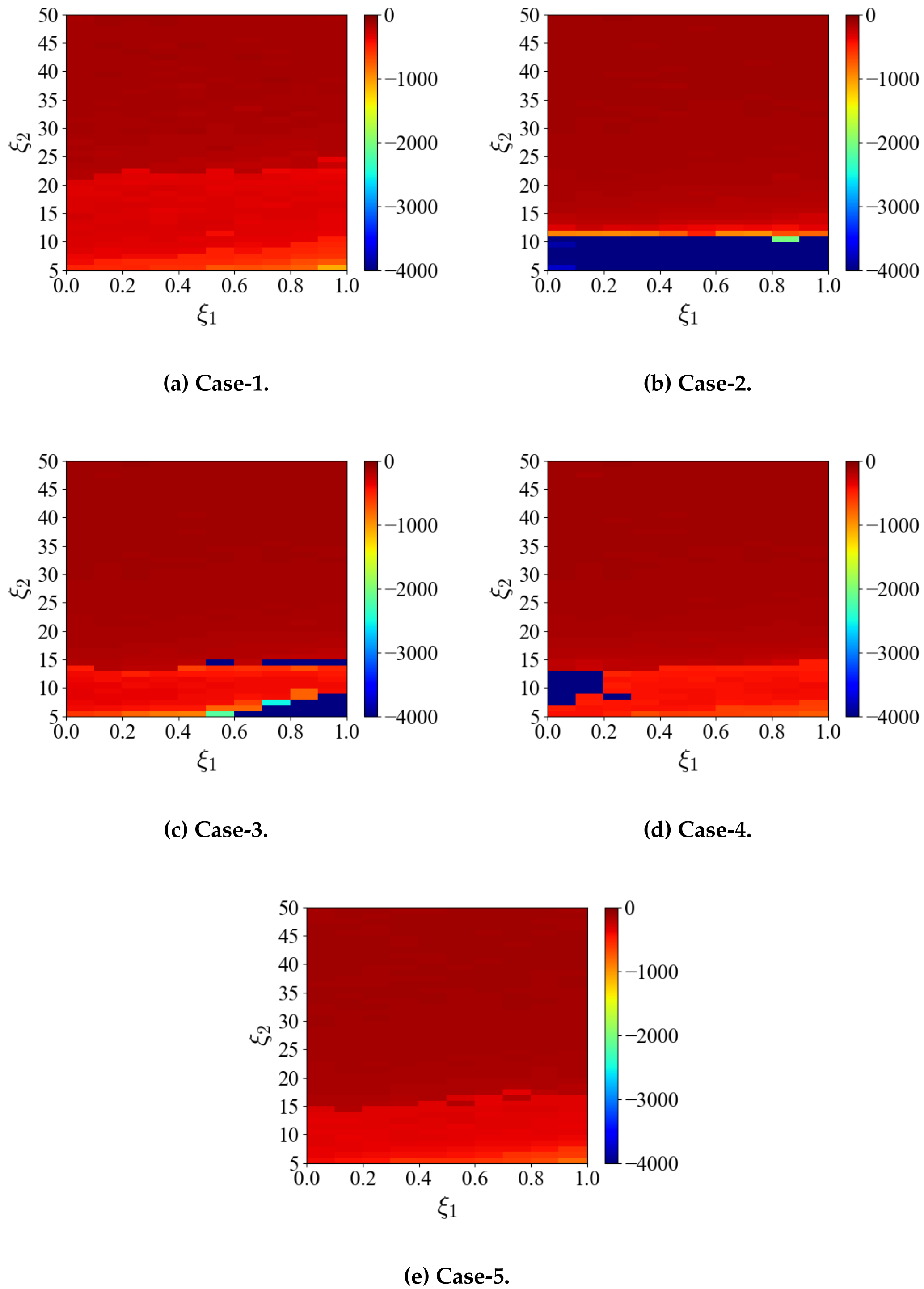
Case number	Choice of basis functions
Case-1	$\{Q_1^*,\ Q_2^*,\ Q_3^*,\ Q_4^*\}$
Case-2	$\{Q_5^*,\ Q_6^*,\ Q_7^*,\ Q_8^*\}$
Case-3	$\{Q_1^*,\ Q_6^*,\ Q_7^*,\ Q_8^*\}$
Case-4	$\{Q_5^*,\ Q_2^*,\ Q_7^*,\ Q_8^*\}$
Case-5	$\{Q_1^*,\ Q_2^*,\ Q_7^*,\ Q_8^*\}$

First, we assume that $N = 4$ and consider the five cases summarized in **Table 3.1** for the choice of basis functions. For each case, **Fig. 3.6** shows scores of policies learned by our proposed algorithm for real systems with $\xi \in \Xi_{\text{plot}}$. The scores for **Case-1** are shown in **Fig. 3.6(a)**. The agent with $\{Q_1^*, Q_2^*, Q_3^*, Q_4^*\}$ learns policies that perform well for real systems with $(\xi_1, \xi_2) \in \Xi_{\text{plot}}$. For example, we show the time response for a real system with $(\xi_1, \xi_2) = (0.95, 5.5)$ in **Fig. 3.7**. It can be seen that the agent stabilizes the real system around the target state. The scores for **Case-2** are shown in **Fig. 3.6(b)**. The policy learned with $\{Q_5^*, Q_6^*, Q_7^*, Q_8^*\}$ does not perform well for a system if its system parameter ξ_2 is less than 10. For example, the time response of the real system with $(\xi_1, \xi_2) = (0.95, 5.5)$ is shown in **Fig. 3.8**. It shows that the agent cannot stabilize the real system. Intuitively, when the parameter ξ_2 is large, the effect of parameter ξ_1 on the system (3.14) is small. On the other hand, when ξ_2 is small, the effect is large, that is, the difference in ξ_1 is sensitive. Thus, both Q_1^* and Q_2^* are necessary to represent the Q-function for a real system with a small ξ_2. We consider **Case-3** and **Case-4** to confirm that Q_1^* and Q_2^* are necessary to stabilize a real system whose system parameter ξ_2 is small. The scores for **Case-3** and **Case-4** are shown in **Figs. 3.6(c)** and **3.6(d)**, respectively. They show that the learned policies do not perform well for some real systems. Consequently, we need both Q_1^* and Q_2^* as basis functions. In **Case-5** where we use both Q_1^* and Q_2^*, the agent learns its policy that performs well for systems with $\xi \in \Xi_{\text{plot}}$, as shown in **Fig. 3.6(e)**.

Next, we consider an adequate number of basis functions chosen to achieve a good performance. The learned policies μ_3^*, μ_4^*, μ_5^*, μ_6^*, μ_7^*, and μ_8^* perform well for real systems if ξ_2 is larger than 35, as shown in **Fig. 3.4**. If we choose at least one of $\{Q_3^*, Q_4^*, ..., Q_8^*\}$ as the basis functions, the agent may learn a policy that performs well for such real systems. We then consider the case in which we choose $\{Q_1^*, Q_2^*, Q_4^*\}$ as the basis functions. The scores of the policies learned for real systems with $\xi \in \Xi_{\text{plot}}$ are shown in **Fig. 3.9(b)**. The agent with $\{Q_1^*, Q_2^*, Q_4^*\}$ learns policies that perform well for systems with $\xi \in \Xi_{\text{plot}}$. On the other hand, if we choose Q_1^* and Q_2^* only, the agent does not learn policies that perform well for real systems if ξ_2 is larger than 35, as shown in **Fig. 3.9(a)**. The results indicate that it is sufficient to choose Q_1^*, Q_2^*, and at least one of $\{Q_3^*, Q_4^*, ..., Q_8^*\}$. Moreover, we consider the case where we choose all the approximated optimal Q-functions learned for virtual systems as basis functions. The scores of the policies learned for real systems with $\xi \in \Xi_{\text{plot}}$ are shown in **Fig. 3.9(c)**. Although the agent performs well for most real systems, it does not learn the policy that performs well for the real system with $(\xi_1, \xi_2) = (0.95, 5.5)$. If we choose basis functions redundantly, the proposed algorithm may not perform well for the real system.

In our proposed method, through interactions with the real system, the agent learns the approximated linear Q-function whose basis functions are approximated as optimal Q-functions learned for virtual systems. To achieve a good performance for a set of system parameter vectors as large as possible, we choose a set of approximated optimal Q-functions such that system parameter sets stabilized by the approximated optimal Q-functions are complementary to each other. Moreover, it is desirable to reduce the number of basis functions to the extent possible.

In general, the choice of basis functions depends on the characteristics of the system. Therefore, we choose basis functions through trial and error. An important direction of future work is to propose a method for the choice of adequate basis functions that achieve the desired control performance.



(a) Case-1.

(b) Case-2.

(c) Case-3.

(d) Case-4.

(e) Case-5.

Fig. 3.6: Scores of policies learned by our proposed method online for real systems with $\xi = (\xi_1, \xi_2) \in \Xi_{\text{plot}}$. Each grid shows the score $G(\mu(\cdot|w)|\xi)$ for the real system with ξ.

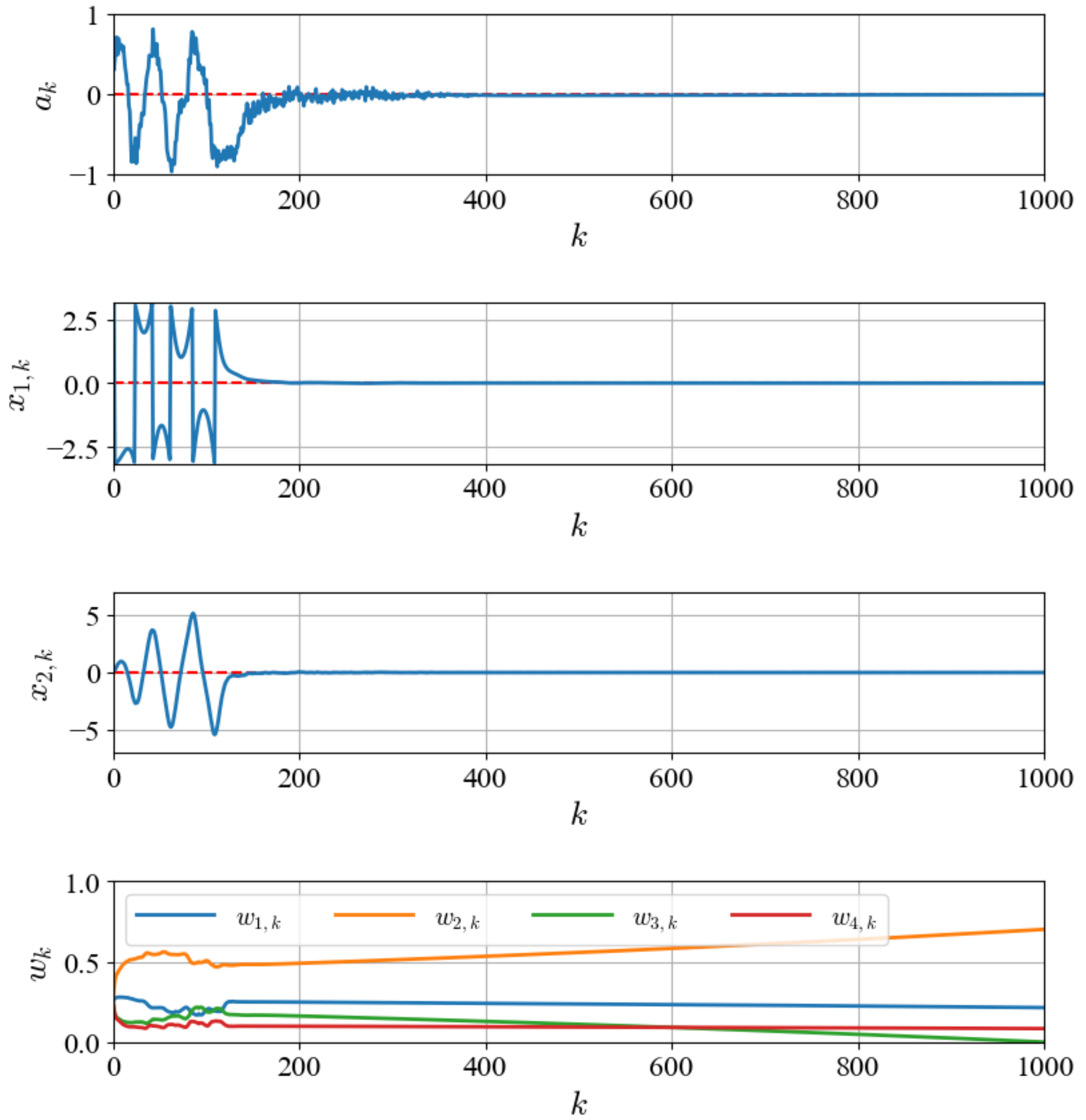


Fig. 3.7: Time response of the real system with $(\xi_1, \xi_2) = (0.95, 5.5)$ controlled by the agent that learns its policy with $\{Q_1^*, Q_2^*, Q_3^*, Q_4^*\}$ using our proposed method online, where w_j is the weight of the approximated optimal Q-function Q_j^*.

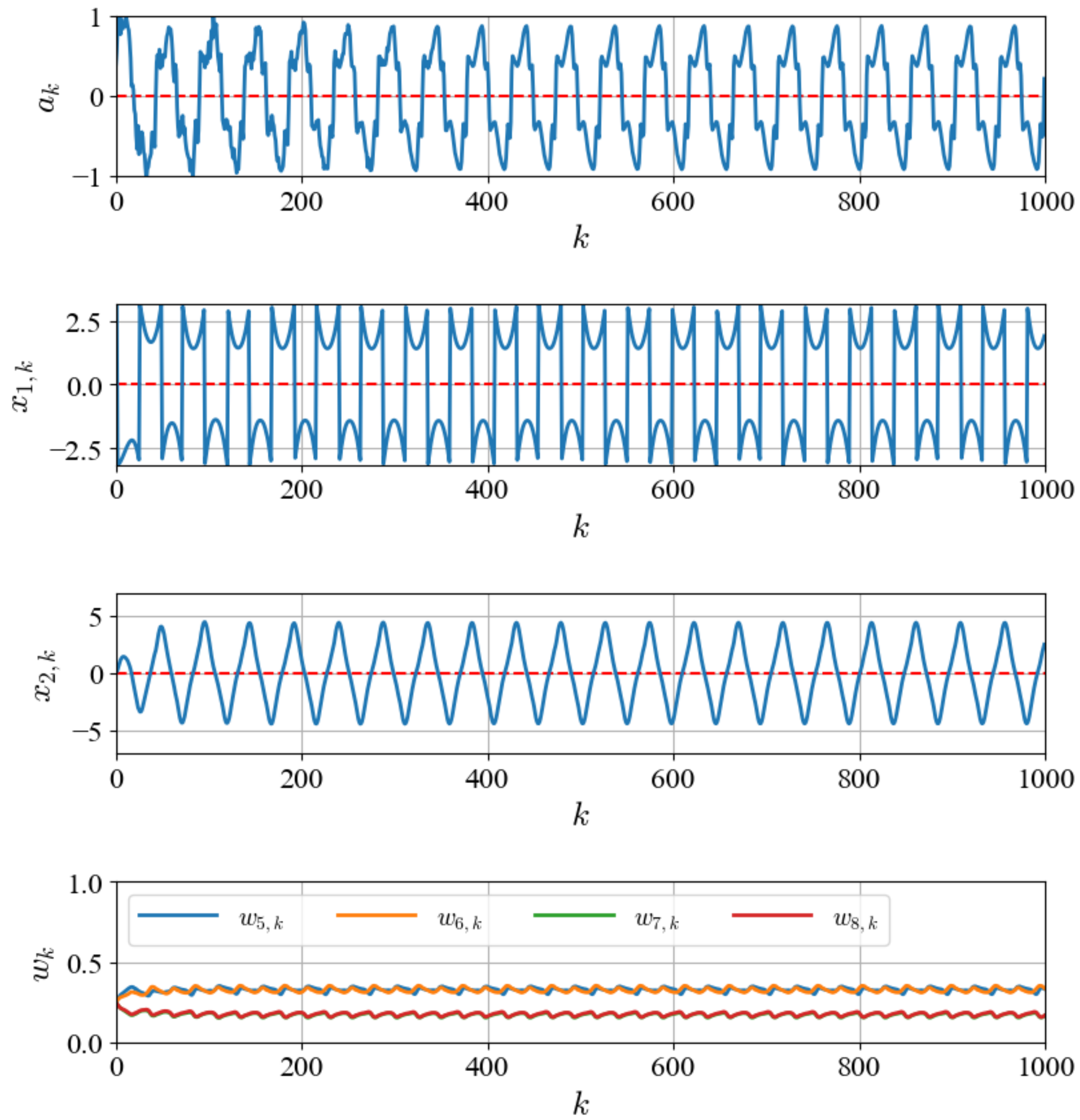
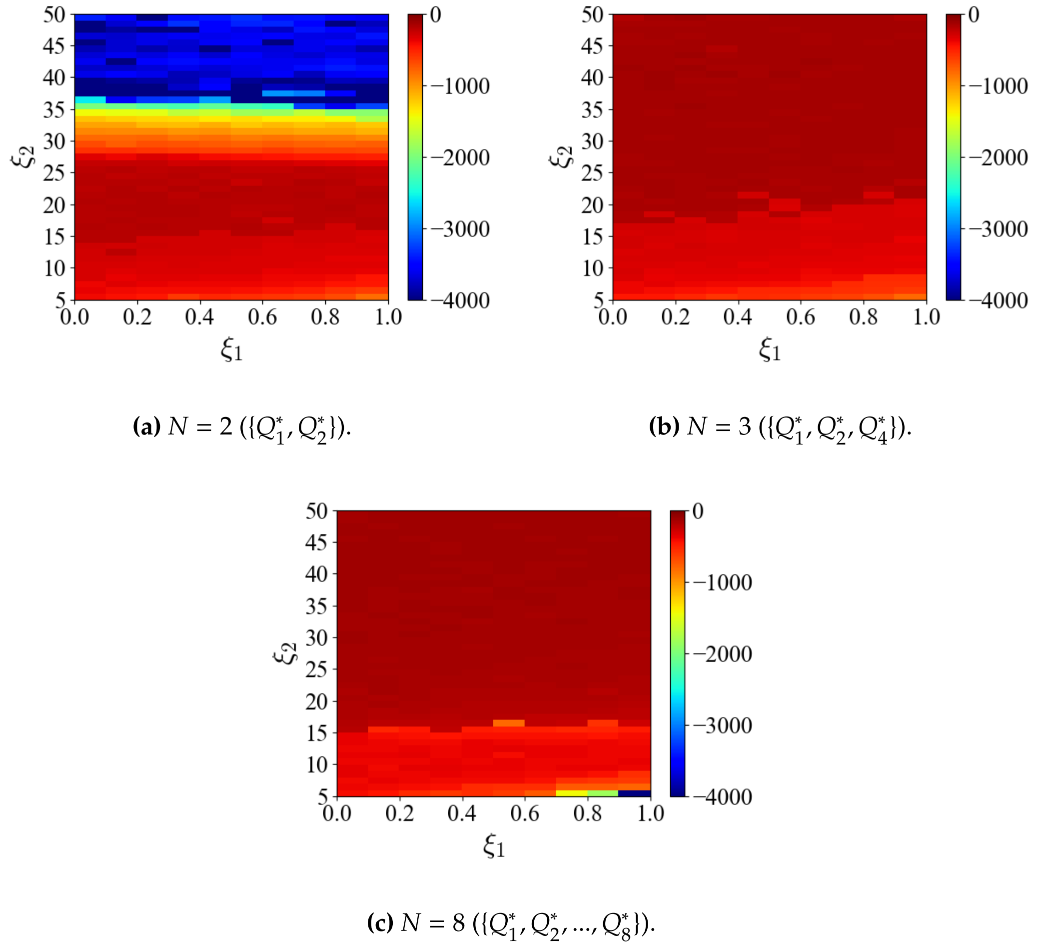


Fig. 3.8: Time response of the real system with $(\xi_1, \xi_2) = (0.95, 5.5)$ controlled by the agent that learns its policy with $\{Q_5^*, Q_6^*, Q_7^*, Q_8^*\}$ using our proposed method online, where w_j is the weight of the approximated optimal Q-function Q_j^*.



(a) $N = 2$ ($\{Q_1^*, Q_2^*\}$).

(b) $N = 3$ ($\{Q_1^*, Q_2^*, Q_4^*\}$).

(c) $N = 8$ ($\{Q_1^*, Q_2^*, ..., Q_8^*\}$).

Fig. 3.9: Scores of policies learned by our proposed method online for real systems with $\xi = (\xi_1, \xi_2) \in \Xi_{\text{plot}}$. We consider three cases: $\{Q_1^*, Q_2^*\}$, $\{Q_1^*, Q_2^*, Q_4^*\}$, and $\{Q_1^*, Q_2^*, ..., Q_8^*\}$. Each grid shows $G(\mu(\cdot|w)|\xi)$ for the real system with $\xi = (\xi_1, \xi_2) \in \Xi_{\text{plot}}$.

3.3.2 Adaptivity under Variation of the System Parameter Vector

We show that our proposed method can be applied to a real system whose system parameter vector varies slowly. In the following, we choose $\{Q_1^*, Q_2^*, Q_4^*\}$ as the basis functions. The initial parameter vector of the Q-function is $[1/3\ 1/3\ 1/3]^\top$. We add exploration noise to actions if $\|x\|_2 \geq 0.05$. The noise is generated from a standard normalized distribution $\mathcal{N}(0, 1)$ and multiplied by 0.1. The initial state is $[\pi\ 0]^\top$. The learning rate is $\alpha = 5.0 \times 10^{-5}$. The maximum step is $K = 1000$.

First, it is assumed that the system parameter ξ_2 increases gradually from 5.0 to 50.0 until $k = 200$, where $\xi_1 = 1.0$. The time response of the real system is shown in **Fig. 3.10**. Although the Euclidean norm of the system's state becomes larger than 0.05 after $k = 200$, the agent adds the exploration noise to its action and learns the parameter vector w.

Second, it is assumed that the system parameter ξ_2 decreases gradually from 50.0 to 5.0 until $k = 200$, where $\xi_1 = 1.0$. The time response of the real system is shown in **Fig. 3.11**. Although the Euclidean norm of the state becomes larger than 0.05, between $k = 200$ and $k = 700$, the agent can control the real system to the target state by learning w online.

The above results indicate that the agent can adapt to a real system whose system parameter vector varies within the premised set Ξ by online learning the parameter vector w.

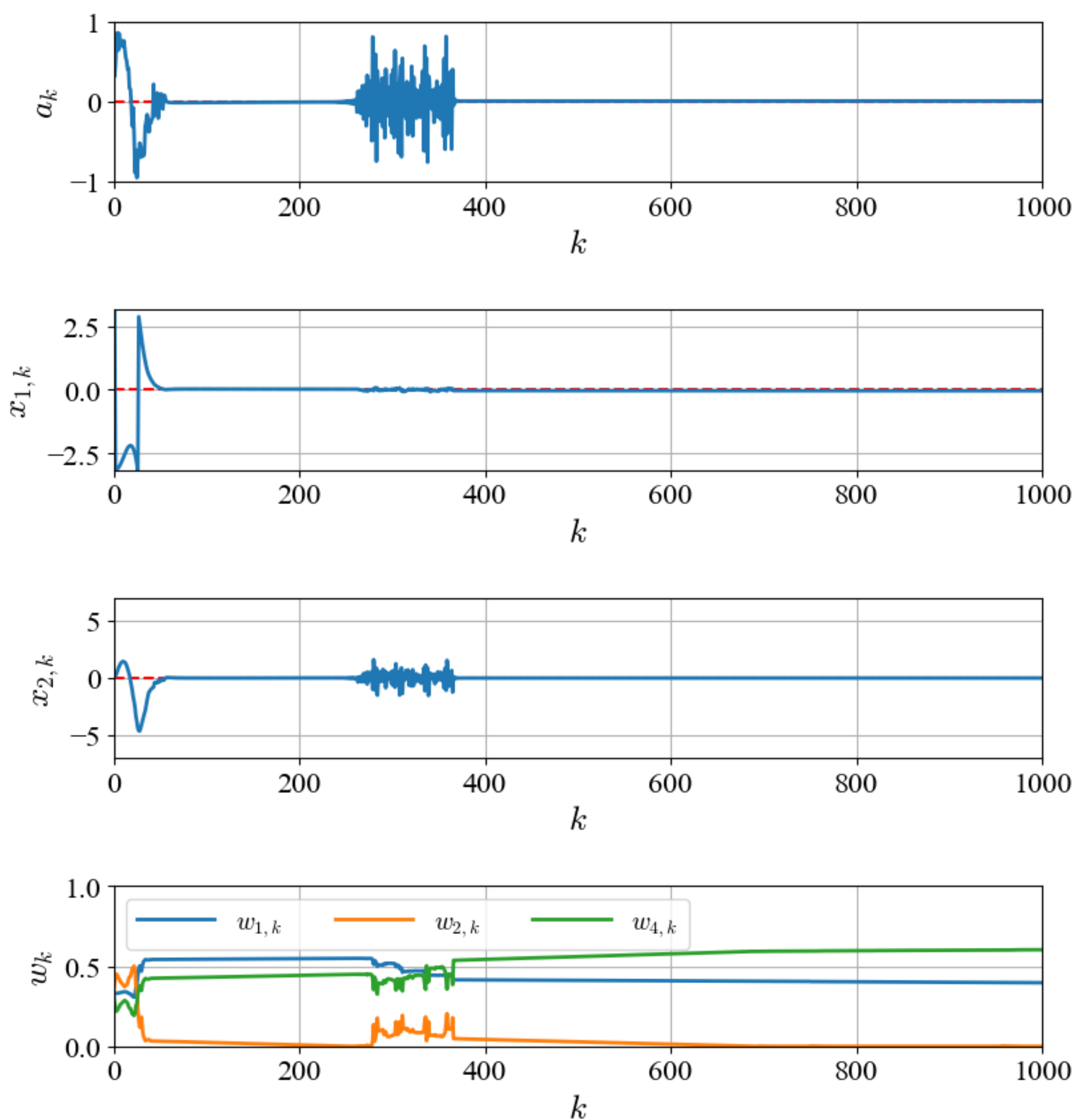


Fig. 3.10: The time response of the varying real system controlled by the agent that learns its policy using our proposed method. It is assumed that $\xi_1 = 1.0$ and ξ_2 varies from 5.0 to 50.0 slowly until $k = 200$.

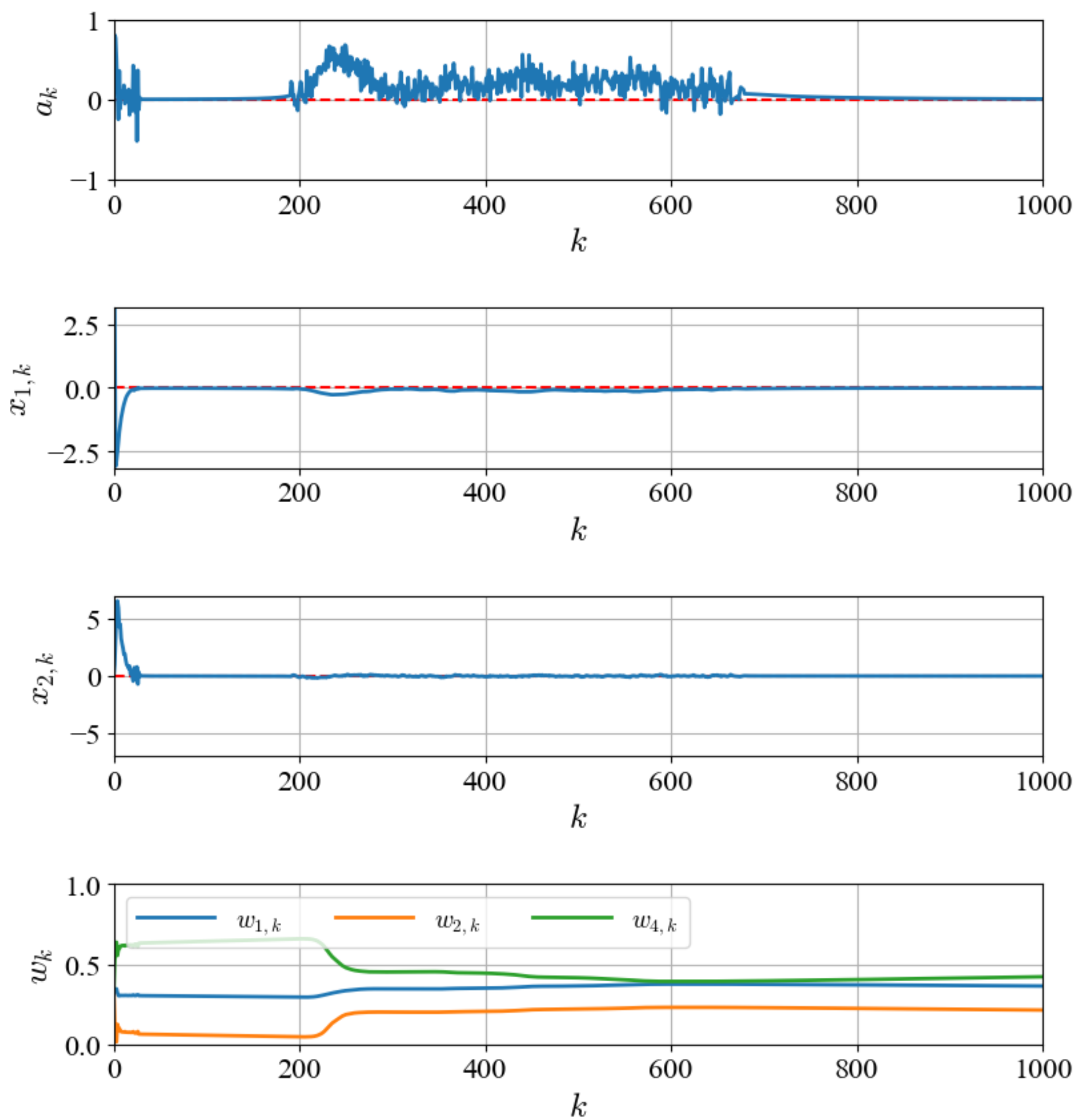


Fig. 3.11: The time response of the varying real system controlled by the agent that learns its policy using our proposed method. It is assumed that $\xi_1 = 1.0$ and ξ_2 varies from 50.0 to 5.0 slowly until $k = 200$.

3.3.3 Effects of Pre-Training

We compare our proposed method with a standard continuous deep Q-learning algorithm without pre-training. For the standard continuous deep Q-learning algorithm, we use the same DNN architecture for the virtual systems. The size of the mini-batch is $I = 128$, that is, the agent begins to learn its policy at $k = 128$. The parameter vector of the DNN is updated by Adam, where its learning rate is $\alpha = 5.0 \times 10^{-5}$. The maximum step is $K = 1000$. The initial state is $[\pi\ 0]^\top$.

The time response of the real system with $(\xi_1, \xi_2) = (0.95, 5.5)$ controlled by the agent that learns its policy by standard continuous deep Q-learning without pre-training is shown in **Fig. 3.12**. Exploration noise is generated by (3.17). The parameter vector of the deep Q-network θ^Q is initialized randomly. Although the agent can stabilize the system for 1000 steps using our proposed method, as shown in **Fig. 3.7**, the agent cannot stabilize it using the standard continuous deep Q-learning algorithm without pre-training. It is shown that pre-training with the simulator accelerates the policy learning.

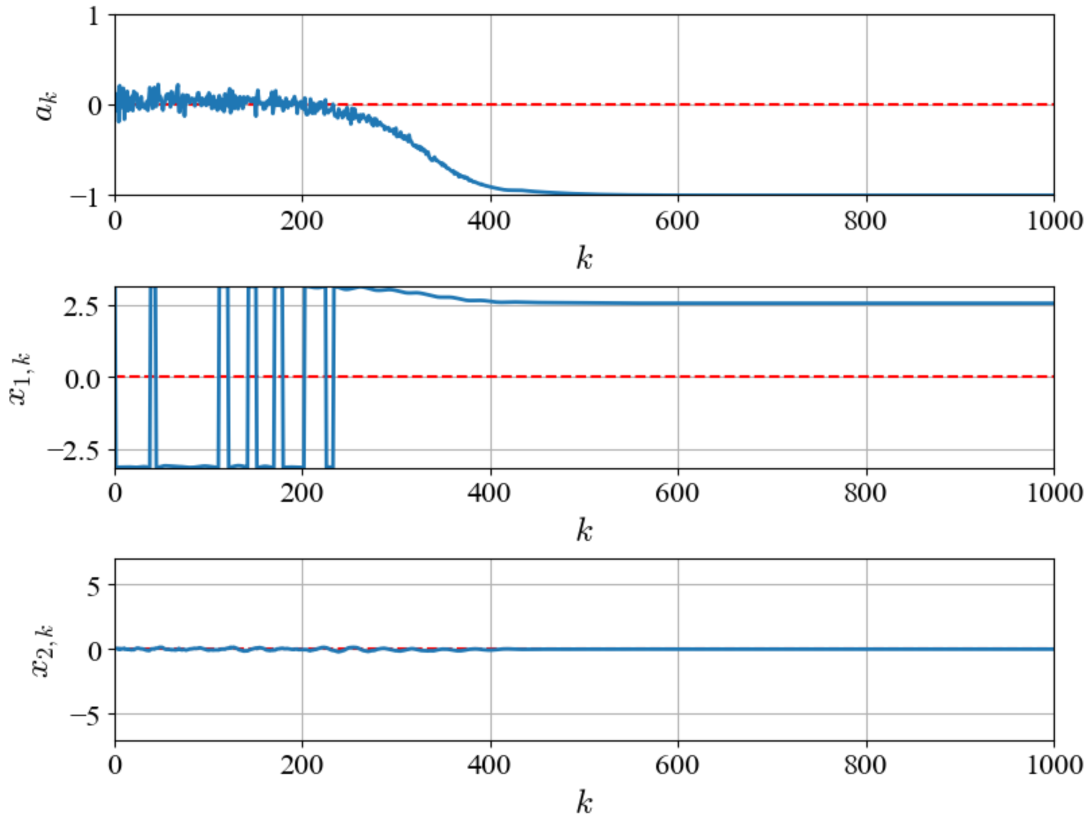


Fig. 3.12: The time response of the real system $(\xi_1, \xi_2) = (0.95, 5.5)$ controlled by the agent that learns its policy using the standard continuous deep Q-learning without pre-training. The agent cannot stabilize the system through 1000 interactions.

Moreover, we apply standard continuous deep Q-learning to a real system whose system parameter vector varies slowly, as in **Section 3.3.2**. We add exploration noise to actions if $||x||_2 \geq 0.05$. First, it is assumed that the system parameter ξ_2 increases gradually from 5.0 to 50.0 until $k = 200$, where $\xi_1 = 1.0$. The DNN parameter vector is initialized by θ^{Q_2}. The time response of the real system is shown in **Fig. 3.13**. Second, it is assumed that the system parameter ξ_2 decreases gradually from 50.0 to 5.0 until $k = 200$, where $\xi_1 = 1.0$. The DNN parameter vector is initialized by θ^{Q_4}. The time response of the real system is shown in **Fig. 3.14**. We see that, in both the cases, the agent cannot adapt to each varying system. In the standard continuous deep Q-learning algorithm, the agent adjusts many DNN's parameters based on experiences from the real system to learn its policy using a stochastic gradient descent method such as Adam. Thus, the agent cannot quickly adjust its DNN parameter vector and cannot adapt to variations in the system parameter vector. On the other hand, because it is relatively easy to adjust the parameters of the approximated linear Q-function, the agent can stabilize a real system whose system parameter vector varies.

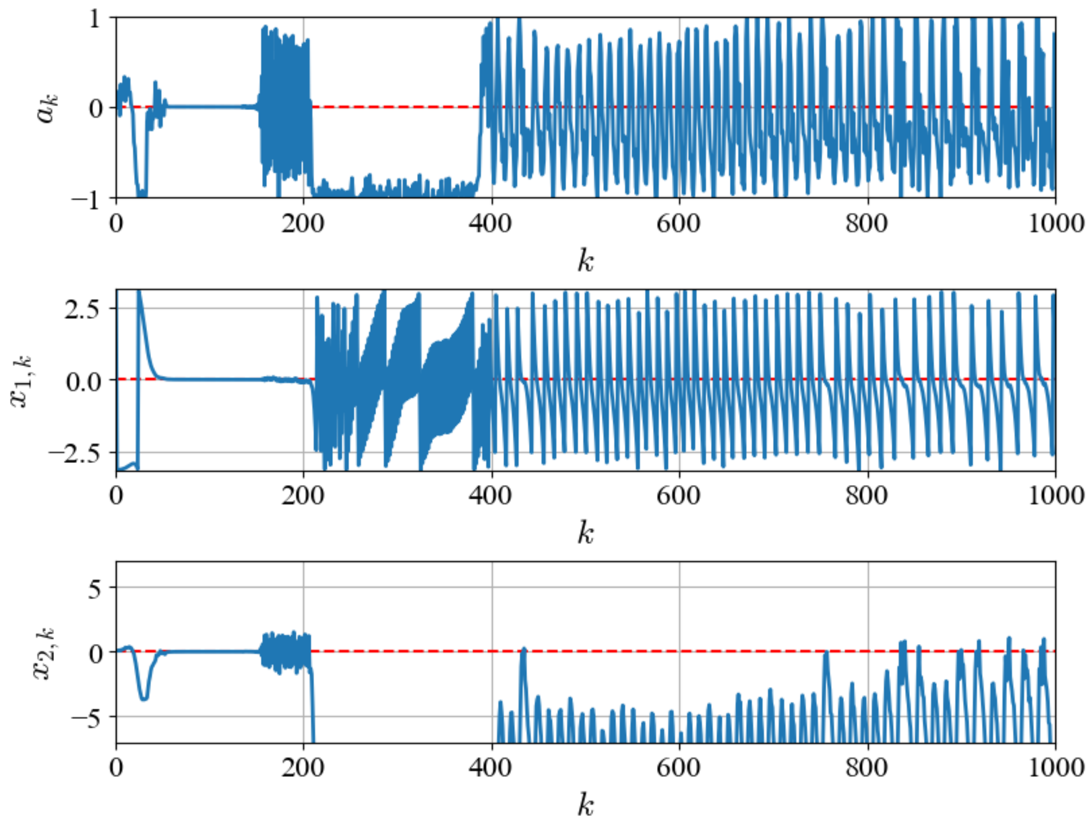


Fig. 3.13: The time response of the real system with the variation of the system parameter vector controlled by the agent that learns its policy using the standard continuous deep Q-learning algorithm. It is assumed that $\xi_1 = 1.0$ and ξ_2 varies from 5.0 to 50.0 slowly until $k = 200$.

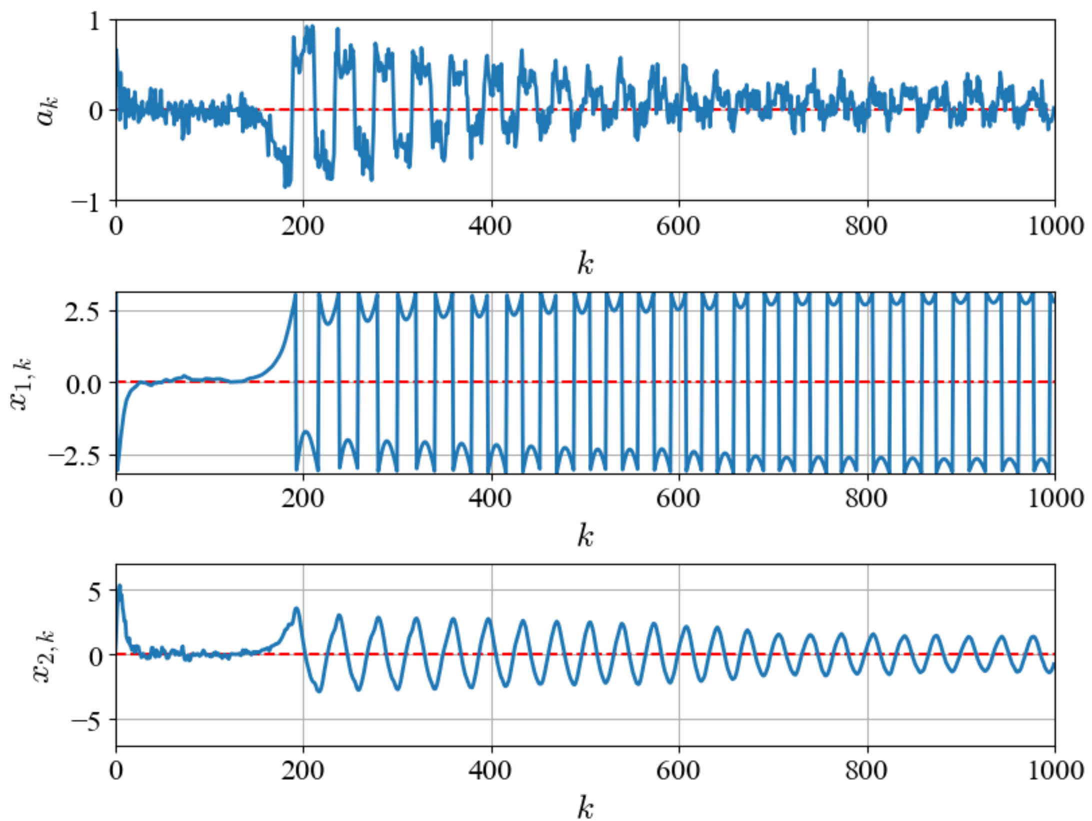


Fig. 3.14: The time response of the real system with the variation of the system parameter vector controlled by the agent that learns its policy using the standard continuous deep Q-learning algorithm. It is assumed that $\xi_1 = 1.0$ and ξ_2 varies from 5.0 to 50.0 slowly until $k = 200$.

