NAIST-IS-DD0861020

Doctoral Dissertation

Highly Modularized Learning System for Behavior Acquisition of Functional Robots

Akihiko Yamaguchi

March 17, 2011

Department of Information Systems
Graduate School of Information Science
Nara Institute of Science and Technology

A Doctoral Dissertation
submitted to Graduate School of Information Science,
Nara Institute of Science and Technology
in partial fulfillment of the requirements for the degree of
Doctor of ENGINEERING

Akihiko Yamaguchi

Thesis Committee:

Professor Tsukasa Ogasawara	*(Supervisor)*
Professor Kazushi Ikeda	*(Co-supervisor)*
Associate Professor Jun Takamatsu	*(Co-supervisor)*

Highly Modularized Learning System for Behavior Acquisition of Functional Robots[*]

Akihiko Yamaguchi

Abstract

An essential ability for highly functional robots, typified by humanoid robots, is acquiring a behavior by itself. This ability enables the end-users to teach their wishes easily. This thesis aims to realize an objective-based task design.

Concretely, this thesis proposes a "highly modularized learning system" where reinforcement learning (RL) methods and related methods are modularized to construct a learning system for a robot. The proposed system has two distinct features: (1) learning strategies (LSs), methods to improve RL, are also modularized, and (2) multiple LSs are applied to each task of the robot multiple times. Examples of LSs are dimension reduction, hierarchical RL, and transfer learning. Using the LSs, the system automatically generates multiple behavior modules to learn policies of a task. For feature (2), Boltzmann selection method with upper confidence bound (UCB) is introduced to select a behavior module that is actually used to control the robot. Some transfer LSs generate new modules from previously learned modules. Thus, each behavior module is generated through a sequence of LSs. The core algorithm is referred to as LS fusion.

This thesis also proposes some elemental technologies to define LSs; mainly, a discrete action set named DCOB and a method to decompose a dynamics model. The discrete action set DCOB has a capability for large domains. The name is derived from the fact that an action is a trajectory Directed to the Center Of a target Basis function. DCOB is extended to learn continuous actions keeping the exploration ability of the discrete set; the method is named WF-DCOB because of using wire-fitting. Some transfer LSs are defined by using WF-DCOB. The method to decompose a dynamics model can extract a task invariant element, which enables to transfer a dynamics model of a task to one of the other tasks.

[*]Doctoral Dissertation, Department of Information Systems, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-DD0861020, March 17, 2011.

Additionally, a highly modularized RL library named SkyAI is developed to implement the proposed methods, and is distributed under an open source license.

A number of experiments were performed to verify the proposed methods. DCOB and WF-DCOB outperformed the conventional methods especially in motion learning tasks of a simulated humanoid robot. DCOB also demonstrated an outstanding performance in a crawling task of a real spider robot. The method to decompose a dynamics model improved the learning speed in maze tasks. Though WF-DCOB demonstrated an outstanding performance, LS fusion additionally improved the performance. The scalability of LS fusion was verified in a maze task of the simulated humanoid robot. In this task, the robot learned a path to a goal hierarchically using a crawling and a turning motions acquired in previous learning.

Through these experiments, this thesis concludes that the highly modularized learning system is a realistic approach to realize the objective-based task design.

Keywords:

Learning System, Reinforcement Learning, Modularization, Motion Learning, Humanoid Robot

高機能ロボットの行動獲得のための高度にモジュール化された学習システム[*]

山口 明彦

内容梗概

ヒューマノイドに代表される高機能ロボットに不可欠な機能のひとつとして，ロボット自身が行動を獲得することがあげられる．このような機能は，エンドユーザがロボットに要求を伝えることを容易にする．本論文は，目的に基づくタスク設計システムの開発を目的とする．

具体的には，本論文は「高度にモジュール化された学習システム」を提案する．このシステムでは，強化学習手法及び関連手法群がモジュール化され，学習システムを構成する．提案するシステムは次の 2 つの特徴を持つ.（1）強化学習を改善する学習戦略群もモジュール化される.（2）単一のロボットの各々のタスクに対し，複数の学習戦略群が複数回適用される．学習戦略とは，例えば次元縮約，階層化，転移学習である．学習戦略群を用いて，システムはタスクの方策を学習する行動モジュール群を自動的に生成する．特徴（2）について，実際にロボットを制御する行動モジュールを選択するために，upper confidence bound（UCB）をもとにした Boltzmann 選択手法が導入される．転移学習モジュール群は既に学習したモジュールから新たなモジュールを生成する．このように，各行動モジュールは複数の学習戦略を通して生成される．中心となるアルゴリズムを学習戦略フュージョンと呼ぶ．

本論文では，学習戦略を定義するための要素技術として，離散行動集合 DCOB 及びダイナミクスモデルの分離手法を提案する．離散行動集合 DCOB は規模の大きいタスクを扱うことができる．DCOB は Directed to the Center Of a target Basis function を表し，目標の基底関数の中心に向かうように行動が生成されることを意味する．DCOB は，離散集合の探索能力を保ったまま，連続行動を学習するように拡張される．この手法は wire-fitting を用いるため WF-DCOB と名付けられる．一部の転移学習戦略群は，WF-DCOB を用いて定義される．ダイナミクスモデルの分離手法は，タスク不変の要素の抽出を可能にする．これによって，あるタスクで学習したダイナミクスモデルをほかのタスクのモデルに転移することが可能となる．

[*]奈良先端科学技術大学院大学 情報科学研究科 情報システム学専攻 博士論文，NAIST-IS-DD0861020, 2011 年 3月 17 日.

加えて，提案手法を実装するために，高度にモジュール化された強化学習ライブラリ SkyAI を開発し，オープンソースで公開する．

提案手法群を検証するため，多数の実験を行った．DCOB と WF-DCOB は，特にシミュレーションのヒューマノイドの運動学習において，従来手法を上回った．DCOB は実機蜘蛛型ロボットの匍匐タスクにおいても，優れたパフォーマンスを示した．ダイナミクスモデルの分離手法は，迷路タスクにおいて学習速度を改善した．WF-DCOB は優れたパフォーマンスを示したが，学習戦略フュージョンはさらにパフォーマンスを向上させた．学習戦略フュージョンの拡張性は，シミュレーションのヒューマノイドの迷路タスクにおいて検証された．このタスクでは，ロボットは，過去に学習した匍匐や旋回を用いて，ゴールまでの経路を学習した．

これらの実験を通して，本論文は，高度にモジュール化された学習システムが，目的に基づくタスク設計を実現する現実的なアプローチのひとつであると結論する．

キーワード

学習システム, 強化学習, モジュール化, 行動学習, ヒューマノイドロボット

Acknowledgements

First of all, I wish to thank Prof. Tsukasa Ogasawara. He supervised me during the five years of my master's and doctor's course. He did not only advise me about research but also supported my life as a researcher. In these past five years, I kept believing in my intuition (research vision), which was very hard to do. I faced many difficult situations, and to tell the truth, I considered giving up. One of the major factors that helped me to overcome the difficulties was the Ogasawara sensei's support.

I am deeply grateful to the other members of my thesis committee, Prof. Kazushi Ikeda and Prof. Jun Takamatsu, for taking time to review my thesis and giving helpful comments.

I have been supported by many professors and staff in the laboratories I have stayed in other than Prof. Ogasawara. Prof. Jun Takamatsu supervised me during my doctoral studies. His comments were insightful which indubitably improved my research. Prof. Mitsuo Kawato supervised me during my master's course. He is one of most brilliant researchers I have met. His sense about human intelligence affected my research so much. I thank Dr. Norikazu Sugimoto who also supervised me during my master's course. A special thanks goes to Dr. Albert Causo who is my friend and my English teacher. He reviewed my papers many times, and gave a lot of helpful comments.

I thank Prof. Sang-Ho Hyon who is my collaborator. We discussed about learning methodology of humanoid locomotion, which improved my research. I am indebted to the current/past staff in Robotics Laboratory, Dr. Kentaro Takemura, Dr. Yuichi Kurita, Prof. Yoshio Matsumoto, Prof. Etsuko Ueda, Prof. Jun Ueda, Dr. Jun'ichi Ido, Ms. Miyuki Yamaguchi, Ms. Michiyo Owaki, Ms. Megumi Kanaoka, and the staff in the Computational Neuroscience Laboratories of ATR, Prof. Yukiyasu Kamitani, Dr. Jun Morimoto, Dr. Masahiko Haruno, Ms. Kazuko Davis, Ms. Akiko Nonaka. I thank all of my friends in Robotics Laboratory, NAIST, and ATR. Dr. Takamitsu Matsubara gave me helpful comments during my master's course. Dr. Ding Ming, Dr. Tsuyoshi Suenaga, and Dr. Atsutoshi Ikeda were my reliable colleagues in Robotics Laboratory. I also would like to thank Prof. Takashi Matsuyama and Dr. Hiroaki Kawashima who supervised me during my bachelor's course. My life as a researcher started at Matsuyama Laboratory, and I feel their sense is still staying in my mind.

Finally, I thank my parents in heaven. They gave me a sound body and mind enough for research life, and supported me a long time. I dedicate this thesis to them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and Objective

Highly functional robots, typified by humanoid robots, that have been developed so far are still being improved. They are expected to play an important role in the future society; Kajita *et al.* forecast that by 2025, such future comes true (Kajita and Sugihara 2009). An essential ability for the robots is acquiring a behavior by itself only from the task objective. This ability enables the end-users to teach their wishes easily. Such an objective-based task design also improves the adaptability of robots to their environment.

The ultimate goal of this research is establishing a behavior acquisition mechanism equivalent to that of a human brain, and embedding it on robots. Machine learning provides a myriad of methods for this goal—especially, reinforcement learning (RL) is a core technology for the objective-based task design. Through interaction with the environment, an RL agent learns a behavior from the *reward* signal that represents the task objective. Many promising RL methods have been proposed and applied to robotics (e.g. (Kober and Peters 2009; Takahashi, Noma, and Asada 2008)).

The major issue of RL is its learning cost in large domains, which exponentially increases with respect to the task complexity, such as the degree-of-freedom of a robot. Naturally, tasks of humanoid robots are large domains. Thus, many researchers have tackled this issue and proposed effective methods. The following is a part of major solutions to improve RL (Figure 1.1).

Dimension reduction: reducing the dimensionality of a state-action space to accelerate the learning speed (Morimoto, Hyon, Atkeson, and Cheng 2008).

Hierarchical RL: decomposing a task into subtasks to make it easy to learn a complicated task (Barto and Mahadevan 2003; Morimoto and Doya 2001; Takahashi and Asada 2003).

Transfer learning: reusing knowledge that is learned in previous tasks (Torrey

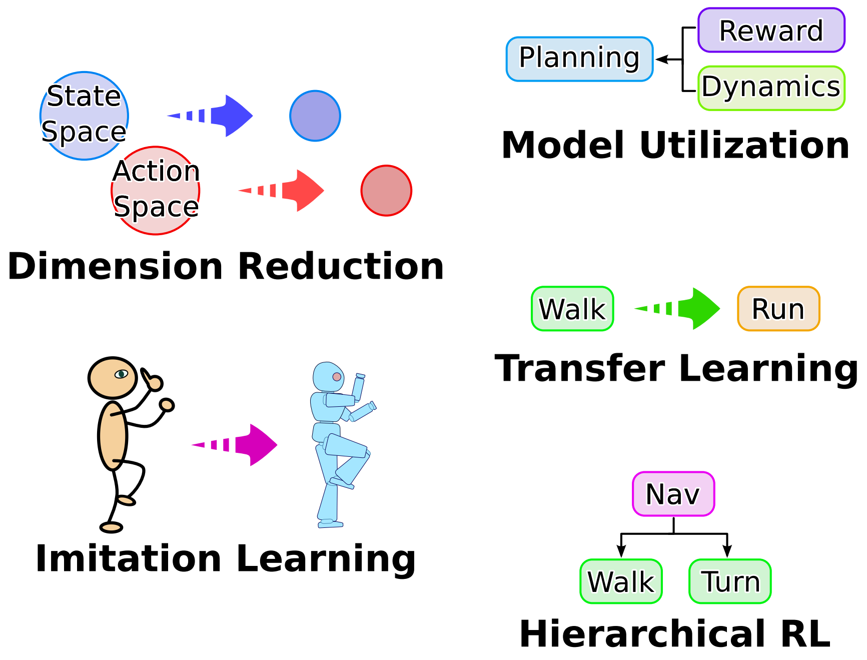


Figure 1.1 A part of major solutions to improve RL.

and Shavlik 2009; Zhang and Rössler 2004).

Model utilization: introducing models for model-based RL methods (Sutton, Szepesvári, Geramifard, and Bowling 2008; Sutton 1990; Farahmand, Shademan, Jägersand, and Szepesvári 2009).

Imitation learning: initializing the policy from a human demonstration (Peters, Vijayakumar, and Schaal 2003; Kober and Peters 2009).

We refer to such a method as a *learning strategy (LS)*.

These LSs improve RL, however, their effectiveness or applicability depends on a task. For instance, a dimension reduction by using a pattern generator may improve cyclic motions (e.g. (Nakamura, Mori, Sato, and Ishii 2007)); on the other hand, this method may restrict the capability to learn episodic motions such as jumping. The dependency of LSs on tasks means that the end-users should select a proper LS for each task. This selection may be difficult for ordinary users. Therefore, there are two possible approaches for the objective-based task design: (1) making a single LS that universally improves RL regardless of tasks, (2) making an algorithm to select proper LSs for each task from a set of predefined LSs.

This thesis utilizes the latter approach, which seems to be more feasible, and aims to develop a framework where the algorithms are modularized and automatically selected for each task. The thesis mainly treats motion learning tasks of robots, looking ahead to extend the framework for more general tasks.

1.2 Highly Modularized Learning System

This thesis tackles to develop:

> **"A highly modularized learning system that is suitable for the objective-based task design."**

Here, a *module* consists of an algorithm and its data. The purpose of modularization is to construct a learning agent with a set of modules. The proposed system aims to realize that *multiple* LSs are applied to each task of a single robot *multiple* times, where the ordering of the LSs is automatically decided.

For this purpose, the highly modularized learning system mainly consists of the following four elements:

Behavior module: deciding the behavior of a robot.

Fundamental module type: constructing behavior modules, such as a module of an RL method.

LS module: generating behavior modules from the fundamental module types according to its learning strategy.

UCB-Boltzmann selection method: choosing a behavior module actually used in each learning stage.

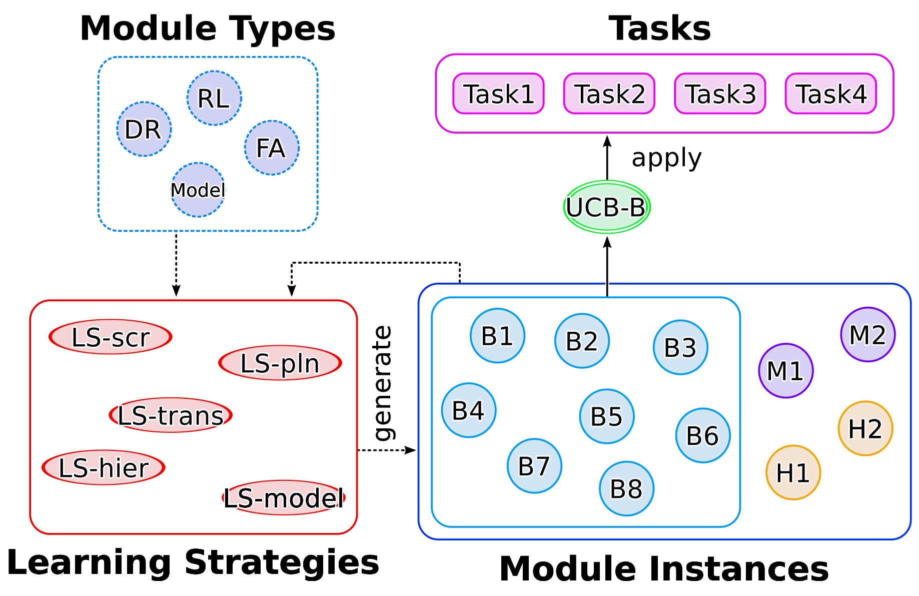


Figure 1.2 Overview of the highly modularized learning system. Every small circles, ellipses, and rectangles indicate modules. The abbreviations denote the following. RL: reinforcement learning, DR: dimension reduction, FA: function approximator, LS-scr: learning from scratch, LS-trans: transfer learning, LS-pln: planning, LS-hier: making a hierarchy, LS-model: making a model, B: behavior module, M: dynamics/reward model, H: hierarchical action space, UCB-B: UCB-Boltzmann selection method.

Figure 1.2 illustrates the overview of the system. The system maintains a set of module types. The LS modules generate behavior modules from the types (Module Instances). Each behavior module is used to decide a behavior of the robot and learn a task. Some of the LS modules generate behavior modules from existing behavior modules (e.g. a transfer LS). The system has multiple behavior modules for each task. Each behavior module has an upper confidence bound (UCB) as the evaluation value which is calculated from the reward observation. In each learning stage, a behavior module that actually decides the movement of the robot is chosen from the whole set of behavior modules by UCB-Boltzmann selection method. UCB-Boltzmann selection method is Boltzmann selection method with the UCB, which enables a probabilistic exploration. The selected behavior module is trained during the learning stage. Thus, each behavior module is generated through a sequence of LSs, and the selection of behavior modules signifies the selection of a proper LS sequence for the task. The core algorithm is referred to as *LS fusion* method.

1.2.1 Developers and Users

The system aims to enable ordinary users to design a task by its objective. Thus, it is desired that the users specify only the task. Since it is difficult to make a learning system common for every robot, we assume the roles of *developers* and *users* as follows:

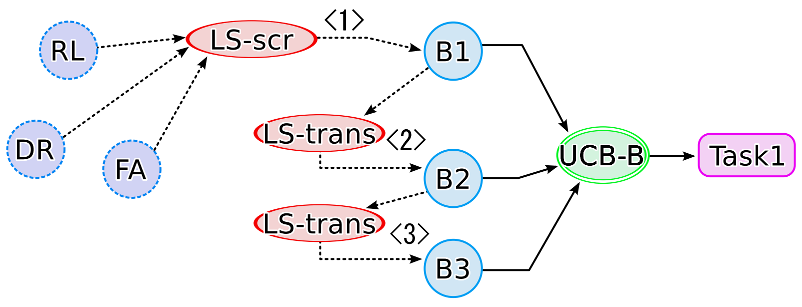
Developers: implement every module types and every LS modules, and tuning their parameters for the robot.

Users: only define tasks. Each task consists of a reward function and an episode definition.

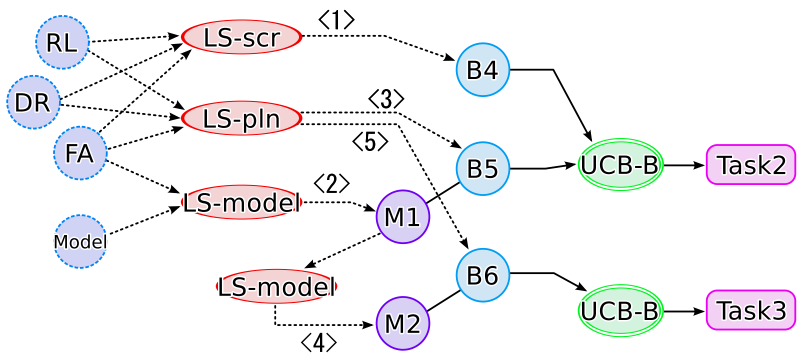
Sensors and actuators of the robot are hardly changed by the users. Thus, the robot-specific parameters, such as control gains and state-action space definitions, can be predefined by the developers. Behavior and the other modules are automatically generated by the system.

1.2.2 Examples of Learning Strategy Fusion

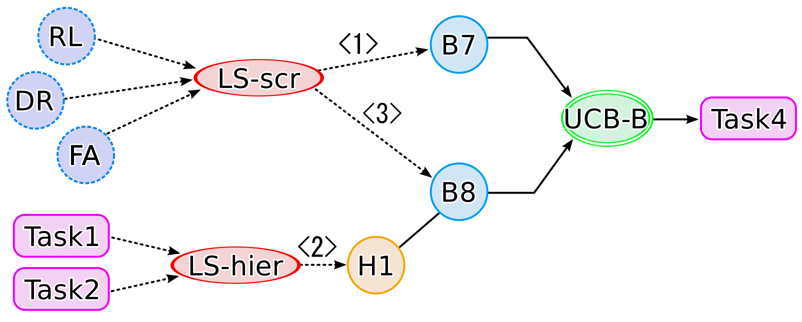
A key feature of LS fusion is that multiple LSs can be applied to a single task multiple times—moreover, a proper set of LSs and their ordering are automatically decided. Some conceptual examples are illustrated in Figure 1.3. In Figure 1.3(a), first, the task (e.g. walking) is learned by a behavior module B1 which is gener-



(a) Example of applying a transfer-learning strategy in several times.