Chapter 4

Application of DRL to NCSs with Uncertain Network Delays

NCSs have attracted much attention thanks to the development of network technology. On the other hand, there are network delays caused by data transmissions in NCSs. These network delays may degrade control performances. In general, the network delays may fluctuate randomly. Additionally, for nonlinear systems, it is difficult to identify the models precisely and to design controllers analytically. Thus, we propose a DRL-based networked controller design taking network delays into consideration.

This chapter is mainly based on “Application of deep reinforcement learning to networked control systems with uncertain network delays” [38] which appeared in *Nonlinear Theory and Its Applications*, © 2020 IEICE.

4.1 Problem Formulation

We consider control of the following continuous-time nonlinear system through a network as shown in **Fig. 4.1**.

$$\dot{x}(t) = f(x(t), u(t)), \tag{4.1}$$

$$y_k = h(x(k\Delta)), \quad k \in \{0, 1, 2, ...\}, \tag{4.2}$$

where

- $x(t) \in \mathcal{X} \subseteq \mathbb{R}^{n_x}$ is the system’s state at the continuous time $t \in [0, \infty)$,
- $u(t) \in \mathcal{U} \subseteq \mathbb{R}^{n_u}$ is the control input at the continuous time $t \in [0, \infty)$,
- $\Delta > 0$ is the sampling period,
- $y_k \in \mathcal{Y} \subseteq \mathbb{R}^{n_y}$ is the k-th output observed by the sensor,
- $f : \mathcal{X} \times \mathcal{U} \to \mathcal{X}$ describes the mathematical model of the system that is local Lipschitz with respect to x, and
- $h : \mathcal{X} \to \mathcal{Y}$ is the output function.

It is assumed that the mathematical models f and h are unknown.

In the NCS, the controller computes control inputs from data sent by the sensor and sends them to the actuator. Additionally, it is assumed that there exist two types of network delays due to data transmissions. One is caused by transmissions of observed outputs from the sensor to the controller. The other is caused by transmissions of determined control inputs from the controller to the actuator. The k-th network delays are denoted by $\tau_{\mathrm{sc},k}$ and $\tau_{\mathrm{ca},k}$, respectively. These network delays randomly fluctuate, where these delays are bounded by the maximum delays $\tau_{\mathrm{sc}}^{\max}(= n_{\mathrm{sc}}\Delta)$ and $\tau_{\mathrm{ca}}^{\max}(= n_{\mathrm{ca}}\Delta)$, respectively. It is assumed that $n_{\mathrm{sc}}, n_{\mathrm{ca}} \in \mathbb{N}$ are known. In this chapter, it is also assumed that the packet loss does not occur in the networks and all data are received in the same order as their sending order.

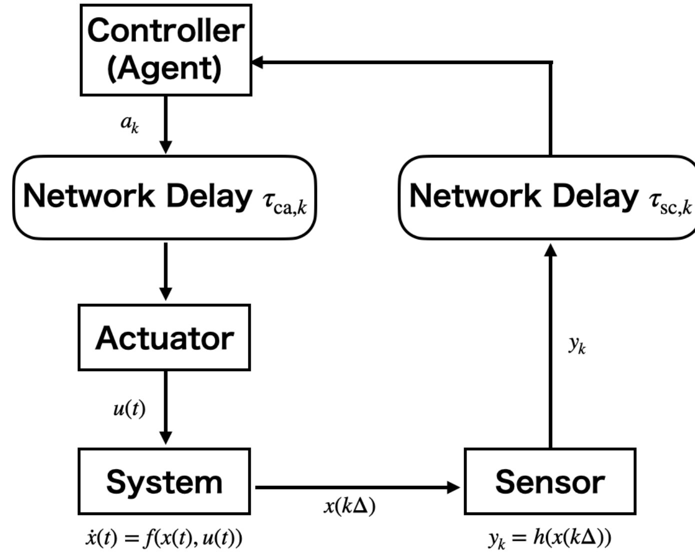


Fig. 4.1: Illustration of an NCS which consists of a system, a sensor, a controller, an actuator, and a network. In this chapter, we regard the controller as an agent and design its control policy using a DRL algorithm. At $t = k\Delta$ ($k = 0, 1, 2, \ldots$), the sensor observes the k-th system's output y_k and sends it to the controller (agent). At $t = k\Delta + \tau_{\mathrm{sc},k}$, the controller (agent) receives the k-th output y_k, computes the k-th control action a_k, and sends it to the actuator. At $t = k + \tau_{\mathrm{sc},k} + \tau_{\mathrm{ca},k}$, the actuator receives the k-th control action a_k and updates the control input to the system $u(t) = a_k$.

In this chapter, we apply DRL to design of a digital controller for stabilization of the nonlinear system without the system's model. Then, we regard the controller as an agent. The agent receives the k-th state x_k or the k-th output y_k and determines the k-th control action a_k. The k-th control action a_k is held until the actuator receives the

next control action a_{k+1}, that is,

$$u(t) = \begin{cases} a_k, & k\Delta + \tau_{\mathrm{sc},k} + \tau_{\mathrm{ca},k} \leq t < (k+1)\Delta + \tau_{\mathrm{sc},k+1} + \tau_{\mathrm{ca},k+1}, \\ 0_{n_u}, & 0 \leq t < \tau_{\mathrm{sc},0} + \tau_{\mathrm{ca},0}, \end{cases} \tag{4.3}$$

where 0_{n_u} is the zero vector in $\mathbb{R}^{n_u}$. Note that the agent cannot observe the true current system's state and the control action determined by the agent cannot be inputted to the system right away due to network delays.

4.2 Design of Networked Controller Using Deep Reinforcement Learning

We propose a DRL-based controller design for stabilization of a nonlinear system (4.1) at an equilibrium point. We consider the two types of learning methods: *state-based learning* and *output-based learning*.

4.2.1 State-Based Learning

We consider the case where the sensor can observe all state variables of the system, which is called full observation. For simplicity, it is assumed that $h(x) = x$. In an RL (or DRL)-based controller design, we often regard a system as an environment. However, in the NCS problem, there are uncertain network delays. If we ignore the effects of network delays and apply RL (or DRL) to design of a networked controller for stabilization, the learned policy may not stabilize the system. In RL, the state of the environment must include sufficient information to determine desired actions. We must appropriately specify the environment for the NCS problem with network delays.

We consider the worst case, where, for any $k \in \{0, 1, 2, \ldots\}$, $\tau_{\mathrm{sc},k} = n_{\mathrm{sc}}\Delta$ and $\tau_{\mathrm{ca},k} = n_{\mathrm{ca}}\Delta$. The network delays in data transmissions are shown in **Fig. 4.2**. The sensor observes the k-th observed state $x_k = x(k\Delta)$ at $t = k\Delta$. The k-th observed state x_k is sent to the agent through the network. The agent receives x_k and determines the k-th control action a_k at $t = (k + n_{\mathrm{sc}})\Delta$. The k-th control action a_k is sent to the actuator. The actuator receives the k-th control action a_k and updates the control input to the system $u(t) = a_k$ at $t = (k + \tau)\Delta$, where $\tau = n_{\mathrm{sc}} + n_{\mathrm{ca}}$. Then, the system's state is $x((k + \tau)\Delta)$ that is the future state at $t = k\Delta$. The agent must predict the future state $x((k + \tau)\Delta)$ and determine the k-th control action a_k based on available information. If we could identify a mathematical model of the system f, we would predict a future state of the system based on the model f as follows:

$$x((k+\tau)\Delta) = x(k\Delta) + \int_{k\Delta}^{(k+\tau)\Delta} f(x(t), u(t))\mathrm{d}t. \tag{4.4}$$

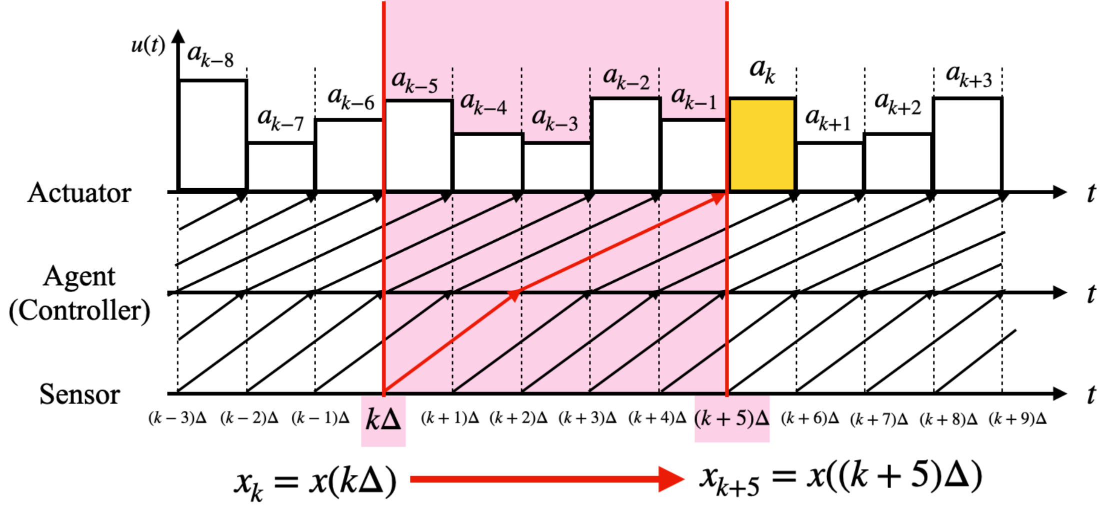


Fig. 4.2: Illustration of the network delays in the worst case, where $\tau_{\mathrm{sc}}^{\max} = 2\Delta$ and $\tau_{\mathrm{ca}}^{\max} = 3\Delta$ ($n_{\mathrm{sc}} = 2$ and $n_{\mathrm{ca}} = 3$). In the case, at $t = (k+2)\Delta$, the agent should predict $x((k+5)\Delta)$ and determine a_k using the k-th observed state x_k and the previously determined actions $a_{k-5}, \dots, a_{k-1}$.

Note that, for $k\Delta \leq t < (k + \tau)\Delta$, the previously determined control actions $a_{k-\tau}$, $a_{k-\tau+1}$, ..., a_{k-1} are inputted to the system as shown in **Fig. 4.2**. Fortunately, the agent can use the previously determined control actions $a_{k-\tau}$, $a_{k-\tau+1}$, ..., a_{k-1} at $t = (k + n_{\mathrm{sc}})\Delta$. Therefore, we use not only the k-th observed state x_k but also these previously determined control actions $a_{k-\tau}$, $a_{k-\tau+1}$, ..., a_{k-1} for the agent to learn its control policy. We define the following extended state $z_k \in \mathcal{X} \times \mathcal{U}^{\tau} \subseteq \mathbb{R}^{n_x+\tau n_u}$.

$$z_k = \begin{bmatrix} x_k \\ a_{k-1} \\ a_{k-2} \\ \vdots \\ a_{k-\tau} \end{bmatrix}. \tag{4.5}$$

In the state-based learning, we regard the extended state z_k as the state of the environment.

Network delays are not necessarily the maximum values $n_{\mathrm{sc}}\Delta$ and $n_{\mathrm{ca}}\Delta$. In the case, the agent also learns the control policy adaptively using the extended state z_k that has sufficient information for the worst case.

4.2.2 Output-Based Learning

We assume that the sensor cannot observe all state variables of the system, which is called partial observation. Then, the agent cannot directly use the system's state

$x(k\Delta)$ to determine an action. The agent must estimate the system's state based on available information. It is assumed that, if we could identify the models f and h, we would predict the full state x_k based on some previously determined control actions and some previously observed outputs. Particularly, for a discrete-time linear system which is controllable and observable, we can estimate the state x_k based on the past outputs $y_{k-1}, y_{k-2}, ..., y_{k-\varsigma}$ and the past control actions $a_{k-1}, a_{k-2}, ..., a_{k-\varsigma}$ as shown in **Appendix A**, where ς is larger than the observability index q of the linear system. Aangenent *et al.* proposed a data-based optimal control method [104], Lewis *et al.* proposed an RL-based method for a partial observable linear system [105], and Fujita and Ushio applied the Lewis's method to the design of an optimal networked controller considering network delays [37]. On the other hand, in the previous studies, we deal with linear systems. The estimation of the current state can be done by solving a linear equation consisting of the previous control actions and outputs whose number is larger than the observability index. However, for a nonlinear system, it is difficult to derive a nonlinear equation whose solution is the current system's state. Thus, we design the approximated optimal controller for a nonlinear system with partial observability using a DRL algorithm. Then, we select $\varsigma \in \mathbb{N}$ as a meta-parameter beforehand and give the agent previously determined control actions $a_{k-1}, a_{k-2}, ..., a_{k-\varsigma}$ and previously observed outputs $y_k, y_{k-1}, y_{k-2}, ..., y_{k-\varsigma}$ as the state of the environment. In general, if we set the meta-parameter ς to a small value, the performance of the learned control policy may be limited. On the other hand, if we set the meta-parameter ς to a large value, the agent may fail to learn its control policy due to the large dimensionality.

Remark: In this chapter, we aim to stabilize a nonlinear system at an equilibrium point. Although the system's dynamics is nonlinear, we can linearize the nonlinear model around the equilibrium point. Thus, we can select the meta-parameter ς with reference to the observability index as the condition around the equilibrium point. However, if the system's model is unknown, we cannot precisely obtain the observability index. On the other hand, the observability index q satisfies $q \leq n_x$. In the worst case, we need the previously observed outputs $y_{k-1}, ..., y_{k-n_x}$ in order to stabilize the system even around its equilibrium point. The dimension of the system's state n_x can be an index for selecting the meta-parameter ς.

Additionally, we consider the effect of network delays. For simplicity, we consider the worst case where all network delays are $n_{\mathrm{sc}}\Delta$ and $n_{\mathrm{ca}}\Delta$ as shown in **Fig. 4.3**. When we use previous outputs $y_{k-1}, y_{k-2}, ..., y_{k-\varsigma}$, we must also consider the control input $u(t)$ for $(k-\varsigma)\Delta \leq t < (k+\tau)\Delta$. Actually, the previous control actions $a_{k-1}, a_{k-2}, ..., a_{k-(\tau+\varsigma)}$ are inputted to the system as the control input $u(t)$, $(k-\varsigma)\Delta \leq t < (k+\tau)\Delta$. Thus, we define the following extended state

$z_k^{\varsigma} \in \mathcal{Y}^{\varsigma+1} \times \mathcal{U}^{\varsigma+\tau} \subseteq \mathbb{R}^{(\varsigma+1)n_y+(\varsigma+\tau)n_u}$.

$$z_k^{\varsigma} = \begin{bmatrix} y_k \\ y_{k-1} \\ y_{k-2} \\ \vdots \\ y_{k-\varsigma} \\ a_{k-1} \\ a_{k-2} \\ \vdots \\ a_{k-(\tau+\varsigma)} \end{bmatrix}. \tag{4.6}$$

In the output-based learning, we regard the extended state z_k^{ς} as the state of the environment.

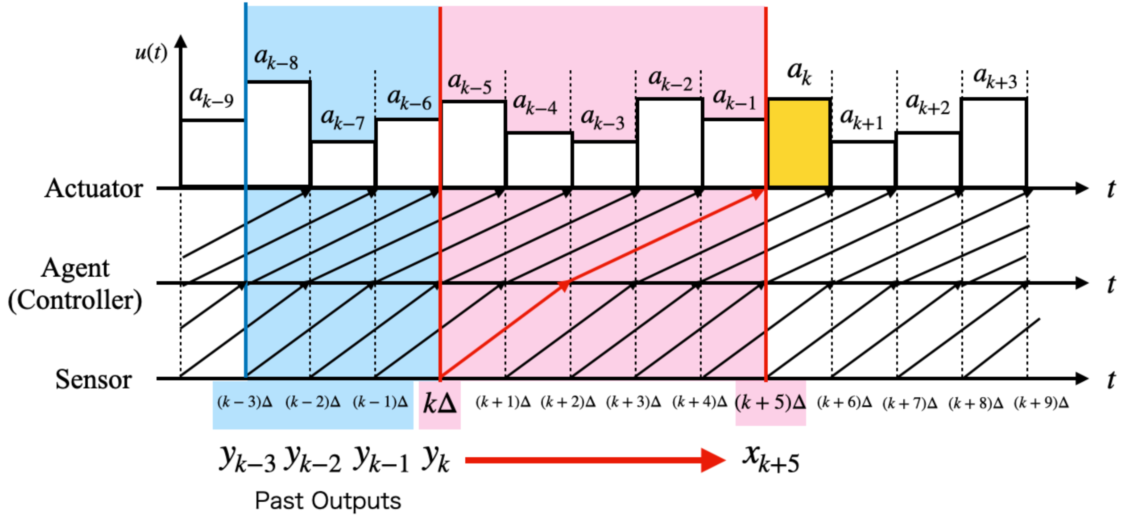


Fig. 4.3: Illustration of the network delays in the worst case, where $\tau_{\text{sc}}^{\max} = 2\Delta$ and $\tau_{\text{ca}}^{\max} = 3\Delta$ ($n_{\text{sc}} = 2$ and $n_{\text{ca}} = 3$). If we select the meta-parameter $\varsigma = 3$, then we must consider the control input $u(t)$ for $(k-3)\Delta \leq t < (k+5)\Delta$.

4.2.3 Learning Algorithm

We apply the DDPG algorithm [11] as a DRL algorithm. As shown in **Fig. 4.4**, we regard the past control action list and the past output list as a part of the environment to construct an extended state z^{ς}. The algorithm is shown in **Algorithm 2**. The outline is as follows: From line 1 to 4, we select a meta-parameter ς, initialize

parameter vectors of main DNNs and target DNNs, and initialize a replay buffer $\mathcal{D}$ for the experience replay. From line 5 to 27, the agent learns its policy through interactions with the environment that consists of the system, the past control action list, and the past output list. From line 6 to 9, at the start of an episode, the agent receives the initial output y_0, adds it to the past output list, constructs the initial extended state $z_0^\varsigma = [y_0 \ \cdots \ y_0 \ 0 \ \cdots \ 0]^\top$, and sets a random process p_ϵ for generating exploration noises. From line 10 to 26, the agent learns its policy and interacts with the environment to collect experiences. From line 11 to 16, the agent receives the reward r_{k-1} for the transition $(z_{k-1}^\varsigma, a_{k-1}, z_k^\varsigma)$ and stores the experience $(z_{k-1}^\varsigma, a_{k-1}, z_k^\varsigma, r_{k-1})$ to the replay buffer $\mathcal{D}$. In line 17, the agent determines the control action a_k based on the actor DNN $\mu(z_k^\varsigma|\theta_\mu)$, where the agent adds the exploration noise $\epsilon_k \sim p_\epsilon$ to a_k. In line 18, the agent adds the control action a_k to the past control action list and sends it to the actuator. From line 19 to 25, the agent updates parameter vector of DNNs using the DDPG algorithm stated in **Section 2.5**.

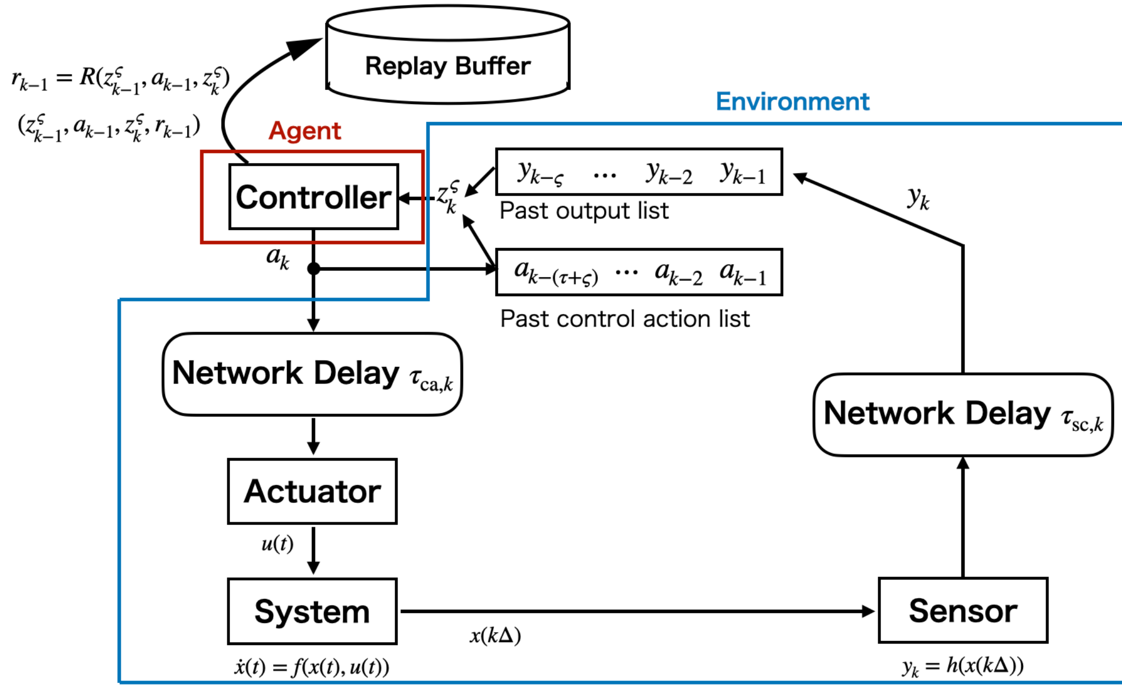


Fig. 4.4: Illustration of the agent and the environment. The agent interacts with the environment to collect experiences. The experiences are stored to the replay buffer. The agents updates parameter vectors of DNNs using experiences selected from the replay buffer randomly.

Remark: In our proposed method, we can use another DRL algorithm, where we should select an off-policy DRL algorithm with the experience replay such as continuous deep Q-learning [15], TD3 [12], and SAC [13, 14].

Algorithm 2 DRL algorithm for design of a networked controller under partial observation

1: Select the length of the past output list ς.
2: Randomly initialize the parameter vectors of an actor DNN θ_μ and a critic DNN θ_Q.
3: Initialize parameter vector of target networks $\theta_\mu^- \leftarrow \theta_\mu$ and $\theta_Q^- \leftarrow \theta_Q$.
4: Initialize the replay buffer $\mathcal{D}$.
5: **for** episode = $1, 2, ...$ **do**
6: Receive the initial observed output y_0.
7: Add the observed output y_0 to the past output list.
8: Generate the initial extended state z_0^ς, where $u_i = 0$ $(i < 0)$, $y_i = y_0$ $(i < 0)$.
9: Initialize a random process p_ϵ for exploration.
10: **for** $k = 0, 1, ..., K$ **do**
11: **if** $k > 0$ **then**
12: Receive the k-th output y_k.
13: Add the observed output y_k to the past output list.
14: Generate the extended state z_k^ς with past control actions and past outputs, where $a_i = 0$ $(i < 0)$, $y_i = y_0$ $(i < 0)$.
15: Receive the reward r_{k-1} for the tuple $(z_{k-1}^\varsigma, a_{k-1}, z_k^\varsigma)$ and store $(z_{k-1}^\varsigma, a_{k-1}, z_k^\varsigma, r_{k-1})$ to the replay buffer $\mathcal{D}$.
16: **end if**
17: Determine the control action a_k based on the actor DNN and add the exploration noise $\epsilon_k \sim p_\epsilon$ to the action $(a_k \leftarrow a_k + \epsilon_k)$.
18: Add the control action a_k to the past control action list and send it to the actuator.
19: **for** iteration = $1, 2, ..., I$ **do**
20: Select N experiences $(z^{\varsigma,(n)}, a^{(n)}, z'^{\varsigma,(n)}, r^{(n)})$, $n = 1, 2, ..., N$ randomly.
21: Set $t^{(n)} = r^{(n)} + \gamma Q_{\theta_Q^-}(z'^{\varsigma,(n)}, \mu_{\theta_\mu^-}(z'^{\varsigma,(n)}))$.
22: Update θ_Q by decreasing the loss (2.23).
23: Update θ_μ by decreasing the loss (2.24).
24: Update target DNNs by soft update (2.18).
25: **end for**
26: **end for**
27: **end for**

4.3 Example

We consider the following examples.

1. **Pendulum**

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} = \begin{bmatrix} x_2(t) \\ g \sin x_1(t) - \kappa x_2(t) + u(t) \end{bmatrix}, \tag{4.7}$$

where $g = 9.81$ and $\kappa = 0.05$. The system's state is $x(t) = [x_1(t)\ x_2(t)]^\top \in [-\pi, \pi] \times \mathbb{R}$ and the control input is $u(t) \in [-2, 2]$. The goal is the stabilization of the pendulum at the unstable equilibrium point $[0\ 0]^\top$.

2. **Lorenz dynamics**

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{bmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \end{bmatrix} = \begin{bmatrix} p_1(x_2(t) - x_1(t)) \\ -x_1(t)x_3(t) + p_2 x_1(t) - x_2(t) \\ x_1(t)x_2(t) - p_3 x_3(t) \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 2 & 0 \\ 0 & 3 \end{bmatrix}\begin{bmatrix} u_1(t) \\ u_2(t) \end{bmatrix}, \tag{4.8}$$

where $p_1 = 10$, $p_2 = 28$, and $p_3 = 8/3$. The uncontrolled behavior is a chaotic attractor as shown in **Fig. 4.5**. The system's state is $x(t) = [x_1(t)\ x_2(t)\ x_3(t)]^\top \in \mathbb{R}^3$ and the control input is $u(t) = [u_1(t)\ u_2(t)]^\top \in [-7, 7]^2$. The goal is the stabilization of the Lorenz dynamics at one of the unknown equilibrium points.

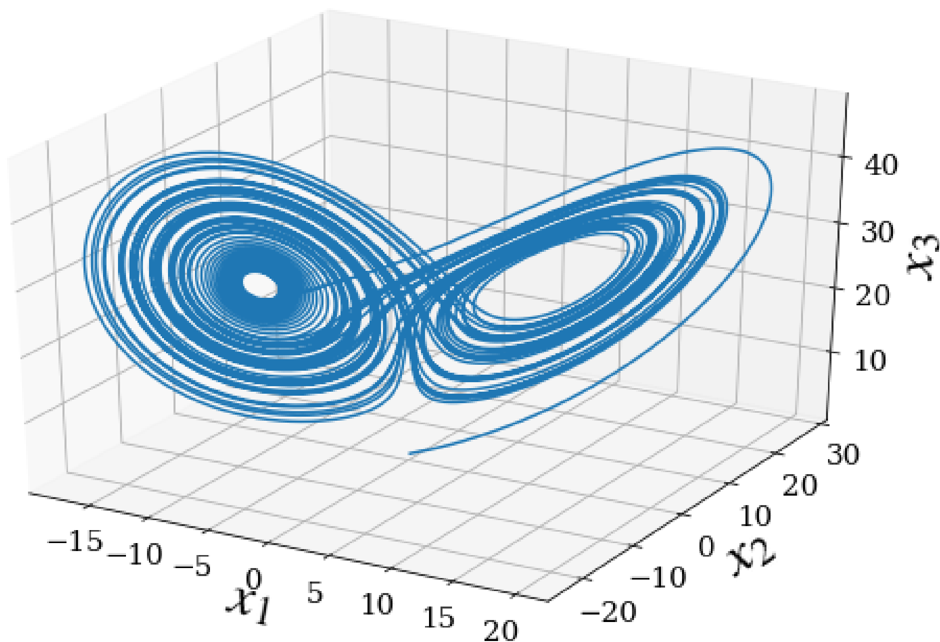


Fig. 4.5: Illustration of the Lorenz attractor.

We assume that both $\tau_{\mathrm{sc},k}$ and $\tau_{\mathrm{ca},k}$ are generated by the uniform distribution $U(3\Delta, 5\Delta)$, where we cannot identify the distribution beforehand while these maximum values $\tau_{\mathrm{sc}}^{\max} = 5\Delta$ and $\tau_{\mathrm{ca}}^{\max} = 5\Delta$ are known. Thus, in all simulations, $\tau = 10$ unless otherwise noted. It is assumed that the sampling period is $\Delta = 2^{-4}$ and the terminal of a learning episode is at time $t = 25.0$, that is, K in **Algorithm 2** is the number of interactions with the system during $0 \leq t \leq 25.0$. We use the *Runge-Kutta method* [106] to solve the differential equation for all simulations stated in **Appendix B**. We use exploration noises generated by the discrete-time Ornstein Uhlenbeck process [99]

$$\epsilon_k = \epsilon_{k-1} - 0.15\epsilon_{k-1} + p_{\mathcal{N}}\epsilon_{\mathcal{N}},\ \epsilon_0 = 0,$$

where $\epsilon_{\mathcal{N}}$ is a noise generated by the standard normal distribution $\mathcal{N}(0, 1)$. For the pendulum, we set $p_{\mathcal{N}} = 0.3$. After 400 episodes, we weight the generated noise as follows:

$$\epsilon_k' \leftarrow \frac{400}{ep}\epsilon_k, \tag{4.9}$$

where ep (> 400) denotes the current episode number. For the Lorenz dynamics, we set $p_{\mathcal{N}} = 0.5$ before the 50th episode. After that, we set $p_{\mathcal{N}} = 0.3$. We double the size of generated noise ϵ_k before the 400th episode. After that, we weight the generated noise as follows:

$$\epsilon_k' \leftarrow \frac{800}{ep}\epsilon_k. \tag{4.10}$$

All experiments run on a computer with an Intel(R) Core(TM) i7-10700 @ 2.9GHz processor and 32GB of memory and were conducted using Python software. The implementation for the experiments is shown in **Appendix C** in detail.

In order to evaluate the learned policy, we utilize the learning curve that shows the average reward $\frac{1}{K}\sum_{k=0}^{K} r_k$ obtained by the learned policy for an episode, where for the pendulum, let the initial state be $[\pi\ 0]^\top$ and, for the Lorenz dynamics, let the initial state be $[9\ -9\ 18]^\top$.

4.3.1 State-Based Learning

We assumed that the sensor can observe all state variables of the system, where the k-th observed state is $x_k = x(k\Delta)$.

Pendulum

In the experiment, we use a critic DNN and an actor DNN with two hidden layers, where all hidden layers have 128 units and all layers are fully connected. The activation functions are ReLU functions except for the output layers. In regards to the activation functions of the output layers, we use a hyperbolic tangent function, where

the output is doubled, for the actor DNN and a linear function for the critic DNN. The size of the replay buffer $\mathcal{D}$ is 1.0×10^6 and the mini-batch size is $N = 128$. The parameter vector of the actor DNN and the critic DNN are updated by Adam [103], where learning step sizes are set to 1.0×10^{-4} for the actor DNN and 1.0×10^{-3} for the critic DNN, respectively. The soft update rate of target networks is $\xi = 0.001$. The discount factor γ is 0.99. The initial state is randomly selected for each episode, where $-\pi \leq x_1(0) \leq \pi$ and $-3 \leq x_2(0) \leq 3$. We input $x_k = [-\sin x_1(k\Delta)\ \cos x_1(k\Delta), x_2(k\Delta)]^\top$ to DNNs instead of $[x_1(k\Delta),\ x_2(k\Delta)]^\top$ such as *pendulum-v0* in *Open AI gym* [107].

At first, we design a policy using the latest observed system's state only. The reward function $R_{\mathrm{pendulum},0} : \mathcal{X} \times \mathcal{U} \to \mathbb{R}$ is given by

$$R_{\mathrm{pendulum},0}(x, a) = -R_{\mathrm{pendulum},x}(x) - R_{\mathrm{pendulum},a}(a), \tag{4.11}$$

where

$$\begin{aligned} R_{\mathrm{pendulum},x}(x) &= \max\left\{|x_1|, x_1^2\right\} + 0.1 \max\left\{|x_2|, x_2^2\right\}, \\ R_{\mathrm{pendulum},a}(a) &= \max\left\{|a_1|, a_1^2\right\}, \end{aligned}$$

$x = [x_1\ x_2]^\top$, and $a = [a_1]^\top$. As shown by the blue curve in **Fig. 4.6**, in the case where for all k, $\tau_{\mathrm{sc},k},\ \tau_{\mathrm{ca},k} = 0$, the agent can learn its policy. However, as shown by the red curve in **Fig. 4.6**, in the case where, for all k, $\tau_{\mathrm{sc},k},\ \tau_{\mathrm{ca},k} \sim U(3\Delta, 5\Delta)$, the agent cannot proceed with its policy learning because it cannot deal with network delays using a latest observed state only.

To solve the problem, we utilize the extended state z as the state of the environment. The reward function $R_{\mathrm{pendulum}} : (\mathcal{X} \times \mathcal{U}^{10}) \times \mathcal{U} \to \mathbb{R}$ is given by

$$R_{\mathrm{pendulum}}(z, a) = -R_{\mathrm{pendulum},z}(z) - R_{\mathrm{pendulum},a}(a), \tag{4.12}$$

where

$$\begin{aligned} R_{\mathrm{pendulum},z}(z) &= \max\left\{|z_1|, z_1^2\right\} + 0.1 \max\left\{|z_2|, z_2^2\right\} + \sum_{l=3}^{12} 0.05 \max\left\{|z_l|, z_l^2\right\}, \\ R_{\mathrm{pendulum},a}(a) &= \max\left\{|a_1|, a_1^2\right\}, \end{aligned}$$

$z = [z_1\ z_2\ \ldots\ z_{12}]^\top$, and $a = [a_1]^\top$. Note that $[z_1\ z_2]^\top$ is the latest observed state and $z_3, \ldots, z_{12}$ are previously determined control actions.

The learning curve is shown in **Fig. 4.7**. It is shown that the agent can learn a control policy that obtains a high average of rewards using the extended state z. Moreover, the time response of the pendulum controlled by the learned policy is shown in **Fig. 4.8**. It is shown that the agent that sufficiently learned its policy by our proposed algorithm can stabilize the pendulum at the equilibrium point $[0\ 0]^\top$.

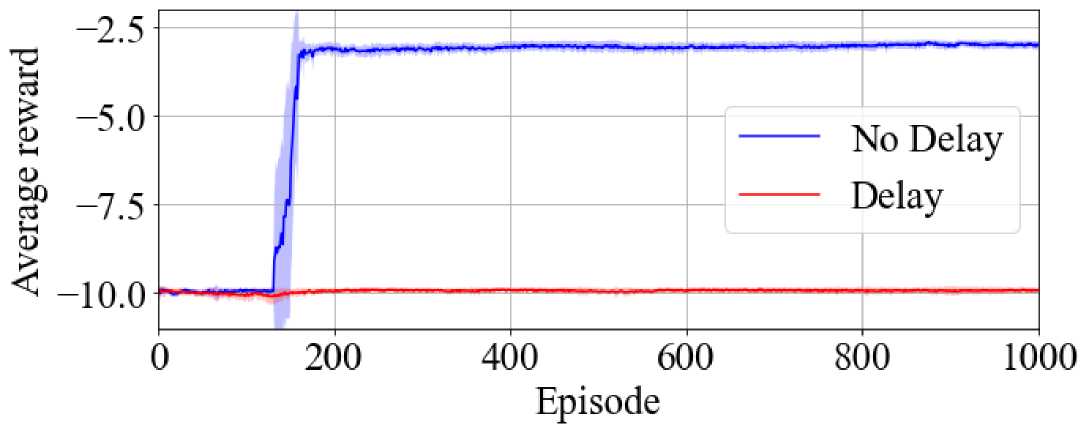


Fig. 4.6: Learning curves for a pendulum using the latest observed system's state only as the state of the environment. The blue curve shows the result in the case where there are no delays. The red curve shows the result in the case where there are random delays. The solid curves and the shades represent the average results and the standard deviations over 10 trials with different random seeds, respectively.

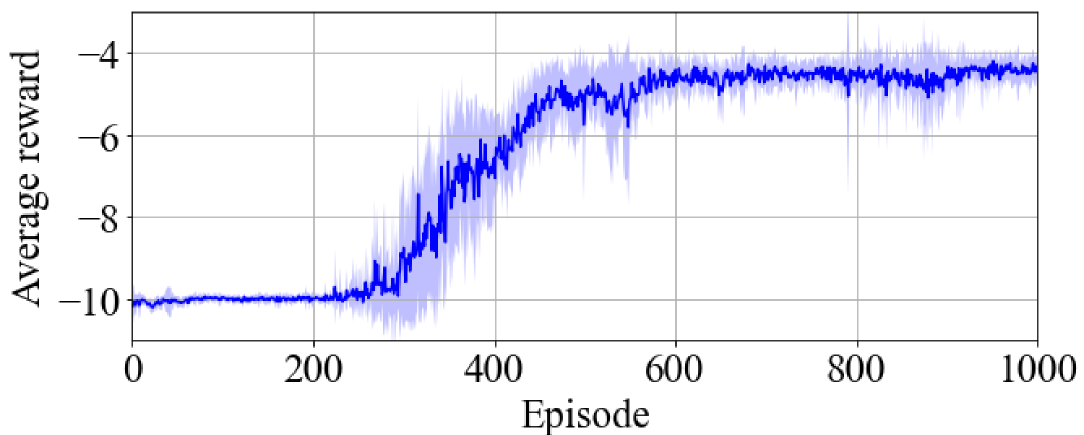


Fig. 4.7: Learning curve for a pendulum using the extended state z as the state of the environment. The solid curve and the shade represent the average results and the standard deviations over10 trials with different random seeds, respectively.

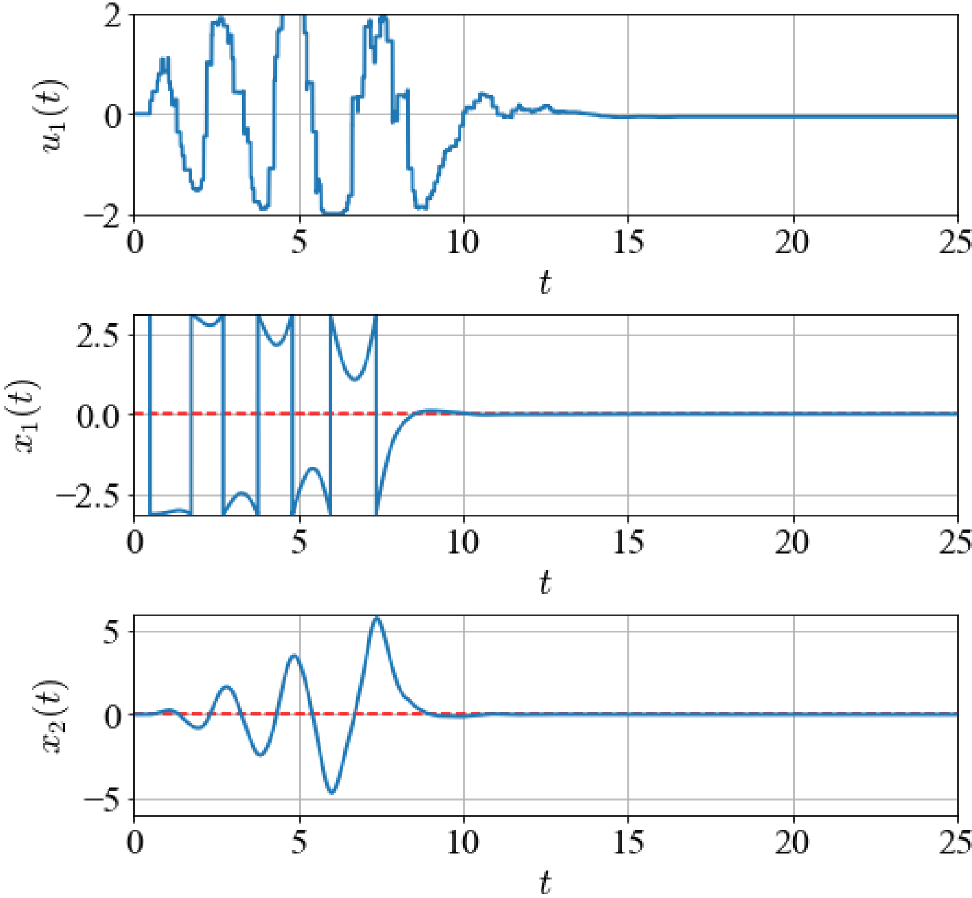


Fig. 4.8: Time response of the pendulum controlled by the learned policy.