(b) Example of reusing a model in learning the other task.



(c) Example of learning with a hierarchical task space.

Figure 1.3 Examples of LS fusion. ⟨*number*⟩ shows the ordering of generating modules (see Figure 1.2 for the abbreviations). Dotted arrow shows a generation, and solid arrow shows a connection; they are automatically performed by the system.

ated by a learning-from-scratch strategy (i.e. learning without prior knowledge). After some trial and error, B2 is generated from B1 by a transfer-learning strategy. Actual scenario is as follows: a slower walking motion B1 is learned first because of the ease of learning, then an accelerated behavior B2 is generated to walk faster. Similarly, B3 is generated. Note that the acceleration does not always improve the performance since the dynamics of the robot is not considered. Instead, UCB-Boltzmann selection method chooses a behavior module that has a higher possibility to perform well.

Figure 1.3(b) shows reusing a model in learning the other task. B4 is learned from scratch since learning without a model sometimes works better than learning with a model. A model M1 is trained from samples, and B5 updates its policy using M1. M1 can be learned using the samples obtained during B4's learning. Alternatively, B5 can use a Dyna architecture (Sutton, Szepesvári, Geramifard, and Bowling 2008) which enables to simultaneously execute an RL algorithm, training a model, and planning with the model. In learning Task3 after Task2, a model M2 is generated from M1 if they have common elements. For example, learning walking and learning turning can have a similar dynamics model if their state-action spaces are the same.

Figure 1.3(c) illustrates learning with a hierarchical architecture. A hierarchical action space H1 where some tasks are handled as subtasks is generated by LS-hier. Here, H1 assumes that the behavior modules for the subtasks are previously learned. A behavior module B8 is generated by a learning-from-scratch strategy where H1 is used as an action space. B7 uses an ordinary action space.

1.3 Contributions

Making the highly modularized learning system results the following technologies.

SkyAI: an open source software library of RL methods to implement the proposed system. The distinct feature is its modular architecture which realizes high execution-speed enough for real robot systems and high flexibility to design learning systems. SkyAI is designed so that the modular structure can be changed during execution.

DCOB: a method to generate a discrete action space from a set of basis functions given to approximate a value function. The distinct feature is its applicability to large domains. The name is derived from the fact that an action is a trajectory Directed to the Center Of a target Basis function. **WF-DCOB** is also developed as an extension of DCOB to search continuous actions around each discrete action of DCOB; the name comes from using wire-fitting (Baird and Klopf 1993). WF-DCOB is also designed to be suitable for the learning strategies.

Model decomposition: a method to decompose a dynamics model into task specific and task invariant elements. This method enables to transfer a dynamics model of a task to one of the other tasks.

LS Fusion: the core method for the highly modularized learning system. In addition, LSs are defined by using the proposed methods, WF-DCOB and the model decomposition.

Learning humanoid locomotion: research for applying the highly modularized learning system to learning locomotion by a human-size humanoid robot. A new learning scheme is studied where the robot is embedded with a primitive balancing controller during learning.

1.4 Thesis Outline

Figure 1.4 shows the outline of this thesis and the relation of the chapters. Chapter 2 describes preliminaries of this thesis, which includes a brief introduction to RL methods and definitions of benchmark tasks commonly used in this thesis. Chapter 3 introduces the RL library, SkyAI, which is used to implement the algorithms proposed in this thesis. Chapter 4 proposes DCOB and WF-DCOB, and Chapter 5 proposes a model decomposition method. These methods are used to define the LSs in Chapter 6. LS fusion is also proposed in Chapter 6. In Chapter 7, an application to learning locomotion by a human-size humanoid robot is demonstrated. Finally, Chapter 8 concludes the thesis.

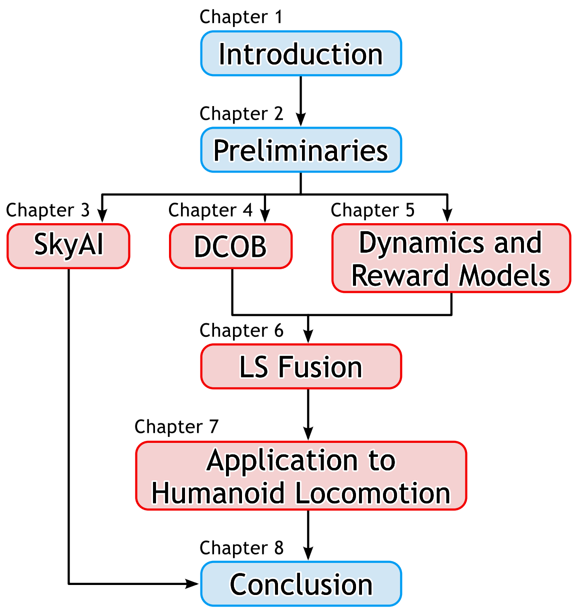


Figure 1.4 The outline of this thesis and the relation of the chapters.

Chapter 2

Preliminaries

This chapter describes the preliminaries of this thesis. Concretely, Section 2.1 introduces two major RL methods, Section 2.2 describes two function approximators used in this thesis, Section 2.3 provides three basis function allocation methods, and finally, Section 2.4 defines three benchmark tasks.

2.1 Reinforcement Learning

The purpose of RL is for a learning system (agent) whose input is a state $x_n \in \mathcal{X}$ and a reward $R_n \in \mathbb{R}$, and whose output is an action $u_n \in \mathcal{U}$, to acquire the policy $\pi(x_n) : \mathcal{X} \to \mathcal{U}$ that maximizes the expected discounted return $\mathbb{E}\left[\sum_{k=1}^{\infty} \gamma^{k-1} R_{n+k}\right]$ where $n \in \mathbb{N} = \{0, 1, \dots\}$ denotes the time step and $\gamma \in [0, 1)$ denotes a discount factor. In value-function-based RL algorithms, an action value function $Q(x, u) : \mathcal{X} \times \mathcal{U} \to \mathbb{R}$ is learned to represent the expected discounted return by taking an action u from a state x. Then, the optimal action rule is obtained from the greedy policy $\pi(x) = \arg\max_u Q(x, u)$.

Another approach to find the optimal policy is searching directly in the policy space. The Natural Actor Critic (NAC) (Peters, Vijayakumar, and Schaal 2003) is a typical example. However, this kind of approach strongly depends on the initial value of the policy parameter, especially in large domains. In our learning-from-scratch case, we assume that the value-function-based RL methods may obtain better policy since they can have richer policy parameterization. Thus, we test Peng's Q(λ)-learning algorithm (Peng and Williams 1994) and fitted Q iteration algorithm (Ernst, Geurts, and Wehenkel 2003; Ernst, Geurts, and Wehenkel 2005).

In general, fitted Q iteration requires appropriate samples, but it is difficult to obtain such samples with a random policy. In our preliminary experiments, though we applied fitted Q iteration in batch mode of small sizes, we were aware that Q(λ)-learning is better in the early stage of learning. A possible reason is that since we reduce the set of basis functions of a function approximator to improve the learning speed, the system loses the Markov property. Thus, we mainly use Q(λ)-learning, and apply fitted Q iteration to the same function approximator only with sample sequences of higher return.

2.1.1 Peng's Q(λ)-learning

The Peng's Q(λ)-learning algorithm (Peng and Williams 1994) is an on-line RL method. The update procedure for a generic function approximator $Q(x,u)$ of the parameter $\theta \in \Theta$ is written as follows:

$$e_n = R_n + \gamma V_n(x_{n+1}) - V_n(x_n), \tag{2.1a}$$

$$e'_n = R_n + \gamma V_n(x_{n+1}) - Q_n(x_n, u_n), \tag{2.1b}$$

$$\theta_{n+1} = \theta_n + \alpha e_n Tr_n + \alpha e'_n \nabla_\theta Q_n(x_n, u_n), \tag{2.1c}$$

$$Tr_{n+1} = (\gamma\lambda)(Tr_n + \nabla_\theta Q_n(x_n, u_n)), \tag{2.1d}$$

where $\nabla_\theta Q(x,u)$ denotes the derivative of $Q(x,u)$ w.r.t. the parameter θ, α denotes a step-size parameter, Tr denotes the eligibility trace ($Tr_0 = \mathbf{0} \in \Theta$), and $V_n(x) \triangleq \max_u Q_n(x,u)$. This update procedure is applied after each action.

In order to make learning stable, *replacing trace* (Singh and Sutton 1996) is applied after each update of the eligibility trace. Note that replacing trace is effective only for a linear action value function and applicable BFs are limited as Tsitsiklis *et al.* pointed out (Tsitsiklis and Roy 1997). Thus, in this thesis, replacing trace is applied to only Q(λ)-learning with a linear function approximators.

2.1.2 Fitted Q Iteration

The fitted Q iteration algorithm (Ernst, Geurts, and Wehenkel 2003; Ernst, Geurts, and Wehenkel 2005) is a batch mode RL method to learn from sample trajectories whose element is a four-tuple $F_n = (x_n, u_n, x_{n+1}, R_n)$. The idea of fitted Q iteration is as follows: first, we build a training set from the current function approximator and a set of four tuples. Then, we train the next function approximator with the training set by a supervised learning method. Iterating these two steps, the function approximator will converge to the action value function.

The action value function approximator Q_0 is initialized so that $Q_0(x,u) = 0$ for all $(x,u) \in \mathcal{X} \times \mathcal{U}$. In the N-th iteration, the training set $\{i_n, o_n\}$ is built from the current function approximator Q_{N-1} and the set of four tuples $\{F_n\}$ by

$$i_n = (x_n, u_n), \tag{2.2}$$

$$o_n = R_n + \gamma \max_u Q_{N-1}(x_{n+1}, u). \tag{2.3}$$

Then, Q_N is trained with $\{i_n, o_n\}$. We implement this supervised learning with a gradient descent for the least squares.

As mentioned above, we combine fitted Q iteration and Q(λ)-learning. At the end of each action, we apply the update rule of Q(λ)-learning and store the sample. At the end of every N_{FQI} episodes, we execute an iteration of fitted Q iteration with the samples in the top N_{smpl} episodes ranked by the return of the episode.

2.2 Function Approximators

Next, we describe some function approximators for the action value functions $Q(x, u)$. For a continuous state space $\mathcal{X}$ and a discrete action space $\mathcal{U}$, we use a linear function approximator because of its stability. If both the state and the action spaces are continuous, we employ wire-fitting (Baird and Klopf 1993). The notable feature of wire-fitting is that we can maximize $Q(x, u)$ w.r.t. u by evaluating only on a fixed number of points.

2.2.1 Linear Function Approximator (LFA) with NGnet

For a continuous state $x \in \mathcal{X}$ and a discrete action $u \in \mathcal{U}$, we let $Q(x, u) = \theta_u^\top \phi(x)$, where $\phi(x) = (\phi_1(x), \ldots, \phi_{|\mathcal{K}|}(x))^\top$ denotes the feature vector of a state x, $\mathcal{K} = \{\phi_k \mid k = 1, 2, ..\}$ denotes a set of basis functions, and $\theta_u \in \mathbb{R}^{|\mathcal{K}| \times 1}$ denotes a parameter related to an action u. The parameter vector is defined as $\theta = (\theta_1^\top, \ldots, \theta_{|\mathcal{U}|}^\top)^\top \in \mathbb{R}^{|\mathcal{K}||\mathcal{U}| \times 1}$. The derivative of the $Q(x, u)$ w.r.t. θ is given by $\nabla_\theta Q_n(x, u) = (\delta_{1u}\phi_1^\top, \ldots, \delta_{|\mathcal{U}|u}\phi_{|\mathcal{U}|}^\top)^\top$ where δ denotes the Kronecker's delta. Note that in the learning-from-scratch case, the parameter θ is initialized by zero.

As basis functions, we use Normalized Gaussian Network (NGnet) (Sato and Ishii 2000) which is sometimes used as the basis functions of function approximators in RL applications (Morimoto and Doya 2001). In NGnet, $\phi_k(x)$ is given by

$$\phi_k(x) = \frac{\mathrm{G}(x; \mu_k, \Sigma_k)}{\sum_{k' \in \mathcal{K}} \mathrm{G}(x; \mu_{k'}, \Sigma_{k'})}, \tag{2.4}$$

where $\mathrm{G}(x;\mu,\Sigma)$ denotes a Gaussian with mean μ and covariance matrix Σ. In the case of a linear function approximator, $\mathcal{K}$ is predefined and $\{\mu_k, \Sigma_k \mid k \in \mathcal{K}\}$ are treated as fixed parameters.

In the following, we refer to the linear function approximator with NGnet as *LFA-NGnet*.

Action Selection for LFA-NGnet

As an exploration policy, we introduce Boltzmann selection method (Sutton and Barto 1998). An action $u \in \mathcal{U}$ is selected with the probability

$$\pi(u|x) = \frac{\exp(\frac{1}{\tau}Q(x,u))}{\sum_{u'\in\mathcal{U}}\exp(\frac{1}{\tau}Q(x,u'))}, \tag{2.5}$$

where τ denotes a temperature parameter. Letting $\tau = 0$ gives the greedy policy.

2.2.2 Wire-Fitting

For a continuous state $x \in \mathcal{X}$ and a continuous action $u \in \mathcal{U}$, wire-fitting is defined as:

$$Q(x,u) = \lim_{\epsilon\to 0^+} \frac{\sum_{i\in\mathcal{W}}(d_i+\epsilon)^{-1}q_i(x)}{\sum_{i\in\mathcal{W}}(d_i+\epsilon)^{-1}}, \tag{2.6}$$

$$d_i = \|u - u_i(x)\|^2 + C\big[\max_{i'\in\mathcal{W}}(q_{i'}(x)) - q_i(x)\big]. \tag{2.7}$$

Here, a pair of the functions $q_i(x) : \mathcal{X} \to \mathbb{R}$ and $u_i(x) : \mathcal{X} \to \mathcal{U}$ $(i \in \mathcal{W})$ is called a *control wire*; wire-fitting is regarded as an interpolator of the set of control wires $\mathcal{W}$. C is the smoothing factor of the interpolation; we choose $C = 0.001$ in the experiments. Any function approximator is available for $q_i(x)$ and $u_i(x)$. Regardless of the kind of the function approximators, one of $q_i(x), i \in \mathcal{W}$ is equal to $\max_u Q(x,u)$ and the corresponding $u_i(x)$ is the greedy action at x.

We use NGnet for $q_i(x)$ and a constant vector for $u_i(x)$, that is, we let $q_i(x) = \theta_i^\top \phi(x)$ and $u_i(x) = U_i$, where $\phi(x)$ is the feature vector of the NGnet. The parameter vector θ is defined as $\theta^\top = (\theta_1^\top, U_1^\top, \theta_2^\top, U_2^\top, \dots, \theta_{|\mathcal{W}|}^\top, U_{|\mathcal{W}|}^\top)$, and the gradient $\nabla_\theta Q(x,u)$ can be calculated analytically.

Parameter Initialization and Constraints

In the learning-from-scratch case, $\{\theta_i | i \in \mathcal{W}\}$ are initialized by zero. For $\{U_i | i \in \mathcal{W}\}$, a typical initialization method is assigning random values, but this initialization sometimes leads to undesirable local maxima. Instead, we initialize them on a Grid over a command space. During learning, each U_i is constrained around the corresponding point on the Grid, which improves the learning stability.

Action Selection for Wire-Fitting

As an exploration policy in using wire-fitting, we use the Boltzmann-like selection method. A control wire i is considered to be a discrete action whose action value is $q_i(x)$, and one of the control wires is chosen by Boltzmann selection. Then the corresponding $u_i(x)$ is the selected action. Namely, a control wire $i \in \mathcal{W}$ is selected by the probability

$$\pi(i|x) = \frac{\exp(\frac{1}{\tau}q_i(x))}{\sum_{i' \in \mathcal{W}} \exp(\frac{1}{\tau}q_{i'}(x))}, \tag{2.8}$$

where τ denotes a temperature parameter. Then the corresponding $u_i(x)$ is the selected action. We refer to this method as WF-Boltzmann. The same as Boltzmann selection, letting $\tau = 0$ gives the greedy policy.

2.3 Basis Function Allocation

Allocating BFs is a major factor to determine the learning performance, especially in large domains. This section introduces three allocation methods: grid allocation, spring-damper allocation, and dynamics-based allocation. The grid allocation requires an exponential number of BFs with respect to the dimensionality of a state-action space, while the others can choose the number of BFs.

2.3.1 Grid Allocation

This method allocates BFs on a grid where each dimension of the state-action space is divided independently. Grid allocation is widely used (e.g. (Matsubara, Morimoto, Nakanishi, Hyon, Hale, and Cheng 2007)) for ease of use. However, since the number of BFs increases exponentially w.r.t. the dimensionality of the state-action space, applying this allocation method to large domains is difficult.

2.3.2 Spring-Damper Allocation

This method allocates a given number of BFs so that they spread as widely as possible. Since this method can choose the number of BFs, applying it to large domains is easier than that of grid allocation. In spring-damper allocation, first, we allocate BFs randomly. The covariance matrix of each BF is constrained to $\Sigma = \sigma^2\mathbf{1}$ where $\mathbf{1}$ is a unit matrix. Then, they are re-arranged so that the centers of the BFs spread as widely as possible and σ becomes as large as possible without overlapping. The detailed algorithm is described in Appendix A.

2.3.3 Dynamics-Based Allocation

The dynamics-based allocation is similar to the spring-damper allocation, that is, we first choose the number of BFs. However, the dynamics-based method allocates BFs according to the dynamics of the robot, which may make the allocation better than that of spring-damper method.

Let us remember that we do not have a dynamics model of the robot. In the dynamics-based allocation, we construct a function approximator of given number of BFs as the dynamics model of the robot. Specifically, we first obtain a data set $\{x, \tilde{u} | x \in \mathcal{X}, \tilde{u} \in \tilde{\mathcal{U}}\}$ from random motions where $\mathcal{X}$ is a state space, and $\tilde{\mathcal{U}}$ is a control command space (e.g. torque space). Then, we train the function approximator

$$\dot{x} = \sum_{k \in \mathcal{K}} \left(A_k \begin{pmatrix} x \\ \tilde{u} \end{pmatrix} + b_k \right) \phi_k(x; \mu_k, \Sigma_k) \tag{2.9}$$

with the data set, where $\{A_k, b_k, \mu_k, \Sigma_k | k \in \mathcal{K}\}$ are the parameters of the approximator, $\mathcal{K}$ is a set of BFs of the given number, and ϕ_k is a normalized Gaussian (eq. (2.4)). The parameters are trained by EM algorithm with *unit manipulations*[1] (Sato and Ishii 2000). The obtained BFs $\{\mu_k, \Sigma_k | k \in \mathcal{K}'\}$ are used for RL where $\mathcal{K}'$ denotes a new set of BFs.

The advantage of this allocation is explained as follows. In general, a higher resolution policy is required in state regions of higher nonlinear dynamics, while a lower resolution policy is enough in state regions of near linear dynamics. On the other hand, by using the dynamics-based allocation, more BFs are allocated over state regions of higher nonlinear dynamics, while less BFs are allocated over state regions of near linear dynamics. Thus, this allocation is considered to be suitable to represent a policy. Nevertheless, since the policy also depends on the reward function, this allocation is not the best, but proper when allocating without task information. This method is based on the idea of MOSAIC model (Wolpert and Kawato 1998; Doya, Samejima, Katagiri, and Kawato 2002).

[1]Actually, unit division and unit deletion are implemented.

2.4 Benchmark Tasks

This section describes some benchmark tasks to test the proposed methods. In the following tasks, the reward function $r(t)$ is calculated at each time step t rather than at each action. The purpose is to compare different action sets evenly since they have different durations. The reward for an action is obtained by summing $r(t)$ during the action.

2.4.1 Task Maze2D

This task is a navigation task of an omniwheel mobile robot. The robot can move in any direction on a 2-dimensional plane (x_1, x_2), $x_1, x_2 \in [-1, 1]$ (Figure 2.1). This task is performed in simulation. The state of the robot is its global position which is expressed as

$$x = (x_1, x_2)^\top, \tag{2.10}$$

and its control input is the state transition in a time step $\delta\mathrm{t} = 0.01$ which is expressed as

$$\tilde{u} = (\Delta x_1, \Delta x_2)^\top. \tag{2.11}$$

In this environment, there is some *wind* that changes the behavior of the robot in the direction of the arrows as shown in Figure 2.1. There are also *walls* which the robot can not pass through. There are four types of mazes, referred to as easy1, easy2, middle, and hard, as shown in Figure 2.2. The specific calculation of the dynamics of the environment is described in Appendix B.