Lorenz

In the experiment, we use a critic DNN and an actor DNN with two hidden layers, where all hidden layers have 128 units and all layers are fully connected. The activation functions are ReLU functions except for the output layers. In regards to the activation functions of the output layers, we use a hyperbolic tangent function, where the output is multiplied by 7, for the actor DNN and a linear function for the critic DNN. The size of the replay buffer $\mathcal{D}$ is 1.0×10^6 and the mini-batch size is $N = 128$. The parameter vector of the actor DNN and the critic DNN are updated by Adam, where learning step sizes are set to 1.0×10^{-5} for the actor DNN and 1.0×10^{-4} for the critic DNN, respectively. The soft update rate of target DNNs is $\xi = 0.001$. The discount factor γ is 0.99. The initial state is randomly selected for each episode, where $-15 \leq x_1(0) \leq 15$, $-15 \leq x_2(0) \leq 15$, and $10 \leq x_3(0) \leq 50$. We input $x_k = [x_1(k\Delta)\ x_2(k\Delta)\ 0.1x_3(k\Delta)]^\top$ to DNNs instead of $[x_1(k\Delta)\ x_2(k\Delta)\ x_3(k\Delta)]^\top$ because the size of x_3 tends to be larger than x_1 and x_2.

At first, we design a networked controller using the latest observed system's state only. The reward function $R_{\text{Lorenz},0} : \mathcal{X} \times \mathcal{U} \times \mathcal{X} \to \mathbb{R}$ is given by

$$R_{\text{Lorenz},0}(x, a, x') = -R_{\text{Lorenz},x}(x, x') - R_{\text{Lorenz},a}(a), \tag{4.13}$$

where

$$\begin{aligned} R_{\text{Lorenz},x}(x, x') &= \max\left\{|x_1' - x_1|, (x_1' - x_1)^2\right\} + \max\left\{|x_2' - x_2|, (x_2' - x_2)^2\right\} \\ &\quad +8.0 \max\left\{|x_3' - x_3|, (x_3' - x_3)^2\right\}, \\ R_{\text{Lorenz},a}(a) &= 1.5 \max\left\{|a_1|, a_1^2\right\} + 2.5 \max\left\{|a_2|, a_2^2\right\}, \end{aligned}$$

$x = [x_1\ x_2\ x_3]^\top$, $a = [a_1\ a_2]^\top$, and $x' = [x_1'\ x_2'\ x_3']^\top$. As shown by the blue curve in **Fig. 4.9**, in the case where for all k, $\tau_{\text{sc},k}$, $\tau_{\text{ca},k} = 0$, the agent can learn its policy. As shown by the red curve in **Fig. 4.9**, in the case where, for all k, $\tau_{\text{sc},k}$, $\tau_{\text{ca},k} \sim U(3\Delta, 5\Delta)$, the agent cannot proceed with its policy learning.

We use the extended state z as the state of the environment. It is assumed that the reward function $R_{\text{Lorenz}} : (\mathcal{X} \times \mathcal{U}^{10}) \times \mathcal{U} \times (\mathcal{X} \times \mathcal{U}^{10}) \to \mathbb{R}$ is given by

$$R_{\text{Lorenz}}(z, a, z') = -R_{\text{Lorenz},z}(z, z') - R_{\text{Lorenz},a}(a), \tag{4.14}$$

where

$$\begin{aligned} R_{\text{Lorenz},z}(z, z') &= \max\left\{|z_1' - z_1|, (z_1' - z_1)^2\right\} + \max\left\{|z_2' - z_2|, (z_2' - z_2)^2\right\} \\ &\quad +8.0 \max\left\{|z_3' - z_3|, (z_3' - z_3)^2\right\} + \sum_{l=4}^{23} 0.15 \max\left\{|z_l|, z_l^2\right\}, \\ R_{\text{Lorenz},a}(a) &= 1.5 \max\left\{|a_1|, a_1^2\right\} + 2.5 \max\left\{|a_2|, a_2^2\right\}, \end{aligned}$$

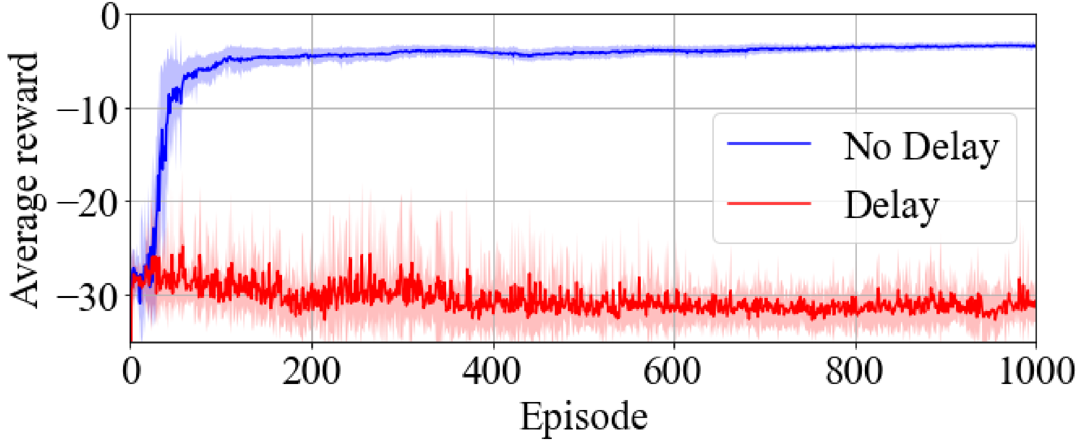


Fig. 4.9: Learning curves for the Lorenz dynamics using the latest observed system's state as the state of the environment. The blue curve shows the result in the case where there are no delays. The red curve shows the result in the case where there are random delays. The solid curves and the shades represent the average results and the standard deviations over 10 trials with different random seeds, respectively.

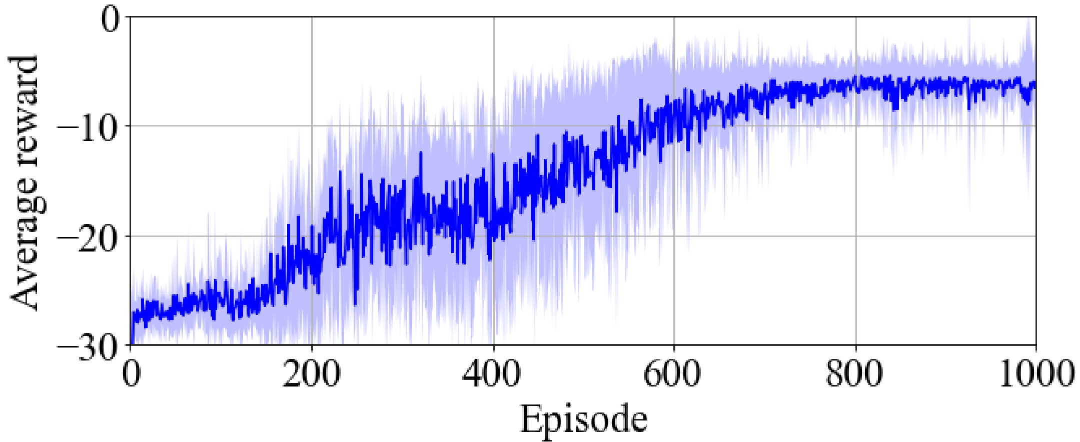


Fig. 4.10: Learning curve for the Lorenz dynamics using the extended state z as the state of the environment. The solid curve and the shade represent the average results and the standard deviations over 10 trials with different random seeds, respectively.

$z = [z_1\ z_2\ ...\ z_{23}]^\top$, $a = [a_1\ a_2]^\top$, and $z' = [z'_1\ z'_2\ ...\ z'_{23}]^\top$. Note that $[z_1\ z_2\ z_3]^\top$ is the latest observed state and $[z_4\ z_5]^\top$, ..., $[z_{22}\ z_{23}]^\top$ are previously determined control actions.

The learning curve is shown in **Fig. 4.10**. It is shown that the agent can learn a control policy that obtains a high average of rewards using the extended state z. Moreover, the time response of the Lorenz dynamics controlled by the learned policy is shown in **Fig. 4.11**. It is shown that the agent that sufficiently learned its policy by our proposed algorithm can stabilize the Lorenz dynamics at the equilibrium point.

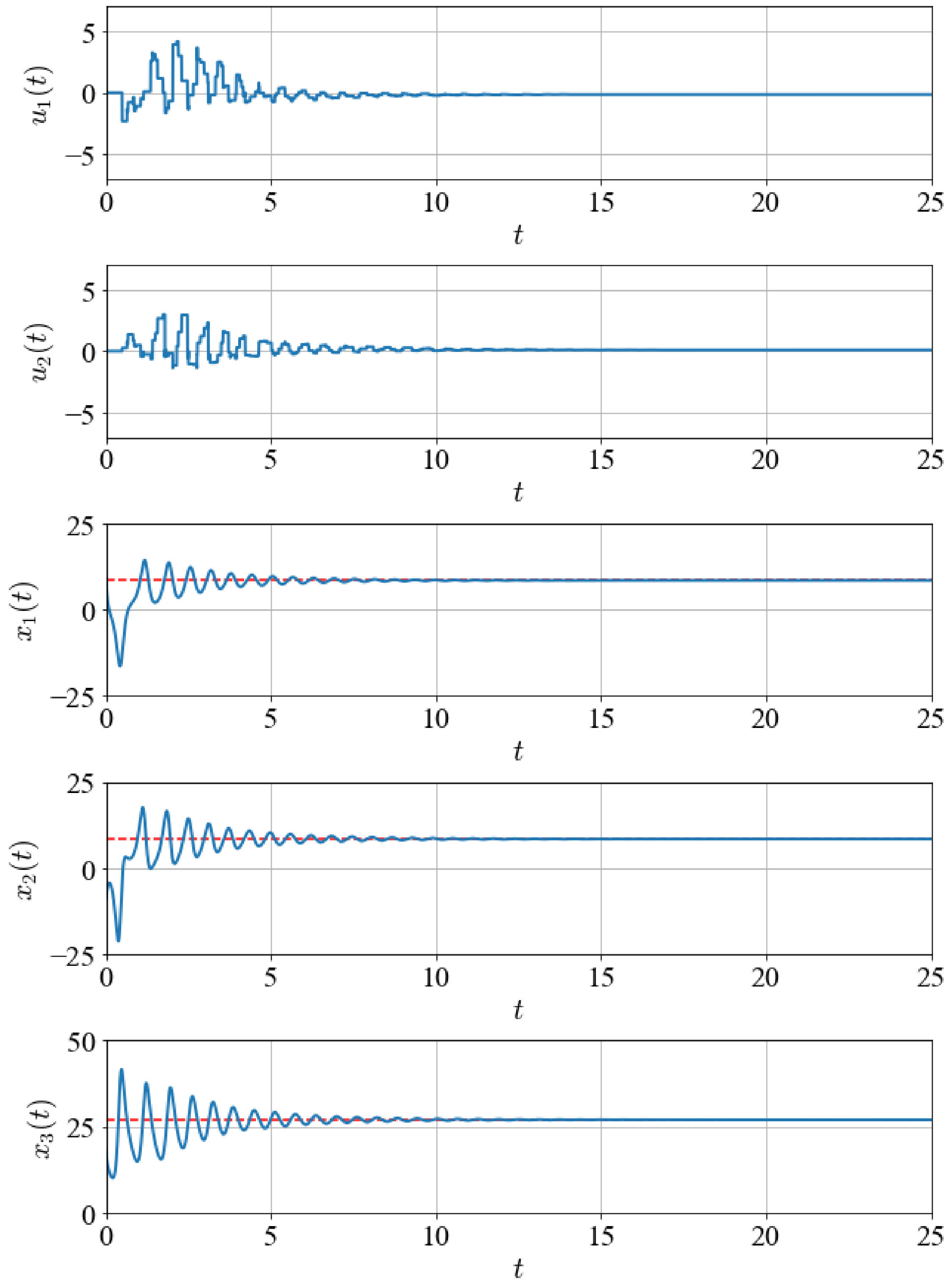


Fig. 4.11: Time response of the Lorenz dynamics controlled by the learned policy.

4.3.2 Effect of τ

We discuss the effect of τ. In this section, we show simulations using the pendulum (4.7), where $\tau_{\mathrm{sc},k}$ and $\tau_{\mathrm{ca},k}$ are generated by the uniform distribution $U(3\Delta, 5\Delta)$. The experimental setting other than the number of τ is same as **Section 4.3.1**. The reward function $R_{\mathrm{pendulum}} : (\mathcal{X} \times \mathcal{U}^{\tau}) \times \mathcal{U} \to \mathbb{R}$ is given by

$$R_{\mathrm{pendulum}}(z, a) = -R_{\mathrm{pendulum},z}(z) - R_{\mathrm{pendulum},a}(a), \tag{4.15}$$

where

$$\begin{aligned} R_{\mathrm{pendulum},z}(z) &= \max\left\{|z_1|, z_1^2\right\} + 0.1 \max\left\{|z_2|, z_2^2\right\} + \sum_{l=3}^{\tau+2} 0.05 \max\left\{|z_l|, z_l^2\right\}, \\ R_{\mathrm{pendulum},a}(a) &= \max\left\{|a_1|, a_1^2\right\}, \end{aligned}$$

$z = [z_1 \ z_2 \ \ldots \ z_{\tau+2}]^{\top}$, and $a = [a_1]^{\top}$. Note that $[z_1 \ z_2]^{\top}$ is the latest observed state and $z_3, \ \ldots, \ z_{\tau+2}$ are previous control actions. The results of $\tau = 4, 6, 8, 16$ are shown in **Fig. 4.12**. If the number of previous control actions is not sufficient to learn a policy considering network delays such as $\tau = 4, 6$, the agent cannot proceed with its policy learning as shown in **Fig. 4.12(a)** and **(b)**. On the other hand, although the number of previous determined control actions is less than 10 in the case with $\tau = 8$, the agent can learn its policy as well as the case $\tau = 10$ as shown in **Fig. 4.12(c)** since the mean of $\tau_{\mathrm{sc},k} + \tau_{\mathrm{ca},k}$ is 8Δ. Moreover, even if we set a large number for the worst case scenario such as $\tau = 16$, the agent can learn its policy as shown in **Fig. 4.12(d)**. From these results, if we know the worst case delay roughly, we can design a policy using a DRL algorithm with our proposed extended state.

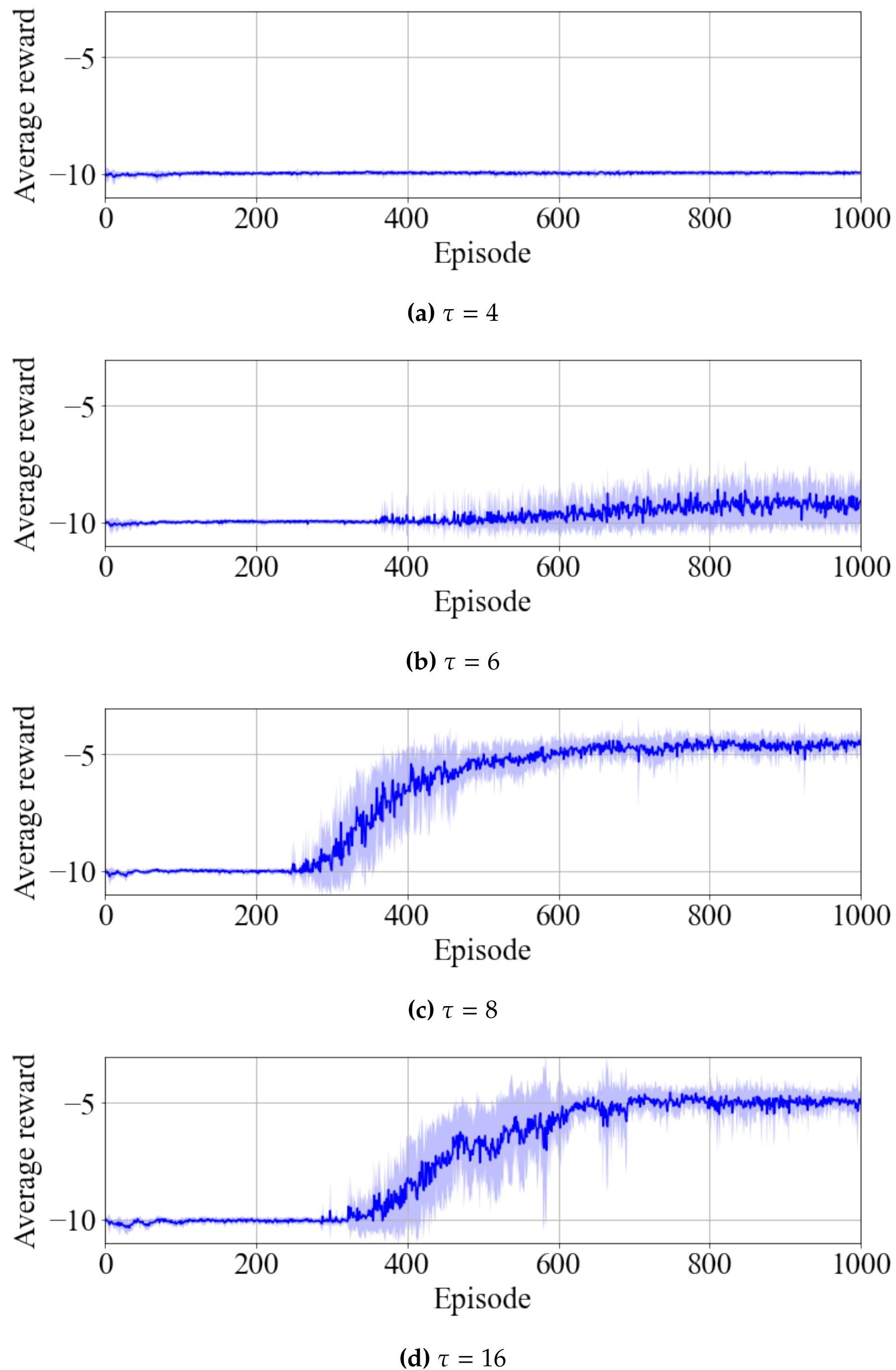


Fig. 4.12: Learning curves for the pendulum in the cases $\tau = 4, 6, 8, 16$, where $\tau_{\mathrm{sc},k}$ and $\tau_{\mathrm{ca},k}$ are generated by the uniform distribution $U(3\Delta, 5\Delta)$. The solid curves and the shades represent the average results and the standard deviations over 10 trials with different random seeds, respectively.

4.3.3 Output-Based Learning

Pendulum
The output is given by

$$y_k = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x_1(k\Delta) \\ x_2(k\Delta) \end{bmatrix}. \tag{4.16}$$

In the experiment, we use a critic DNN and an actor DNN with two hidden layers, where all hidden layers have 256 units and all layers are fully connected. The activation functions are ReLU functions except for the output layers. In regards to the activation functions of the output layers, we use a hyperbolic tangent function, where the output is doubled, for the actor DNN and a linear function for the critic DNN. The size of the replay buffer $\mathcal{D}$ is 1.0×10^6 and the mini-batch size is $N = 128$. The parameter vector of the actor DNN and the critic DNN are updated by Adam, where learning step sizes are set to 1.0×10^{-4} for the actor DNN and 1.0×10^{-3} for the critic DNN. The update rate of target networks is $\xi = 0.001$. The discount factor γ is 0.99. The initial state is randomly selected for each episode, where $-\pi \le x_1(0) \le \pi$ and $-3 \le x_2(0) \le 3$.

We use the extended state z^{ς} as the state of the environment, where we consider the cases $\varsigma = 0, 2$. The reward function $R^{\varsigma}_{\text{pendulum}} : (\mathcal{Y}^{(\varsigma+1)} \times \mathcal{U}^{(\varsigma+10)}) \times \mathcal{U} \to \mathbb{R}$ is given by

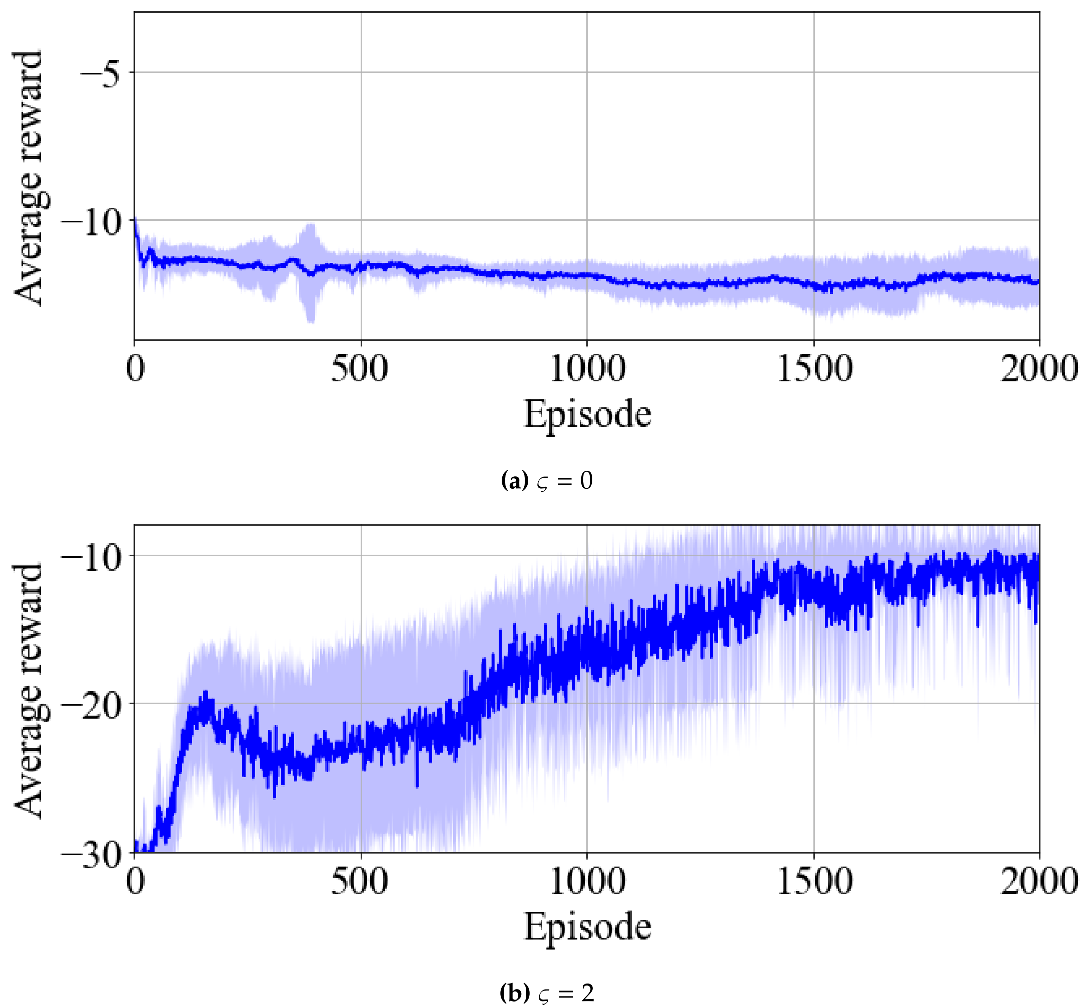
$$R^{\varsigma}_{\text{pendulum}}(z^{\varsigma}, a) = -R^{\varsigma}_{\text{pendulum},z^{\varsigma}}(z^{\varsigma}) - R^{\varsigma}_{\text{pendulum},a}(a), \tag{4.17}$$

where

$$R^{\varsigma}_{\text{pendulum},z^{\varsigma}}(z^{\varsigma}) = \sum_{m=1}^{\varsigma+1} \max\left\{|z^{\varsigma}_m|, (z^{\varsigma}_m)^2\right\} + \sum_{l=(\varsigma+1)+1}^{(\varsigma+1)+(\varsigma+10)} 0.05 \max\left\{|z^{\varsigma}_l|, (z^{\varsigma}_l)^2\right\},$$
$$R^{\varsigma}_{\text{pendulum},a}(a) = \max\left\{|a_1|, a_1^2\right\}.$$

$z^{\varsigma} = [z^{\varsigma}_1 \, z^{\varsigma}_2 \, ... \, z^{\varsigma}_{2\varsigma+11}]^\top$, and $a = [a_1]^\top$. Note that z^{ς}_1, z^{ς}_2, ..., $z^{\varsigma}_{\varsigma+1}$ are previously observed outputs and $z^{\varsigma}_{\varsigma+2}$, $z^{\varsigma}_{\varsigma+3}$..., $z^{\varsigma}_{2\varsigma+11}$ are previously determined control actions.

The learning curve for $\varsigma = 0$ is shown in **Fig. 4.13(a)**. The agent cannot proceed with its policy learning. On the other hand, the learning curve for $\varsigma = 2$ is shown in **Fig. 4.13(b)**. The agent can learn the policy that obtains a high average of rewards. The time response of the pendulum controlled by the learned policy with $\varsigma = 2$ is shown in **Fig. 4.14**. The policy can stabilize the pendulum at $[0 \; 0]^\top$.



(a) $\varsigma = 0$

(b) $\varsigma = 2$

Fig. 4.13: Learning curves for the pendulum in the cases with $\varsigma = 0, 2$. The solid curves and the shades represent the average results and the standard deviations over 10 trials with different random seeds, respectively.

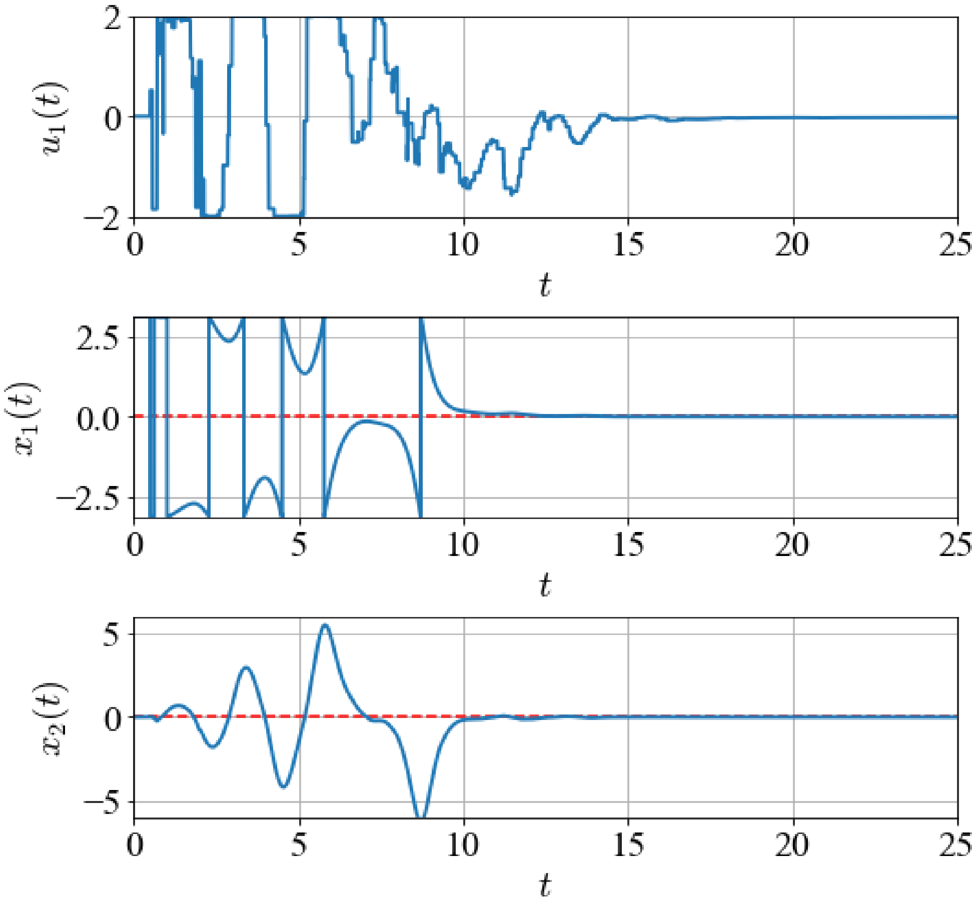


Fig. 4.14: Time response of the pendulum controlled by the learned policy with $\tau = 10$ and $\varsigma = 2$.

Lorenz
The output is given by

$$y_k = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0.1 \end{bmatrix} \begin{bmatrix} x_1(k\Delta) \\ x_2(k\Delta) \\ x_3(k\Delta) \end{bmatrix}. \tag{4.18}$$

In the experiment, we use a critic DNN and an actor DNN with two hidden layers, where all hidden layers have 128 units and all layers are fully connected. The activation functions are ReLU functions except for the output layers. In regards to the activation functions of the output layers, we use a hyperbolic tangent function, where the output is multiplied by 7, for the actor DNN and a linear function for the critic DNN. The size of the replay buffer $\mathcal{D}$ is 1.0×10^6 and the mini-batch size is $N = 128$. The parameter vectors of the actor DNN and the critic DNN are updated by Adam, where learning step sizes are set to 1.0×10^{-5} for the actor DNN and 1.0×10^{-4} for the critic DNN. The soft update rate of target networks is $\xi = 0.001$. The discount factor γ is 0.99. The initial state is randomly selected for each episode, where $-15 \leq x_1(0) \leq 15$, $-15 \leq x_2(0) \leq 15$, and $10 \leq x_3(0) \leq 50$.

We use the extended state z^ς as the state of the environment, where we consider the cases $\varsigma = 0, 3$. The reward function $R^\varsigma_{\text{Lorenz}} : (\mathcal{Y}^{(\varsigma+1)} \times \mathcal{U}^{(\varsigma+10)}) \times \mathcal{U} \times (\mathcal{Y}^{(\varsigma+1)} \times \mathcal{U}^{(\varsigma+10)}) \to \mathbb{R}$ is given by

$$R^\varsigma_{\text{Lorenz}}(z^\varsigma, a, z^{\varsigma\prime}) = -R^\varsigma_{\text{Lorenz},z^\varsigma}(z^\varsigma, z^{\varsigma\prime}) - R^\varsigma_{\text{Lorenz},a}(a), \tag{4.19}$$

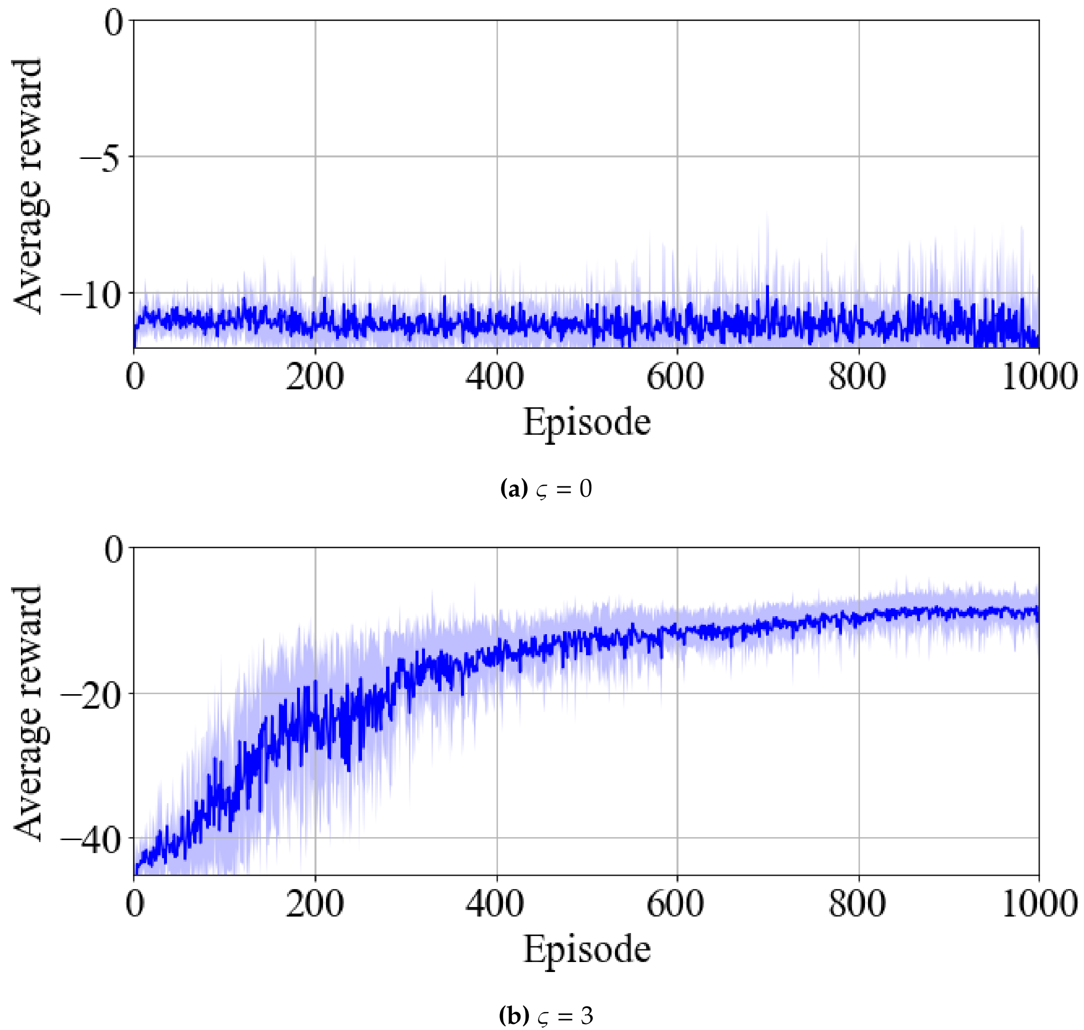
where

$$\begin{aligned} R^\varsigma_{\text{Lorenz},z^\varsigma}(z^\varsigma, z^{\varsigma\prime}) = \sum_{m=1}^{\varsigma+1} \Big(& \max\left\{|z^{\varsigma\prime}_{2m-1} - z^\varsigma_{2m-1}|, (z^{\varsigma\prime}_{2m-1} - z^\varsigma_{2m-1})^2\right\} \\ & +8.0 \max\left\{|z^{\varsigma\prime}_{2m} - z^\varsigma_{2m}|, (z^{\varsigma\prime}_{2m} - z^\varsigma_{2m})^2\right\}\Big) \\ & + \sum_{l=2(\varsigma+1)+1}^{2(\varsigma+1)+2(\varsigma+10)} 0.15 \max\left\{|z^\varsigma_l|, (z^\varsigma_l)^2\right\}, \\ R^\varsigma_{\text{Lorenz},a}(a) = 1.5 \max\left\{|a_1|, a_1^2\right\} & + 2.5 \max\left\{|a_2|, a_2^2\right\}, \end{aligned}$$

$z^\varsigma = [z^\varsigma_1 \ z^\varsigma_2 \ \dots \ z^\varsigma_{4\varsigma+22}]^\top$, $a = [a_1 \ a_2]^\top$, and $z^{\varsigma\prime} = [z^{\varsigma\prime}_1 \ z^{\varsigma\prime}_2 \ \dots \ z^{\varsigma\prime}_{4\varsigma+22}]^\top$. Note that $[z^\varsigma_1 \ z^\varsigma_2]^\top$, $[z^\varsigma_3 \ z^\varsigma_4]^\top$, ..., $[z^\varsigma_{2\varsigma+1} \ z^\varsigma_{2\varsigma+2}]^\top$ are previously observed output and $[z^\varsigma_{2\varsigma+3} \ z^\varsigma_{2\varsigma+4}]^\top$, $[z^\varsigma_{2\varsigma+5} \ z^\varsigma_{2\varsigma+6}]^\top$, ..., $[z^\varsigma_{4\varsigma+21} \ z^\varsigma_{4\varsigma+22}]^\top$ are previously determined control actions.

The learning curve for $\varsigma = 0$ is shown in **Fig. 4.15(a)**. The agent cannot proceed with its policy learning. On the other hand, the learning curve for $\varsigma = 3$ is shown in **Fig. 4.15(b)**. The agent can learn the policy that obtains a high average of rewards.

The time response of the Lorenz dynamics controlled by the learned policy with $\varsigma = 3$ is shown in **Fig. 4.16**. The policy can stabilize the Lorenz dynamics at an equilibrium point.



(a) $\varsigma = 0$

(b) $\varsigma = 3$

Fig. 4.15: Learning curves for the Lorenz dynamics in the cases with $\varsigma = 0, 3$. The solid curves and the shades represent the average results and the standard deviations over 10 trials with different random seeds, respectively.

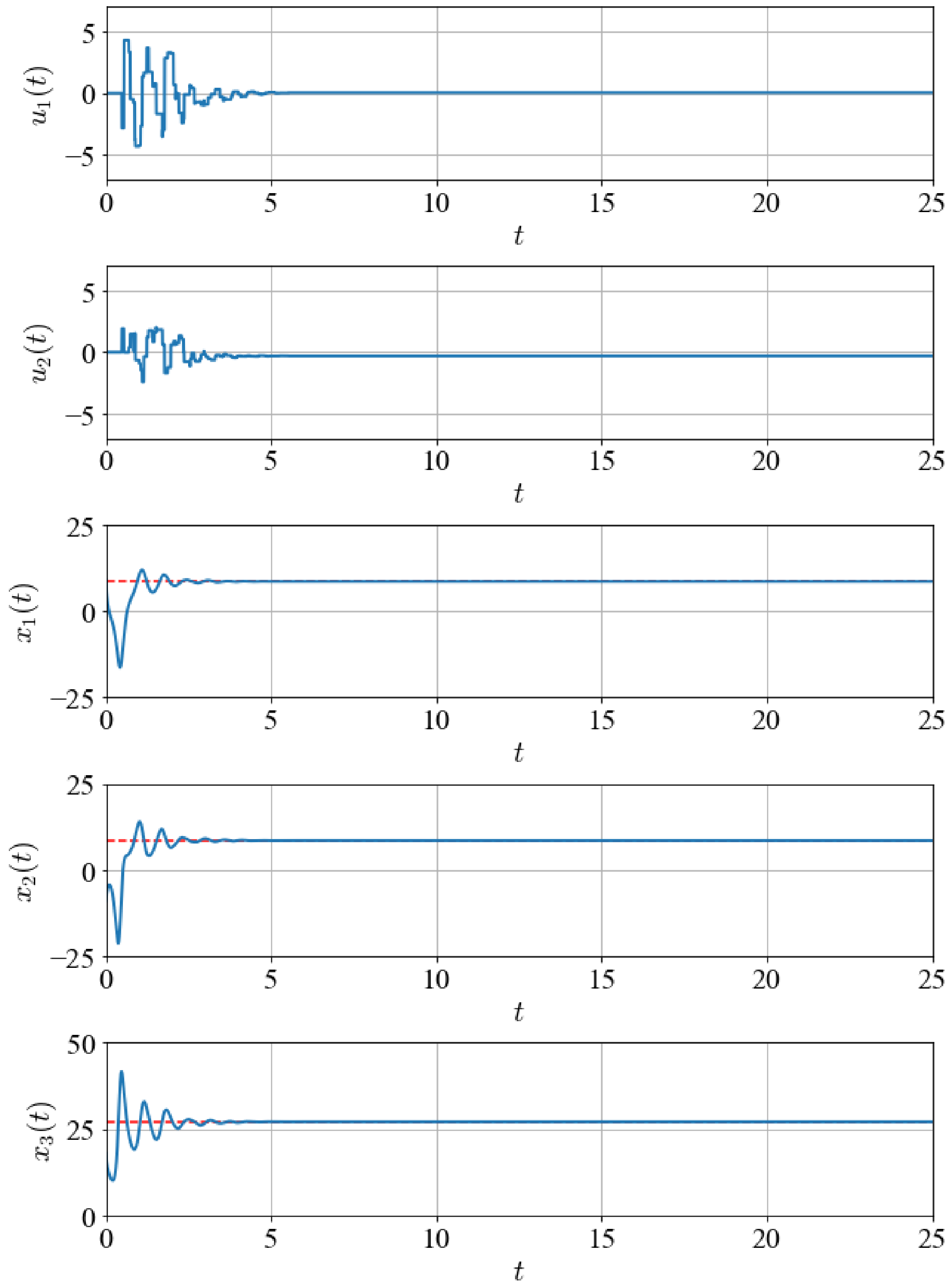


Fig. 4.16: Time response of the Lorenz dynamics controlled by the learned policy with $\tau = 10$ and $\varsigma = 3$.

Chapter 5

DRL for STL Tasks under Uncertain Network Delays

We apply DRL to design of a networked controller to complete a temporal control task described by an STL formula. STL is useful to specify a temporal control task with a bounded time interval mathematically. In general, an agent needs not only the current system's state but also the past system's behavior to determine a desired control action for completing the given STL task. Additionally, in an NCS, we need to consider the effect of network delays. Thus, we propose an extended MDP using some past system's states and control actions, which is called a τd-MDP, so that the agent can evaluate the satisfaction of the STL formula considering the effect of network delays. Thereafter, we apply a DRL algorithm to design of a networked controller for completing the STL task using the τd-MDP.

This chapter is based on "Deep reinforcement learning based networked control with network delays for signal temporal logic specifications" [54] which appeared in *Proceedings of IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*, © 2022 IEEE.

5.1 Signal Temporal Logic (STL)

In this chapter, the desired behavior of a discrete-time system is described by an STL formula with the following *syntax*.

$$
\begin{aligned}
\Phi &::= G_{[0,T_e]}\phi \mid F_{[0,T_e]}\phi, \\
\phi &::= \phi \wedge \phi \mid \phi \vee \phi \mid G_{[t_s,t_e]}\varphi \mid F_{[t_s,t_e]}\varphi, \\
\varphi &::= \psi \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi,
\end{aligned}
\tag{5.1}
$$

where $T_e, t_s, t_e \in \mathbb{N}_{\geq 0}$ are nonnegative constants for the time bounds. Φ, ϕ, φ, and ψ are STL formulae. ψ is an inequality-formed predicate such as $h(x) \leq y$, where $h : \mathcal{X} \to \mathbb{R}$ is a function of the system's state, and $y \in \mathbb{R}$ is a constant. $\mathcal{X}$ denotes the

system's state space. The Boolean operators $\neg, \wedge, \vee$ are negation, conjunction, and disjunction, respectively. The temporal operator $G_{\mathcal{T}}$ and $F_{\mathcal{T}}$ refer to *Globally* (always) and *Finally* (eventually), respectively. $\mathcal{T}$ denotes the time bound of the temporal operator. $\phi_i = G_{[t_s^i, t_e^i]}\varphi_i$ (or $F_{[t_s^i, t_e^i]}\varphi_i$), $i = 1, 2, ..., M$ are called *STL sub-formulae*.

For a finite trajectory $x_0 x_1 x_2 ... x_T$ whose length is $T+1$, x_t and $x_{t_1:t_2}$ denote the state at the discrete-time t and the partial trajectory for a discrete-time interval $[t_1, t_2]$, where $0 \le t_1 \le t_2 \le T$, that is, $x_{t_1:t_2} = x_{t_1} x_{t_1+1} ... x_{t_2-1} x_{t_2}$. The *Boolean semantics* of STL is recursively defined as follows:

$$
\begin{aligned}
& x_{t:T} \models \psi \Leftrightarrow h(x_t) \le y, \\
& x_{t:T} \models \neg\psi \Leftrightarrow \neg(x_{t:T} \models \psi), \\
& x_{t:T} \models \phi_1 \wedge \phi_2 \Leftrightarrow x_{t:T} \models \phi_1 \text{ and } x_{t:T} \models \phi_2, \\
& x_{t:T} \models \phi_1 \vee \phi_2 \Leftrightarrow x_{t:T} \models \phi_1 \text{ or } x_{t:T} \models \phi_2, \\
& x_{t:T} \models G_{[t_s, t_e]}\phi \Leftrightarrow x_{t':T} \models \phi, \ \forall t' \in [t + t_s, t + t_e], \\
& x_{t:T} \models F_{[t_s, t_e]}\phi \Leftrightarrow \exists t' \in [t + t_s, t + t_e], \text{ s.t. } x_{t':T} \models \phi.
\end{aligned}
\tag{5.2}
$$

The *quantitative semantics* of STL, which is called robustness, is recursively defined as follows:

$$
\begin{aligned}
\rho(x_{t:T}, \psi) &= y - h(x_t), \\
\rho(x_{t:T}, \neg\psi) &= -\rho(x_{t:T}, \psi) \\
\rho(x_{t:T}, \phi_1 \wedge \phi_2) &= \min\left\{\rho(x_{t:T}, \phi_1), \rho(x_{t:T}, \phi_2)\right\}, \\
\rho(x_{t:T}, \phi_1 \vee \phi_2) &= \max\left\{\rho(x_{t:T}, \phi_1), \rho(x_{t:T}, \phi_2)\right\}, \\
\rho(x_{t:T}, G_{[t_s, t_e]}\phi) &= \min_{t' \in [t+t_s, t+t_e]} \rho(x_{t':T}, \phi), \\
\rho(x_{t:T}, F_{[t_s, t_e]}\phi) &= \max_{t' \in [t+t_s, t+t_e]} \rho(x_{t':T}, \phi),
\end{aligned}
\tag{5.3}
$$

which quantifies how well the system's trajectory satisfies the given STL formulae. If the robustness $\rho(x_{t:T}, \phi)$ is positive, the trajectory $x_{t:T}$ satisfies the formula ϕ.

The horizon length of an STL formula is recursively defined as follows:

$$
\begin{aligned}
\mathrm{hrz}(\psi) &= 0, \\
\mathrm{hrz}(\phi) &= t_e, \text{ for } \phi = G_{[t_s, t_e]}\varphi \text{ or } F_{[t_s, t_e]}\varphi, \\
\mathrm{hrz}(\neg\phi) &= \mathrm{hrz}(\phi), \\
\mathrm{hrz}(\phi_1 \wedge \phi_2) &= \max\left\{\mathrm{hrz}(\phi_1), \mathrm{hrz}(\phi_2)\right\}, \\
\mathrm{hrz}(\phi_1 \vee \phi_2) &= \max\left\{\mathrm{hrz}(\phi_1), \mathrm{hrz}(\phi_2)\right\}, \\
\mathrm{hrz}(G_{[t_s, t_e]}\phi) &= t_e + \mathrm{hrz}(\phi), \\
\mathrm{hrz}(F_{[t_s, t_e]}\phi) &= t_e + \mathrm{hrz}(\phi).
\end{aligned}
\tag{5.4}
$$

$\mathrm{hrz}(\phi)$ is the length of the state sequence which is required to verify the satisfaction of the STL formula ϕ.

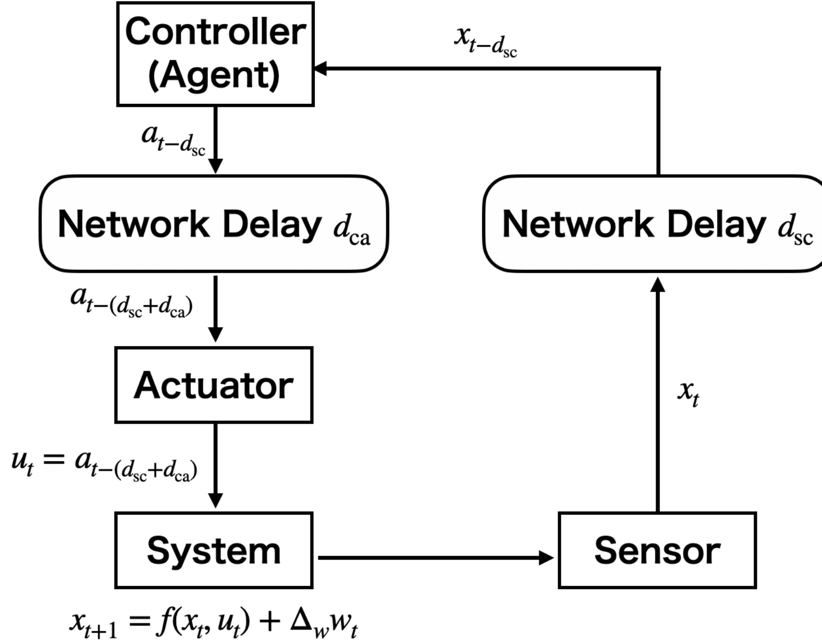


Fig. 5.1: Illustration of an NCS with a stochastic discrete-time system. (Copyright © 2022 IEEE)

5.2 Problem Formulation

We design a networked controller for the following stochastic discrete-time dynamical system as shown in **Fig. 5.1**.

$$x_{t+1} = f(x_t, u_t) + \Delta_w w_t, \tag{5.5}$$

where $x_t \in \mathcal{X}$, $u_t \in \mathcal{U}$, and $w_t \in \mathcal{W}$ are the system state, the control input, and the system noise at the discrete-time $t \in \{0, 1, ..., T\}$. $\mathcal{X} = \mathbb{R}^{n_x}$, $\mathcal{U} \subseteq \mathbb{R}^{n_u}$, and $\mathcal{W} = \mathbb{R}^{n_x}$ are the state space, the control input space, and the system's noise space, respectively. The system's noise w_t is an independent and identically distributed random variable with a probability density $p_w : \mathcal{W} \to \mathbb{R}_{\geq 0}$. Δ_w is a regular matrix that is a weighting factor of the system's noise. $f : \mathcal{X} \times \mathcal{U} \to \mathcal{X}$ is a function that describes the system's dynamics without a system's noise. Then, we have the transition probability density

$$p_f(x'|x, u) = |\Delta_w^{-1}| p_w(\Delta_w^{-1}(x' - f(x, u))).$$

The initial state $x_0 \in \mathcal{X}$ is sampled from a probability density $p_0 : \mathcal{X} \to \mathbb{R}_{\geq 0}$. The goal is to design a control policy such that $x_{0:T}^{\pi} \models \Phi$, where $x_{0:T}^{\pi}$ is a system's trajectory controlled by a control policy π, Φ is a given STL formula, and $T = T_e + \mathrm{hrz}(\phi)$.

Additionally, in an NCS, there exist two types of network delays: a sensor-to-controller delay $d_{\mathrm{sc}} \in \mathbb{N}$ caused by the transmission of an observed state and a controller-to-actuator delay $d_{\mathrm{ca}} \in \mathbb{N}$ caused by the transmission of a control input computed by the controller. In this chapter, it is assumed that these delays are unknown constants, where they are bounded by the maximum delays $d_{\mathrm{sc}}^{\max} \in \mathbb{N}$ and $d_{\mathrm{ca}}^{\max} \in \mathbb{N}$, respectively. Then, the controller computes the k-th control input a_k at $t = k + d_{\mathrm{sc}}$. Actually, the control input a_k is inputted to the system as follows:

$$u_t = \begin{cases} a_k & t = k + d_{\mathrm{sc}} + d_{\mathrm{ca}}, \\ 0_{n_u} & 0 \leq t < d_{\mathrm{sc}} + d_{\mathrm{ca}}, \end{cases} \tag{5.6}$$

where 0_{n_u} is a zero-vector of the Euclidean space $\mathbb{R}^{n_u}$, that is, the actuator does not input a control input until receiving a_0. The controller computes control inputs $a_0, a_1, ..., a_{T-d_{\mathrm{sc}}-d_{\mathrm{ca}}}$ to satisfy the given STL formula ϕ.

In this chapter, we assume that the mathematical models f and p_w are unknown. Thus, we apply RL to design of a networked controller for satisfying an STL formula Φ. We regard the controller as the agent and call a control input determined by the agent a control action. In addition, we need to identify the environment's state space for a temporal control task described by an STL formula beforehand and to design a reward function to evaluate the satisfaction of the given STL formula appropriately. Aksaray *et al.* introduced an extended MDP, which is called *τ-MDP*, and proposed the classical Q-learning based algorithm for satisfying an STL formula [49]. However, the classical Q-learning algorithm cannot be directly applied to the problem in this chapter due to the following problems.

(i) The classical Q-learning algorithm cannot deal with a continuous state-action space directly.
(ii) There are uncertain network delays in the NCS.

5.3 Q-Learning for Satisfying an STL Formula Using a τ-MDP

In this section, we review the Q-learning algorithm to learn a policy satisfying an STL formula [49]. Although we often regard the current system's state as the environment's state for RL, the current system's state is not enough to determine an action for satisfying a given STL formula Φ. Thus, Aksaray *et al.* defined the following extended state using some previous system's state.

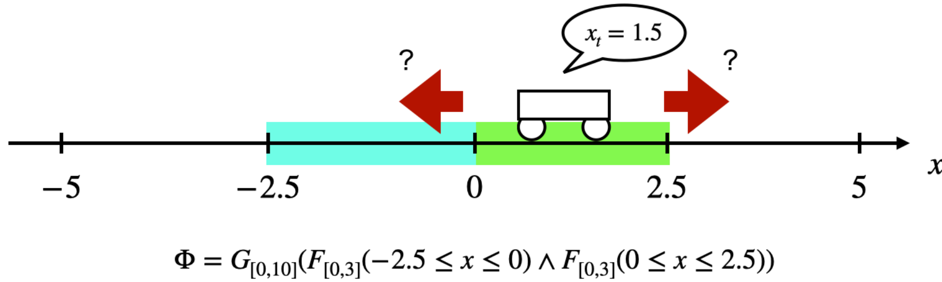
$$x_t^{\tau} = [x_{t-\tau+1}^{\top} \; x_{t-\tau+2}^{\top} \; ... \; x_t^{\top}]^{\top} \in \mathcal{X}^{\tau},$$

where $\tau = \mathrm{hrz}(\phi) + 1$ for the given STL formula $\Phi = G_{[0,T_e]}\phi$ (or $F_{[0,T_e]}\phi$) and $\mathcal{X}^{\tau}$ is an extended state space. We show a simple example in **Fig. 5.2**. We operate a

one-dimensional dynamical system to satisfy the STL formula

$$\Phi = G_{[0,10]}(F_{[0,3]}(-2.5 \leq x \leq 0) \wedge F_{[0,3]}(0 \leq x \leq 2.5)).$$

At any time in the time interval $[0, 10]$, the system must enter both the blue region and the green region before 3 time steps elapsed, where there is no constraint for the order of the visits. Let the current system's state be $x_t = 1.5$. Note that the desired action for the STL formula Φ is different depending on the past system's states. For example, in the case where $x_{t-3:t} = -0.5,\ 0.5,\ 1.0,\ 1.5$, we should operate the system to the blue region right away. On the other hand, in the case where $x_{t-3:t} = -1.5,\ -2.5,\ -0.5,\ 1.5$, we do not need to move it. Therefore, we regard not only the current system's state but also some previous system's states as an environment's state for RL.



Case 1

x_{t-3}	x_{t-2}	x_{t-1}	x_t
−0.5	0.5	1.0	1.5

Case 2

x_{t-3}	x_{t-2}	x_{t-1}	x_t
−1.5	−2.5	−0.5	1.5

Fig. 5.2: Illustration of a simple example of a temporal control task described by an STL formulae.

Additionally, Aksaray *et al.* designed the reward function $R_{STL} : \mathcal{X}^{\tau} \to \mathbb{R}$ using robustness and the log-sum-exp approximation. The robustness of a trajectory $x_{0:T}$ with respect to the given STL formula Φ is as follows:

$$\begin{aligned}
\rho(x_{0:T}, \Phi) &= \begin{cases} \min \left\{ \rho(x_{0:\tau-1}, \phi),\ \dots\ \rho(x_{T-\tau+1:T}, \phi) \right\} & \text{for } \Phi = G_{[0,T_e]}\phi, \\ \max \left\{ \rho(x_{0:\tau-1}, \phi),\ \dots\ \rho(x_{T-\tau+1:T}, \phi) \right\} & \text{for } \Phi = F_{[0,T_e]}\phi, \end{cases} \\
&= \begin{cases} \min \left\{ \rho(x^{\tau}_{\tau-1}, \phi),\ \dots\ \rho(x^{\tau}_{T}, \phi) \right\} & \text{for } \Phi = G_{[0,T_e]}\phi, \\ \max \left\{ \rho(x^{\tau}_{\tau-1}, \phi),\ \dots\ \rho(x^{\tau}_{T}, \phi) \right\} & \text{for } \Phi = F_{[0,T_e]}\phi. \end{cases}
\end{aligned} \tag{5.7}$$

We consider the following problem.

$$\max_{\pi} Pr\left[x^{\pi}_{0:T} \models \Phi \right] = \max_{\pi} E\left[\mathbf{1}(\rho(x^{\pi}_{0:T}, \Phi)) \right], \tag{5.8}$$

where $x^{\pi}_{0:T}$ is the system's trajectory controlled by the policy π and the function $\mathbf{1} : \mathbb{R} \to \{0, 1\}$ is an indicator defined by

$$\mathbf{1}(y) = \begin{cases} 1 & \text{if } y \geq 0, \\ 0 & \text{if } y < 0. \end{cases} \tag{5.9}$$

Since $\mathbf{1}(\min\{y_1, ..., y_n\}) = \min\{\mathbf{1}(y_1), ..., \mathbf{1}(y_n)\}$ and $\mathbf{1}(\max\{y_1, ..., y_n\}) = \max\{\mathbf{1}(y_1), ..., \mathbf{1}(y_n)\}$,

$$\begin{aligned} \max_{\pi} E\left[\mathbf{1}(\rho(x^{\pi}_{0:T}, \Phi))\right] &= \begin{cases} \max_{\pi} E\left[\mathbf{1}(\min_{\tau-1 \leq t \leq T} \rho(x^{\tau}_t, \phi))\right] & \text{if } \Phi = G_{[0,T_e]}\phi, \\ \max_{\pi} E\left[\mathbf{1}(\max_{\tau-1 \leq t \leq T} \rho(x^{\tau}_t, \phi))\right] & \text{if } \Phi = F_{[0,T_e]}\phi, \end{cases} \\ &= \begin{cases} \max_{\pi} E\left[\min_{\tau-1 \leq t \leq T} \mathbf{1}(\rho(x^{\tau}_t, \phi))\right] & \text{if } \Phi = G_{[0,T_e]}\phi, \\ \max_{\pi} E\left[\max_{\tau-1 \leq t \leq T} \mathbf{1}(\rho(x^{\tau}_t, \phi))\right] & \text{if } \Phi = F_{[0,T_e]}\phi. \end{cases} \end{aligned} \tag{5.10}$$

Then, we use the following log-sum-exp approximation.

$$\min\{y_1, ..., y_n\} \simeq -\frac{1}{\beta} \log \sum_{i=1}^{n} \exp(-\beta y_i), \tag{5.11}$$

$$\max\{y_1, ..., y_n\} \simeq \frac{1}{\beta} \log \sum_{i=1}^{n} \exp(\beta y_i), \tag{5.12}$$

where $\beta > 0$ is an approximation parameter. We can approximate $\min\{\cdots\}$ or $\max\{\cdots\}$ with arbitrary accuracy by selecting a large β. Then, (5.10) can be approximated as follows:

$$\max_{\pi} E[\mathbf{1}(\rho(x^{\pi}_{0:T}, \Phi))] \simeq \begin{cases} \max_{\pi} E\left[-\frac{1}{\beta} \log \sum_{t=\tau-1}^{T} \exp(-\beta \mathbf{1}(\rho(x^{\tau}_t, \phi)))\right] & \text{if } \Phi = G_{[0,T_e]}\phi, \\ \max_{\pi} E\left[\frac{1}{\beta} \log \sum_{t=\tau-1}^{T} \exp(\beta \mathbf{1}(\rho(x^{\tau}_t, \phi)))\right] & \text{if } \Phi = F_{[0,T_e]}\phi. \end{cases} \tag{5.13}$$

Since the log function is a strictly monotonic function and $1/\beta > 0$ is a constant, we have

$$\begin{aligned} &\begin{cases} \arg\max_{\pi} E\left[-\frac{1}{\beta} \log \sum_{t=\tau-1}^{T} \exp(-\beta \mathbf{1}(\rho(x^{\tau}_t, \phi)))\right] & \text{if } \Phi = G_{[0,T_e]}\phi, \\ \arg\max_{\pi} E\left[\frac{1}{\beta} \log \sum_{t=\tau-1}^{T} \exp(\beta \mathbf{1}(\rho(x^{\tau}_t, \phi)))\right] & \text{if } \Phi = F_{[0,T_e]}\phi. \end{cases} \\ \Leftrightarrow &\begin{cases} \arg\max_{\pi} E\left[-\sum_{t=\tau-1}^{T} \exp(-\beta \mathbf{1}(\rho(x^{\tau}_t, \phi)))\right] & \text{if } \Phi = G_{[0,T_e]}\phi, \\ \arg\max_{\pi} E\left[\sum_{t=\tau-1}^{T} \exp(\beta \mathbf{1}(\rho(x^{\tau}_t, \phi)))\right] & \text{if } \Phi = F_{[0,T_e]}\phi. \end{cases} \end{aligned} \tag{5.14}$$

Thus, we use the following reward function $R_{STL} : \mathcal{X}^{\tau} \to \mathbb{R}$ to satisfy the given STL formula Φ.

$$R_{STL}(x^{\tau}) = \begin{cases} -\exp(-\beta \mathbf{1}(\rho(x^{\tau}, \phi))) & \text{if } \Phi = G_{[0,T_e]}\phi, \\ \exp(\beta \mathbf{1}(\rho(x^{\tau}, \phi))) & \text{if } \Phi = F_{[0,T_e]}\phi. \end{cases} \tag{5.15}$$

Remark: Note that the agent cannot determine a control action until $t = \tau - 1$. If a partial trajectory $x_{0:\tau-1}$ does not satisfy ϕ, where $\mathrm{hrz}(\phi) = \tau - 1$, then the agent cannot satisfy the STL specification $\Phi = G_{[0,T_e]}\phi$. Thus, we assume that $x_{<0} = x_0$, that is, the system keeps an initial state x_0 before $t = 0$. The agent can also operate the system at $t = 0, 1, ..., \tau - 1$.

To design a policy satisfying an STL formula Φ using the Q-learning algorithm [6], Aksaray *et al.* proposed a τ-MDP as follows:

Definition 5.1 (τ-MDP)
We consider an STL formula $\Phi = G_{[0,T_e]}\phi$ (or $F_{[0,T_e]}\phi$), where $\mathrm{hrz}(\Phi) = T$ and ϕ consists of multiple STL sub-formulae ϕ_i, $i \in \{1, 2, ..., M\}$. Subsequently, we set $\tau = \mathrm{hrz}(\phi) + 1$, that is, $T = T_e + \tau - 1$. A τ-MDP is defined by a tuple $\mathcal{M}_\tau = \left\langle \mathcal{X}^\tau, \mathcal{U}, p_0^\tau, p^\tau, R_{STL} \right\rangle$, where

- $\mathcal{X}^\tau$ is an extended state space which we regard as an environment's state space for RL. The extended state $x^\tau \in \mathcal{X}^\tau$ is a vector of multiple system's states $x^\tau = [x^\tau[0]^\top \ x^\tau[1]^\top \ ... \ x^\tau[\tau - 1]^\top]^\top$, $x^\tau[i] \in \mathcal{X}$, $\forall i \in \{0, 1, ..., \tau - 1\}$.
- $\mathcal{U}$ is an agent's control action space.
- p_0^τ is a probability density for an initial extended state x_0^τ with $x_0^\tau[i] = x_0$, $\forall i \in \{0, 1, ..., \tau - 1\}$, where x_0 is generated from p_0.
- p^τ is a transition probability density for the extended state x^τ. In the case where the system's state is updated by $x' \sim p_f(\cdot|x, a)$, the extended state is updated by $x^{\tau'} \sim p^\tau(\cdot|x^\tau, a)$ as follows:

$$x^{\tau'}[i] = x^\tau[i + 1], \ \forall i \in \{0, 1, ..., \tau - 2\},$$
$$x^{\tau'}[\tau - 1] \sim p_f(\cdot|x^\tau[\tau - 1], a).$$

  Fig. 5.3 shows an example of the transition. We consider the sequence that consists of states $x_{t-\tau+1}, \ x_{t-\tau+2}, \ \cdots, \ x_t$ as the extended state at t. In the transition, the head system's state $x_{t-\tau+1}$ is removed from the sequence and other system's states $x_{t-\tau+2}, \ \cdots, \ x_t$ are shifted to the left. After that, the next system's state x_{t+1} updated by $p_f(\cdot|x_t, a_t)$ is inputted to the tail of the sequence. The next extended state x_{t+1}^τ depends on the current extended state x_t^τ and the agent's control action a_t.
- $R_{STL} : \mathcal{X}^\tau \to \mathbb{R}$ is a reward function defined by (5.15).

5.4 DRL for Satisfying an STL Formula under Network Delays

Aksaray *et al.* introduced a τ-MDP using a finite past state sequence and designed a reward function (5.15) to satisfy a given STL formula Φ. However, the previous

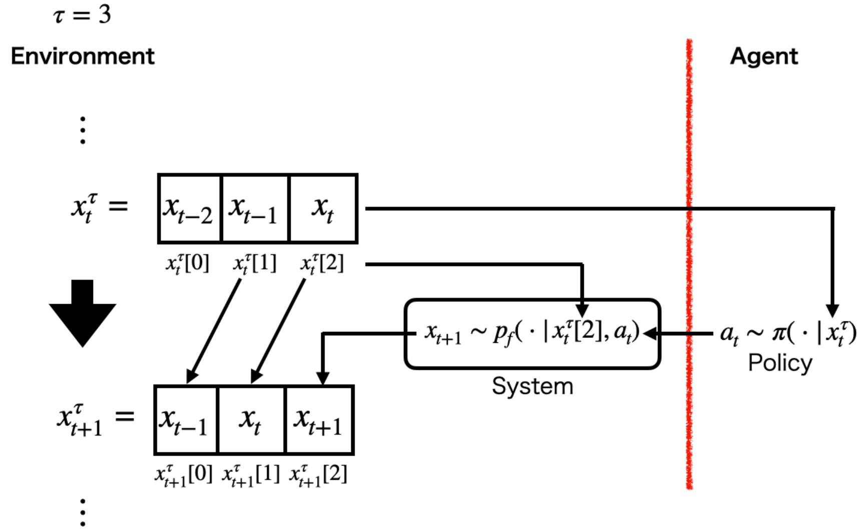


Fig. 5.3: Illustration of an extended state's transition. We show the case $\tau = 3$ as an example. The $(t+1)$-th extended state x^{τ}_{t+1} depends on the t-th extended state x^{τ}_{t} and the t-th control action a_t.

learning algorithm cannot directly handle control tasks with continuous state-action spaces since it is based on the classical Q-learning algorithm. To resolve (i) in **Section. 5.2**, we extend the previous learning algorithm using a DRL algorithm. We apply DRL algorithms derived from Q-learning for problems with continuous state-action spaces such as TD3 [12] and SAC [13,14].

Remark: The standard DRL algorithm based on Q-learning is the DQN algorithm [10]. However, the DQN algorithm cannot handle continuous action spaces due to its DNN architecture.

Additionally, we consider the effect of network delays. In **Chapter 4**, we proposed an extended state that consists of the latest observed state and previously determined control actions to design a policy for stabilization of a nonlinear system considering network delays. Thus, to solve (ii) in **Section. 5.2**, we use not only the extended state proposed in [49] but also previously determined control actions. For simplicity, we consider the worst case scenario, that is, $d_{\mathrm{sc}} = d^{\mathrm{max}}_{\mathrm{sc}}$, $d_{\mathrm{ca}} = d^{\mathrm{max}}_{\mathrm{ca}}$ as shown in **Fig. 5.4**. Let $d = d^{\mathrm{max}}_{\mathrm{sc}} + d^{\mathrm{max}}_{\mathrm{ca}}$. At $t = k$, the sensor observes the k-th system state x_k and sends it to the agent. At $t = k + d^{\mathrm{max}}_{\mathrm{sc}}$, the agent receives x_k, determines the k-th control action a_k, and sends it to the actuator. At $t = k + d$, the actuator receives a_k and updates the control input $u_t = a_k$. Note that the k-th control action a_k is actually inputted to the system at $t = k + d$. The k-th control action a_k must be determined to satisfy the given STL formula, that is, the agent needs the partial

trajectory $x_{k+\tau-1+d}, \ldots, x_{k+d-1}, x_{k+d}$ to determine a desire control action at $t = k + d_{\text{sc}}^{\max}$. However, $x_{k+1}, \ldots, x_{k+d-1}, x_{k+d}$ are not available. The agent should predict these states. In order to predict them, the agent needs control inputs u_t for $t \in [k, k+d]$, which are control actions $a_{k-d}, a_{k-d+1}, \ldots, a_{k-1}$, even if the mathematical model of the system is known. Thus, we include the previously determined actions $a_{k-d}, \ldots, a_{k-2}, a_{k-1}$ to the extended state $x_k^{\tau} = [x_{k-\tau+1}^{\top}, \cdots, x_k^{\top}]^{\top}$ as follows:

$$z_k = [x_{k-\tau+1}^{\top} \; x_{k-\tau+2}^{\top} \; \cdots \; x_k^{\top} \; a_{k-d}^{\top} \; a_{k-d+1}^{\top} \; \cdots \; a_{k-1}^{\top}]^{\top} \in \mathcal{X}^{\tau} \times \mathcal{U}^{d}.$$

We consider z_k as an environment's state for RL in the NCS problem.

Network delays d_{sc} and d_{ca} are not necessarily maximum values $d_{\text{sc}}^{\max}$ and $d_{\text{ca}}^{\max}$, respectively. Then, the agent learns its policy adaptively using z_k that has sufficient information for the worst case scenario. Although the dimensionality of the extended state received by the agent becomes large due to the extension with previous control actions, it is not problematic in our proposed method because DRL are effective for high dimensional tasks.

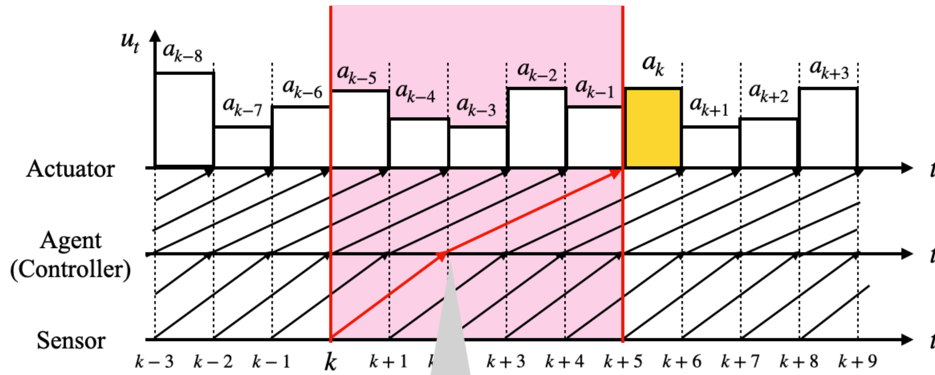


Fig. 5.4: Illustration of the network delays in the worst case, where $d_{\text{sc}}^{\max} = 2$ and $d_{\text{ca}}^{\max} = 3$. In the case, at $t = k+2$, the agent should predict the states $x_{k+1}, x_{k+2}, x_{k+3}, x_{k+4}, x_{k+5}$ using the k-th state x_k and the previously determined control action $a_{k-5}, a_{k-4}, a_{k-3}, a_{k-2}, a_{k-1}$ and determine the k-th control action a_k based on the previously observed states and the predictions of $x_{k+1}, x_{k+2}, x_{k+3}, x_{k+4}, x_{k+5}$ for satisfying the given STL formula Φ.

Remark: In the worst case $d_{\text{sc}} + d_{\text{ca}} = d$, the agent does not need $x_{k-\tau+1}, x_{k-\tau+2}, \ldots, x_{k-\tau+d}$ because it determines a desired control action based on $x_{k-\tau+1+d}, x_{k-\tau+2+d}, \ldots, x_{k+d}$,

where the agent needs to predict $x_{k+1}, \cdots, x_{k+d}$. In general, however, $d_{sc} + d_{ca}$ may not be the maximum delay d. As shown in **Fig. 5.5**, when true network delays are $d_{sc} + d_{ca} (< d)$, the agent needs $x_{k-\tau+1+d_{sc}+d_{ca}}, x_{k-\tau+2+d_{sc}+d_{ca}}, \ldots, x_{k+d_{sc}+d_{ca}}$. In other words, the agent also needs $x_{k-\tau+1+d_{sc}+d_{ca}}, x_{k-\tau+2+d_{sc}+d_{ca}}, \ldots, x_{k-\tau+d}$ other than $x_{k-\tau+1+d}, x_{k-\tau+2+d}, \ldots, x_{k+d_{sc}+d_{ca}}$. To adapt the unknown network delay $0 \leq d_{sc} + d_{ca} \leq d$ using sufficient information, we construct the extended state z using the partial trajectory $x_{k-\tau+1}, x_{k-\tau+2}, \ldots, x_k$ and the previously determined actions $a_{k-1}, a_{k-2}, \ldots, a_{k-d}$.

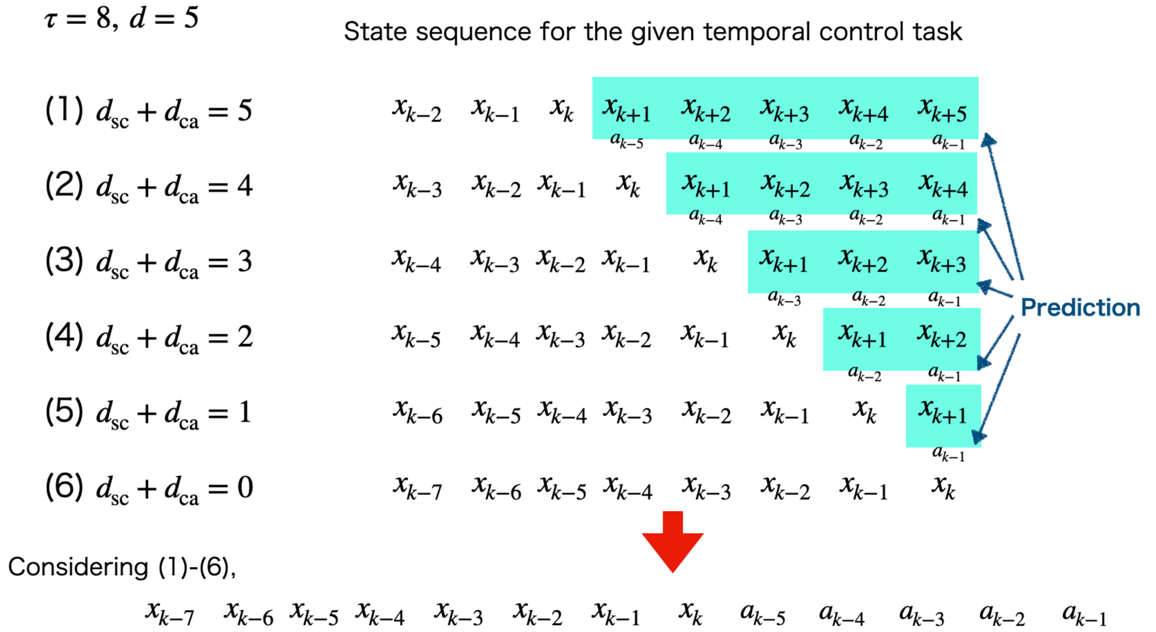


Fig. 5.5: Illustration of the state sequences for the given temporal control task in the cases $d_{sc} + d_{ca} = 0, 1, 2, 3, 4, 5$, where $\tau = 8$ and $d = 5$. In the worst case scenario, we need $x_{k-2}, x_{k-1}, x_k, a_{k-5}, a_{k-4}, a_{k-3}, a_{k-2}, a_{k-1}$. On the other hand, in the case where $d_{sc} + d_{ca} < 5$, we need $x_{k-7+d_{sc}+d_{ca}}, \cdots, x_{k-3}$ other than x_{k-2}, x_{k-1}, x_k. To adapt the uncertain network delay $0 \leq d_{sc} + d_{ca} \leq d$ using sufficient information, we construct the extended state z_k using $x_{k-7}, x_{k-6}, \cdots, x_k, a_{k-5}, a_{k-4}, \cdots, a_{k-1}$.

5.4.1 τd-Markov Decision Process (τd-MDP)

To solve the problem formulated in **Section. 5.2**, we model the interactions between the agent and the system as the following extended MDP, which is called a τd-MDP.

Definition 5.2 (τd-MDP)

We consider an STL formula $\Phi = G_{[0,T_e]}\phi$ (or $F_{[0,T_e]}\phi$), where $\mathrm{hrz}(\Phi) = T$ and ϕ consists of multiple STL sub-formulae ϕ_i, $i \in \{1, 2, \ldots, M\}$. Subsequently, we set $\tau = \mathrm{hrz}(\phi) + 1$, that is, $T = T_e + \tau - 1$. It is assumed that $d_{sc}^{\max} + d_{ca}^{\max} = d$. A τd-MDP is defined by a tuple $\mathcal{M}_{\tau,d} = \left\langle \mathcal{Z}, \mathcal{U}, p_0^z, p^z, R_{STL} \right\rangle$, where

- $\mathcal{Z}$ $(= \mathcal{X}^{\tau} \times \mathcal{U}^{d})$ is an extended state space. An extended state is denoted by $z = [(x^{\tau})^{\top}\ (a^{d})^{\top}]^{\top}$, where $x^{\tau} = [x^{\tau}[0]^{\top}\ x^{\tau}[1]^{\top}\ \dots\ x^{\tau}[\tau-1]^{\top}]^{\top}$ and $a^{d} = [a^{d}[0]^{\top}\ a^{d}[1]^{\top}\ \dots\ a^{d}[d-1]^{\top}]^{\top}$ are a previous system's state sequence and a previously determined control action sequence, respectively, that is, $x^{\tau}[i] \in \mathcal{X},\ \forall i \in \{0,1,\dots,\tau-1\}$ and $a^{d}[j] \in \mathcal{U},\ \forall j \in \{0,1,\dots,d-1\}$.
- $\mathcal{U}$ is an agent's control action space.
- p_0^z is a probability density for an initial extended state $z_0 = [(x_0^{\tau})^{\top}\ (a_0^{d})^{\top}]^{\top}$ with $x_0^{\tau}[i] = x_0,\ \forall i \in \{0,1,\dots,\tau-1\}$ and $a_0^{d}[j] = 0_{n_u},\ \forall j \in \{0,1,\dots,d-1\}$, where x_0 is generated from p_0.
- p^z is a transition probability density for the extended state z. In the case where the system's state is updated by $x' \sim p_f(\cdot|x,a)$, the extended state is updated by $z' \sim p^z(\cdot|z,a)$ as follows:

$$
\begin{aligned}
&a^{d'}[j] = a^{d}[j+1],\ \forall j \in \{0,1,\dots,d-2\},\\
&a^{d'}[d-1] = a,\\
&x^{\tau'}[i] = x^{\tau}[i+1],\ \forall i \in \{0,1,\dots,\tau-2\},\\
&x^{\tau'}[\tau-1] \sim \begin{cases} p_f(\cdot|x^{\tau}[\tau-1], a^{d'}[d-1-d_{\mathrm{sc}}-d_{\mathrm{ca}}]), & d \neq d_{\mathrm{sc}}+d_{\mathrm{ca}} \\ p_f(\cdot|x^{\tau}[\tau-1], a^{d}[0]), & d = d_{\mathrm{sc}}+d_{\mathrm{ca}} \end{cases}
\end{aligned}
$$

  where $z = [(x^{\tau})^{\top}\ (a^{d})^{\top}]^{\top}$ and $z' = [(x^{\tau'})^{\top}\ (a^{d'})^{\top}]^{\top}$ are the current extended state and the next extended state, respectively. **Fig. 5.6** shows a transition of an extended state of the τd-MDP.
- $R_{STL} : \mathcal{Z} \to \mathbb{R}$ is a reward function defined by

$$
R_{STL}(z) = \begin{cases} -\exp(-\beta \mathbf{1}(\rho(x^{\tau},\phi))) & \text{if } \Phi = G_{[0,T_e]}\phi, \\ \exp(\beta \mathbf{1}(\rho(x^{\tau},\phi))) & \text{if } \Phi = F_{[0,T_e]}\phi, \end{cases} \tag{5.16}
$$

  where $z = [(x^{\tau})^{\top}\ (a^{d})^{\top}]^{\top}$.