The objective of the navigation task is to acquire a path from the *start* to the *goal*. According to this objective, the reward function is designed by,

$$r(t) = r_\mathrm{g}(t) - r_\mathrm{sc}(t) - r_\mathrm{os}(t) \tag{2.12}$$

$$r_\mathrm{g}(t) = \begin{cases} 1 & (\|x_\mathrm{g} - x'\| < \rho_\mathrm{g}) \\ 0 & (\text{otherwise}) \end{cases} \tag{2.13}$$

$$r_\mathrm{sc}(t) = 25\|\tilde{u}(t)\|^2\delta\mathrm{t} \tag{2.14}$$

$$r_\mathrm{os}(t) = \begin{cases} 0.5 & (x' \notin \mathcal{X}_\mathrm{pl}) \\ 0 & (\text{otherwise}) \end{cases} \tag{2.15}$$

where $x' = x(t + \delta\mathrm{t})$, $\mathcal{X}_\mathrm{pl} = \{(x_1, x_2) \mid x_1 \in [-1, 1], x_2 \in [-1, 1]\}$. $r_\mathrm{g}(t)$ is the reward for getting closer to the goal, $r_\mathrm{sc}(t)$ is the step cost, $r_\mathrm{os}(t)$ is the penalty

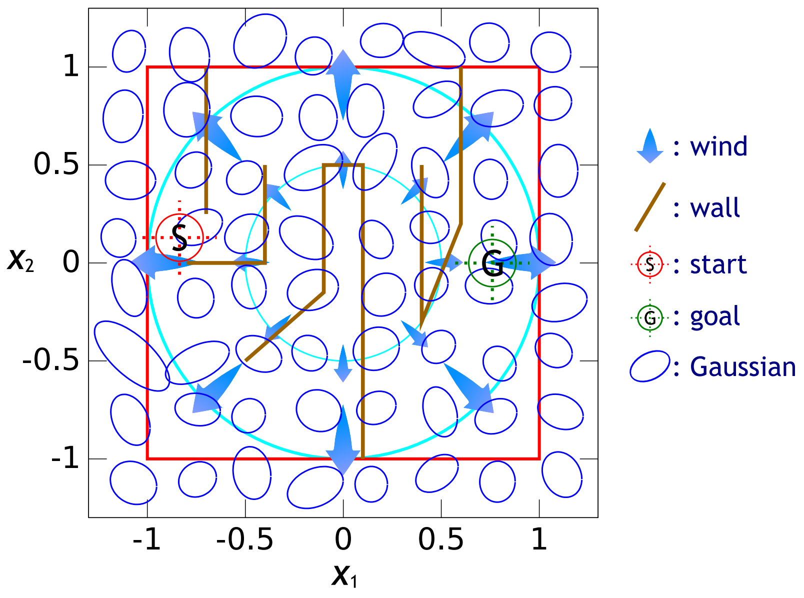


Figure 2.1 Environment of the robot navigation task.

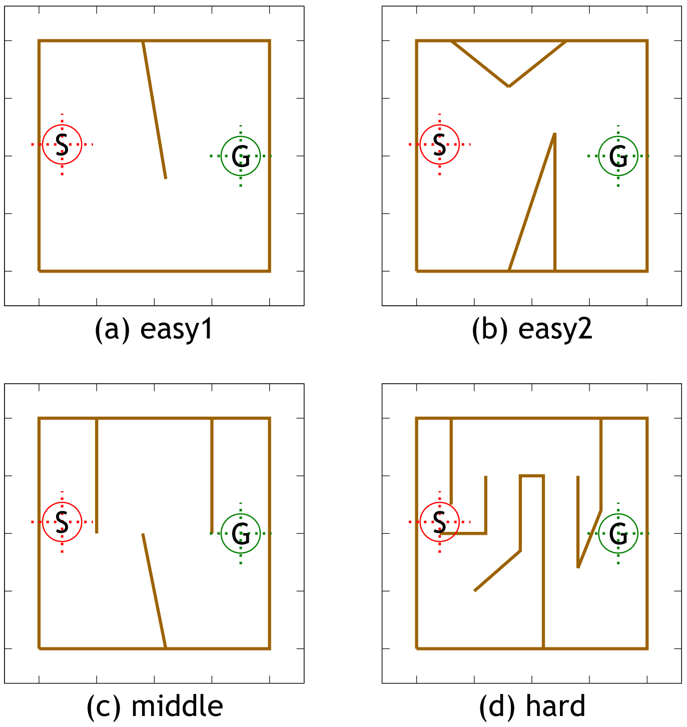


Figure 2.2 Types of mazes.

for going out of the plane. x_{g} denotes the *goal* state, and $\rho_{\mathrm{g}} = 0.15$ denotes the radius of the *goal*. Note that the goal reward and the out-of-plane penalty are given once in each episode. Each episode begins with the *start* state $x(0) = x_{\mathrm{s}}$, and ends if the robot has reached the *goal*, gone outside ($x' \notin \mathcal{X}_{\mathrm{pl}}$), or $t > 12$.

We use NGnet with 64 BFs allocated as shown in Figure 2.1 to approximate the action value function. These BFs are allocated on a 8×8 grid with added random noise to each center and covariance.

2.4.2 Task HumanoidML-crawling, turning

HumanoidML is a motion learning task of a humanoid robot. Experiments are performed in simulation using a dynamics simulator ODE (Open Dynamics Engine (Smith 2006)). Figure 2.3 shows the simulation model. Its height is 0.328m. It weights 1.20kg. Each joint torque is limited to 1.03Nm, and a PD-controller is embedded on it. The dynamics simulation is calculated with a time step $\delta t = 0.2[\mathrm{ms}]$. A crawling and a turning task are performed with this robot.

The whole-body state x_{w} of the robot consists of the global position (c_{0x}, c_{0y}, c_{0z}) (the center-of-mass of the body link), the global orientation in quaternion (q_w, q_x, q_y, q_z), their velocities $(\dot{c}_{0x}, \dot{c}_{0y}, \dot{c}_{0z}, \omega_x, \omega_y, \omega_z)$, joint angles $(q_0, \dots, q_{16})$, and joint angular velocities $(\dot{q}_0, \dots, \dot{q}_{16})$. Thus, the whole-body state x_{w} is a 47-dimensional vector. The corresponding control command input $\tilde{u}_{\mathrm{w}}$ is the target joint angles, which is denoted as $(q_0^{\mathrm{trg}}, \dots, q_{16}^{\mathrm{trg}})$. Note that we can also directly control the joint torques.

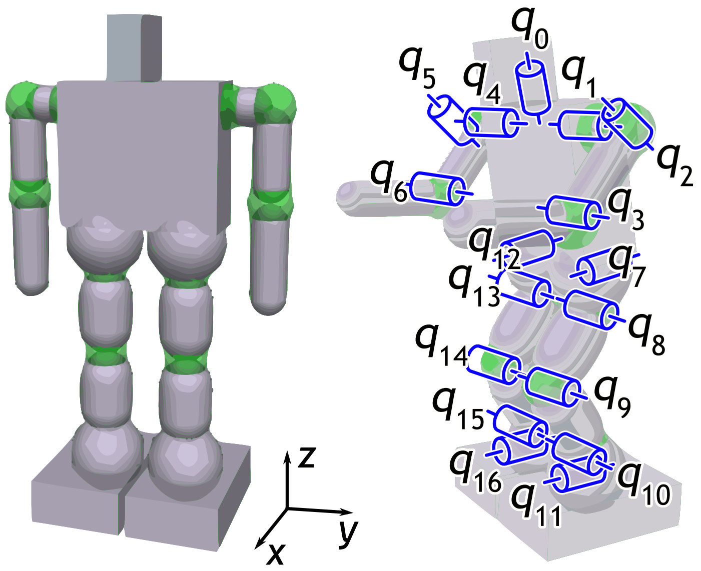


Figure 2.3 Simulation model of a humanoid robot.

DoF Configurations

For RL, it is difficult to directly handle all 17-DoFs. Thus, some reduced DoF configurations are defined. This dimension reduction is performed by defining constant matrices $C_{\mathcal{X}}$ and $C_{\tilde{\mathcal{U}}}$ such that $x = C_{\mathcal{X}} x_{\mathrm{w}}$, $\tilde{u}_{\mathrm{w}} = C_{\tilde{\mathcal{U}}} \tilde{u}$ where $x \in \mathcal{X}$, $x_{\mathrm{w}} \in \mathcal{X}_{\mathrm{w}}$, $\tilde{u} \in \tilde{\mathcal{U}}$, and $\tilde{u}_{\mathrm{w}} \in \tilde{\mathcal{U}}_{\mathrm{w}}$. The pair $(\mathcal{X}_{\mathrm{w}}, \tilde{\mathcal{U}}_{\mathrm{w}})$ denotes the overall (whole-body) state-command space. The pair $(\mathcal{X}, \tilde{\mathcal{U}})$ denotes the reduced state-command space.

The following are the prepared DoF configurations (Figure 2.4).

3-DoF ($\mathcal{X}_3, \tilde{\mathcal{U}}_3$) : Joint pairs $\{q_1, q_3, q_4, q_6\}$, $\{q_8, q_{13}\}$, $\{q_9, q_{10}, q_{14}, q_{15}\}$ are coupled respectively, which gives a bilateral symmetry. The default BFs are allocated on a $5 \times 5 \times 5$ grid over the reduced joint angle space.

4-DoF ($\mathcal{X}_4, \tilde{\mathcal{U}}_4$) : Joint pairs $\{q_1, q_3\}$, $\{q_4, q_6\}$, $\{q_8, q_9, q_{10}\}$, $\{q_{13}, q_{14}, q_{15}\}$ are coupled respectively, which means a single DoF for each leg. The default BFs are allocated on a $4 \times 4 \times 4 \times 4$ grid over the reduced joint angle space.

5-DoF ($\mathcal{X}_5, \tilde{\mathcal{U}}_5$) : Joint pairs $\{q_1, q_4\}$, $\{q_3, q_6\}$, $\{q_8, q_{13}\}$, $\{q_9, q_{14}\}$, $\{q_{10}, q_{15}\}$ are coupled respectively, which gives a bilateral symmetry. The default BFs are allocated by the dynamics-based allocation method over the global state space and the reduced joint angle/angular velocity space. Specifically, 202 BFs are allocated.