We apply a DRL algorithm derived from the Q-learning algorithm to design of a control policy satisfying a given STL formula Φ using the τd-MDP.

5.4.2 Pre-Process

If τ is a large value, it is difficult for the agent to learn its policy due to the large dimensionality of the extended state space $\mathcal{Z}$. Then, *pre-process* is useful in order to reduce the dimension, which is related to [50]. In the previous study, a flag state for each sub-formula is defined as a discrete state. The discrete flag state space is combined with the discrete system's state space as a pre-processed state. On the other hand, in this chapter, it is assumed that the system's state space is continuous. If we use the discrete flag states, the pre-processed state space is a hybrid state space that has discrete values and continuous values. Thus, we consider the flag state as a continuous value and input it to DNNs as shown in **Fig. 5.7**.

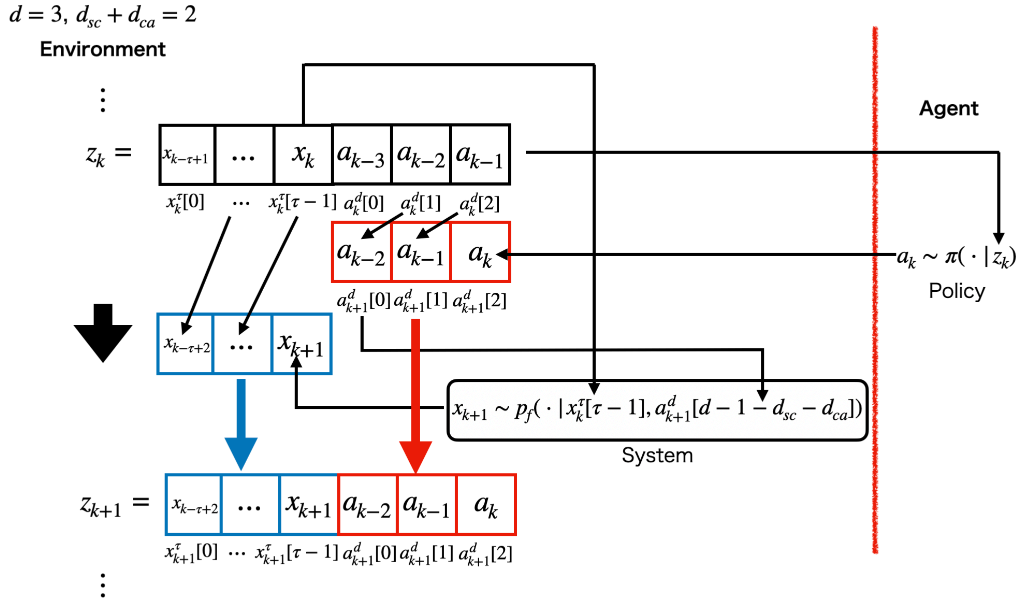


Fig. 5.6: Illustration of an extended state's transition. We consider the case where the true delays are $d_{\mathrm{sc}} = 1$, $d_{\mathrm{ca}} = 1$. It is assumed that $d = 3$. The $(k+1)$-th extended state z_{k+1} depends on the k-th extended state z_k and the k -th control action a_k.

At first, we consider the case without network delays $d = 0$. We introduce a flag value f^i for each STL sub-formula ϕ_i.

Definition 5.3 (Pre-Processing)
For an extended state x^τ, the flag value f^i of an STL sub-formula ϕ_i is defined as follows:
(i) For $\phi_i = G_{[t_s^i, t_e^i]}\varphi_i$,

$$f^i = \max\left\{ \frac{t_e^i - l + 1}{t_e^i - t_s^i + 1} \,\middle|\, l \in \{t_s^i, \dots, t_e^i\} \wedge (\forall l' \in \{l, \dots, t_e^i\},\ x^\tau[l'] \models \varphi_i) \right\}. \tag{5.17}$$

(ii) For $\phi_i = F_{[t_s^i, t_e^i]}\varphi_i$,

$$f^i = \max\left\{ \frac{l - t_s^i + 1}{t_e^i - t_s^i + 1} \,\middle|\, l \in \{t_s^i, \dots, t_e^i\} \wedge x^\tau[l] \models \varphi_i \right\}, \tag{5.18}$$

where $\max \emptyset = -\infty$. The flag value represents the normalized time lying in $(0, 1] \cap \{-\infty\}$. Intuitively, for $\phi_i = G_{[t_s^i, t_e^i]}\varphi_i$, the flag value indicates the time duration in which ϕ_i is always satisfied, whereas, for $\phi_i = F_{[t_s^i, t_e^i]}\varphi_i$, the flag value indicates the instant when φ_i is satisfied. The flag values f^i, $i \in \{1, 2, \dots, M\}$ calculated by (5.17) or (5.18) are transformed into $\hat{f}^i$ as follows:

$$\hat{f}^i = \begin{cases} f^i - \frac{1}{2} & \text{if } f^i \neq -\infty, \\ -\frac{1}{2} & \text{otherwise.} \end{cases} \tag{5.19}$$

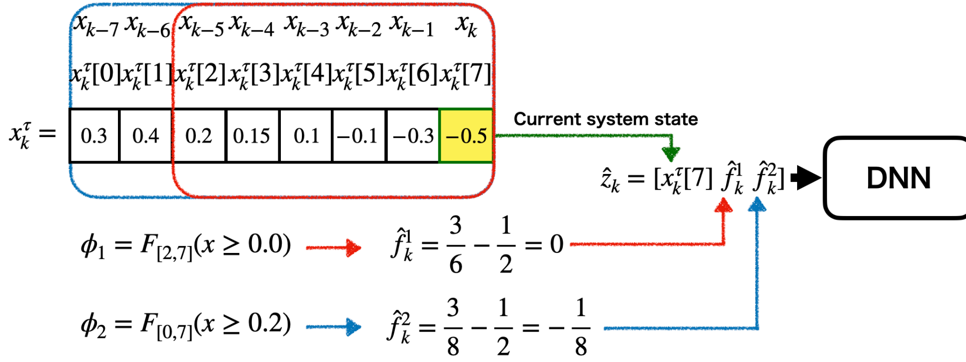


Fig. 5.7: Example of constructing a pre-processed state. For simplicity, we show the case without network delays $d = 0$. We consider the 1-dimensional system and the two STL sub-formulae: $\phi_1 = F_{[2,7]}(x \geq 0.0)$ and $\phi_2 = F_{[0,7]}(x \geq 0.2)$. For the sub-formulae ϕ_1 and ϕ_2, we compute the transformed flag values $\hat{f}_k^1$ and $\hat{f}_k^2$ using the extended state x_k^τ, which are regarded as continuous values in $[-0.5, 0.5]$. After that, we construct the pre-processed state using $x_k^\tau[\tau-1](= x_k)$, $\hat{f}_k^1$, and $\hat{f}_k^2$ and input it to DNNs.

The transformed flag values $\hat{f}^i$, $i = 1, 2, ..., M$ are actually used as inputs to DNNs to prevent positive biases of the flag values f^i, $i = 1, 2, ..., M$ and inputting $-\infty$ to DNNs. We compute the transformed flag value for each STL sub-formula and construct a flag state $\hat{f} = [\hat{f}^1\ \hat{f}^2\ ...\ \hat{f}^M]^\top$, which is called pre-processing. In this definition, it is assumed that $t_e^i = \tau - 1$, $\forall i \in \{1, 2, ..., M\}$. Then, we use the pre-processed state $\hat{z} = [x^\tau[\tau-1]^\top\ \hat{f}^\top]^\top$ instead of the extended state x^τ. Although $\hat{f}^i$, $i = 1, 2, ..., M$ are actually discrete values, we can regard the values as continuous values since τ is a large value.

Remark: It is important to ensure the Markov property of the pre-processed state $\hat{z}$ for the agent to learn its policy. If $t_e^i = \tau - 1$, $\forall i \in \{1, 2, ..., M\}$, then the pre-processed state $\hat{z}$ satisfies the Markov property. Let the current pre-processed state and the next pre-processed state be $\hat{z} = [x^\tau[\tau-1]^\top\ \hat{f}^\top]^\top$ and $\hat{z}' = [x^{\tau'}[\tau-1]^\top\ \hat{f}']^\top$, respectively. $x^{\tau'}[\tau-1]$ is generated by $p_f(\cdot|x^\tau[\tau-1], a)$, where a is the current control action. Therefore, $x^{\tau'}[\tau-1]$ depends on $x^\tau[\tau-1]$ and the current control action a. For each transformed flag value $\hat{f}^i$, $i \in \{1, 2, ..., M\}$, it is updated by

1. $\phi_i = G_{[t_s^i, \tau-1]}\varphi_i$

$$\hat{f}^{i'} = T^i(\hat{f}^i, x') = \begin{cases} \min\left\{\hat{f}^i + \frac{1}{\tau - t_s^i}, \frac{1}{2}\right\}, & x' \models \varphi_i, \\ -\frac{1}{2}, & x' \not\models \varphi_i, \end{cases} \tag{5.20}$$

Table 5.1: Dimensionality (Dim.) of extended state spaces.

	Without Pre-processing z	With Pre-processing $\hat{z}$ ($t_e^i = \tau - 1,\ \forall i \in \{1,2,...,M\}$)	With Pre-processing $\hat{z}$ ($t_e^{\max} - t_e^{\min} \geq 1$)
Dim.	$\tau n_x + dn_u$	$n_x + M + dn_u$	$(t_e^{\max} - t_e^{\min})n_x + M + dn_u$

2. $\phi_i = F_{[t_s^i, \tau-1]}\varphi_i$

$$\hat{f^i}' = T^i(\hat{f^i}, x') = \begin{cases} \frac{1}{2}, & x' \models \varphi_i, \\ \max\left\{\hat{f^i} - \frac{1}{\tau - t_s^i}, -\frac{1}{2}\right\}, & x' \not\models \varphi_i, \end{cases} \tag{5.21}$$

where $x' \sim p_f(\cdot|x^\tau[\tau-1], a)$. The transformed flag values are updated by the next system state x'. Therefore, the next transformed flag values $\hat{f^i}'$ depend on $\hat{f^i}$, $x^\tau[\tau-1]$, and the current action a.

On the other hand, in the case where $t_e^{\max} - t_e^{\min} \geq 1$, for an STL sub-formula ϕ_i with $t_e^i < t_e^{\max}$, the transformed flag value $\hat{f^i}$ is updated by

1. $\phi_i = G_{[t_s^i, t_e^i]}\varphi_i$

$$\hat{f^i}' = T^i(\hat{f^i}, x^\tau[t_e^i + 1]) = \begin{cases} \min\left\{\hat{f^i} + \frac{1}{t_e^i - t_s^i + 1}, \frac{1}{2}\right\}, & x^\tau[t_e^i + 1] \models \varphi_i, \\ -\frac{1}{2}, & x^\tau[t_e^i + 1] \not\models \varphi_i, \end{cases} \tag{5.22}$$

2. $\phi_i = F_{[t_s^i, t_e^i]}\varphi_i$

$$\hat{f^i}' = T^i(\hat{f^i}, x^\tau[t_e^i + 1]) = \begin{cases} \frac{1}{2}, & x^\tau[t_e^i + 1] \models \varphi_i, \\ \max\left\{\hat{f^i} - \frac{1}{t_e^i - t_s^i + 1}, -\frac{1}{2}\right\}, & x^\tau[t_e^i + 1] \not\models \varphi_i. \end{cases} \tag{5.23}$$

Then, we must include $x^\tau[\tau - t_e^{\max} + t_e^{\min}], ..., x^\tau[\tau - 1]$ to the pre-processed state $\hat{z}$ in order to ensure the Markov property, where $t_e^{\max} = \max_{i \in \{1,2,...,M\}} t_e^i$ $(= \tau - 1)$ and $t_e^{\min} = \min_{i \in \{1,2,...,M\}} t_e^i$. For example, as shown in **Fig. 5.8**, there may be some transformed flag values that are updated with information other than $[x^\tau[\tau-1]^\top\ \hat{f}]^\top$ and the current action a. Note that, in the case $t_e^{\max} = t_e^j + 1$ as shown in **Fig. 5.9**, the transformed flag value $\hat{f^j}$ is updated by $[x^\tau[\tau-1]^\top\ \hat{f}]^\top$, that is, the agent with DNNs can learn its policy using $[x^\tau[\tau-1]^\top\ \hat{f}]^\top$ when $t_e^{\max} = t_e^{\min} + 1$. As the difference $t_e^{\max} - t_e^{\min}$ increases, we need to include more previous system's states in the pre-processed state. For simplicity, in this chapter, we focus on the case where $t_e^i = \tau - 1$, $\forall i \in \{1,2,...,M\}$. Then, the pre-processing is most effective in terms of reducing the dimensionality of the extended state space as shown in **Table 5.1**.

Next, we consider the case with network delays $d \neq 0$. Then, we also need the

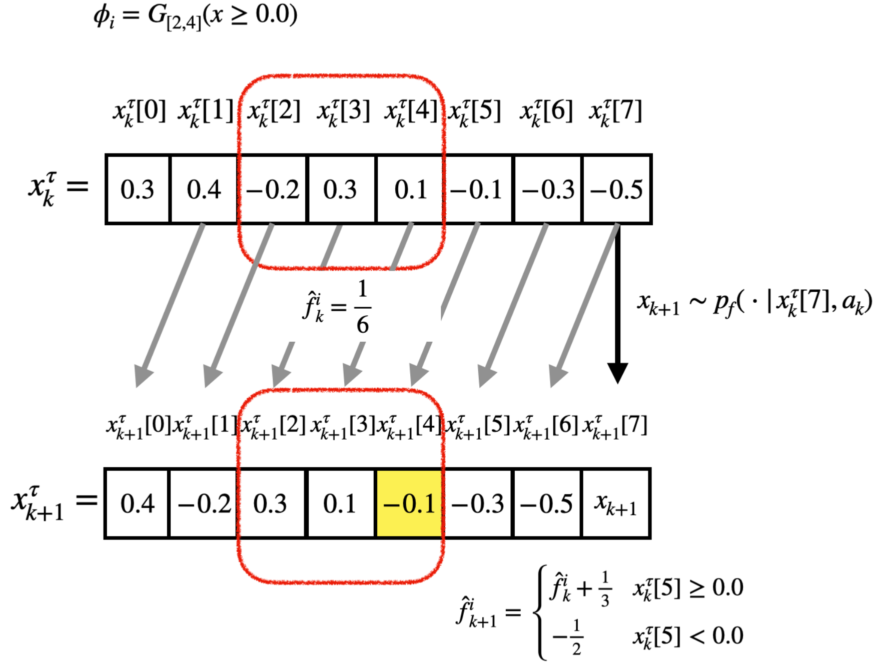


Fig. 5.8: Example of a sub-formula ϕ_i with $t_e^{\max} \geq t_e^i + 1$. We consider the 1-dimensional system and $t_e^{\max} = 7$ ($\tau = 8$). For the sub-formula $\phi_i = G_{[2,4]}(x \geq 0.0)$, $x_{k+1}^\tau[7](= x_{k+1})$ depends on $x_k^\tau[7](= x_k)$ and a_k. However, $\hat{f}_{k+1}^i$ depends on $\hat{f}_k^i$ and $x_k^\tau[5]$. If the pre-processed state is given by $[x_k^\tau[7]\ \hat{f}_k]^\top$, the agent with DNNs observes the environment partially. Then, the agent also needs $x_k^\tau[5]$ and $x_k^\tau[6]$ as parts of the pre-processed state.

previously determined control actions. We define the pre-processing considering the effect of network delays as follows:

Definition 5.4 (Pre-Processing with Network Delays)
For an extended state $z = [(x^\tau)^\top\ (a^d)^\top]^\top$, the flag values f^i, $i \in \{1, 2, ..., M\}$ of STL sub-formulae ϕ_i, $\in \{1, 2, ..., M\}$ are defined by (5.17) or (5.18). The flag values f^i, $i \in \{1, 2, ..., M\}$ are transformed into $\hat{f}^i$ by (5.19). Then, we construct a flag state $\hat{f} = [\hat{f}^1\ \hat{f}^2\ ...\ \hat{f}^M]^\top$. We use the pre-processed state $\hat{z} = [x^\tau[\tau-1]^\top\ \hat{f}^\top\ (a^d)^\top]^\top$ instead of the extended state z, where it is assumed that $t_e^i = \tau - 1$, $\forall i \in \{1, 2, ..., M\}$. The pre-processing is presented in **Algorithm 3**.

As shown in **Fig. 5.10**, in the case where there exist network delays, the agent needs the future state x_{k+d} and the future transformed flag values $\hat{f}_{k+d}^i$, $i \in \{1, 2, ..., M\}$ to determine a desired control action for satisfying the given STL formula Φ. If we know the mathematical models, x_{k+d} and $\hat{f}_{k+d}^i$, $i \in \{1, 2, ..., M\}$ can be predicted based on x_k, $\hat{f}_k^i$, $i \in \{1, 2, ..., M\}$, and $a_{k-d}, a_{k-d+1}, ..., a_{k-1}$. Thus, we give the agent not only the pre-processed state for the case without delays $[x_k^\tau[\tau-1]^\top\ \hat{f}_k^\top]^\top$ but also the previous control actions $a_{k-d}, a_{k-d+1}, ..., a_{k-1}$. Actually, although the agent cannot use

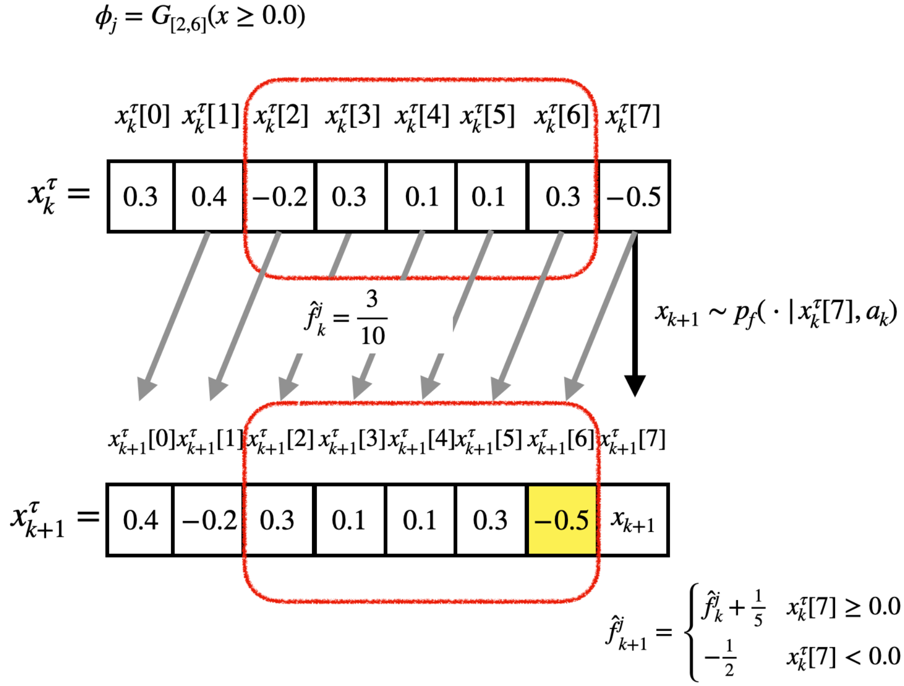


Fig. 5.9: Example of the sub-formula ϕ_j with $t_e^{\max} = t_e^j + 1$. We consider the 1-dimensional system and $t_e^{\max} = 7$ $(\tau = 8)$. For $\phi_j = G_{[2,6]}(x \geq 0.0)$, that is $t_e^{\max} - t_e^j = 1$, the transformed flag value can be updated by $[x_k^\tau[7]\ \hat{f}_k]^\top$ only.

the mathematical model since the model p_f is unknown, it learns its policy through interactions with the system using sufficient information $[x^\tau[\tau-1]^\top\ \hat{f}^\top\ (a^d)^\top]^\top$.

5.4.3 Proposed Algorithm

We propose a DRL-based algorithm to design a policy for satisfying an STL formula shown in **Algorithm 4**. From lines 1 to 3, we initialize the parameter vectors of DNNs and a replay buffer $\mathcal{D}$. In line 5, we initialize the state of the system $x_0 \sim p_0$. From lines 6 to 18, the agent interacts with the system and learns its policy for an episode. In line 7, at $t = k + d_{\mathrm{sc}}$, the agent receives the k-th state x_k. In line 8, the k-th extended state z_k is constructed using x_k, z_{k-1}, and a_{k-1}. In line 9, the k-th pre-processed state $\hat{z}_k$ is computed by **Algorithm 3**. In line 11, if $t > d_{\mathrm{sc}}$, the $(k-1)$-th reward r_{k-1} is computed by (5.16). In line 12, the agent stores the experience $(\hat{z}_{k-1}, a_{k-1}, \hat{z}_k, r_{k-1})$ to the replay buffer $\mathcal{D}$. In line 14, the agent determines the k-th exploration action a_k based on the k-th pre-processed state $\hat{z}_k$. In line 15, the agent sends a_k to the actuator. From lines 16 to 19, the agent updates the DNNs based on a DRL algorithm. In line 16, the agent samples I past experiences $\{(\hat{z}^{(i)}, a^{(i)}, \hat{z}'^{(i)}, r^{(i)})\}_{i=1}^{I}$ from the replay buffer $\mathcal{D}$ randomly. In line 17, the agent updates the parameter vectors of the DNNs based on the DRL algorithm such as TD3 [12] and SAC [13,14].

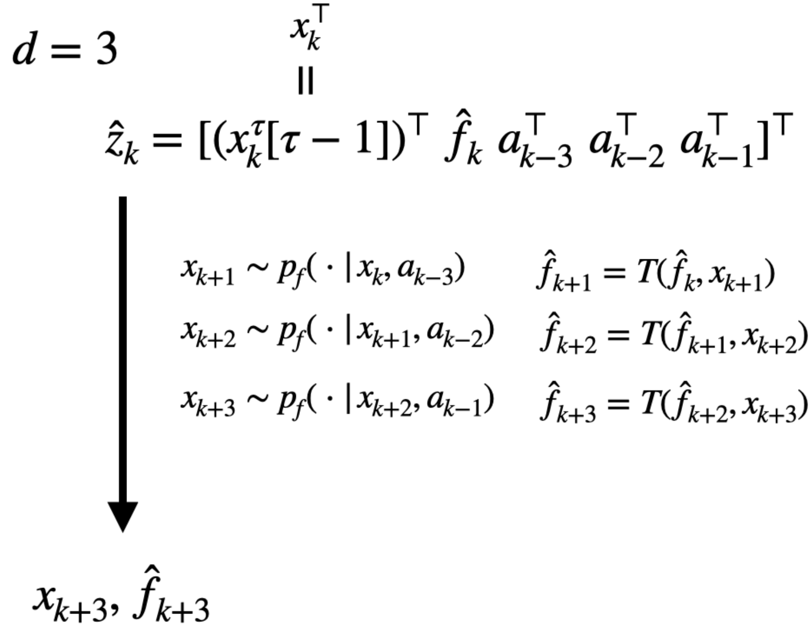


Fig. 5.10: Example of the prediction of a future pre-processed state x_{k+d} and $\hat{f}_{k+d}$ considering network delays. We show the case where $d = 3$ and the number of sub-formulae is $M = 1$. The agent needs x_{k+3} and $\hat{f}_{k+3}$ to determine a desire control action a_k for satisfying a given STL formula. The states can be predicted based on x_k, a_{k-3}, a_{k-2}, and a_{k-1} if the transition models are known. Although the agent cannot actually predict the states using the models, we give the agent sufficient information and make the agent learn its policy through interactions with the system.

5.5 Example

Consider a two-wheeled mobile robot as shown in **Fig. 5.11**. A discrete-time model of the robot is given by

$$\begin{bmatrix} x_{t+1}^{(0)} \\ x_{t+1}^{(1)} \\ x_{t+1}^{(2)} \end{bmatrix} = \begin{bmatrix} x_t^{(0)} + \Delta u_t^{(0)} \cos(x_t^{(2)}) \\ x_t^{(1)} + \Delta u_t^{(0)} \sin(x_t^{(2)}) \\ x_t^{(2)} + \Delta u_t^{(1)} \end{bmatrix} + \Delta_w \begin{bmatrix} w_t^{(0)} \\ w_t^{(1)} \\ w_t^{(2)} \end{bmatrix}, \tag{5.24}$$

where $x_t = [x_t^{(0)} \ x_t^{(1)} \ x_t^{(2)}]^\top \in \mathbb{R}^3$, $u_t = [u_t^{(0)} \ u_t^{(1)}]^\top \in \mathbb{R}^2$, and $w_t = [w_t^{(0)} \ w_t^{(1)} \ w_t^{(2)}]^\top \in \mathbb{R}^3$. $w_t^{(i)}$, $i \in \{0, 1, 2\}$ are sampled from a standard normal distribution $\mathcal{N}(0, 1)$. We assume that $\Delta = 0.1$ and $\Delta_w = 0.01 I_3$, where I_3 is the 3×3 identity matrix. The initial state of the system is sampled randomly in the region $0 \le x^{(0)} \le 2.5,\ 0 \le x^{(1)} \le 2.5,\ -\pi/2 \le x^{(2)} \le \pi/2$. The region 1 is $\{(x^{(0)}, x^{(1)}) |\ 3.75 \le x^{(0)} \le 5,\ 3.75 \le x^{(1)} \le 5\}$ and the region 2 is $\{(x^{(0)}, x^{(1)}) |\ 3.75 \le x^{(0)} \le 5,\ 1.25 \le x^{(1)} \le 2.5\}$. We consider the following temporal

Algorithm 3 Pre-processing of the extended state z

1: **Input:** The extended state $z = [(x^{\tau})^{\top} \ (a^d)^{\top}]^{\top}$ and the STL sub-formulae $\{\phi_i\}_{i=1}^{M}$.
2: **for** $i = 1, ..., M$ **do**
3: **if** $\phi_i = G_{[t_s^i, \tau-1]}\varphi_i$ **then**
4: Compute the flag value f^i by (5.17).
5: **end if**
6: **if** $\phi_i = F_{[t_s^i, \tau-1]}\varphi_i$ **then**
7: Compute the flag value f^i by (5.18).
8: **end if**
9: **end for**
10: Set the flag state $\hat{f} = [\hat{f}^1 \ \hat{f}^2 \ ... \ \hat{f}^M]$ by (5.19).
11: **Output:** The pre-processed state $\hat{z} = [x^{\tau}[\tau-1]^{\top} \ \hat{f}^{\top} \ (a^d)^{\top}]^{\top}$.

control task.
Recurrence: *At any time in the time interval* $[0, 900]$, *the robot visits both the region 1 and the region 2 before* 99 *time steps are elapsed, where there is no constraint for the order of the visits.*

We describe the recurrence task as the following STL formula.

$$\Phi = G_{[0,900]}(F_{[0,99]}\varphi_1 \wedge F_{[0,99]}\varphi_2), \tag{5.25}$$

where

$$\varphi_1 = ((3.75 \le x^{(0)} \le 5) \wedge (3.75 \le x^{(1)} \le 5)),$$
$$\varphi_2 = ((3.75 \le x^{(0)} \le 5) \wedge (1.25 \le x^{(1)} \le 2.5)).$$

The length of x^{τ} is $\tau = 100$. We conduct simulations using the TD3 algorithm [12] and the SAC algorithm [13, 14].

TD3-based learning: The actor and critic DNNs have two hidden layers, all of which have 256 units, and all layers are fully connected. The activation functions for the hidden layers and the outputs of the actor DNN are the ReLU functions and hyperbolic tangent functions, respectively. The size of the replay buffer $\mathcal{D}$ is 1.0×10^5 and the size of the mini-batch is $I = 64$. We use Adam [103] as the optimizers for all main DNNs. The learning rates of all optimizers multiplier are 3.0×10^{-4}. The soft update rate of the target network is $\xi = 0.01$. The discount factor is $\gamma = 0.99$. We use the following Ornstein-Uhlenbeck process for generating exploration noises.

$$\epsilon_{k+1} = \epsilon_t - p_1(\epsilon_k - p_2) + p_3\epsilon_{\mathcal{N}},$$

where $\epsilon_{\mathcal{N}}$ is a noise generated by a standard normal distribution $\mathcal{N}(0, 1)$. We set the parameters $(p_1, p_2, p_3) = (0.15, 0, 0.3)$. The target policy smoothing regularization is

Algorithm 4 DRL-based policy design for satisfying an STL formula taking network delays in consideration

1: Initialize the parameter vectors of main DNNs.
2: Initialize the parameter vectors of target DNNs.
3: Initialize a replay buffer $\mathcal{D}$.
4: **for** Episode = $1, ...,$ MAX EPISODE **do**
5: Initialize the system state $x_0 \sim p_0$.
6: **for** $k = 0, ..., T$ **do**
7: Receive the k-th observed state x_k.
8: Construct the extended state z_k.
9: Compute the next pre-processed state $\hat{z}_k$ by **Algorithm 1**.
10: **if** $k > 0$ **then**
11: Compute the reward $r_{k-1} = R_{STL}(z_{k-1})$.
12: Store the experience

$$(\hat{z}_{k-1}, a_{k-1}, \hat{z}_k, r_{k-1})$$

in the replay buffer $\mathcal{D}$.
13: **end if**
14: Determine the action a_k based on the pre-processed state $\hat{z}_k$.
15: Send the k-th control action a_k to the actuator.
16: Sample I experiences

$$\{(\hat{z}^{(i)}, a^{(i)}, \hat{z}'^{(i)}, r^{(i)})\}_{i=1,...,I}$$

from the replay buffer $\mathcal{D}$ randomly.
17: Update the DNNs (and the entropy temperature) based on the DRL algorithm.
18: **end for**
19: **end for**

implemented by adding noises sampled from the normal distribution $\mathcal{N}(0, 0.2)$ to the action determined by the target actor DNN, which is clipped to $[-0.5, 0.5]$. The agent updates the actor DNN and the target DNNs every 2 learning steps.

SAC-based learning: The actor and critic DNNs have two hidden layers, all of which have 256 units, and all layers are fully connected. The activation functions for the hidden layers and the outputs of the actor DNN are the ReLU functions and hyperbolic tangent functions, respectively. The size of the replay buffer $\mathcal{D}$ is 1.0×10^5 and the size of the mini-batch is $I = 64$. We use Adam as the optimizers for all main DNNs and the entropy temperature. The learning rates of all optimizers are 3.0×10^{-4}. The soft update rate of the target network is $\xi = 0.01$. The discount factor

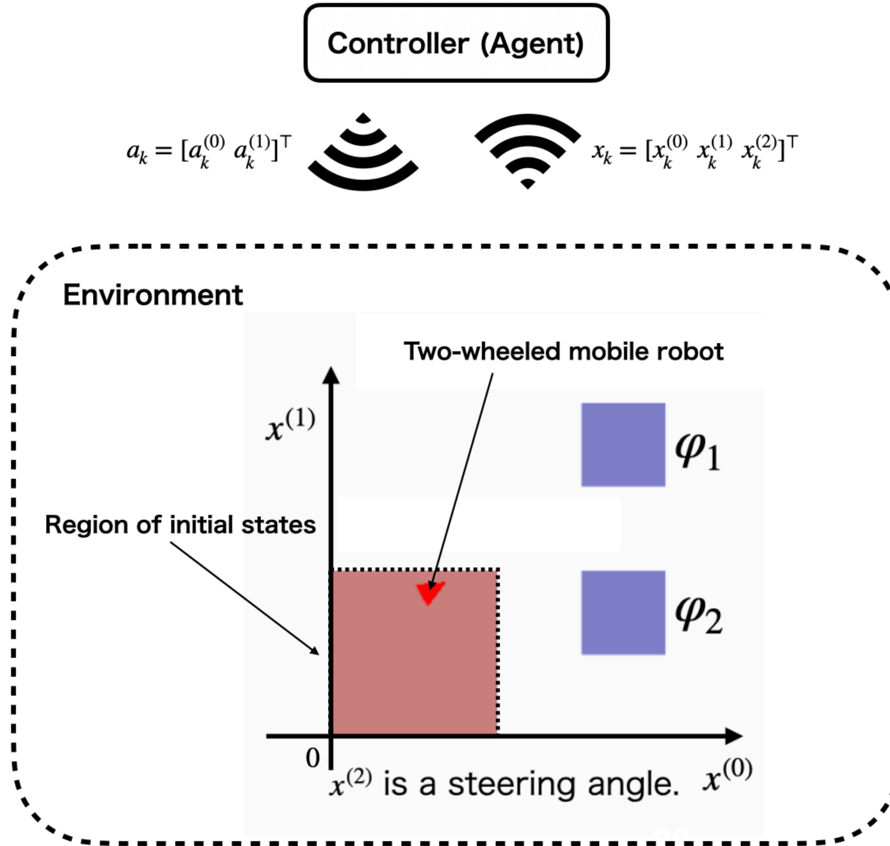


Fig. 5.11: The agent learns the policy for the two-wheeled mobile robot to satisfy the given STL formula considering the effect of network delays. (Copyright © 2022 IEEE)

is $\gamma = 0.99$. The target for updating the entropy temperature $\mathcal{H}_0$ is -2. The initial entropy temperature is 1.0.

The STL-reward parameter is $\beta = 100$. The agent learns its policy for 6.0×10^5 steps. We normalize $x^{(0)}$ and $x^{(1)}$ as $x^{(0)} - 2.5$ and $x^{(1)} - 2.5$ before inputting to DNNs, respectively. In order to evaluate learned policies, we introduce the following two indices:

- a **learning curve** shows the mean of returns $\sum_{k=0}^{T} \gamma^k R_z(z_k)$ for 100 trajectories, and
- a **success rate** shows the number of trajectories satisfying a given STL formula for 100 trajectories.

We prepare 100 initial states sampled from p_0 and generate 100 trajectories using the learned policy for each evaluation. All simulations run on a computer with AMD Ryzen 9 3950X 16-core processor, NVIDIA (R) GeForce RTX 2070 super, and 32GB of

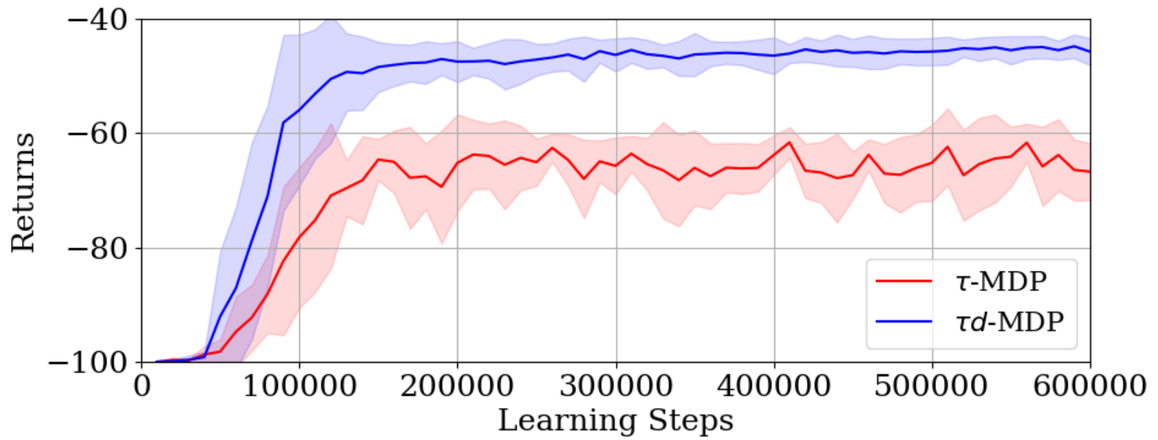
memory and are conducted using the Python software.

5.5.1 Result

We demonstrate the effect of using previously determined control actions as a part of an extended state, where we use pre-processing introduced in **Section 5.4.2**. It is assumed that $d_{\mathrm{sc}} = 3$ and $d_{\mathrm{ca}} = 4$ ($u_t = a_{t-7}$), where these values are unknown, but we know that $d_{\mathrm{sc}} \leq 5$ and $d_{\mathrm{ca}} \leq 5$ beforehand. Then, the length of the past control action sequence a^d is $d = 10$.

The learning curves and the success rates for the τ-MDP case (without previously determined control actions) and the τd-MDP case (with previously determined control actions) are shown in **Figs. 5.12** and **5.13**, respectively. If we do not use previously determined control actions, the success rate of the learned policy is not increasing as shown in **Fig. 5.13** for both the TD3-based algorithm and the SAC-based algorithm. On the other hand, if we utilize previously determined control actions, the agent can learn the policy such that obtained returns and success rate are higher than the policy learned without previously determined control actions. The result concludes that the agent should use not only previous system's states but also previously determined control actions to learn its policy considering the effect of network delays.

Additionally, we compare the TD3-based algorithm and the SAC-based algorithm. The results in **Figs. 5.12** and **5.13** show that the TD3-based algorithm is better than the SAC-based algorithm regarding both the learning curves and the success rates. This is because, in the SAC-based algorithm, the agent learns its policy based on not only sum of rewards but also the entropy. On the other hand, in the SAC-based algorithm, we need not choose a model of exploration noises and the agent can learn how to generate exploration noises. If it is difficult to choose how to generate exploration noises beforehand, the SAC-based algorithm may be more helpful than the TD3-based algorithm. Furthermore, it is known that a maximum entropy algorithm such as the SAC algorithm improves explorations by acquiring diverse behaviors and has the robustness for estimation errors.

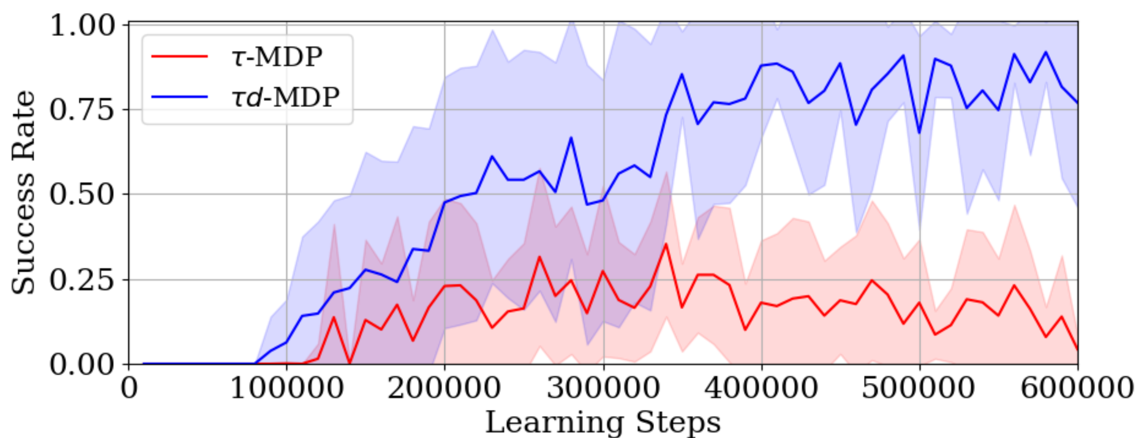


(a) TD3

(b) SAC

Fig. 5.12: Learning curves for the τ-MDP case and the τd-MDP case. The solid curves and the shades represent the average results and the standard deviations over 10 trials with different random seeds, respectively.