6-DoF ($\mathcal{X}_6, \tilde{\mathcal{U}}_6$) : A coupled joint pair $\{q_7, q_{12}\}$ is added to the 5-DoF, which also gives a bilateral symmetry. The default BFs are allocated by the spring-damper allocation method over the reduced joint angle space. Specifically, 300 BFs are allocated.

7-DoF ($\mathcal{X}_7, \tilde{\mathcal{U}}_7$) : A coupled joint pair $\{q_2, q_5\}$ is added to the 6-DoF, which also gives a bilateral symmetry. The default BFs are allocated by the spring-damper allocation method over the reduced joint angle space. Specifically, 600 BFs are allocated.

16-DoF ($\mathcal{X}_{16}, \tilde{\mathcal{U}}_{16}$) : Only the head link is fixed, while the other joints can move independently. The default BFs are not allocated on $\mathcal{X}_{16}$.

For the N_{D}-DoF configuration, let $\mathbf{q}_{N_{\mathrm{D}}}$ denote the reduced joint angle vector. The command input space of N_{D}-DoF configuration is a target value of $\mathbf{q}_{N_{\mathrm{D}}}$ or the corresponding torques.

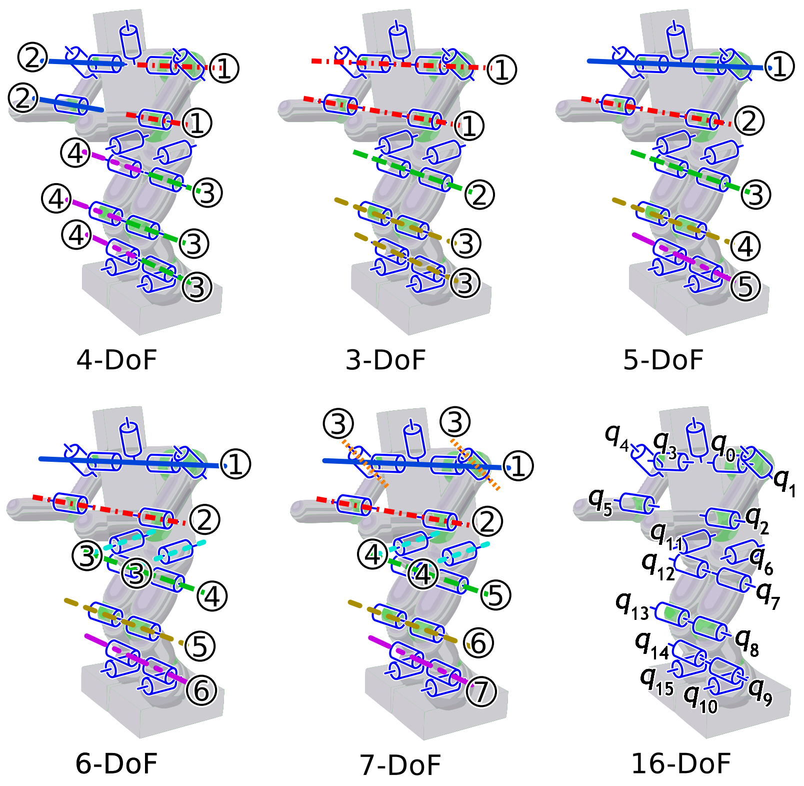


Figure 2.4 DoF configurations of the humanoid robot.

Task HumanoidML-crawling

The objective of the crawling task is to move forward along the x-axis as far as possible. According to the objective, the reward is designed as follows:

$$r(t) = r_{\mathrm{mv}}(t) - r_{\mathrm{sc}}(t) - r_{\mathrm{fd}}(t) \tag{2.16}$$

$$r_{\mathrm{mv}}(t) = 50\dot{c}_{0x}(t) \tag{2.17}$$

$$r_{\mathrm{sc}}(t) = 2 \times 10^{-5}\|\tilde{u}(t)\| \tag{2.18}$$

where $r_{\mathrm{mv}}(t)$ is the reward for forward movement, $r_{\mathrm{sc}}(t)$ is the step cost, $r_{\mathrm{fd}}(t)$ is the penalty for falling down. $r_{\mathrm{fd}}(t)$ takes 4 if the body or the head link touches the ground, otherwise it takes 0. The penalty for falling down is given once in each action. Each episode begins with the initial state where the robot is standing up and stationary, and ends if $t > 20[\mathrm{s}]$ or the sum of reward is less than -40.

Task HumanoidML-turning

The objective of the turning task is to turn around the z-axis as fast as possible. According to the objective, the reward is designed as follows:

$$r(t) = r_{\mathrm{tn}}(t) - r_{\mathrm{sc}}(t) - r_{\mathrm{fd}}(t) \tag{2.19}$$

$$r_{\mathrm{tn}}(t) = 2.5\omega_z(t) \tag{2.20}$$

$$r_{\mathrm{sc}}(t) = 2 \times 10^{-5}\|\tilde{u}(t)\| + 0.5\|(\dot{c}_{0x}, \dot{c}_{0y})\| \tag{2.21}$$

where $r_{\mathrm{tn}}(t)$ is the reward for turning, $r_{\mathrm{sc}}(t)$ is the step cost for the torque usage and the $x - y$ global movement, $r_{\mathrm{fd}}(t)$ is the penalty for falling down. $r_{\mathrm{fd}}(t)$ takes 4 if the body link touches the ground, takes 0.1 if the head link touches the ground; otherwise it takes 0. $r_{\mathrm{fd}}(t)$ is given once in each action. Each episode begins with the initial state where the robot is standing up and stationary, and ends if $t > 20$[s] or the sum of reward is less than -40.

2.4.3 Task BioloidML-crawling

BioloidML is a motion learning task of an actual small robot, Bioloid, made by ROBOTIS co. We use a King Spider model of Bioloid as shown in Figure 2.5. A crawling task is performed with this robot.

The whole-body state x_{w} of the robot consists of only the joint angles $(q_1, \ldots, q_{18})$, specifically, it does not include the global position and orientation. The absence of these observations may break the Markov property of the task. The

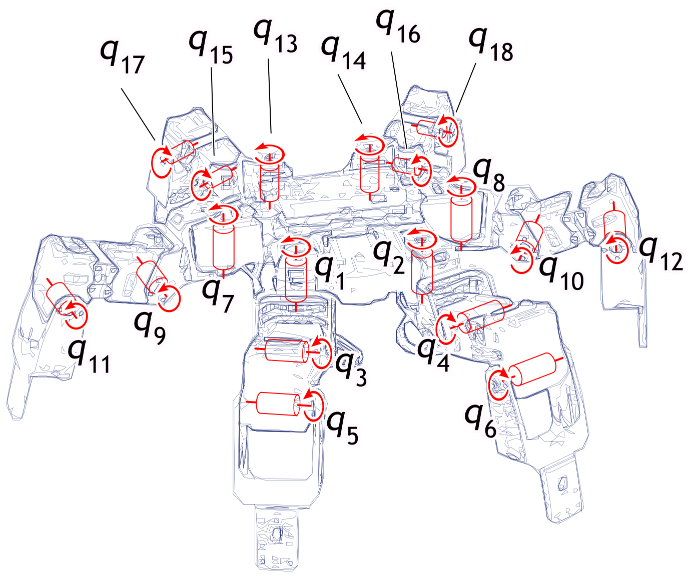


Figure 2.5 King Spider (ROBOTIS Bioloid) which has 18 DoF.

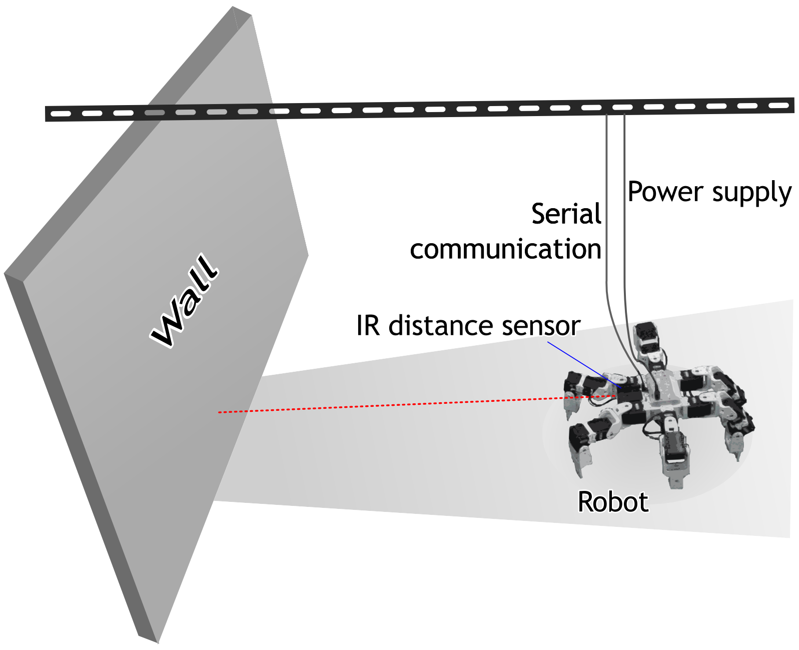


Figure 2.6 Setup of experimental environment.

reasons for these absences are that (1) using the Bioloid product as it is makes verification experiments easy for the other researchers, and (2) we can verify the applicability to POMDPs (Partially Observable Markov Decision Processes). Thus, the whole-body state x_{w} is a 18-dimensional vector. The corresponding control command input $\tilde{u}_{\mathrm{w}}$ is the target joint angles, which is denoted as $(q_1^{\mathrm{trg}}, \ldots, q_{18}^{\mathrm{trg}})$.

As similar to the tasks HumanoidML, a reduced DoF configuration is defined as follows.

5-DoF ($\mathcal{X}_5$, $\tilde{\mathcal{U}}_5$) : Joint pairs $\{q_1, q_2\}$, $\{q_3, q_4, q_5, q_6\}$, $\{q_7, q_8\}$, $\{q_9, q_{10}, q_{11}, q_{12}\}$, $\{q_{15}, q_{16}\}$ are coupled respectively, which gives a bilateral symmetry. The other joints are fixed to a neutral angle; $q_{13} = q_{14} = q_{17} = q_{18} = 0$. The default BFs are allocated on a $3 \times 3 \times 3 \times 3 \times 3$ grid over the reduced joint angle space.