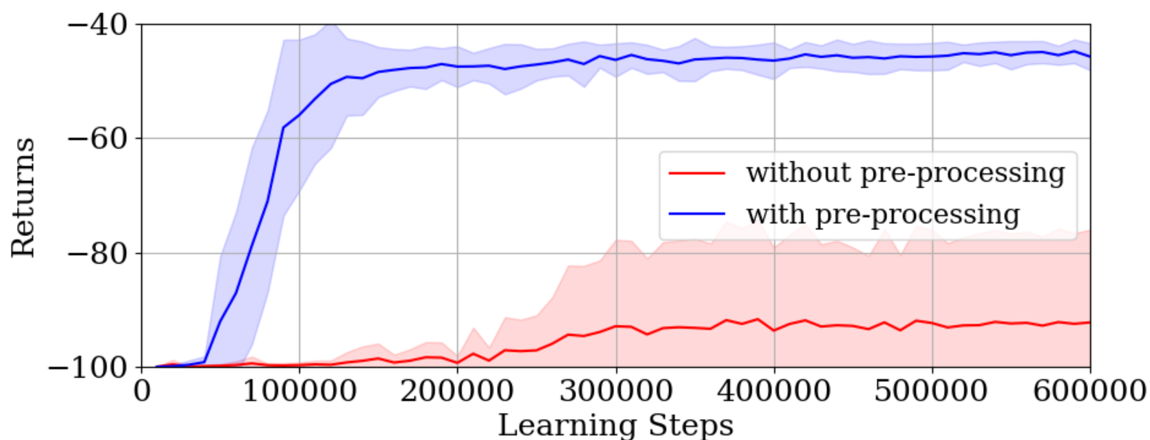
(a) TD3



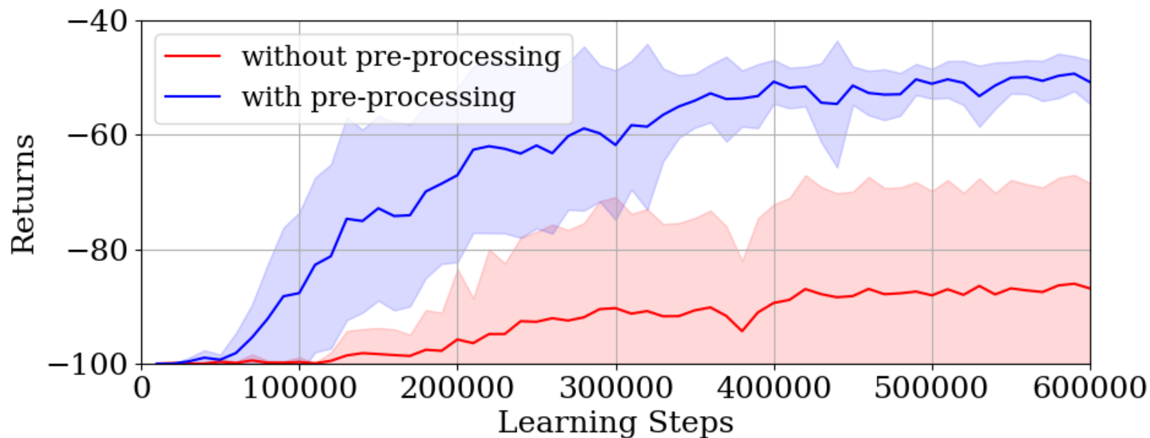
(b) SAC

Fig. 5.13: Success rates for the τ-MDP case and the τd-MDP case. The solid curves and the shades represent the average results and the standard deviations over 10 trials with different random seeds, respectively.

5.5.2 Ablation Study for Pre-Processing



(a) TD3



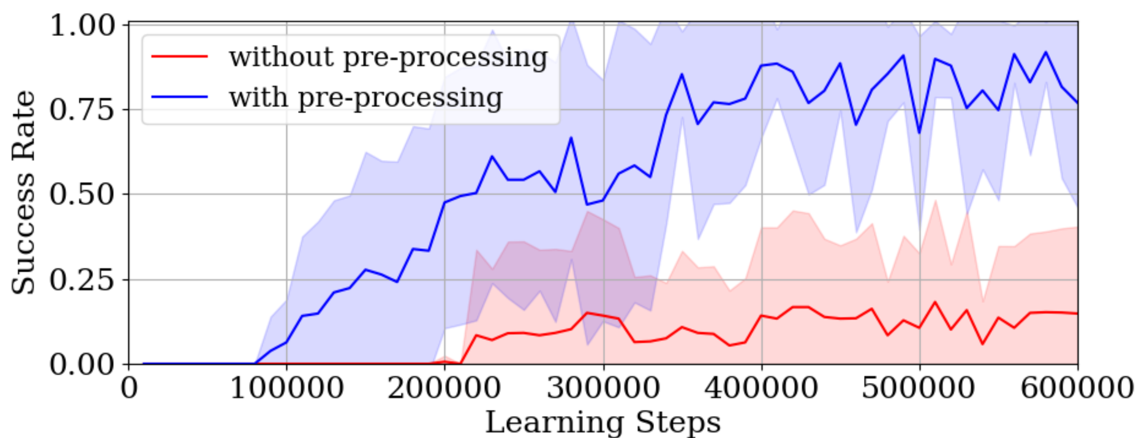
(b) SAC

Fig. 5.14: Learning curves for the cases with and without pre-processing. The solid curves and the shades represent the average results and the standard deviations over 10 trials with different random seeds, respectively.

We show the improvement in the learning performance by pre-processing. In the case without pre-processing, the dimensionality of the extended state is 320 and, in the case with pre-processing, the dimensionality of the extended state is 25. As shown in **Fig. 5.14**, the agent cannot improve the performance of its policy without pre-processing. Then, the learned policy has a low success rate as shown in **Fig. 5.15**. Conversely, in the case with pre-processing, the learned policy has a high success rate. The result concludes that pre-processing is necessary in our proposed method

for satisfying the STL formula Φ with a large τ.

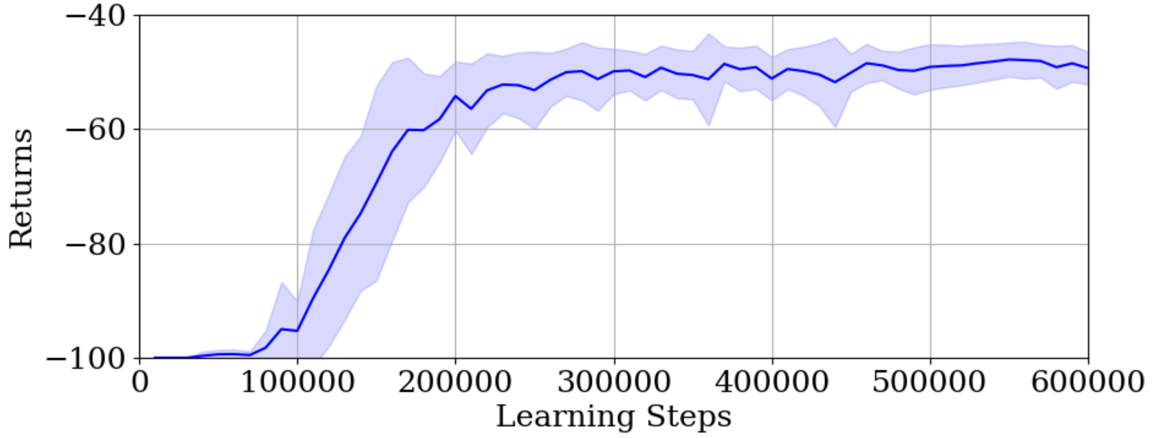
(a) TD3



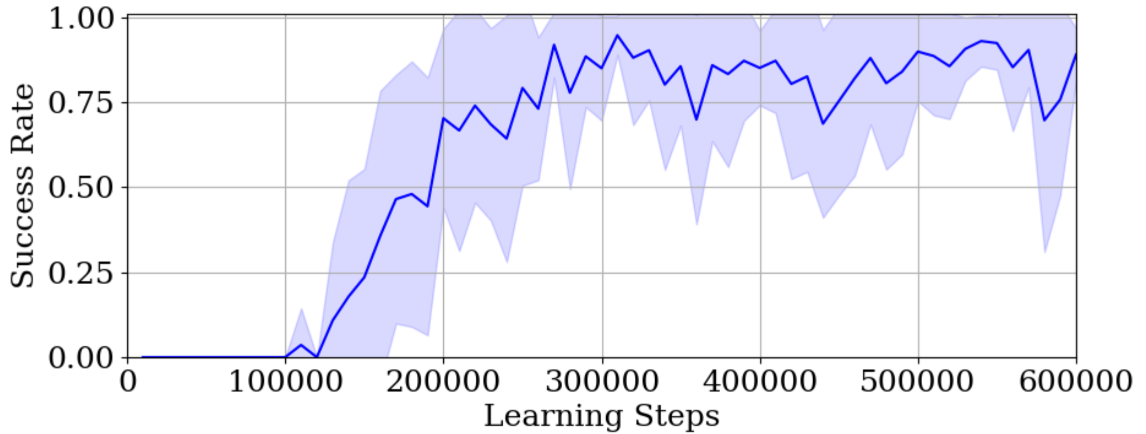
(b) SAC

Fig. 5.15: Success rates for the cases with and without pre-processing. The solid curves and the shades represent the average results and the standard deviations over 10 trials with different random seeds, respectively.

5.6 Application to a Problem with Random Delays



(a) Learning curve for the environment with random delays. The solid curve and the shade represent the average results and the standard deviations over 10 trials with different random seeds, respectively.



(b) Success rate for the environment with random delays. The solid curve and the shade represent the average results and the standard deviations over 10 trials with different random seeds, respectively.

Fig. 5.16: Results for the problem with random delays.

In this section, we apply our proposed method to a problem with random delays. Except for network delay setting, we consider the same problem setting in **Section 5.5**. It is assumed that d_{sc} and d_{ca} are random values, where these values are natural numbers and bounded by $d_{sc}^{max} = 5$ and $d_{ca}^{max} = 5$, respectively. On the other hand, in our proposed method, we need previous state's sequence to evaluate satisfying STL sub-formulae. In the case where d_{sc} is a random value, we may not obtain the previous state's sequence in order. Thus, we assume that d_{sc} always is d_{sc}^{max}. d_{ca}

is an independent and identically distributed random variable with the following probability function:

$$P(d_{\mathrm{ca}} = 1) = \frac{1}{4},\ P(d_{\mathrm{ca}} = 2) = \frac{1}{2},\ P(d_{\mathrm{ca}} = 3) = \frac{1}{8},\ P(d_{\mathrm{ca}} = 4) = \frac{1}{12},\ P(d_{\mathrm{ca}} = 5) = \frac{1}{24}.$$

If the actuator receives a_k before receiving a_l ($l < k$), it does not input the control action a_l to the system as a control input. If the actuator does not receive a new control action at time t, it keeps the latest received control action as a control input u_t.

The length of the previous control action sequence is $d = 10$. In the example, we apply the SAC-based algorithm, where DNN architectures and hyper parameters are same as **Section 5.5**. The learning curve and the success rate are shown in **Figs. 5.16(a)** and **(b)**, respectively. Our proposed method is also useful to design a policy for satisfying a given STL formula Φ with random network delays.

Chapter 6

DRL under STL Constraints Using Lagrangian Relaxation

DRL has attracted much attention as an approach to solve optimal control problems without mathematical models of dynamical systems. On the other hand, in general, constraints may be imposed on optimal control problems. In this chapter, we consider the optimal control problems with constraints to complete temporal control tasks. The constraints are described by STL formulae. To deal with the STL constraints, we introduce an extended CMDP, which is called a τ-CMDP. We formulate the STL-constrained optimal control problem as the τ-CMDP and propose a two-phase constrained DRL algorithm with the Lagrangian relaxation.

This chapter is based on "Deep reinforcement learning under signal temporal logic constraints using Lagrangian relaxation" [56] which appeared in *IEEE Access*.

6.1 Problem Formulation

We consider the following discrete-time stochastic dynamical system.

$$x_{k+1} = f(x_k, a_k) + \Delta_w w_k, \tag{6.1}$$

where $x_k \in \mathcal{X}$, $a_k \in \mathcal{A}$, and $w_k \in \mathcal{W}$ are the system's state, the control action, and the system noise at $k \in \{0, 1, ...\}$. $\mathcal{X} = \mathbb{R}^{n_x}$, $\mathcal{A} \subseteq \mathbb{R}^{n_a}$, and $\mathcal{W} = \mathbb{R}^{n_x}$ are the system's state space, the control action space, and the system noise space, respectively. The system noise w_k is an independent and identically distributed random variable with a probability density $p_w : \mathcal{W} \to \mathbb{R}_{\geq 0}$. Δ_w is a regular matrix that is a weighting factor of the system noise. $f : \mathcal{X} \times \mathcal{A} \to \mathcal{X}$ is a function that describes the system dynamics. Then, we have the transition probability density

$$p_f(x'|x, a) := |\Delta_w^{-1}| p_w(\Delta_w^{-1}(x' - f(x, a))).$$

The initial state $x_0 \in \mathcal{X}$ is sampled from a probability density $p_0 : \mathcal{X} \to \mathbb{R}_{\geq 0}$. For a finite system's trajectory whose length is $K + 1$, $x_{k_1:k_2}$ denotes the partial trajectory for

the time interval $[k_1, k_2]$, where $k_1, k_2 \in \{0, 1, ...\}$ and $0 \leq k_1 \leq k_2 \leq K$.

In this chapter, we consider an optimal control problem with a constraint to complete a given temporal control task. The constraint is described by an STL formula Φ, where we consider the syntax (5.1) and the semantics (5.2) and (5.3) in **Chapter 5**. We formulate the optimal control problem as follows:

$$\begin{aligned} &\text{maximize}_{\pi} \ E_{p_0, p_f, \pi}\left[\sum_{k=0}^{K} \gamma^k R(x_k, a_k)\right], \\ &\text{subject to} \quad x_{0:K} \models \Phi, \end{aligned} \tag{6.2}$$

where $\gamma \in [0, 1)$ is a discount factor and $R : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$ is a reward function for a given control performance index. $E_{p_0, p_f, \pi}[\cdot]$ is the expected value with respect to the initial state probability density $x_0 \sim p_0$, the transition probability density $x_{k+1} \sim p_f(\cdot|x_k, a_k)$, $k \geq 0$, and the policy $a_k \sim \pi(\cdot|x_k)$, $k \geq 0$. In this chapter, it is assumed that the mathematical model of the system is unknown. Thus, we apply a DRL algorithm to design of the STL-constrained optimal policy, where we regard the controller as the agent. The agent needs not only a current system's state but also a previous system's behavior to determine a desire control action at each step for satisfying the STL formula Φ. Thus, a finite sequence of previous system's states is regarded as an environment's state like **Chapter 5**. In order to design the STL-constrained optimal policy using a DRL algorithm, we introduce a τ-*CMDP* derived from a τ-MDP [49] and a CMDP [55].

6.2 Constrained Markov Decision Process (CMDP)

A CMDP is a standard formulation for a sequential decision making problem with some constraints [55]. It is defined by a tuple $\mathcal{CM} = \left\langle \mathcal{X}, \mathcal{A}, p_0, p_f, R, \{C_i\}_{i=1}^{I} \right\rangle$, where

- $\mathcal{X} \in \mathbb{R}^{n_x}$ is an environment's state space.
- $\mathcal{A} \in \mathbb{R}^{n_a}$ is an agent's action space.
- $p_0 : \mathcal{X} \to \mathbb{R}_{\geq 0}$ is a probability density for an initial state.
- $p_f(\cdot|x, a) : \mathcal{X} \to \mathbb{R}_{\geq 0}$ is a probability density for a transition under a state $x \in \mathcal{X}$ and an action $a \in \mathcal{A}$.
- $R : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$ is a reward function.
- $\{C_i\}_{i=1}^{I}$ is a set of cost functions for constraints. $C_i : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$ is the i-th a cost function.

For the CMDP $\mathcal{CM}$, we consider the following problem.

$$\max_{\pi} E_{p_0,p_f,\pi}\left[\sum_{k=0}^{\infty}\gamma^k R(x_k,a_k)\right],$$
$$\text{subject to} \quad E_{p_0,p_f,\pi}\left[\sum_{k=0}^{\infty}\gamma^k C_i(x_k,a_k)\right] \le l_i,\ i\in\{1,2,...,I\},$$

where $l_i,\ i\in\{1,2,...,I\}$ are given thresholds.

6.3 τ-Constrained Markov Decision Process (τ-CMDP)

To solve an optimal control problem with a constraint to satisfying an STL formula Φ, we introduce the following extended CMDP.

Definition 6.1 (τ-CMDP)
We consider an STL formula $\Phi = G_{[0,K_e]}\phi$ (or $\Phi = F_{[0,K_e]}\phi$) as a constraint for the system (6.1), where $\mathrm{hrz}(\Phi) = K$ and ϕ consists of multiple STL sub-formulae $\phi_i,\ i\in\{1,2,...,M\}$. Subsequently, we set $\tau = \mathrm{hrz}(\phi)+1$, that is, $K = K_e+\tau-1$.
To solve the optimal control problem with the STL constraint, a τ-CMDP is defined by a tuple $\mathcal{CM}_\tau = \left\langle \mathcal{Z},\mathcal{A},p_0^z,p^z,R_{STL},R_z\right\rangle$, where

- $\mathcal{Z}$ is an extended state space which we regard as an environment's state space for RL. The extended state $z\in\mathcal{Z}$ is a vector of multiple system's states $z = [z[0]^\top\ z[1]^\top\ ...\ z[\tau-1]^\top]^\top,\ z[i]\in\mathcal{X},\ \forall i\in\{0,1,...,\tau-1\}$.
- $\mathcal{A}$ is an agent's control action space.
- p_0^z is a probability density for the initial extended state z_0 with $z_0[i]=x_0,\ \forall i\in\{0,1,...,\tau-1\}$, where x_0 is generated from p_0.
- p^z is a transition probability density for an extended state. When the system's state is updated by $x'\sim p_f(\cdot|x,a)$, the extended state is updated by $z'\sim p^z(\cdot|z,a)$ as follows:

$$z'[i] = z[i+1],\ \forall i\in\{0,1,..,\tau-2\},$$
$$z'[\tau-1]\sim p_f(\cdot|z[\tau-1],a).$$

- $R_{STL}:\mathcal{Z}\to\mathbb{R}$ is an STL-reward function[*1] defined by (5.15) for satisfying the given STL formula Φ.
- $R_z:\mathcal{Z}\times\mathcal{A}\to\mathbb{R}$ is a reward function as follows:

$$R_z(z,a) = R(z[\tau-1],a),$$

[*1] In the standard CMDP formulation, the functions for constraints are defined as cost functions. On the other hand, in this chapter, we define the function for an STL constraint as a reward function different from the main reward function R_z.

where $R : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$ is a reward function for a given control performance index.

We design an optimal policy with respect to R_z under satisfying the STL formula Φ using a model-free constrained DRL algorithm [86]. Then, we define the following functions.

$$J(\pi) = E_{p_0,p_f,\pi}\left[\sum_{k=0}^{K} \gamma^k R_z(z_k, a_k)\right],$$
$$J_{STL}(\pi) = E_{p_0,p_f,\pi}\left[\sum_{k=0}^{K} \gamma^k R_{STL}(z_k)\right],$$

where $\gamma \in [0, 1)$ is a discount factor close to 1. To apply a constrained DRL algorithm, we reformulate the problem (6.2) as follows:

$$\pi^* \in \arg\max_{\pi \in \Pi_{STL}} J(\pi), \tag{6.3}$$
$$\Pi_{STL} = \{\pi \mid J_{STL}(\pi) \geq l_{STL}\}, \tag{6.4}$$

where $l_{STL} \in \mathbb{R}$ is a lower threshold. In this chapter, l_{STL} is a hyper-parameter for adjusting the satisfiability of the given STL formula Φ. The larger l_{STL} is, the more conservatively the agent learns a policy to satisfy the STL formula Φ. We call the constrained optimal control problem with (6.3) and (6.4) a *τ-CMDP problem*. In the next section, we propose a constrained DRL algorithm with the Lagrangian relaxation to solve the τ-CMDP problem.

6.4 Deep Reinforcement Learning under an STL Constraint

We propose a constrained DRL algorithm with the Lagrangian relaxation to obtain an optimal policy for the τ-CMDP problem. Our proposed algorithm is based on the DDPG algorithm [11] or the SAC algorithm [13, 14], which are DRL algorithms derived from the Q-learning algorithm for problems with continuous state-action spaces. In both algorithms, we parameterize an agent's policy π using a DNN, which is called an actor DNN. The agent updates the parameter vector of the actor DNN based on $J(\pi)$. However, the agent cannot directly use $J(\pi)$ since the mathematical model of the system p_f is unknown. Thus, we approximate $J(\pi)$ using another DNN, which is called a critic DNN. Additionally, we use the experience replay and the target network technique stated in **Section. 2.3** in order to reduce correlations among experience data and improve the learning stability.

On the other hand, we cannot directly apply the DDPG algorithm and the SAC algorithm to the τ-CMDP problem since these are algorithms for unconstrained

problems. Thus, we consider the following Lagrangian relaxation [108].

$$\min_{\kappa \geq 0} \max_{\pi} \mathcal{L}(\pi, \kappa), \tag{6.5}$$

where $\mathcal{L}(\pi, \kappa)$ is a *Lagrangian function* given by

$$\mathcal{L}(\pi, \kappa) = J(\pi) + \kappa(J_{STL}(\pi) - l_{STL}), \tag{6.6}$$

and $\kappa \geq 0$ is a *Lagrange multiplier*. We can relax the constrained problem into the unconstrained problem. To solve the unconstrained minimax problem, we use the iterative primal-dual approach where, in each learning step, we update the policy π and the Lagrangian multiplier κ in turn.

6.4.1 DDPG-Lagrangian

We parameterize a deterministic policy μ using a DNN as shown in **Fig. 6.1**, which is an actor DNN. Its parameter vector is denoted by θ_μ. In the DDPG-Lagrangian algorithm, the parameter vector θ_μ is updated by maximizing (6.6). However, $J(\mu_{\theta_\mu})$ and $J_{STL}(\mu_{\theta_\mu})$ are unknown. Thus, as shown in **Fig. 6.2**, $J(\mu_{\theta_\mu})$ and $J_{STL}(\mu_{\theta_\mu})$ are approximated by two separate critic DNNs, which are a *reward critic DNN* and an *STL-reward critic DNN*, respectively. The parameter vectors of the reward critic DNN and the STL-reward critic DNN are denoted by θ_r and θ_s, respectively. The parameter vectors are updated by decreasing the following critic loss functions.

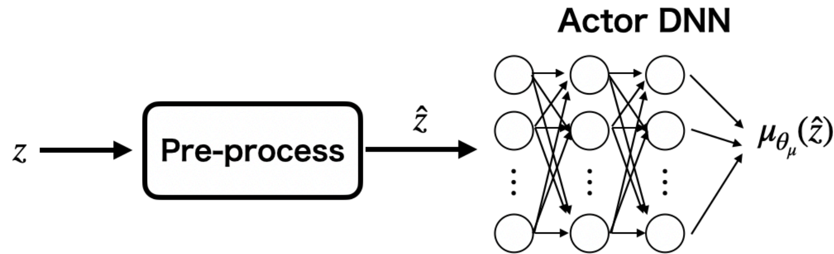


Fig. 6.1: Illustration of an actor DNN for the DDPG Lagrangian algorithm. Actually, we input a pre-processed state $\hat{z}$ stated in **Section 6.4.4** to the DNN instead of an extended state z.

$$J_{rc}(\theta_r) = E_{(z,a,z')\sim\mathcal{D}}\left[(Q_{\theta_r}(z,a) - t_r)^2\right], \tag{6.7}$$

$$J_{sc}(\theta_s) = E_{(z,a,z')\sim\mathcal{D}}\left[(Q_{\theta_s}(z,a) - t_s)^2\right], \tag{6.8}$$

where $Q_{\theta_r}(\cdot,\cdot)$ and $Q_{\theta_s}(\cdot,\cdot)$ are the outputs of the reward critic DNN and the STL-reward critic DNN, respectively. $E_{(z,a,z')\sim\mathcal{D}}[\cdot]$ is the expected value under the random

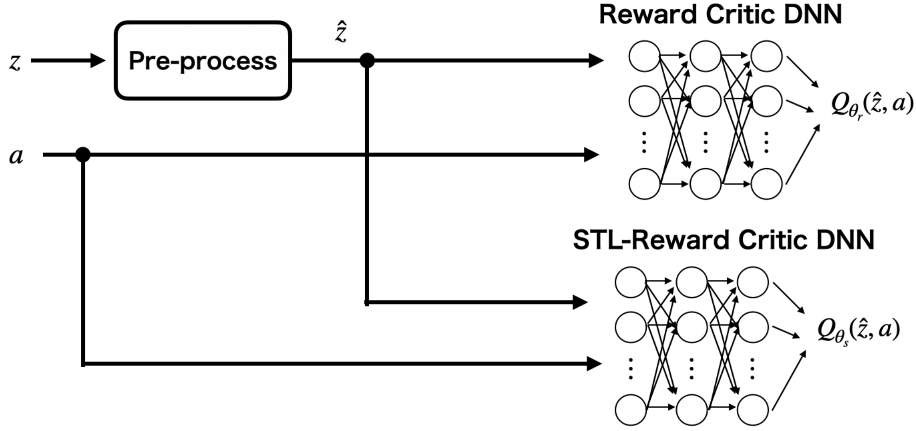


Fig. 6.2: Illustration of the two-type critic DNNs (the reward critic DNN and the STL-reward critic DNN). In the DDPG-Lagrangian algorithm, the reward critic DNN and the STL-reward critic DNN estimate the terms $J(\mu_{\theta_\mu})$ and $J_{STL}(\mu_{\theta_\mu})$ in (6.6), respectively. In the SAC-Lagrangian algorithm, the reward critic DNN and the STL-reward critic DNN estimate the terms $J_{ent}(\pi_{\theta_\pi})$ and $J_{STL}(\pi_{\theta_\pi})$ in (6.13), respectively. Actually, we input a pre-processed state $\hat{z}$ stated in **Section 6.4.4** to the DNN instead of an extended state z.

sampling of the experiences (z, a, z') from $\mathcal{D}$. The target values t_r and t_s are given by

$$\begin{aligned} t_r &= R_z(z, a) + \gamma Q_{\theta_r^-}(z', \mu_{\theta_\mu^-}(z')), \\ t_s &= R_{STL}(z) + \gamma Q_{\theta_s^-}(z', \mu_{\theta_\mu^-}(z')). \end{aligned}$$

$Q_{\theta_r^-}(\cdot, \cdot)$ and $Q_{\theta_s^-}(\cdot, \cdot)$ are the outputs of the target reward critic DNN and the target STL-reward critic DNN, respectively, and $\mu_{\theta_\mu^-}(\cdot)$ is the output of target actor DNN. θ_r^-, θ_s^-, and θ_μ^- are parameter vectors of the target reward critic DNN, the target STL-reward critic DNN, and the target actor DNN, respectively. Their parameter vectors are slowly updated by the following soft update.

$$\begin{aligned} \theta_r^- &\leftarrow \xi\theta_r + (1 - \xi)\theta_r^-, \\ \theta_s^- &\leftarrow \xi\theta_s + (1 - \xi)\theta_s^-, \\ \theta_\mu^- &\leftarrow \xi\theta_\mu + (1 - \xi)\theta_\mu^-, \end{aligned} \tag{6.9}$$

where $\xi > 0$ is a sufficiently small positive constant. In the standard DDPG algorithm [11], the parameter vector of the actor DNN is updated by decreasing

$$J_a(\theta_\mu) = E_{z\sim\mathcal{D}}[-Q_{\theta_r}(z, \mu_{\theta_\mu}(z))],$$

where $E_{z\sim\mathcal{D}}[\cdot]$ is the expected value with respect to z sampled from $\mathcal{D}$ randomly. However, in the DDPG-Lagrangian algorithm, we consider (6.6) as an objective

instead of $J(\mu_{\theta_\mu})$. Thus, the parameter vector of the actor DNN θ_μ is updated by decreasing the following actor loss function.

$$J_a(\theta_\mu) = E_{z\sim\mathcal{D}}[-(Q_{\theta_r}(z, \mu_{\theta_\mu}(z)) + \kappa Q_{\theta_s}(z, \mu_{\theta_\mu}(z)))]. \tag{6.10}$$

The Lagrange multiplier κ is updated by decreasing the following loss function.

$$J_L(\kappa) = E_{z_0\sim p_0^z}\left[\kappa(Q_{\theta_s}(z_0, \mu_{\theta_\mu}(z_0)) - l_{STL})\right], \tag{6.11}$$

where $E_{z_0\sim p_0^z}[\cdot]$ is the expected value with respect to p_0^z.

6.4.2 SAC-Lagrangian

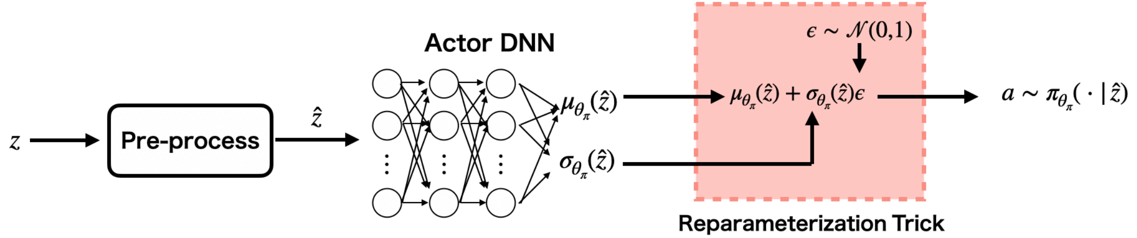


Fig. 6.3: Illustration of an actor DNN with a reparameterization trick for the SAC Lagrangian algorithm. The DNN outputs the mean $\mu_{\theta_\pi}(\hat{z})$ and the standard deviation $\sigma_{\theta_\pi}(\hat{z})$ parameters. We use the reparameterization trick to sample an action a, where ϵ is sampled from a standard normal distribution $\mathcal{N}(0,1)$. Actually, we input a pre-processed state $\hat{z}$ stated in **Section 6.4.4** to the DNN instead of an extended state z.

The SAC algorithm is a maximum entropy DRL algorithm that obtains a policy to maximize both the expected sum of rewards and the expected entropy of the policy. It is known that a maximum entropy algorithm improves explorations by acquiring diverse behaviors and has the robustness for estimation errors [13, 14]. In the SAC algorithm, we design a stochastic policy π. We use the following objective with an entropy term instead of $J(\pi)$.

$$\begin{aligned} J_{ent}(\pi) &= E_{p_0,p_f,\pi}\left[\sum_{k=0}^{K}\gamma^k(R_z(z_k,a_k) + \alpha_{ent}\mathcal{H}(\pi(\cdot|z_k)))\right], \\ &= J(\pi) + E_{p_0,p_f,\pi}\left[\sum_{k=0}^{K}\gamma^k\alpha_{ent}\mathcal{H}(\pi(\cdot|z_k))\right], \end{aligned} \tag{6.12}$$

where $\mathcal{H}(\pi(\cdot|z_k)) = E_{a\sim\pi}[-\log\pi(a|z_k)]$ is an entropy of the stochastic policy π and $\alpha_{ent} \geq 0$ is an entropy temperature. The entropy temperature determines the relative importance of the entropy term against the sum of rewards.

We use the Lagrangian relaxation for the SAC algorithm such as [88–90] to solve the τ-CMDP problem. Then, a Lagrangian function with the entropy term is given by

$$\mathcal{L}(\pi, \kappa) = J_{ent}(\pi) + \kappa(J_{STL}(\pi) - l_{STL}). \tag{6.13}$$

We model the stochastic policy using a Gaussian with the mean and the standard deviation output by a DNN with a reparameterization trick [101] as shown in **Fig. 6.3**, which is called an actor DNN π_{θ_π}. The parameter vector is denoted by θ_π. Additionally, we need to estimate $J_{ent}(\pi_{\theta_\pi})$ and $J_{STL}(\pi_{\theta_\pi})$ to update the parameter vector θ_π like the DDPG Lagrangian algorithm. Thus, $J_{ent}(\pi_{\theta_\pi})$ and $J_{STL}(\pi_{\theta_\pi})$ are also approximated by two separate critic DNNs as shown in **Fig. 6.2**. Note that, in the SAC-Lagrangian algorithm, the reward critic DNN estimates not only $J(\pi_{\theta_\pi})$ but also the entropy term. The parameter vectors are also updated using the experience replay and the target network technique. θ_r and θ_s are updated by decreasing the following critic loss functions.

$$J_{rc}(\theta_r) = E_{(z,a,z')\sim\mathcal{D}}\left[\left(Q_{\theta_r}(z,a) - \left(r + \gamma V_{\theta_r^-}(z')\right)\right)^2\right], \tag{6.14}$$

$$J_{sc}(\theta_s) = E_{(z,a,z')\sim\mathcal{D}}\left[\left(Q_{\theta_s}(z,a) - \left(s + \gamma V_{\theta_s^-}(z')\right)\right)^2\right], \tag{6.15}$$

where $r = R_z(z,a)$ and $s = R_{STL}(z)$. $Q_{\theta_r}(\cdot,\cdot)$ and $Q_{\theta_s}(\cdot,\cdot)$ are the outputs of the reward critic DNN and the STL-reward critic DNN, respectively. The target values are computed by

$$V_{\theta_r^-}(z') = E_{a'\sim\pi_{\theta_\pi}}\left[Q_{\theta_r^-}(z',a') - \alpha_{ent}\log\pi_{\theta_\pi}(a'|z')\right],$$
$$V_{\theta_s^-}(z') = E_{a'\sim\pi_{\theta_\pi}}\left[Q_{\theta_s^-}(z',a')\right],$$

where $Q_{\theta_r^-}(\cdot,\cdot)$ and $Q_{\theta_s^-}(\cdot,\cdot)$ are outputs of the target reward critic DNN and the target STL-reward critic DNN, respectively. $E_{a'\sim\pi_{\theta_\pi}}[\cdot]$ is the expected value with respect to π_{θ_π}. Their parameter vectors θ_r^-, θ_s^- are slowly updated by (6.9). In the standard SAC algorithm, the parameter vector of the actor DNN θ_π is updated by decreasing

$$J_a(\theta_\pi) = E_{z\sim\mathcal{D},a\sim\pi_{\theta_\pi}}[\alpha_{ent}\log(\pi_{\theta_\pi}(a|z)) - Q_{\theta_r}(z,a)],$$

where $E_{z\sim\mathcal{D},a\sim\pi_{\theta_\pi}}[\cdot]$ is the expected value with respect to the experiences z sampled from $\mathcal{D}$ and π_{θ_π}. However, in the SAC-Lagrangian algorithm, we consider (6.13) as the objective instead of (6.12). Thus, the parameter vector of the actor DNN θ_π is updated by decreasing the following actor loss function.

$$J_a(\theta_\pi) = E_{z\sim\mathcal{D},a\sim\pi_{\theta_\pi}}[\alpha_{ent}\log(\pi_{\theta_\pi}(a|z)) - (Q_{\theta_r}(z,a) + \kappa Q_{\theta_s}(z,a))]. \tag{6.16}$$

The Lagrange multiplier κ is updated by decreasing the following loss function.

$$J_L(\kappa) = E_{z\sim p_0^z,a\sim\pi_{\theta_\pi}}\left[\kappa(Q_{\theta_s}(z,a) - l_{STL})\right], \tag{6.17}$$

where $E_{z\sim p_0^z, a\sim\pi_{\theta_\pi}}[\cdot]$ is the expected value with respect to p_0^z and π_{θ_π}. The entropy temperature α_{ent} is updated by decreasing the following loss function.

$$J_{temp}(\alpha_{ent}) = E_{z\sim\mathcal{D}, a\sim\pi_{\theta_\pi}}\left[\alpha_{ent}(-\log(\pi_{\theta_\pi}(a|z)) - \mathcal{H}_0)\right], \tag{6.18}$$

where $\mathcal{H}_0$ is a lower bound which is a hyper parameter. Actually, in [14], the parameter $\mathcal{H}_0$ is selected based on the dimensionality of the action space. Additionally, in the SAC algorithm, to mitigate the positive bias in updates of θ_π, the double Q-learning technique [12,59] is adapted. Thus, in the SAC-Lagrangian algorithm, we also adopt the technique.

6.4.3 Pre-Training and Fine-Tuning Method

In this chapter, it is important to satisfy the given STL constraint. In order to learn a policy satisfying the constraint, the agent needs many experiences satisfying the STL formula Φ. However, it is difficult to collect the experiences considering both the control performance index and the STL constraint in the early learning stage since the agent may prioritize to optimize its policy with respect to the control performance index. Thus, we propose a two-phase learning algorithm. In the first phase which is called *pre-train*, the agent focuses on learning a policy satisfying a given STL formula Φ to store experiences receiving high STL-rewards to a replay buffer $\mathcal{D}$, that is, the agent learns its policy considering only STL-rewards.

Pre-Training for DDPG-Lagrangian

The parameter vector of the actor DNN θ_μ is updated by decreasing

$$J_a(\theta_\mu) = E_{z\sim\mathcal{D}}\left[-Q_{\theta_s}(z, \mu_{\theta_\mu}(z))\right] \tag{6.19}$$

instead of (6.10). On the other hand, θ_s is updated by (6.8).

Pre-Training for SAC-Lagrangian

The parameter vector of the actor DNN θ_π is updated by decreasing the following function.

$$J_a(\theta_\pi) = E_{z\sim\mathcal{D}, a\sim\pi_{\theta_\pi}}\left[\alpha_{ent}\log(\pi_{\theta_\pi}(a|z)) - Q_{\theta_s}(z, a)\right] \tag{6.20}$$

instead of (6.16). On the other hand, θ_s is updated by (6.15), where $V_{\theta_s}^-$ is computed by

$$V_{\theta_s^-}(z') = E_{a'\sim\pi_{\theta_\pi}}[Q_{\theta_s^-}(z', a') - \alpha_{ent}\log(\pi_{\theta_\pi}(a'|z'))].$$

In the second phase which is called *fine-tune*, the agent learns the optimal policy constrained by the given STL formula Φ. In the DDPG-Lagrangian algorithm, the

actor DNN θ_μ is updated by (6.10). In the SAC-Lagrangian algorithm, the actor DNN θ_π is updated by (6.16).

Remark: The two-phase learning may become unstable temporally because it discontinuously changes the objective functions. In such a case, we may start the second phase with changing the objective functions from those used in the first phase smoothly and slowly.

6.4.4 Pre-Process

We also use pre-processing (**Definition 5.3** in **Section 5.4.2**) to reduce the dimensionality of the extended state z such as **Chapter 5**. For simplicity, we focus on the case where $k_e^i = \tau - 1$ for all STL sub-formulae $\phi_i = G_{[k_s^i, k_e^i]}\varphi_i$ (or $F_{[k_s^i, k_e^i]}\varphi_i$), $i \in \{1, 2, ..., M\}$, which is the most effective case in terms of reducing dimensionality as shown in **Table 6.1**.

Table 6.1: Dimensionality (Dim.) of the extended state spaces.

	Without Pre-processing z	With Pre-processing $\hat{z}$ ($k_e^i = \tau - 1,\ i \in \{1, 2, ..., M\}$)	With Pre-processing $\hat{z}$ ($k_e^{\max} - k_e^{\min} \geq 1$)
Dim.	τn_x	$n_x + M$	$(k_e^{\max} - k_e^{\min}) n_x + M$

6.4.5 Algorithm

Our proposed algorithm to design an optimal policy under an STL constraint is presented in **Algorithm 5**. In line 1, we select a DRL algorithm such as the DDPG algorithm and the SAC algorithm. From line 2 to 4, we initialize the parameter vectors of the DNNs, the entropy temperature α_{ent} (if the algorithm is the SAC-Lagrangian algorithm), and the Lagrange multiplier κ. In line 5, we initialize a replay buffer $\mathcal{D}$. In line 6, we set the number of the repetition of pre-training K_{pre}. In line 7, we initialize a counter for updates. In line 9, the agent receives an initial state $x_0 \sim p_0$. From line 10 to 11, the agent sets the initial extended state $z_0 = [x_0^\top \ ... \ x_0^\top]^\top$ and computes the pre-processed state $\hat{z}_0$. One learning step is done between line 13 and 25. In line 13, the agent determines an action a_k based on the pre-processed state $\hat{z}_k$ for an exploration. In line 14, the system's state changes depending on the determined action a_k and the agent receives the next state x_{k+1}, the reward r_k, and the STL-reward s_k. From line 15 to 16, the agent constructs the next extended state z_{k+1} using x_{k+1} and z_k and computes the next pre-processed state $\hat{z}_{k+1}$. In line 17, the agent stores the experience $(\hat{z}_k, a_k, \hat{z}_{k+1}, r_k, s_k)$ in the replay buffer $\mathcal{D}$. In line 18, the agent samples I experiences $\{(\hat{z}^{(i)}, a^{(i)}, \hat{z}'^{(i)}, r^{(i)}, s^{(i)})\}_{i=1}^{I}$ from the replay buffer $\mathcal{D}$ randomly. If the learning counter is $c < K_{pre}$, the agent pre-trains the parameter vectors of DNNs in **Algorithm 7**. Then,

the parameter vectors of the reward critic DNN θ_r and the STL reward critic DNN θ_s are updated by (6.7) and (6.8) (or (6.14) and (6.15)), respectively. The parameter vector of the actor DNN θ_μ (or θ_π) is updated by (6.19) (or (6.20)). In the SAC-based algorithm, the entropy temperature α_{ent} is updated by (6.18). On the other hand, if the learning counter is $c \geq K_{pre}$, the agent fine-tunes the parameter vectors of DNNs in **Algorithm 8**. Then, the parameter vector of the actor DNN θ_μ (or θ_π) is updated by (6.10) (or (6.16)) and the other parameter vectors are updated same as the case $c < K_{pre}$. The Lagrange multiplier κ is updated by (6.11) (or (6.17)). In line 24, the agent updates the parameter vectors of target DNNs by (6.9). In line 25, the learning counter is updated. The agent repeats the process between lines 13 and 25 in a learning episode.