Sensors

Other than the joint angle sensors, we use an infrared ray (IR) sensor to observe the distance from the robot to an obstacle. Let d_{IR} denote the distance processed by a low-pass filter.

Experimental Environment

The experimental environment is configured as shown in Figure 2.6. The robot is put in front of a wall; the IR sensor measures the distance from the robot to the wall. The robot is connected to a computer[2] and communicates with it through a serial protocol. In each $\delta\mathrm{t} = 0.1[\mathrm{s}]$, the target joint angles $\tilde{u}$ are sent to the robot.

Task BioloidML-crawling

The objective of the crawling task is to move forward as far as possible, whose reward is designed as follows:

$$r(t) = v_{\mathrm{IR}}(t) - 0.15, \tag{2.22}$$

where $v_{\mathrm{IR}}(t)$ denotes the velocity of the robot calculated from $d_{\mathrm{IR}}(t)$. Thus, in the summation of the robot (return), the $v_{\mathrm{IR}}(t)$ term indicates a moving distance and the constant (0.15) term denotes a penalty for elapsed time. Each episode begins with an initial pose ($q_1 = -q_2 = -\pi/9, q_{3,\dots,18} = 0$), and ends if $t > 50[\mathrm{s}]$, the robot touches the wall (determined by the operator), or some problems arise.

[2]A laptop PC: Pentium M, 2[GHz], 512[MB] RAM, Debian Linux.

Chapter 3

SkyAI : Highly Modularized RL Library

This chapter introduces a software library of reinforcement learning (RL) methods, named SkyAI. This library is a highly modularized RL library for real and simulated robots to learn behaviors. SkyAI aims to provide an implementation of the proposed methods in this thesis, especially the learning strategy fusion method. For this purpose, SkyAI realizes two conflicting features: high execution-speed enough for real robot systems and high flexibility to design learning systems. This chapter describes the concepts, the requirements, and the current implementation of SkyAI.

3.1 Introduction

As described in Chapter 1, multiple learning strategies (LSs) are applied to a single task multiple times. Namely, the structure of the modules is modified in runtime. Thus, an implementation of the LSs and the modules is required to be highly flexible to modify the modular structure in addition to high execution-speed, enough for real robot systems.

Many of existing software libraries of RL are written in script languages, whose execution-speed is relatively slow. Some existing libraries written in compiler languages do not have flexibility.

Thus, a compact middleware is developed to realize high flexibility and high execution-speed. RL algorithms and the proposed methods in this thesis are implemented as modules on the middleware. Concretely, the middleware and every modules are written in C++, and a script interface is provided to modify the modular structure in runtime. The implemented software library is an open source software library released as SkyAI.

In the rest of this chapter, Section 3.2 describes the overview. In Section 3.3, we discuss about the related works. In Section 3.4, an experiment of speed comparison with the other implementation styles is demonstrated. Finally, Section 3.5

concludes this chapter.

3.2 Overview

This section describes the principal concepts of SkyAI, the requirements basing on the concepts, the solutions, and an overview of a system with SkyAI.

3.2.1 Principal Concepts

High modularization

The approach of the SkyAI is *modularization* of the RL or the other machine-learning algorithms. Modular architecture enables the following features:

High extensibility: Modular architecture makes it easy to create a new module by inheriting the other module. Adding new functions, or specializing some functions are realized with a little modification. Thus, the library can be highly extensible.

High reusability of implementation: Modular architecture can separate a task (problem) specific implementation into modules. Typical examples are reward modules and low-level robot controller. In addition, we can make generic, i.e. highly reusable, modules.

High reusability of learned knowledge: Modular architecture can also enhance the reusability of learned knowledge, such as a learned policy by an RL algorithm, a dynamics model, and a reward model.

High execution-speed and high flexibility

SkyAI must be executed in high speed, and should also be highly flexible. These are very important features for SkyAI to be applied to real robot systems. Real robot systems require high-speed execution. On the other hand, we need high flexibility like script languages. Generally, these two features are conflicting.

Developer friendly

Highly-modularized architecture has many benefits as mentioned above, however, it sometimes makes development difficult. SkyAI pursues a developer-friendly implementation to boost the participation of many researchers and developers.

3.2.2 Requirements and Solutions

Writing in a compiler language

To achieve the high execution-speed, SkyAI should be written in a compiler language. We choose C++. In general, a C++ source code is compiled to an executable code whose execution speed is almost of the highest level. This is very suitable for real robot systems.

Each module is implemented as a class of C++. Generally, communication between classes is performed by member functions[1]. We basically use *call-by-reference* for the functions, which enables high-speed communication.

Once compiled, reconfigurable infinitely

Once a C++ source code is compiled, it is difficult to modify its scheme. Thus, SkyAI *wraps the C++ class system* and *provides a script interface* so that the modular structure can be changed by the script after compiling the source code.

To change the modular structure dynamically, the member functions for the communication between classes are needed to be connected and disconnected. Thus, each member function is encapsulated as a *port* class. Each module can have any number of ports. Ports can be connected and disconnected at any time in execution, which enables the reconfiguration of the modular structure.

A script language is defined to provide an interface of modular manipulations during execution. Specifically, instantiating modules, connecting ports, and setting parameters of the modules (e.g. a learning rate) can be described in the script language.

Using standard preprocessor and compiler

To make a module program compatible with the modular architecture, the program should follow some rules and regulations of SkyAI. Some similar software platforms, such as (Ando, Suehiro, and Kotoku 2008), provide their own programs to generate system-compatible source code. However, such a system often makes modification complicated.

In contrast, SkyAI provides some macros and templates[2] to support the developers to easily write system-compatible source code. The macros and the templates can be processed by a standard preprocessor and compiler; code genera-

[1]Public member variables are also available, but, they can be replaced by so-called *accessors*.

[2]Strictly, template functions and template classes of C++.

tors are not required. Thus, it is easy for the developers to modify source code with the system-compatibility.

3.2.3 Overview of System with SkyAI

The center of a software using SkyAI is an agent class. The agent class manages whole module instances, and has a parser for the script language. The agent class is provided as a generic one available for any applications. A user instantiates the agent class and calls the parser from the C++ source code (basically, in the main function).

Figure 3.1 illustrates an example of a modular structure around an RL module. In an on-line learning system, there are several kinds of cycles, such as an episode, an action, and a time step of a low-level controller. SkyAI's modular architecture can handle any number of cycles as shown in Figure 3.1.

SkyAI's architecture enables the modularization of RL algorithms as generic ones. Users of SkyAI should implement modules specific to tasks and robots, such as a low-level robot controller. Thus, in order to apply SkyAI, (1) the user implements some modules specific to tasks and robots, then, (2) the user builds them with the provided modules and the agent class. Finally, (3) the user writes a script for a specific task.

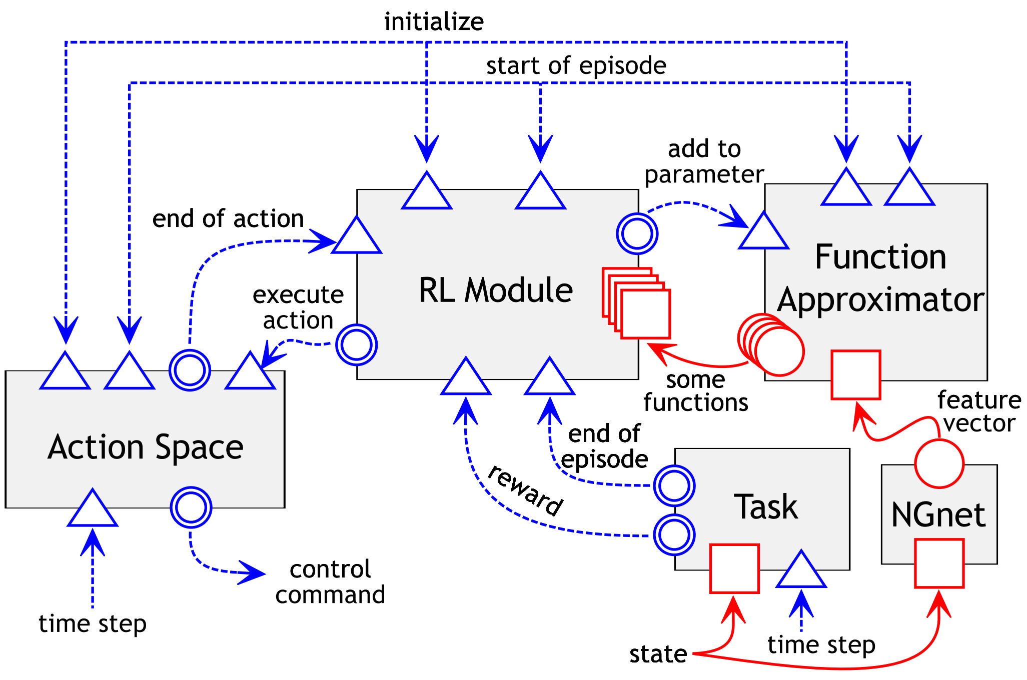


Figure 3.1 Example modular structure around an RL module.

3.3 Related Works

There are similar works to develop libraries for RL or other machine-learning methods. Compared to the libraries written in script languages, such as Python and MATLAB, SkyAI has an advantage in execution speed. SkyAI is therefore considered to be more suitable for real robot systems.

Some libraries mainly written in a script language use a compiler language in crucial bottleneck parts. For example, PyBrain (Schaul, Bayer, Wierstra, Sun, Felder, Sehnke, Rückstieß, and Schmidhuber 2010) is written in Python, but some parts are written in C/C++ which are referred from the Python code by using SWIG[3]. There are two reasons why SkyAI does not use such a technology but defines a custom script language. First, SkyAI provides a *composite module* architecture to compose some modules, and existing script languages are considered to be not suitable to define a composite module. Second, SkyAI is using some features of C++ that are not supported by SWIG. However, it is considered to be possible and useful that we implement an interface for Python or other script languages. The important fact is that SkyAI is completely written in C++ and provides an interface to manipulate the modular architecture.

The core architecture of SkyAI is a middleware. There are some middlewares for robotics, such as YARP[4], ROS[5], and OpenRTM (Ando, Suehiro, and Kotoku 2008). Another objective of SkyAI is to make a learning system by combining highly reusable modules. A highly reusable module generally becomes small, so SkyAI needs to prevent overhead of communication between modules. Thus, we implement each port of a SkyAI's module as an encapsulated function of call-by-reference, whose overhead may be smaller than that of the middlewares listed above.