6.5 Example

We consider STL-constrained optimal control problems for a two-wheeled mobile robot shown in **Fig. 6.4**, where its working area Ω is $\{(x^{(0)}, x^{(1)}) | \, 0.5 \leq x^{(0)} \leq 4.5, \ 0.5 \leq x^{(1)} \leq 4.5\}$. Let $x^{(2)}$ be the steering angle with $x^{(2)} \in [-\pi, \pi]$. A discrete-time model of the robot is described by

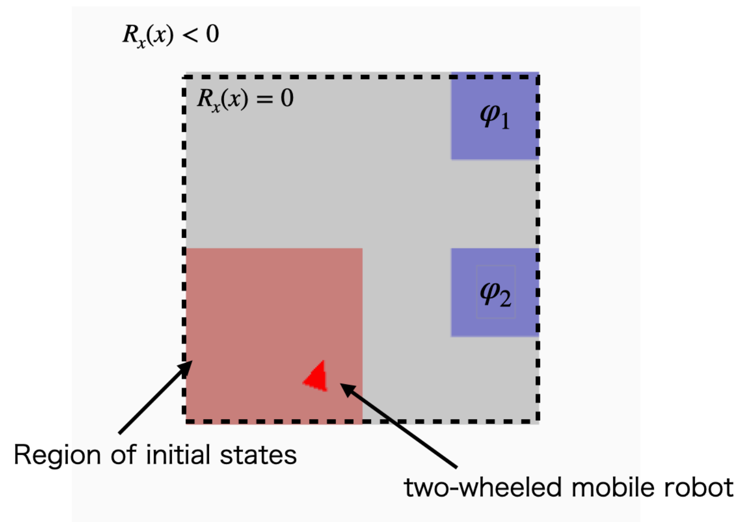


Fig. 6.4: Control of a two-wheeled mobile robot under an STL constraint. The working area is $0.5 \leq x^{(0)} \leq 4.5, \ 0.5 \leq x^{(1)} \leq 4.5$ colored gray. The initial state of the system is sampled randomly in $0.5 \leq x^{(0)} \leq 2.5, \ 0.5 \leq x^{(1)} \leq 2.5, \ -\pi/2 \leq x^{(2)} \leq \pi/2$ colored red. The region 1 labeled by φ_1 is $3.5 \leq x^{(0)} \leq 4.5, \ 3.5 \leq x^{(1)} \leq 4.5$ and the region 2 labeled by φ_2 is $3.5 \leq x^{(0)} \leq 4.5, \ 1.5 \leq x^{(1)} \leq 2.5$. These regions are colored blue.

Algorithm 5 Two-phase DRL-Lagrangian to design an optimal policy under an STL constraint.

1: Select a DRL algorithm such as DDPG and SAC.
2: Initialize parameter vectors of main DNNs.
3: Initialize parameter vectors of target DNNs.
4: Initialize an entropy temperature and a Lagrange multiplier α_{ent}, κ.
5: Initialize a replay buffer $\mathcal{D}$.
6: Set the number of the repetition of pre-training K_{pre}.
7: Initialize learning counter $c \leftarrow 0$.
8: **for** Episode = 1, ..., MAX EPISODE **do**
9: Receive an initial state $x_0 \sim p_0$.
10: Set the initial extended state z_0 using x_0.
11: Compute the pre-processed state $\hat{z}_0$ by **Algorithm 6**.
12: **for** $k = 0, ..., K$ **do**
13: Determine an action a_k based on the state $\hat{z}_k$.
14: Execute a_k and receive the next state x_{k+1} and the reward r_k and the STL-reward s_k.
15: Set the next extended state z_{k+1} using x_{k+1} and z_k.
16: Compute the next pre-processed state $\hat{z}_{k+1}$ by **Algorithm 6**.
17: Store the experience $(\hat{z}_k, a_k, \hat{z}_{k+1}, r_k, s_k)$ in the replay buffer $\mathcal{D}$.
18: Sample I experiences $\{(\hat{z}^{(i)}, a^{(i)}, \hat{z}'^{(i)}, r^{(i)}, s^{(i)})\}_{i=1,...,I}$ from $\mathcal{D}$ randomly.
19: **if** $c < K_{pre}$ **then**
20: Pre-training by **Algorithm 7**.
21: **else**
22: Fine-tuning by **Algorithm 8**.
23: **end if**
24: Update the target DNNs by (6.9).
25: $c \leftarrow c + 1$.
26: **end for**
27: **end for**

$$\begin{bmatrix} x_{k+1}^{(0)} \\ x_{k+1}^{(1)} \\ x_{k+1}^{(2)} \end{bmatrix} = \begin{bmatrix} x_k^{(0)} + \Delta a_k^{(0)} \cos(x_k^{(2)}) \\ x_k^{(1)} + \Delta a_k^{(0)} \sin(x_k^{(2)}) \\ x_k^{(2)} + \Delta a_k^{(1)} \end{bmatrix} + \Delta_w \begin{bmatrix} w_k^{(0)} \\ w_k^{(1)} \\ w_k^{(2)} \end{bmatrix}, \tag{6.21}$$

where $x_k = [x_k^{(0)}\ x_k^{(1)}\ x_k^{(2)}]^\top \in \mathbb{R}^3$, $a_k = [a_k^{(0)}\ a_k^{(1)}]^\top \in [-1, 1]^2$, and $w_k = [w_k^{(0)}\ w_k^{(1)}\ w_k^{(2)}]^\top \in \mathbb{R}^3$. $w_k^{(i)}$, $i \in \{0, 1, 2\}$ are sampled from a standard normal distribution $\mathcal{N}(0, 1)$. We assume that $\Delta = 0.1$ and $\Delta_w = 0.01 I_3$, where I_3 is the 3×3 unit matrix. The initial state of the system is sampled randomly in $0.5 \leq x^{(0)} \leq 2.5$, $0.5 \leq x^{(1)} \leq 2.5$, $-\pi/2 \leq x^{(2)} \leq \pi/2$.

Algorithm 6 Pre-processing

1: **Input:** The extended state z and the STL sub-formulae $\{\phi_i\}_{i=1}^{M}$, where $\phi_i = G_{[k_s^i,\tau-1]}\varphi_i$ (or $F_{[k_s^i,\tau-1]}\varphi_i$), $i = 1,2,...,M$.
2: **for** $i = 1,...,M$ **do**
3: **if** $\phi_i = G_{[k_s^i,\tau-1]}\varphi_i$ **then**
4: Compute the flag value f^i by (5.17).
5: **end if**
6: **if** $\phi_i = F_{[k_s^i,\tau-1]}\varphi_i$ **then**
7: Compute the flag value f^i by (5.18).
8: **end if**
9: **end for**
10: Set the flag state $\hat{f} = [\hat{f}^1\ \hat{f}^2\ ...\ \hat{f}^M]$ by (5.19).
11: **Output:** The pre-processed state $\hat{z} = [z[\tau-1]^\top\ \hat{f}^\top]^\top$.

Algorithm 7 Pre-training

1: **Input:** The experiences $\{(\hat{z}^{(i)}, a^{(i)}, \hat{z}'^{(i)}, r^{(i)}, s^{(i)})\}_{i=1,2,...,I}$ and parameters $\theta_\pi,\ \theta_r,\ \theta_s,\ \alpha_{ent}$.
2: The parameter vector θ_r is updated by (6.7) or (6.14).
3: The parameter vector θ_s is updated by (6.8) or (6.15) .
4: The parameter vector θ_π is updated by (6.19) or (6.20).
5: **if** SAC-based algorithm **then**
6: The entropy temperature α is updated by (6.18).
7: **end if**
8: **Output:** $\theta_\pi,\ \theta_r,\ \theta_s,\ \alpha$

The region 1 is $\{(x^{(0)}, x^{(1)})|\ 3.5 \le x^{(0)} \le 4.5,\ 3.5 \le x^{(1)} \le 4.5\}$ and the region 2 is $\{(x^{(0)}, x^{(1)})|\ 3.5 \le x^{(0)} \le 4.5,\ 1.5 \le x^{(1)} \le 2.5\}$. We consider the following two constraints.

Constraint 1 (Recurrence): *At any time in the time interval* $[0, 900]$, *the robot visits both the regions 1 and 2 before* 99 *time steps are elapsed, where there is no constraint for the order of the visits.*
Constraint 2 (Stabilization): *The robot visits the region 1 or 2 in the time interval* $[0, 450]$ *and stays there for* 49 *time steps.*

These constraints are described by the following STL formulae.
STL formula 1:

$$\Phi_1 = G_{[0,900]}(F_{[0,99]}\varphi_1 \wedge F_{[0,99]}\varphi_2), \tag{6.22}$$

Algorithm 8 Fine-tuning

1: **Input:** The experiences $\{(\hat{z}^{(i)}, a^{(i)}, \hat{z}'^{(i)}, r^{(i)}, s^{(i)})\}_{i=1,2,...,I}$ and parameters $\theta_\pi,\ \theta_r,\ \theta_s,\ \alpha_{ent},\ \kappa$.
2: The parameter vector θ_r is updated by (6.7) or (6.14).
3: The parameter vector θ_s is updated by (6.8) or (6.15).
4: The parameter vector θ_π is updated by (6.10) or (6.16).
5: **if** SAC-based algorithm **then**
6: The entropy temperature α is updated by (6.18).
7: **end if**
8: The Lagrange multiplier κ is updated by (6.11) or (6.17).
9: **Output:** $\theta_\pi,\ \theta_r,\ \theta_s,\ \alpha,\ \kappa$

STL formula 2:

$$\Phi_2 = F_{[0,450]}(G_{[0,49]}\varphi_1 \vee G_{[0,49]}\varphi_2), \tag{6.23}$$

where

$$\varphi_1 = ((3.5 \le x^{(0)} \le 4.5) \wedge (3.5 \le x^{(1)} \le 4.5)),$$
$$\varphi_2 = ((3.5 \le x^{(0)} \le 4.5) \wedge (1.5 \le x^{(1)} \le 2.5)).$$

We consider the following reward function

$$R_z(z, a) = R_x(z[\tau - 1]) + R_a(a), \tag{6.24}$$

where

$$R_x(x) = \min\{x^{(0)} - 0.5, 4.5 - x^{(0)}, x^{(1)} - 0.5,\ 4.5 - x^{(1)},\ 0.0\}, \tag{6.25}$$
$$R_a(a) = -||a||_2^2. \tag{6.26}$$

(6.25) is the term for keeping the working area Ω. As the agent moves away from the working area Ω, the agent receives a larger negative reward. (6.26) is the term for fuel costs.

6.5.1 Evaluation

We apply the SAC-Lagrangian algorithm to design a policy constrained by an STL formula Φ. In all simulations, the DNNs have two hidden layers, all of which have 256 units, and all layers are fully connected. The activation functions for the hidden layers and the outputs of the actor DNN are the ReLU functions and hyperbolic tangent functions, respectively. We normalize $x^{(0)}$ and $x^{(1)}$ as $x^{(0)} - 2.5$ and $x^{(1)} - 2.5$, respectively. The size of the replay buffer $\mathcal{D}$ is 1.0×10^5, and the size of the mini-batch is $I = 64$. We use Adam [103] as the optimizers for all main DNNs, the entropy

temperature, and the Lagrange multiplier. The learning rate of the optimizer for the Lagrange multiplier is 1.0×10^{-5} and the learning rates of the other optimizers are 3.0×10^{-4}. The soft update rate of the target network is $\xi = 0.01$. The discount factor is $\gamma = 0.99$. The target for updating the entropy temperature $\mathcal{H}_0$ is -2.0. The STL-reward parameter is $\beta = 100$. The agent learns its control policy for 6.0×10^5 steps. The initial parameters of both the entropy temperature and the Lagrange multiplier are 1.0. To evaluate the performance of the learned policy, we introduce the following three indices:

- a **reward learning curve** shows the mean of the sum of rewards $\sum_{k=0}^{K} \gamma^k R_z(z_k, a_k)$ for 100 trajectories,
- an **STL-reward learning curve** shows the mean of the sum of STL-rewards $\sum_{k=0}^{K} \gamma^k R_{STL}(z_k)$ for 100 trajectories, and
- a **success rate** shows the number of trajectories satisfying the given STL constraint for 100 trajectories.

We prepare 100 initial states sampled from p_0 and generate 100 trajectories using the learned policy for each evaluation. We show the results for $K_{pre} = 0$ (**Case 1**) and $K_{pre} = 300000$ (**Case 2**). We do not utilize pre-training in **Case 1**. All simulations run on a computer with AMD Ryzen 9 3950X 16-core processor, NVIDIA (R) GeForce RTX 2070 super, and 32GB of memory and were conducted using the Python software.

Formula 1

We consider the case where the constraint is given by (6.22). In this simulation, we set $K = 1000$ and $l_{STL} = -40$. The length of z is $\tau = 100$. The reward learning curves and the STL-rewards learning curves are shown in **Figs. 6.5** and **6.6**, respectively. In **Case 1**, it takes a lot of steps to learn a policy such that the sum of STL-rewards is near the threshold $l_{STL} = -40$. The reward learning curve decreases gradually while the STL-reward curve increases. This is an effect of lacking in experience satisfying the STL formula Φ. If the agent cannot satisfy the STL constraint during its explorations, the Lagrange multiplier κ becomes large as shown in **Fig. 6.7**. Then, the STL term $-\kappa Q_{\theta_s}$ of the actor loss $J(\pi_\theta)$ becomes larger than the other terms. As a result, the agent updates the parameter vector θ_π considering only the STL-rewards. On the other hand, in **Case 2**, the agent can obtain enough experiences satisfying the STL formula in 300000 pre-training steps. The agent learns the policy such that the sum of the STL-rewards is near the threshold relatively quickly and fine-tunes the policy under the STL constraint after pre-training. According to the results in the both cases, our proposed method is useful to learn the optimal policy under the STL constraint. Additionally, as the sum of STL-rewards obtained by the learned policy is increasing, the success rate for the given STL formula is also increasing as shown in **Fig. 6.8**.

Fig. 6.5: Reward learning curves for the formula Φ_1. The red and blue curves show the results of $K_{pre} = 0$ (**Case 1**) and $K_{pre} = 300000$ (**Case 2**), respectively. The solid curves and the shades represent the average results and the standard deviations over 10 trials with different random seeds, respectively. The gray line shows 300000 steps.

Fig. 6.6: STL-reward learning curves for the formula Φ_1. The red and blue curves show the results of $K_{pre} = 0$ (**Case 1**) and $K_{pre} = 300000$ (**Case 2**), respectively. The solid curves and the shades represent the average results and the standard deviations over 10 trials with different random seeds, respectively. The dashed line shows the threshold $l_{STL} = -40$. The gray line shows 300000 steps.

Formula 2

We consider the case where the constraint is given by (6.23). In this simulation, we set $K = 500$ and $l_{STL} = 35$. The length of z is $\tau = 50$. We use the reward function $R_{STL}(z) = \exp(\beta \mathbf{1}(\rho(z, \phi)))/\exp(\beta)$ instead of (5.15) to prevent the sum of STL-rewards diverging to infinity. The reward learning curves and the STL-rewards learning

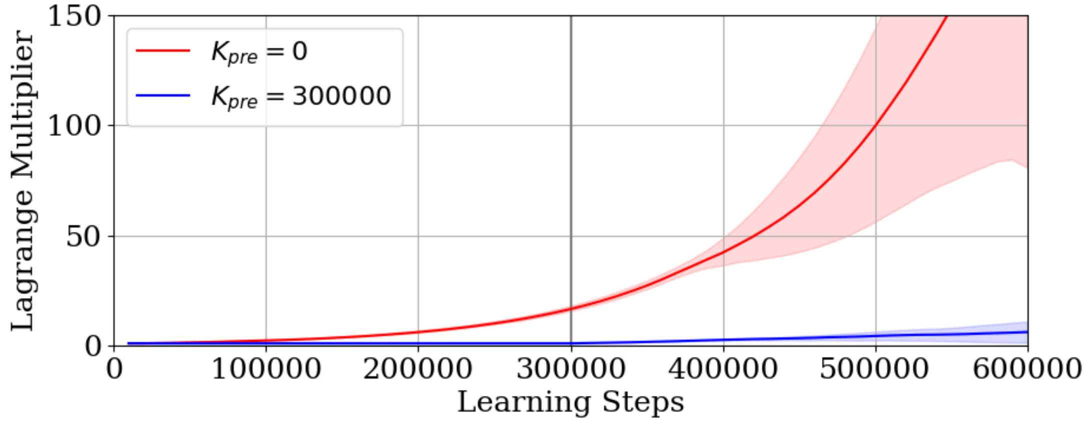


Fig. 6.7: Curves of Lagrange multiplier κ for the formula Φ_1. The red and blue curves show the results of $K_{pre} = 0$ (**Case 1**) and $K_{pre} = 300000$ (**Case 2**), respectively. The solid curves and the shades represent the average results and the standard deviations over 10 trials with different random seeds, respectively. The gray line shows 300000 steps.

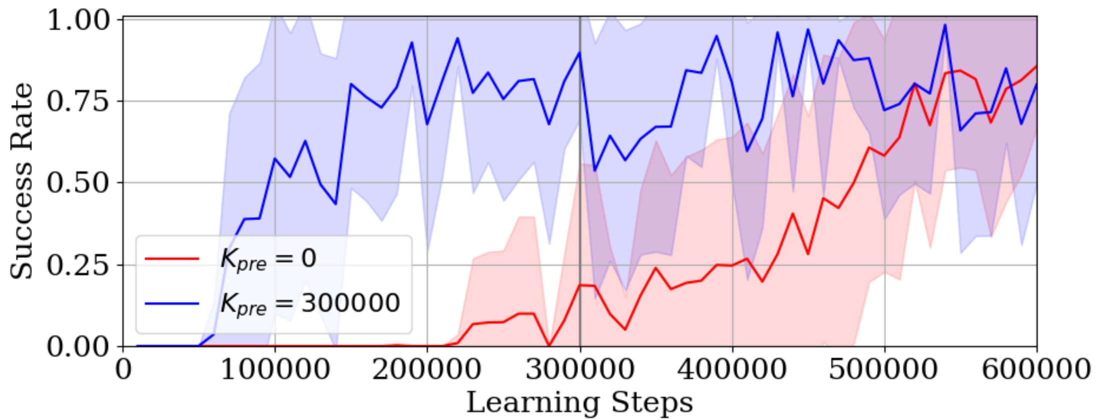


Fig. 6.8: Success rates for the formula Φ_1. The red and blue curves show the results of $K_{pre} = 0$ (**Case 1**) and $K_{pre} = 300000$ (**Case 2**), respectively. The solid curves and the shades represent the average results and the standard deviations over 10 trials with different random seeds, respectively. The gray line shows 300000 steps.

curves are shown in **Figs. 6.9** and **6.10**, respectively. In **Case 1**, although the reward learning curve maintains more than -20, the STL-reward learning curve maintains much less than the threshold $l_{STL} = 35$. On the other hand, in **Case 2**, the agent learns the policy such that the sum of STL-rewards is near the threshold $l_{STL} = 35$ and fine-tunes the policy under the STL constraint after pre-training. Our proposed method is useful for not only the formula Φ_1 but also the formula Φ_2. Additionally, as the sum of STL-rewards obtained by the learned policy is increasing, the success

rate for the given STL formula is also increasing as shown in **Fig. 6.11**.

Fig. 6.9: Reward learning curves for the formula Φ_2. The red and blue curves show the results of $K_{pre} = 0$ (**Case 1**) and $K_{pre} = 300000$ (**Case 2**), respectively. The solid curves and the shades represent the average results and the standard deviations over 10 trials with different random seeds, respectively. The gray line shows 300000 steps.

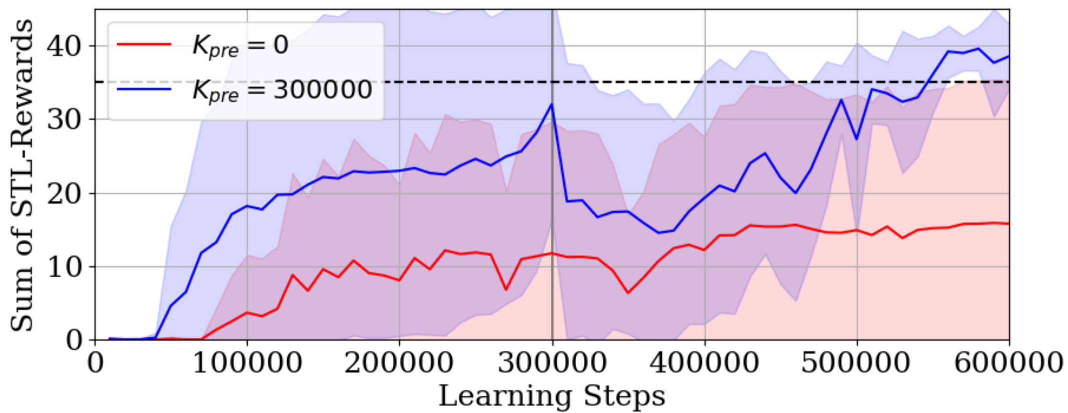


Fig. 6.10: STL-reward learning curves for the formula Φ_2. The red and blue curves show the results of $K_{pre} = 0$ (**Case 1**) and $K_{pre} = 300000$ (**Case 2**), respectively. The solid curves and the shades represent the average results and the standard deviations over 10 trials with different random seeds, respectively. The dashed line shows the threshold $l_{STL} = 35$. The gray line shows 300000 steps.

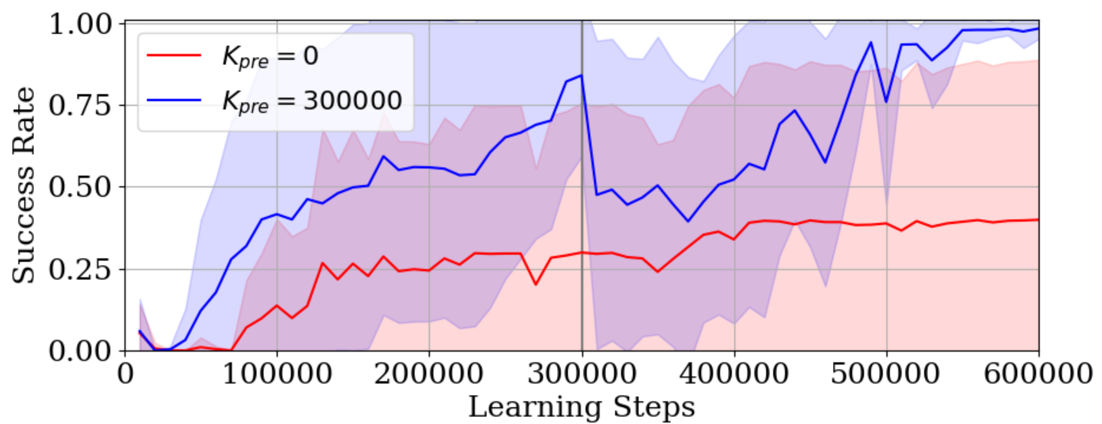


Fig. 6.11: Success rates for the formula Φ_2. The red and blue curves show the results of $K_{pre} = 0$ (**Case 1**) and $K_{pre} = 300000$ (**Case 2**), respectively. The solid curves and the shades represent the average results and the standard deviations over 10 trials with different random seeds, respectively. The gray line shows 300000 steps.

6.5.2 Ablation Studies for Pre-Processing

In this section, we show the ablation study for pre-processing stated in **Section 6.4.4**. We conduct the experiment for Φ_1 using the SAC-Lagrangian algorithm. In the case without pre-processing, the dimensionality of the input to DNNs is 300 and, in the case with pre-processing, the dimensionality of the input to DNNs is 5. The STL-reward learning curves for the two cases are shown in **Fig. 6.12**. The agent without pre-processing cannot improve the performance of its policy with respect to STL-rewards. The result concludes that pre-processing is necessary for a problem constrained by an STL formula Φ with a large τ.

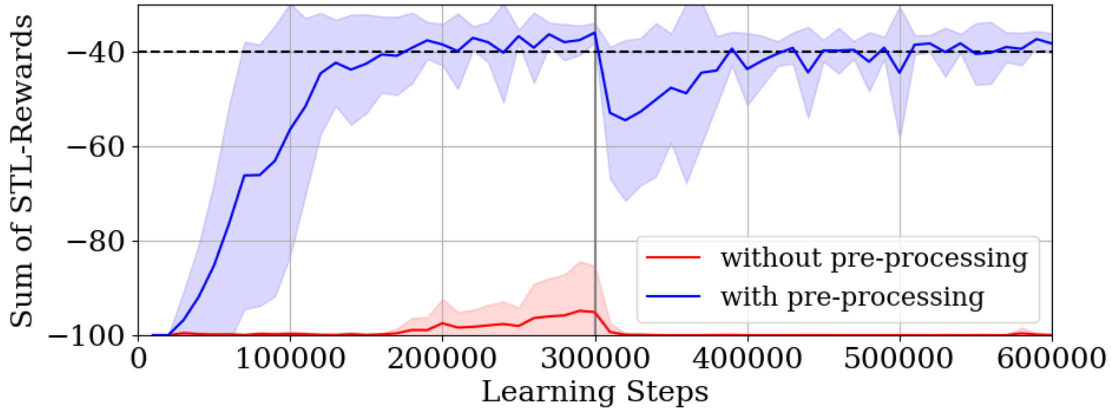


Fig. 6.12: STL-reward learning curves for the case without pre-processing (red) and the case with pre-processing (blue). We consider the formula Φ_1. The solid curves and the shades represent the average results and the standard deviations over 10 trials with different random seeds, respectively. The dashed line shows the threshold $l_{STL} = -40$. The gray line shows 300000 steps.

6.5.3 Comparison of Based Algorithms

In this section, we compare the SAC based algorithm with other off-policy DRL-based algorithms: DDPG [11] and TD3 [12]. For the DDPG-Lagrangian algorithm and the TD3-Lagrangian algorithm, we use the following Ornstein-Uhlenbeck process for generating exploration noises.

$$\epsilon_{k+1} = \epsilon_k - p_1(\epsilon_k - p_2) + p_3\epsilon_{\mathcal{N}},$$

where $\epsilon_{\mathcal{N}}$ is a noise generated by a standard normal distribution $\mathcal{N}(0,1)$. We set the parameters $(p_1, p_2, p_3) = (0.15, 0, 0.3)$. For the TD3-Lagrangian algorithm, the target policy smoothing is implemented by adding noises sampled from the normal

distribution $\mathcal{N}(0, 0.2)$ to the actions chosen by the target actor DNN, clipped to $(-0.5, 0.5)$. The agent updates the actor DNN and the target DNNs every 2 learning steps. Other experimental settings such as hyper-parameters, optimizers, and DNN architectures, are same as the SAC-Lagrangian algorithm.

We conduct experiments for Φ_1. We show the reward learning curves and the STL-reward learning curves in **Figs. 6.13** and **6.14**, respectively. Although all algorithms can improve the policy with respect to rewards after fine-tuning, the DDPG algorithm cannot improve the policy with respect to the STL-rewards. The STL-reward curve of the DDPG-Lagrangian algorithm is much less than the threshold. On the other hand, the TD3-Lagrangian algorithm and the SAC-Lagrangian algorithm can learn the policy such that the STL-rewards are more than the threshold. These results show the importance of the double Q-learning technique to mitigate positive biases for critic estimations in the fine-tuning phase. Actually, the technique is used in both the TD3-Lagrangian algorithm and the SAC-Lagrangian algorithm. Then, we show the result in the case where we do not use the double Q-learning technique in the SAC-Lagrangian in **Fig. 6.15**. Although the agent can learn a policy such that the STL-rewards are near the threshold in the pre-training phase, the performance of the policy with respect to the STL-rewards is degraded in the fine-tuning phase.

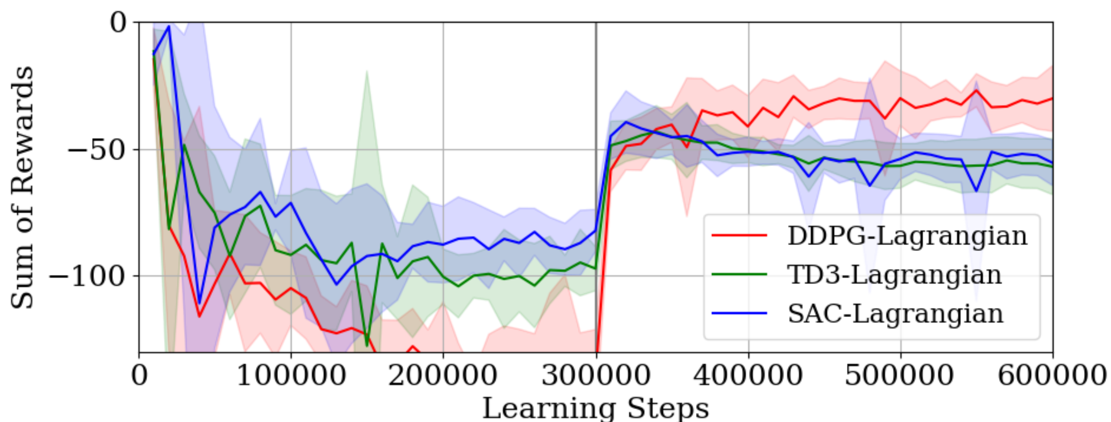


Fig. 6.13: Reward learning curves for the formula Φ_1. The red, blue, and green curves show the results of the DDPG-Lagrangian algorithm, the TD3-Lagrangian algorithm, and the SAC-Lagrangian algorithm, respectively. The solid curves and the shades represent the average results and the standard deviations over 10 trials with different random seeds, respectively.

Fig. 6.14: STL-reward learning curves for the formula Φ_1. The red, blue, and green curves show the results of the DDPG-Lagrangian algorithm, the TD3-Lagrangian algorithm, and the SAC-Lagrangian algorithm, respectively. The solid curves and the shades represent the average results and the standard deviations over 10 trials with different random seeds, respectively.

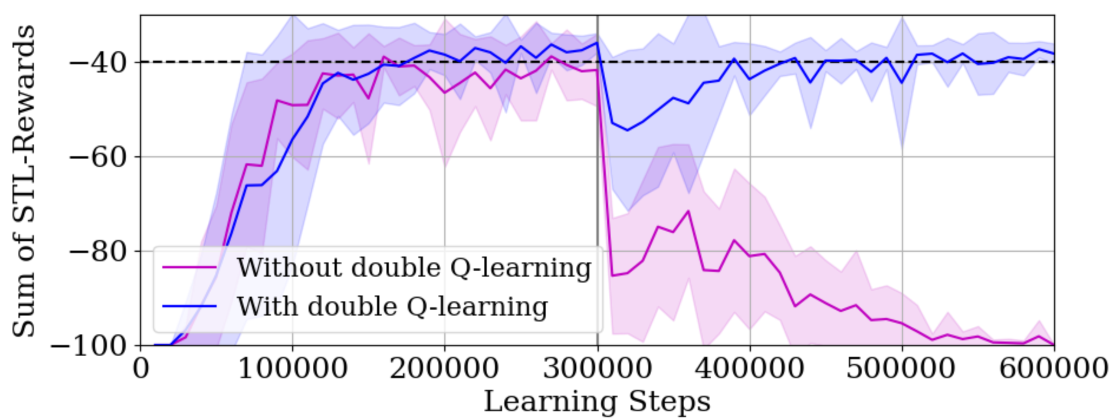


Fig. 6.15: STL-reward learning curves for the formula Φ_1. The purple and blue curves show the results of the SAC-Lagrangian algorithm without and with the double Q-learning technique, respectively. The solid curves and the shades represent the average results and the standard deviations over 10 trials with different random seeds, respectively.

Chapter 7

Conclusions and Future Works

We proposed some DRL-based optimal controller design methods to extend the applicable range of DRL in the real world. DRL has achieved great results for various decision making problems thanks to the development of DNN techniques. Particularly, the development of DRL algorithms for Atari video games is remarkable. Nevertheless, it is often difficult to directly apply DRL to problems in the real world. It is necessary to make ingenuity according to each application. Concretely, in this dissertation, we tackled the following problems.

In **Chapter 3**, we proposed a practical deep Q-learning algorithm for stabilization of nonlinear systems using a simulator to mitigate high sample complexity of DRL. Our proposed method consists of the two stages. In the first stage, we obtain the approximated optimal Q-functions for virtual systems in the simulator using the continuous deep Q-learning algorithm. In the second stage, we represent the Q-function for the real system by the approximated linear function whose basis functions are the approximated optimal Q-functions learned in the first stage. The agent learns the parameter vector of the approximated linear Q-function through online interactions with the real system. We showed that the agent can learn the parameter vector and stabilize the real system at the fixed point. Moreover, we showed that the agent with our proposed algorithm can adapt to a system whose system parameter vector varies slowly.

In **Chapter 4**, we proposed a DRL-based method to design a networked controller that stabilizes a nonlinear system at an equilibrium point considering two types of network delays caused by data transmissions between the system and the controller. At first, we considered the case where the sensor can observe all state variables of the system. Then, we regarded not only the latest observed state but also some previously determined control actions as the state of the environment to stabilize the system. Next, we considered the case where the sensor cannot observe part of state variables of the system. Then, we used not only the latest observed output and some previously determined control actions but also some previously observed outputs as the state of the environment. As examples, we considered the stabilization problems for two nonlinear systems: a pendulum and a Lorenz dynamics, and show that the

agent can learn its policy by our proposed method.

In **Chapter 5**, we proposed a DRL-based networked controller design to complete a given temporal control task described by an STL formula considering the effect of network delays. We introduced an extended MDP, which is called a τd-MDP, and proposed a DRL algorithm to design the networked controller, where it is assumed that the network delays are unknown constants. Additionally, we utilized pre-processing for the DRL algorithm to reduce the dimensionality of the extended state. Furthermore, through numerical simulations, we showed that our proposed algorithm can be also applied to an NCS problem with random network delays.

In **Chapter 6**, we considered a model-free optimal control problem constrained by a given STL formula. We modeled the problem as a τ-CMDP that is an extension of a τ-MDP and a CMDP. To solve the τ-CMDP problem with continuous state-action spaces, we proposed a constrained DRL algorithm with the Lagrangian relaxation. In the algorithm, we relaxed the constrained problem into an unconstrained problem in order to utilize a standard DRL algorithm. Additionally, we proposed a practical two-phased learning algorithm to make it easy to obtain experiences satisfying the given STL formula. Through numerical simulations, we demonstrated the performance of the proposed algorithm. First, we showed that the agent with our proposed two-phase algorithm can learn its policy for the τ-CMDP problem. Next, we conducted ablation studies for pre-processing to reduce the dimensionality of the extended state and showed the usefulness. Finally, we conducted three constrained DRL algorithms and showed the usefulness of the double Q-learning technique in the fine-tuning phase.

We show several future works as follows: For the study in **Chapter 3**, we devise approaches to choose the parameter vectors of virtual systems automatically and to extend the proposed method to the case where the mathematical model of the real system has uncertain parts. Additionally, the second stage of our proposed method is related to online learning methods. The algorithm analysis such as regret analysis for our proposed learning algorithm is also one of important future studies. For the study in **Chapter 4**, the application to complicated systems such as multi robot systems is one of future works. Additionally, the theoretical analysis for partial observation is also an important future work. For the study in **Chapter 5**, the reward may be sparse for some STL formulae and the syntax is restrictive compared with the general STL syntax [45]. Solving the issues is an interesting future work. For the study in **Chapter 6**, our approach cannot guarantee satisfying the STL constraint during learning of the policy. Solving the issue is a future work. Additionally, we further develop the theory of DRL with some constraints and propose practical constrained DRL algorithms to extend the applicable range of DRL in the real world.

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. The MIT Press, 2018.

[2] C. Szepesvari, *Algorithms for Reinforcement Learning*. Morgan and Claypool Publishers, 2010.

[3] C. M. Bishop and N. M. Nasrabadi, *Pattern Recognition and Machine Learning*. Springer, 2006.

[4] T. Lattimore and C. Szepesvári, *Bandit Algorithms*. Cambridge University Press, 2020.

[5] R. Bellman, "The theory of dynamic programming," *Bulletin of the American Mathematical Society*, vol. 60, no. 6, pp. 503–515, 1954.

[6] C. J. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3, pp. 279–292, 1992.

[7] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Proceedings of the 12th International Conference on Neural Information Processing Systems (NIPS 1999)*, 1999, pp. 1057–1063.

[8] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT press, 2016.

[9] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[10] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[11] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[12] S. Fujimoto, H. Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," in *Proceedings of the 35th International Conference on Machine Learning (ICML 2018)*, vol. 80, 2018, pp. 1587–1596.

[13] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," in *Proceedings of the 35th International Conference on Machine Learning (ICML 2018)*,

vol. 80, 2018, pp. 1861–1870.

[14] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel *et al.*, “Soft actor-critic algorithms and applications,” *arXiv preprint arXiv:1812.05905*, 2018.

[15] S. Gu, T. Lillicrap, I. Sutskever, and S. Levine, “Continuous deep Q-learning with model-based acceleration,” in *Proceedings of the 33rd International Conference on Machine Learning (ICML 2016)*, vol. 48, 2016, pp. 2829–2838.

[16] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, “Asynchronous methods for deep reinforcement learning,” in *Proceedings of the 33rd International Conference on Machine Learning (ICML 2016)*, vol. 48, 2016, pp. 1928–1937.

[17] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, “Trust region policy optimization,” in *Proceedings of the 32nd International Conference on Machine Learning (ICML 2015)*, vol. 37, 2015, pp. 1889–1897.

[18] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, “High-dimensional continuous control using generalized advantage estimation,” in *Proceedings of the 4th International Conference on Learning Representations (ICLR 2016)*, 2016.

[19] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” *arXiv preprint arXiv:1707.06347*, 2017.

[20] B. R. Kiran, I. Sobh, V. Talpaert, P. Mannion, A. A. Al Sallab, S. Yogamani, and P. Pérez, “Deep reinforcement learning for autonomous driving: A survey,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 6, pp. 4909–4926, 2021.

[21] S. Gu, E. Holly, T. Lillicrap, and S. Levine, “Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates,” in *Proceedings of the 2017 IEEE International Conference on Robotics and Automation (ICRA 2017)*, 2017, pp. 3389–3396.

[22] W. Zhao, J. P. Queralta, and T. Westerlund, “Sim-to-real transfer in deep reinforcement learning for robotics: a survey,” in *Proceedings of the 2020 IEEE Symposium Series on Computational Intelligence (SSCI 2020)*, 2020, pp. 737–744.

[23] A. A. Rusu, M. Večerík, T. Rothörl, N. Heess, R. Pascanu, and R. Hadsell, “Sim-to-real robot learning from pixels with progressive nets,” in *Proceedings of the 1st Annual Conference on Robot Learning*, 2017, pp. 262–270.

[24] J. Tan, T. Zhang, E. Coumans, A. Iscen, Y. Bai, D. Hafner, S. Bohez, and V. Vanhoucke, “Sim-to-real: Learning agile locomotion for quadruped robots,” *arXiv preprint arXiv:1804.10332*, 2018.

[25] M. A. Bucci, O. Semeraro, A. Allauzen, G. Wisniewski, L. Cordier, and L. Mathelin, “Control of chaotic systems by deep reinforcement learning,” *Proceedings of the Royal Society A*, vol. 475, no. 2231, 2019.

[26] J. Ikemoto and T. Ushio, “Control of discrete-time chaotic systems with policy-based deep reinforcement learning,” *IEICE Transactions on Fundamentals of Elec-*

tronics, Communications and Computer Sciences, vol. 103, no. 7, pp. 885–892, 2020.
[27] X.-Y. Liu, H. Yang, Q. Chen, R. Zhang, L. Yang, B. Xiao, and C. D. Wang, "Finrl: A deep reinforcement learning library for automated stock trading in quantitative finance," *arXiv preprint arXiv:2011.09607*, 2020.
[28] Y. Deng, F. Bao, Y. Kong, Z. Ren, and Q. Dai, "Deep direct reinforcement learning for financial signal representation and trading," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, no. 3, pp. 653–664, 2016.
[29] H. Khalil, *Nonlinear Systems*. Prentice Hall, 2002.
[30] J. Ikemoto and T. Ushio, "Continuous deep Q-learning with a simulator for stabilization of uncertain discrete-time systems," *Nonlinear Theory and Its Applications, IEICE*, vol. 12, no. 4, pp. 738–757, 2021.
[31] J. P. Hespanha, P. Naghshtabrizi, and Y. Xu, "A survey of recent results in networked control systems," *Proceedings of the IEEE*, vol. 95, no. 1, pp. 138–162, 2007.
[32] R. A. Gupta and M.-Y. Chow, "Networked control system: Overview and research trends," *IEEE Transactions on Industrial Electronics*, vol. 57, no. 7, pp. 2527–2535, 2009.
[33] X.-M. Zhang, Q.-L. Han, X. Ge, D. Ding, L. Ding, D. Yue, and C. Peng, "Networked control systems: A survey of trends and techniques," *IEEE/CAA Journal of Automatica Sinica*, vol. 7, no. 1, pp. 1–17, 2019.
[34] R. Yang, G.-P. Liu, P. Shi, C. Thomas, and M. V. Basin, "Predictive output feedback control for networked control systems," *IEEE Transactions on Industrial Electronics*, vol. 61, no. 1, pp. 512–520, 2013.
[35] H. Gao, T. Chen, and J. Lam, "A new delay system approach to network-based control," *Automatica*, vol. 44, no. 1, pp. 39–52, 2008.
[36] T. Fujita and T. Ushio, "Rl-based optimal networked control considering network delay of discrete-time linear systems," in *Proceedings of the 14th annual European Control Conference (ECC 2015)*, 2015, pp. 2476–2481.
[37] T. Fujita and T. Ushio, "Optimal digital control with uncertain network delay of linear systems using reinforcement learning," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. 99, no. 2, pp. 454–461, 2016.
[38] J. Ikemoto and T. Ushio, "Application of deep reinforcement learning to networked control systems with uncertain network delays," *Nonlinear Theory and Its Applications, IEICE*, vol. 11, no. 4, pp. 480–500, 2020.
[39] J. Ikemoto and T. Ushio, "Stabilization of nonlinear systems with uncertain input delays using deep reinforcement learning," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences (Japanese Edition)*, vol. J102-A, no. 10, pp. 268–271, 2019.
[40] J. Ikemoto and T. Ushio, "Networked control of nonlinear systems under partial observation using continuous deep Q-learning," in *Proceedings of the 58th IEEE*

Conference on Decision and Control (CDC 2019), 2019, pp. 6793–6798.

[41] C. Belta, B. Yordanov, and E. A. Gol, *Formal Methods for Discrete-Time Dynamical Systems*. Springer, 2017.

[42] C. Baier and J.-P. Katoen, *Principles of Model Checking*. MIT press, 2008.

[43] M. Hasanbeig, A. Abate, and D. Kroening, "Logically-constrained reinforcement learning," *arXiv preprint arXiv:1801.08099*, 2018.

[44] M. Hasanbeig, D. Kroening, and A. Abate, "Deep reinforcement learning with temporal logics," in *Proceedings of the 18th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS 2020)*, 2020, pp. 1–22.

[45] O. Maler and D. Nickovic, "Monitoring temporal properties of continuous signals," in *Proceedings of Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, 2004, pp. 152–166.

[46] A. Donzé, "On signal temporal logic," in *Proceedings of the 4th International Conference on Runtime Verification*, 2013, pp. 382–383.

[47] V. Raman, A. Donzé, M. Maasoumy, R. M. Murray, A. Sangiovanni-Vincentelli, and S. A. Seshia, "Model predictive control with signal temporal logic specifications," in *Proceedings of the 53rd IEEE Conference on Decision and Control (CDC 2014)*, 2014, pp. 81–87.

[48] L. Lindemann and D. V. Dimarogonas, "Control barrier functions for signal temporal logic tasks," *IEEE control systems letters*, vol. 3, no. 1, pp. 96–101, 2018.

[49] D. Aksaray, A. Jones, Z. Kong, M. Schwager, and C. Belta, "Q-learning for robust satisfaction of signal temporal logic specifications," in *Proceedings of the 55th Conference on Decision and Control (CDC 2016)*, 2016, pp. 6565–6570.

[50] H. Venkataraman, D. Aksaray, and P. Seiler, "Tractable reinforcement learning of signal temporal logic objectives," in *Proceedings of the 2nd Conference on Learning for Dynamics and Control*, 2020, pp. 308–317.

[51] A. Balakrishnan and J. V. Deshmukh, "Structured reward shaping using signal temporal logic specifications," in *Proceedings of the 2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2019)*, 2019, pp. 3481–3486.

[52] P. Kapoor, A. Balakrishnan, and J. V. Deshmukh, "Model-based reinforcement learning from signal temporal logic specifications," *arXiv preprint arXiv:2011.04950*, 2020.

[53] N. Hansen, "The CMA evolution strategy: A tutorial," *arXiv preprint arXiv:1604.00772*, 2016.

[54] J. Ikemoto and T. Ushio, "Deep reinforcement learning based networked control with network delays for signal temporal logic specifications," in *Proceedings of the 27th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2022)*, 2022, pp. 1–8.

[55] E. Altman, *Constrained Markov Decision Processes: Stochastic Modeling*. Routledge, 1999.

[56] J. Ikemoto and T. Ushio, "Deep reinforcement learning under signal temporal

logic constraints using Lagrangian relaxation," *IEEE Access*, vol. 10, pp. 114 814–114 828, 2022.

[57] H. Dong, Z. Ding, and S. Zhang, *Deep Reinforcement Learning Fundamentals, Research and Applications: Fundamentals, Research and Applications*. Springer, 2020.

[58] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "A brief survey of deep reinforcement learning," *arXiv preprint arXiv:1708.05866*, 2017.

[59] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double Q-learning," in *Proceedings of the 30th AAAI conference on artificial intelligence (AAAI 2016)*, vol. 30, no. 1, 2016, pp. 2094–2100.

[60] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[61] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, O. Pieter Abbeel, and W. Zaremba, "Hindsight experience replay," in *Proceedings of the 30th International Conference on Neural Information Processing Systems (NIPS 2017)*, vol. 30, 2017.

[62] M. G. Bellemare, W. Dabney, and R. Munos, "A distributional perspective on reinforcement learning," in *Proceedings of the 34th International Conference on Machine Learning (ICML 2017)*, vol. 70, 2017, pp. 449–458.

[63] M. Fortunato, M. G. Azar, B. Piot, J. Menick, I. Osband, A. Graves, V. Mnih, R. Munos, D. Hassabis, O. Pietquin, C. Blundell, and S. Legg, "Noisy networks for exploration," in *Proceedings of the 6th International Conference on Learning Representations (ICLR 2018)*, 2018.

[64] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, "Dueling network architectures for deep reinforcement learning," in *Proceedings of the 33rd International Conference on Machine Learning (ICML 2016)*, vol. 48, 2016, pp. 1995–2003.

[65] A. S. Polydoros and L. Nalpantidis, "Survey of model-based reinforcement learning: Applications on robotics," *Journal of Intelligent and Robotic Systems*, vol. 86, no. 2, pp. 153–173, 2017.

[66] W. Li and E. Todorov, "Iterative linear quadratic regulator design for nonlinear biological movement systems." in *Proceedings of the 1st International Conference on Informatics in Control, Automation and Robotics (ICINCO 2004)*, 2004, pp. 222–229.

[67] E. Todorov and W. Li, "A generalized iterative LQG method for locally-optimal feedback control of constrained nonlinear stochastic systems," in *Proceedings of the 2005 American Control Conference (ACC 2005)*, 2005, pp. 300–306.

[68] M. P. Deisenroth, D. Fox, and C. E. Rasmussen, "Gaussian processes for data-efficient learning in robotics and control," *IEEE transactions on pattern analysis and machine intelligence*, vol. 37, no. 2, pp. 408–423, 2013.

[69] F. Berkenkamp, M. Turchetta, A. Schoellig, and A. Krause, "Safe model-based

reinforcement learning with stability guarantees," *Proceedings of the 30th International Conference on Neural Information Processing Systems (NIPS 2017)*, vol. 30, 2017.

[70] A. Nagabandi, G. Kahn, R. S. Fearing, and S. Levine, "Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning," in *Proceedings of the 2018 IEEE International Conference on Robotics and Automation (ICRA 2018)*, 2018, pp. 7559–7566.

[71] T. Kurutach, I. Clavera, Y. Duan, A. Tamar, and P. Abbeel, "Model-ensemble trust-region policy optimization," *arXiv preprint arXiv:1802.10592*, 2018.

[72] S. Levine and P. Abbeel, "Learning neural network policies with guided policy search under unknown dynamics," *Proceedings of the 27th International Conference on Neural Information Processing Systems (NIPS 2014)*, vol. 27, 2014.

[73] N. C. Luong, D. T. Hoang, S. Gong, D. Niyato, P. Wang, Y.-C. Liang, and D. I. Kim, "Applications of deep reinforcement learning in communications and networking: A survey," *IEEE Communications Surveys and Tutorials*, vol. 21, no. 4, pp. 3133–3174, 2019.

[74] D. Baumann, J.-J. Zhu, G. Martius, and S. Trimpe, "Deep reinforcement learning for event-triggered control," in *Proceedings of the 57th IEEE Conference on Decision and Control (CDC 2018)*, 2018, pp. 943–950.

[75] B. Demirel, A. Ramaswamy, D. E. Quevedo, and H. Karl, "Deepcas: A deep reinforcement learning algorithm for control-aware scheduling," *IEEE Control Systems Letters*, vol. 2, no. 4, pp. 737–742, 2018.

[76] K. Katsikopoulos and S. Engelbrecht, "Markov decision processes with delays and asynchronous cost collection," *IEEE Transactions on Automatic Control*, vol. 48, no. 4, pp. 568–574, 2003.

[77] T. Walsh, A. Nouri, L. Li, and M. Littman, "Learning and planning in environments with delayed feedback," *Autonomous Agents and Multi-Agent Systems*, vol. 18, pp. 83–105, 2008.

[78] S. Ramstedt, Y. Bouteiller, G. Beltrame, C. J. Pal, and J. Binas, "Reinforcement learning with random delays," in *Proceedings of the 9th International Conference on Learning Representations (ICLR 2021)*, 2021.

[79] M. Hausknecht and P. Stone, "Deep recurrent Q-learning for partially observable MDPs," in *Proceedings of the 2015 AAAI Fall Symposium Series*, 2015.

[80] N. Heess, J. J. Hunt, T. P. Lillicrap, and D. Silver, "Memory-based control with recurrent neural networks," *arXiv preprint arXiv:1512.04455*, 2015.

[81] A. Puranic, J. Deshmukh, and S. Nikolaidis, "Learning from demonstrations using signal temporal logic," in *Proceedings of the 2020 Conference on Robot Learning*, vol. 155, 2021, pp. 2228–2242.

[82] A. G. Puranic, J. V. Deshmukh, and S. Nikolaidis, "Learning from demonstrations using signal temporal logic in stochastic and continuous domains," *IEEE Robotics and Automation Letters*, vol. 6, no. 4, pp. 6250–6257, 2021.

[83] S. Sickert, J. Esparza, S. Jaax, and J. Křetínský, "Limit-deterministic Büchi automata for linear temporal logic," in *Proceedings of the 28th International Conference on Computer Aided Verification (CAV 2016)*, 2016, pp. 312–332.
[84] X. Li, C.-I. Vasile, and C. Belta, "Reinforcement learning with temporal logic rewards," in *Proceedings of the 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2017)*, 2017, pp. 3834–3839.
[85] X. Li, Y. Ma, and C. Belta, "A policy search method for temporal logic specified reinforcement learning tasks," in *Proceedings of the 2018 American Control Conference (ACC 2018)*, 2018, pp. 240–245.
[86] Y. Liu, A. Halev, and X. Liu, "Policy learning with constraints in model-free reinforcement learning: A survey," in *Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI 2021)*, 2021, pp. 4508–4515.
[87] A. Stooke, J. Achiam, and P. Abbeel, "Responsive safety in reinforcement learning by PID Lagrangian methods," in *Proceedings of the 37th International Conference on Machine Learning (ICML 2020)*, vol. 119, 2020, pp. 9133–9143.
[88] S. Ha, P. Xu, Z. Tan, S. Levine, and J. Tan, "Learning to walk in the real world with minimal human effort," *arXiv preprint arXiv:2002.08550*, 2020.
[89] Q. Yang, T. D. Simão, S. Tindemans, and M. T. J. Spaan, "WCSAC: Worst-case soft actor critic for safety-constrained reinforcement learning," in *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI 2021)*, 2021.
[90] W. Wang, N. Yu, Y. Gao, and J. Shi, "Safe off-policy deep reinforcement learning algorithm for Volt-VAR control in power distribution systems," *IEEE Transactions on Smart Grid*, vol. 11, no. 4, pp. 3008–3018, 2020.
[91] J. Achiam, D. Held, A. Tamar, and P. Abbeel, "Constrained policy optimization," in *Proceedings of the 34th International Conference on Machine Learning (ICML 2017)*, vol. 70, 2017, pp. 22–31.
[92] Y. Chow, O. Nachum, E. Duenez-Guzman, and M. Ghavamzadeh, "A Lyapunov-based approach to safe reinforcement learning," in *Proceedings of the 31st International Conference on Neural Information Processing Systems (NeurIPS 2018)*, vol. 31, 2018.
[93] Y. Chow, O. Nachum, A. Faust, E. Duenez-Guzman, and M. Ghavamzadeh, "Lyapunov-based safe policy optimization for continuous control," *arXiv preprint arXiv:1901.10031*, 2019.
[94] K. C. Kalagarla, R. Jain, and P. Nuzzo, "Model-free reinforcement learning for optimal control of Markov decision processes under signal temporal logic specifications," in *Proceedings of the 60th IEEE Conference on Decision and Control (CDC 2021)*, 2021, pp. 2252–2257.
[95] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, vol. 8, no. 3, pp. 229–256, 1992.
[96] F. L. Lewis and D. Vrabie, "Reinforcement learning and adaptive dynamic

programming for feedback control," *IEEE Circuits and Systems Magazine*, vol. 9, no. 3, pp. 32–50, 2009.

[97] S. Levine, A. Kumar, G. Tucker, and J. Fu, "Offline reinforcement learning: Tutorial, review, and perspectives on open problems," *arXiv preprint arXiv:2005.01643*, 2020.

[98] A. Kumar, A. Zhou, G. Tucker, and S. Levine, "Conservative Q-learning for offline reinforcement learning," in *Proceedings of the 33rd International Conference on Neural Information Processing Systems (NeurIPS 2020)*, vol. 33, 2020, pp. 1179–1191.

[99] G. E. Uhlenbeck and L. S. Ornstein, "On the theory of the Brownian motion," *Physical review*, vol. 36, no. 5, pp. 823–841, 1930.

[100] S. Levine, "Reinforcement learning and control as probabilistic inference: Tutorial and review," *arXiv preprint arXiv:1805.00909*, 2018.

[101] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *arXiv preprint arXiv:1312.6114*, 2013.

[102] D. Bertsekas, *Dynamic Programming and Optimal Control: Volume I*. Athena scientific, 2012, vol. 1.

[103] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[104] W. Aangenent, D. Kostic, B. de Jager, R. van de Molengraft, and M. Steinbuch, "Data-based optimal control," in *Proceedings of the 2005 American Control Conference (ACC 2005)*, 2005, pp. 1460–1465.

[105] F. L. Lewis and K. G. Vamvoudakis, "Reinforcement learning for partially observable dynamic processes: Adaptive dynamic programming using measured output data," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 41, no. 1, pp. 14–25, 2010.

[106] L. F. Shampine, *Numerical solution of ordinary differential equations*. Routledge, 2018.

[107] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[108] D. P. Bertsekas, *Nonlinear Programming*, 3rd ed. Academic press, 2016.

Appendix

A Reconstruction of State of Linear Dynamical System

We consider the following linear dynamical system.

$$\dot{x}(t) = A_c x(t) + B_c u(t),$$
$$y(t) = C_c x(t),$$

where $x(t) \in \mathbb{R}^{n_x}$, $u(t) \in \mathbb{R}^{n_u}$, and $y(t) \in \mathbb{R}^{n_y}$ are the state, the control input, and the output at time t, respectively. A_c, B_c, and C_c are matrices of dimensions $n_x \times n_x$, $n_x \times n_u$, and $n_y \times n_x$, respectively. It is assumed that the control input is zero-order hold and the sampling period is $\Delta > 0$. Then, we have the discrete-time linear dynamical system as follows:

$$x_{k+1} = A_d x_k + B_d u_k, \tag{A-1}$$
$$y_k = C_d x_k, \tag{A-2}$$

where $x_k = x(k\Delta)$, $y_k = y(k\Delta)$, and u_k is the k-th control input computed by the digital controller, and

$$A_d = \exp(A_c \Delta),$$
$$B_d = \int_0^{\Delta} \exp(A_c \xi) B_c \mathrm{d}\xi,$$
$$C_d = C_c.$$

We assume that (A_d, B_d) is controllable and (A_d, C_d) is observable.

By (A-1) and (A-2),

$$x_k = A_d^{\varsigma} x_{k-\varsigma} + \begin{bmatrix} B_d & A_d B_d & A_d^2 B_d & \cdots & A_d^{\varsigma-1} B_d \end{bmatrix} \begin{bmatrix} u_{k-1} \\ u_{k-2} \\ \vdots \\ u_{k-\varsigma} \end{bmatrix}, \tag{A-3}$$

$$\begin{bmatrix} y_{k-1} \\ y_{k-2} \\ \vdots \\ y_{k-\varsigma} \end{bmatrix} = \begin{bmatrix} C_d^{\varsigma-1} A_d \\ \vdots \\ C_d A_d \\ C_d \end{bmatrix} x_{k-\varsigma} + \begin{bmatrix} 0 & C_d B_d & C_d A_d B_d & \cdots & C_d A_d^{\varsigma-2} B_d \\ 0 & 0 & C_d B_d & \cdots & C_d A_d^{\varsigma-3} B_d \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & C_d B_d \\ 0 & 0 & 0 & \cdots & 0 \end{bmatrix} \begin{bmatrix} u_{k-1} \\ u_{k-2} \\ \vdots \\ u_{k-\varsigma} \end{bmatrix}, \tag{A-4}$$

where ς is larger than the observability index. Furthermore, we rewrite the two equations as follows:

$$x_k = A_d^{\varsigma} x_{k-\varsigma} + M_c u_{k-1:k-\varsigma}, \tag{A-5}$$

$$y_{k-1:k-\varsigma} = M_o x_{k-\varsigma} + T u_{k-1:k-\varsigma}, \tag{A-6}$$

where

$$M_c = \begin{bmatrix} B_d & A_d B_d & A_d^2 B_d & \cdots & A_d^{\varsigma-1} B_d \end{bmatrix},$$

$$M_o = \begin{bmatrix} C_d^{\varsigma-1} A_d \\ \vdots \\ C_d A_d \\ C_d \end{bmatrix},$$

$$T = \begin{bmatrix} 0 & C_d B_d & C_d A_d B_d & \cdots & C_d A_d^{\varsigma-2} B_d \\ 0 & 0 & C_d B_d & \cdots & C_d A_d^{\varsigma-3} B_d \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & C_d B_d \\ 0 & 0 & 0 & \cdots & 0 \end{bmatrix},$$

$$u_{k-1:k-\varsigma} = \begin{bmatrix} u_{k-1} \\ u_{k-2} \\ \vdots \\ u_{k-\varsigma} \end{bmatrix},$$

$$y_{k-1:k-\varsigma} = \begin{bmatrix} y_{k-1} \\ y_{k-2} \\ \vdots \\ y_{k-\varsigma} \end{bmatrix}.$$

The matrices M_c, M_o, and T are called the controllability matrix, the observability matrix, and the Toeplitz matrix, respectively.

If (A_d, C_d) is observable, then there exists the observability index q such that $rank(M_o) < n_x$ for $\varsigma < q$ and that $rank(M_o) = n_x$ for $\varsigma \geq q$. Thus, let $\varsigma \geq q$, the observability matrix M_o is full column rank n_x. Then, there exists a matrix M such that

$$A_d^{\varsigma} = MM_o. \tag{A-7}$$

Moreover, since M_o is a full column rank matrix, its left inverse matrix is given by

$$M_o^{+} = (M_o^{\top} M_o)^{-1} M_o^{\top}, \tag{A-8}$$

so that, for any matrix Z,

$$M = A_d^{\varsigma} M_o^{+} + Z(I - M_o M_o^{+}). \tag{A-9}$$

The following equation is proved.

$$x_k = \begin{bmatrix} M_u & M_y \end{bmatrix} \begin{bmatrix} u_{k-1:k-\varsigma} \\ y_{k-1:k-\varsigma} \end{bmatrix}, \tag{A-10}$$

where $M_y = A_d^{\varsigma}(M_o^{\top} M_o)^{-1} M_o^{\top}$ and $M_u = M_c - A_d^{\varsigma}(M_o^{\top} M_o)^{-1} M_o^{\top} T$.

Proof According to (A-6), (A-7), and (A-8), we have

$$\begin{aligned} A_d^{\varsigma} x_{k-\varsigma} &= MM_o x_{k-\varsigma} \\ &= M(y_{k-1:k-\varsigma} - Tu_{k-1:k-\varsigma}) \\ &= (A_d^{\varsigma} M_o^{+} + Z(I - M_o M_o^{+}))y_{k-1:k-\varsigma} - (A_d^{\varsigma} M_o^{+} + Z(I - M_o M_o^{+}))Tu_{k-1:k-\varsigma} \\ &= A_d^{\varsigma} M_o^{+} y_{k-1:k-\varsigma} - A_d^{\varsigma} M_o^{+} Tu_{k-1:k-\varsigma} + \underline{Z(I - M_o M_o^{+})y_{k-1:k-\varsigma} - Z(I - M_o M_o^{+})Tu_{k-1:k-\varsigma}}. \end{aligned} \tag{A-11}$$

Then, we consider the underline part of (A-11). From (A-6) and (A-8),

$$\begin{aligned} & Z(I - M_o M_o^{+})y_{k-1:k-\varsigma} - Z(I - M_o M_o^{+})Tu_{k-1:k-\varsigma} \\ &= Z(I - M_o M_o^{+})(M_o x_{k-\varsigma} + Tu_{k-1:k-\varsigma}) - Z(I - M_o M_o^{+})Tu_{k-1:k-\varsigma} \\ &= Z(I - M_o M_o^{+})M_o x_{k-\varsigma} \\ &= Z(M_o - M_o(M_o^{\top} M_o)^{-1} M_o^{\top} M_o)x_{k-\varsigma} \\ &= Z(M_o - M_o)x_{k-\varsigma} \\ &= 0. \end{aligned}$$

Therefore, we have the following equation.

$$A_d^{\varsigma} x_{k-\varsigma} = A_d^{\varsigma} M_o^{+} y_{k-1:k-\varsigma} - A_d^{\varsigma} M_o^{+} Tu_{k-1:k-\varsigma}. \tag{A-12}$$

According to (A-5), (A-10) is proved as follows:

$$\begin{aligned} x_k &= A_d^{\varsigma} M_o^{+} y_{k-1:k-\varsigma} - A_d^{\varsigma} M_o^{+} T u_{k-1:k-\varsigma} + M_c u_{k-1:k-\varsigma} \\ &= (M_c - A_d^{\varsigma}(M_o^{\top} M_o)^{-1} M_o^{\top} T) u_{k-1:k-\varsigma} + A_d^{\varsigma}(M_o^{\top} M_o)^{-1} M_o^{\top} y_{k-1:k-\varsigma} \\ &= M_u u_{k-1:k-\varsigma} + M_y y_{k-1:k-\varsigma}. \end{aligned}$$

□

From the result, we confirm that the controller can estimate the state of the system under the partial observation using sufficient past control inputs $u_{k-1:k-\varsigma}$ and outputs $y_{k-1:k-\varsigma}$.

B Runge-Kutta Method

In **Chapter 4**, we use the Runge-Kutta method to solve the ordinary differential equation. We consider the following initial value problem.

$$\begin{aligned} \dot{x}(t) &= f(x(t)), \\ x(t_0) &= x_0, \end{aligned} \tag{B-13}$$

where $x \in \mathbb{R}^{n_x}$. Let $\delta > 0$ be a step-size. We approximately compute the solution of the equation as follows:

$$x^{(n+1)} = x^{(n)} + \frac{1}{6}(h_1 + 2h_2 + 2h_3 + h_4),$$

where

$$\begin{aligned} h_1 &= \delta f(x^{(n)}), \\ h_2 &= \delta f\left(x^{(n)} + \frac{h_1}{2}\right), \\ h_3 &= \delta f\left(x^{(n)} + \frac{h_2}{2}\right), \\ h_4 &= \delta f(x^{(n)} + h_3). \end{aligned}$$

In simulations of **Chapter 4**, we set $\delta = 2^{-10}$.

Example 7.1 We consider the following pendulum.

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} = \begin{bmatrix} x_2(t) \\ 9.81 \sin x_1(t) - 0.05 x_2(t) \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u_1(t). \tag{B-14}$$

At $t = n\delta$ ($n \in \mathbb{N}$), we compute the system state $x((n+1)\delta)$ by the Runge-Kutta method. Let $x^{(n)} = [x_1^{(n)}\ x_2^{(n)}]^\top \simeq [x_1(n\delta)\ x_2(n\delta)]^\top = x(n\delta)$. It is assumed that the control inputs is constant values for $n\delta \leq t \leq (n+1)\delta$. We compute h_1, h_2, h_3, and $h_4 \in \mathbb{R}^{n_x}$ as follows:

1.

$$h_1 = \begin{bmatrix} h_{1,1} \\ h_{1,2} \end{bmatrix} = \delta \begin{bmatrix} x_2^{(n)} \\ 9.81 \sin x_1^{(n)} - 0.05 x_2^{(n)} + u(n\delta) \end{bmatrix}.$$

2.

$$h_2 = \begin{bmatrix} h_{2,1} \\ h_{2,2} \end{bmatrix} = \delta \begin{bmatrix} x_2^{(n)} + h_{1,2}/2 \\ 9.81 \sin(x_1^{(n)} + h_{1,1}/2) - 0.05(x_2^{(n)} + h_{1,2}/2) + u(n\delta) \end{bmatrix}.$$

3.

$$h_3 = \begin{bmatrix} h_{3,1} \\ h_{3,2} \end{bmatrix} = \delta \begin{bmatrix} x_2^{(n)} + h_{2,2}/2 \\ 9.81\sin(x_1^{(n)} + h_{2,1}/2) - 0.05(x_2^{(n)} + h_{2,2}/2) + u(n\delta) \end{bmatrix}.$$

4.

$$h_4 = \begin{bmatrix} h_{4,1} \\ h_{4,2} \end{bmatrix} = \delta \begin{bmatrix} x_2^{(n)} + h_{3,2} \\ 9.81\sin(x_1^{(n)} + h_{3,1}) - 0.05(x_2^{(n)} + h_{3,2}) + u(n\delta) \end{bmatrix}.$$

Finally, we compute the $x((n+1)\delta)$ as follows:

$$x((n+1)\delta) \simeq \begin{bmatrix} x_1^{(n)} \\ x_2^{(n)} \end{bmatrix} + \frac{1}{6}\left(\begin{bmatrix} h_{1,1} \\ h_{1,2} \end{bmatrix} + 2\begin{bmatrix} h_{2,1} \\ h_{2,2} \end{bmatrix} + 2\begin{bmatrix} h_{3,1} \\ h_{3,2} \end{bmatrix} + \begin{bmatrix} h_{4,1} \\ h_{4,2} \end{bmatrix} \right).$$

Example 7.2 We consider the following Lorenz dynamics.

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{bmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \end{bmatrix} = \begin{bmatrix} 10(x_2(t) - x_1(t)) \\ -x_1(t)x_3(t) + 28x_1(t) - x_2(t) \\ x_1(t)x_2(t) - 8/3x_3(t) \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 2 & 0 \\ 0 & 3 \end{bmatrix}\begin{bmatrix} u_1(t) \\ u_2(t) \end{bmatrix}. \tag{B-15}$$

At $t = n\delta$ ($n \in \mathbb{N}$), we compute the system state $x((n+1)\delta)$ by the Runge-Kutta method. Let $x^{(n)} = [x_1^{(n)}\ x_2^{(n)}\ x_3^{(n)}]^\top \simeq [x_1(n\delta)\ x_2(n\delta)\ x_3(n\delta)]^\top = x(n\delta)$. It is assumed that the control inputs is constant values for $n\delta \le t \le (n+1)\delta$. We compute h_1, h_2, h_3, and $h_4 \in \mathbb{R}^{n_x}$ as follows:

1.

$$h_1 = \begin{bmatrix} h_{1,1} \\ h_{1,2} \\ h_{1,3} \end{bmatrix} = \delta \begin{bmatrix} 10x_2^{(n)} - 10x_1^{(n)} + u_1(n\delta) \\ -x_1^{(n)}x_3^{(n)} + 28x_1^{(n)} - x_2^{(n)} + 2u_1(n\delta) \\ x_1^{(n)}x_2^{(n)} - 8/3x_3^{(n)} + 3u_2(n\delta) \end{bmatrix}.$$

2.

$$h_2 = \begin{bmatrix} h_{2,1} \\ h_{2,2} \\ h_{2,3} \end{bmatrix}$$
$$= \delta \begin{bmatrix} 10(x_2^{(n)} + h_{1,2}/2) - 10(x_1^{(n)} + h_{1,1}/2) + u_1(n\delta) \\ -(x_1^{(n)} + h_{1,1}/2)(x_3^{(n)} + h_{1,3}/2) + 28(x_1^{(n)} + h_{1,1}/2) - (x_2^{(n)} + h_{1,2}/2) + 2u_1(n\delta) \\ (x_1^{(n)} + h_{1,1}/2)(x_2^{(n)} + h_{1,2}/2) - 8/3(x_3^{(n)} + h_{1,3}/2) + 3u_2(n\delta) \end{bmatrix}.$$

3.

$$
\begin{aligned}
h_3 &= \begin{bmatrix} h_{3,1} \\ h_{3,2} \\ h_{3,3} \end{bmatrix} \\
&= \delta \begin{bmatrix} 10(x_2^{(n)} + h_{2,2}/2) - 10(x_1^{(n)} + h_{2,1}/2) + u_1(n\delta) \\ -(x_1^{(n)} + h_{2,1}/2)(x_3^{(n)} + h_{2,3}/2) + 28(x_1^{(n)} + h_{2,1}/2) - (x_2^{(n)} + h_{2,2}/2) + 2u_1(n\delta) \\ (x_1^{(n)} + h_{2,1}/2)(x_2^{(n)} + h_{2,2}/2) - 8/3(x_3^{(n)} + h_{2,3}/2) + 3u_2(n\delta) \end{bmatrix}.
\end{aligned}
$$

4.

$$
h_4 = \begin{bmatrix} h_{4,1} \\ h_{4,2} \\ h_{4,3} \end{bmatrix} = \delta \begin{bmatrix} 10(x_2^{(n)} + h_{3,2}) - 10(x_1^{(n)} + h_{3,1}) + u_1(n\delta) \\ -(x_1^{(n)} + h_{3,1})(x_3^{(n)} + h_{3,3}) + 28(x_1^{(n)} + h_{3,1}) - (x_2^{(n)} + h_{3,2}) + 2u_1(n\delta) \\ (x_1^{(n)} + h_{3,1})(x_2^{(n)} + h_{3,2}) - 8/3(x_3^{(n)} + h_{3,3}) + 3u_2(n\delta) \end{bmatrix}.
$$

Finally, we compute the $x((n+1)\delta)$ as follows:

$$
x((n+1)\delta) \simeq \begin{bmatrix} x_1^{(n)} \\ x_2^{(n)} \\ x_3^{(n)} \end{bmatrix} + \frac{1}{6}\left(\begin{bmatrix} h_{1,1} \\ h_{1,2} \\ h_{1,3} \end{bmatrix} + 2\begin{bmatrix} h_{2,1} \\ h_{2,2} \\ h_{2,3} \end{bmatrix} + 2\begin{bmatrix} h_{3,1} \\ h_{3,2} \\ h_{3,3} \end{bmatrix} + \begin{bmatrix} h_{4,1} \\ h_{4,2} \\ h_{4,3} \end{bmatrix} \right).
$$

C NCSs Simulations

In **Chapter 4**, we consider a network control problem with network delays. We describe the flow from the k-th observation of a system's output y_k to the update of a control input by the k-th control action a_k in detail.

1. **Sensor**
   At $t = k\Delta$, the sensor observes the k-th output of the system. The output function is given by (4.2).
2. **Network delay for the output data transmission**
   The output data y_k observed at $t = k\Delta$ is transmitted from the sensor to the agent. The delay caused by this transmission is denoted by $\tau_{\mathrm{sc},k}$, where $\tau_{\mathrm{sc},k}$ is sampled from an unknown probability distribution.
3. **Agent**
   At $t = k\Delta + \tau_{\mathrm{sc},k}$, the agent receives the k-th output data y_k and computes the k-th control action a_k.
4. **Network Delay for the control action transmission**
   The k-th control action computed at $t = k\Delta + \tau_{\mathrm{sc},k}$ is transmitted from the agent to the actuator. The delay caused by this transmission is denoted by $\tau_{\mathrm{ca},k}$, where $\tau_{\mathrm{ca},k}$ is sampled from an unknown probability distribution.
5. **Actuator**
   The control input $u(t)$ inputted to the system is updated from a_{k-1} to a_k at $t = k\Delta + \tau_{\mathrm{sc},k} + \tau_{\mathrm{ca},k}$.

Additionally, we assume that the order of control actions and observed outputs does not change. Then, the network delays $\{\tau_{\mathrm{sc},k}\}_{k\in\mathbb{N}}$ and $\{\tau_{\mathrm{ca},k}\}_{k\in\mathbb{N}}$ satisfy the following two conditions:

Condition 1: For all $k \in \mathbb{N}$,

$$k\Delta + \tau_{\mathrm{sc},k} \leq (k+1)\Delta + \tau_{\mathrm{sc},k+1},$$

that is,

$$\tau_{\mathrm{sc},k+1} \geq \tau_{\mathrm{sc},k} - \Delta.$$

Then, we define the variable $\hat{\tau}_{\mathrm{sc},k} := \tau_{\mathrm{sc},k+1} - (\tau_{\mathrm{sc},k} - \Delta)$. We need $\hat{\tau}_{\mathrm{sc},k} \geq 0$.

Condition 2: For all $k \in \mathbb{N}$,

$$k\Delta + \tau_{\mathrm{sc},k} + \tau_{\mathrm{ca},k} \leq (k+1)\Delta + \tau_{\mathrm{sc},k+1} + \tau_{\mathrm{ca},k+1},$$

that is,

$$\begin{aligned}\tau_{\mathrm{ca},k+1} &\geq \tau_{\mathrm{ca},k} - (\tau_{\mathrm{sc},k+1} - \tau_{\mathrm{sc},k}) - \Delta \\ &= \tau_{\mathrm{ca},k} - (\tau_{\mathrm{sc},k+1} - \tau_{\mathrm{sc},k} + \Delta) \\ &= \tau_{\mathrm{ca},k} - (\tau_{\mathrm{sc},k+1} - (\tau_{\mathrm{sc},k} - \Delta)) \\ &= \tau_{\mathrm{ca},k} - \hat{\tau}_{\mathrm{sc},k}.\end{aligned}$$

Then, we define the variable $\hat{\tau}_{\mathrm{ca},k} := \tau_{\mathrm{ca},k+1} - (\tau_{\mathrm{ca},k} - \hat{\tau}_{\mathrm{sc},k})$. We need $\hat{\tau}_{\mathrm{ca},k} \geq 0$.

In the dissertation, we actually use the discrete uniform distribution as the probability distribution.

$$DU(m,n) = \frac{1}{n-m+1},\ m,n \in \mathbb{N},\ m < n.$$

Let δ be a step size for numerical integration by the Runge-Kutta method. The network delays τ is sampled as follows:

$$\begin{aligned}\tau &= \epsilon_{NCS}\delta, \\ \epsilon_{NCS} &\sim DU(3\Delta/\delta, 5\Delta/\delta).\end{aligned}$$

In the simulation of the dissertation, we use $\Delta = 2^{-4}$ and $\delta = 2^{-10}$, that is, $DU(192, 320)$.

The sampled network delays $\tau_{\mathrm{sc},k}$ and $\tau_{\mathrm{ca},k}$ may not satisfy the two conditions. Thus, in the simulations, if $\hat{\tau}_{\mathrm{sc},k} < 0$, then $\tau_{\mathrm{sc},k+1} = \tau_{\mathrm{sc},k} - \Delta + \delta$ and if $\hat{\tau}_{\mathrm{ca},k} < 0$, then $\tau_{\mathrm{ca},k+1} = \tau_{\mathrm{ca},k} - \hat{\tau}_{\mathrm{sc},k} + \delta$.

Acknowledgment

First and foremost, I would like to express my sincere gratitude to my supervisor Professor Toshimitsu Ushio, Graduate School of Engineering Science, Osaka University, for his great supports. His guidances and discussions were of inestimable value for my studies.

I deeply appreciate Professor Masahiro Inuiguchi and Professor Youji Iiguni, Graduate school of Engineering Science, Osaka University, for agreeing to be on my dissertation committee. Their suggestions and comments helped me improve this dissertation.

I also would like to thank all past and current members at Ushio Laboratory for their kind help, discussions, and friendship. I spent wonderful days with them.

Last but not least, I would like to convey my gratitude to my family and all friends from university.

Junya Ikemoto

Publication List

Peer-Reviewed Journal Article

[J1] J. Ikemoto and T. Ushio, “Stabilization of nonlinear systems with uncertain input delays using deep reinforcement learning,” *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. J102-A, no. 10, pp. 268–271, Oct. 2019 (in Japanese).

[J2] J. Ikemoto and T. Ushio, “Control of discrete-time chaotic systems with policy-based deep reinforcement learning,” *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E103-A, no. 7, pp. 885–892, Jul. 2020, doi: 10.1587/transfun.2019EAP1154.

[J3] J. Ikemoto and T. Ushio, “Application of deep reinforcement learning to networked control systems with uncertain network delays,” *Nonlinear Theory and Its Applications, IEICE*, vol. 11, no. 4, pp. 480–500, Oct. 2020, doi:10.1587/nolta.11.480.

[J4] J. Ikemoto and T. Ushio, “Continuous deep Q-learning with a simulator for stabilization of uncertain discrete-time systems,” *Nonlinear Theory and Its Applications, IEICE*, vol. 12, no. 4, pp. 738–757, Oct. 2021, doi: 10.1587/nolta.12.738.

[J5] J. Ikemoto and T. Ushio, “Deep reinforcement learning under signal temporal logic constraints using Lagrangian relaxation,” *IEEE Access*, vol. 10, pp. 114814–114828, Oct. 2022, doi: 10.1109/ACCESS.2022.3218216.

Reviewed International Conference Proceeding

[C1] J. Ikemoto and T. Ushio, “Application of continuous deep Q-learning to networked state-feedback control of nonlinear systems with uncertain network delays,” in *Proceedings of 2019 International Symposium on Nonlinear Science and Its Applications (NOLTA 2019)*, Kuala Lumpur, Malaysia, Dec. 2019, pp. 192–195.

[C2] J. Ikemoto and T. Ushio, “Networked control of nonlinear systems under partial observation using continuous deep Q-learning,” in *Proceedings of the 58th IEEE Conference on Decision and Control (CDC 2019)*, Nice, France, Dec. 2019, pp. 6793–6798, doi: 10.1109/CDC40024.2019.9029214.

[C3] J. Ikemoto and T. Ushio, “Deep reinforcement learning based networked control with network delays for signal temporal logic specifications,” in *Proceedings of the 27th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2022)*, Stuttgart, Germany, Sep. 2022, pp. 1–8, doi: 10.1109/ETFA52439.2022.9921505.

Review Article

[R1] J. Ikemoto and T. Ushio, “Application of deep reinforcement learning to control problems,” *The Brain and Neural Networks*, vol. 26, no. 4, pp. 135-144, Dec. 2019.

Domestic Conference

[D1] J. Ikemoto and T. Ushio, “Stabilization of nonlinear systems with uncertain input delays using deep reinforcement learning,” in *IEICE Technical Committee on Nonlinear Problems*, Fukui, Mar. 2019 (in Japanese).

[D2] J. Ikemoto and T. Ushio, “Chaos control of Hénon map using reinforcement learning,” in *IEICE Society Conference*, Toyonaka, Sep. 2019 (in Japanese).

[D3] J. Ikemoto and T. Ushio, “Deep reinforcement learning of continuous control policy for satisfying signal temporal logic specifications,” *IEICE Society Conference*, Online, Sep. 2021 (in Japanese).

Award Recieved

[A1] 2018 NLP Presentation Award of IEICE.