3.4 Experiment: Speed of Modular Communication

The largest execution cost in the SkyAI system is the overhead of modular communication. Thus, we test the speed of modular communication. We implement a modular structure which is equivalent to the C++ code listed in Figure 3.2, where `N1` and `N2` are constant integers to vary the calculation amount. We make two modules (Figure 3.3); `MTest` as an equivalent module of `TTest`,

[3]Simplified Wrapper and Interface Generator: `www.swig.org`

[4]Yet Another Robot Platform: `eris.liralab.it/yarp`

[5]Robot Operating System: `www.ros.org`

and `MRepeater` for repeating. By observing the execution time, we determine the overhead of modular communication compared to the calculation time in the `Step` function.

Table 3.1 shows the execution time in second (the mean of 100 execution). "C++ class" denotes using `TTest` of a normal C++ code (Figure 3.2), "No com" denotes using `MTest` (SkyAI's module) but no modular communication (just calling functions of each port), "SkyAI" denotes using `MTest` and `MRepeater` in SkyAI's manner, and "Python+SWIG" denotes using Python that calls `TTest` of a normal C++ code `N1` times through SWIG[6]. In the $\mathtt{N1} = 10^8, \mathtt{N2} = 10^0$ case, "Modular com" takes more time than "C++ class" and "No com". The reason is that the cost of calculation in the `Step` function is quite small and almost the same as the overhead of the modular communication in SkyAI. In the other case, the execution time of these three are almost the same. Every execution time of "Python+SWIG" is longer than the others. The major reason is considered to be the slowness of Python's repeating process.

Therefore, we figure out that if a calculation process of a port is very small, such as just a scalar calculation, the modular communication cost is relatively large, but for an usual calculation process, such as update of an RL policy, the modular communication cost can be ignored. Thus, SkyAI achieves high-speed communication.

[6]`http://www.swig.org/`

```
class TTest
{
public:
  void Init()  {a_=1;}
  void Step()
    {for(int i(0);i<N2;++i) {a_+=(a_%10==0)?2:1;}}
  void Print() {cerr<<a_<<endl;}
protected:
  int a_;
};
int main(int argc, char**argv)
{
  TTest test;
  test.Init();
  for (int i=0;i<N1;++i)  test.Step();
  test.Print();
  return 0;
}
```

Figure 3.2 C++ code for testing the speed of modular communication.

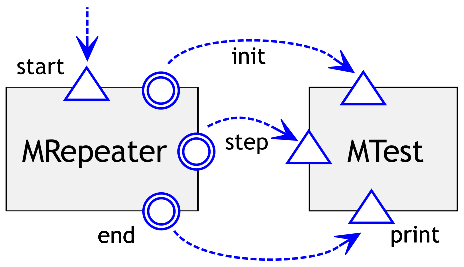


Figure 3.3 Modular structure for testing speed.

Table 3.1 Execution time (second).

N1	N2	C++ class	No com	SkyAI	Python+SWIG
10^8	10^0	0.70	0.69	3.30	113.67
10^7	10^1	0.69	0.68	0.89	11.94
10^6	10^2	0.64	0.66	0.70	1.79
10^5	10^3	0.63	0.65	0.66	0.79

3.5 Conclusion

This chapter described the principal concepts, the requirements, and the solutions of SkyAI. This chapter also demonstrated the speed advantage of SkyAI compared to the other implementation styles.

Thus, we consider that SkyAI already has the ability to handle real world tasks. However, a lot of improvements are possible. The critical ones are

Improving the script language and GUI: A script seems complicated due to its flexibility. We need to simplify it. A graphical user interface is also useful.

Multi-threading: The current program works on only a single thread. Multi-threading is desirable to speed-up the execution of the algorithms.

Transporting to the other platforms: Current implementation is only available on Linux. But, we are going to transport to the other platforms, such as Microsoft Windows and Mac OS.

Naturally, implementing new modules of state-of-the-art RL methods and more benchmark modules is an important future work. We also have a plan to make an interface to the RL-Glue (Tanner and White 2009) which is a language-independent software package for RL experiments. We encourage researchers to join the SkyAI project: **`skyai.org`**.

Chapter 4

DCOB : Action Space for Large DoF Robots

This chapter proposes a discrete action set DCOB which is generated from the basis functions (BFs) given to approximate a value function. Though DCOB is a discrete set, it has an ability to acquire high performance motions. Moreover, by using the dynamics-based BF allocation or the spring-damper BF allocation (Section 2.3), the size of DCOB is reduced, which improves the learning speed. In addition, a method WF-DCOB is proposed to enhance the performance, where wire-fitting (Baird and Klopf 1993) is utilized to search continuous actions around each discrete action of DCOB.

4.1 Introduction

In many RL applications to robot control, the action space $\mathcal{U}$ is defined as a continuous space since the command input space (we just refer to it as the command space) $\tilde{\mathcal{U}}$ is a continuous space. The simplest example of such an action space is one which is the same as the command space ($\mathcal{U} = \tilde{\mathcal{U}}$) (Kimura, Yamashita, and Kobayashi 2001; Kondo and Ito 2004; Gaskett, Fletcher, and Zelinsky 2000; Zhang and Rössler 2004). Some researchers define $\mathcal{U}$ so that it is suitable for RL (Morimoto and Doya 1998; Nakamura, Mori, Sato, and Ishii 2007; Peters, Vijayakumar, and Schaal 2003). However, RL in highly dimensional continuous command space has issues:

(1) It is difficult to initialize parameters properly especially in a learning-from-scratch case, which makes converging to undesirable local maxima.
(2) Sometimes the learning process becomes unstable.
(3) It is difficult to apply a value-function based RL method, since maximizing the action value function w.r.t. an action is computationally costly.

Meanwhile, a discrete action set is often used in some RL methods (Uchibe and Doya 2004; Takahashi and Asada 2003; Tham and Prager 1994; Kirchner

1998). The reasons are considered to be

(A) In general, learning in a discrete action space is more stable than learning in a continuous action space.
(B) Implementing value-function based RL methods is easy.
(C) Some learning architectures can be defined simply, such as a hierarchical architecture (Takahashi and Asada 2003), a multi-module learning system (Uchibe and Doya 2004), and a Dyna architecture (Sutton, Szepesvári, Geramifard, and Bowling 2008).

However, there are few general ways to design a discrete action set. A typical one is to independently divide each dimension of the command space. However, such a method makes the size of the action set increase exponentially w.r.t. the dimensionality of the command space.

Thus, a method is proposed to construct a discrete action set that is compact, has an ability to acquire high performance motions, and is therefore applicable to RL in large domains. Concretely, the proposed action set is named *DCOB* which is generated from a set of BFs given to approximate a value function. DCOB is designed to be able to acquire higher performance than a conventional discrete action set that has the same size as DCOB. Moreover, reducing number of BFs by some allocation methods reduces the size of DCOB since DCOB is generated from the BFs. The examples of such a BF allocation method are the dynamics-based and the spring-damper allocations defined in Section 2.3. Thus, DCOB has advantages in both learning speed and ability to acquire performance.

The key technique of DCOB is the way to generate an action set from BFs. DCOB requires that each BF has a center state as a parameter. Each action in DCOB is designed to be a trajectory that is Directed to the Center Of a target BF (which is the origin of the name). Here, each trajectory is terminated around the nearest BF from the current state. This abbreviation of trajectory makes the actions suitable for the resolution of BFs. The BFs are also used to learn a policy, therefore, it is considered that the abbreviation of trajectory improves the learning.

In addition, as an extension of DCOB, WF-DCOB is also proposed which uses wire-fitting to search continuous actions around each discrete action of DCOB. The aim of WF-DCOB is to acquire higher performance than DCOB while keeping the learning stability and speed in learning-from-scratch cases. The key idea of WF-DCOB is restricting the exploration around the actions in DCOB. This restriction makes the learning process stable and keeps the learning speed.

Since there are some requirements for DCOB, DCOB and WF-DCOB do not work with all kinds of robots. The methods are mainly applicable to articulated robots, such as legged robots including humanoid robots, and manipulators.

The rest of this chapter is organized as follows. Section 4.2 describes an action converter BFTrans commonly used in DCOB and WF-DCOB. Section 4.3 and 4.4 define DCOB and WF-DCOB respectively. Section 4.5 demonstrates the results of some experiments. In Section 4.6, we discuss the convergence, computational cost, available types of BF, and the relation to other works. Section 4.7 concludes this chapter.

4.2 BFTrans

The core system of DCOB is an action converter referred to as BFTrans. BFTrans converts an input action $u = (g, q^{\text{trg}})$ into a sequence of control commands, where $g \in \mathbb{R}$ is called an *interval factor* that decides a speed of motion, and $q^{\text{trg}} \in \mathcal{Q}$ is the target point of a trajectory. The trajectory is determined by $u = (g, q^{\text{trg}})$ and is followed by a low-level controller which outputs the command sequence. The command sequence is terminated after a short time with which the state moves into the nearest BF from the starting state of the action (which is the origin of the name BFTrans; transition between BFs). Thus, BFTrans provides a continuous action space for RL methods. Let $\mathcal{U}_{\text{BFTrans}} \triangleq \mathbb{R} \times \mathcal{Q}$ denote the space.

DCOB is generated by discretizing the interval factor space with a small set of real numbers and $\mathcal{Q}$ with the set of BFs. Note that using the space converter ($\mathcal{X}$ to $\mathcal{Q}$), the centers of the BFs are distributed in $\mathcal{Q}$, which means that $\mathcal{Q}$ is discretized by the BFs. WF-DCOB directly learns within the action space BFTrans using wire-fitting. Figure 4.1 illustrates these relations. The rest of this section describes the detail of BFTrans.

4.2.1 Assumptions

BFTrans assumes the following:

(A) Each BF $k \in \mathcal{K}$ has a fixed center $\mu_k \in \mathcal{X}$.

(B) $\mathcal{Q}$, $\mathrm{C_p}(x)$, and $\mathrm{C_d}(x)$ are defined. $\mathcal{Q}$: a space in which a reference trajectory is calculated (e.g. a joint angle space). $\mathrm{C_p}(x)$: a function that extracts $q \in \mathcal{Q}$ from a state $x \in \mathcal{X}$ as $q = \mathrm{C_p}(x) : \mathcal{X} \to \mathcal{Q}$. $\mathrm{C_d}(x)$: a function that extracts the derivative of $q \in \mathcal{Q}$ from a state $x \in \mathcal{X}$ as $\dot{q} = \mathrm{C_d}(x)$.

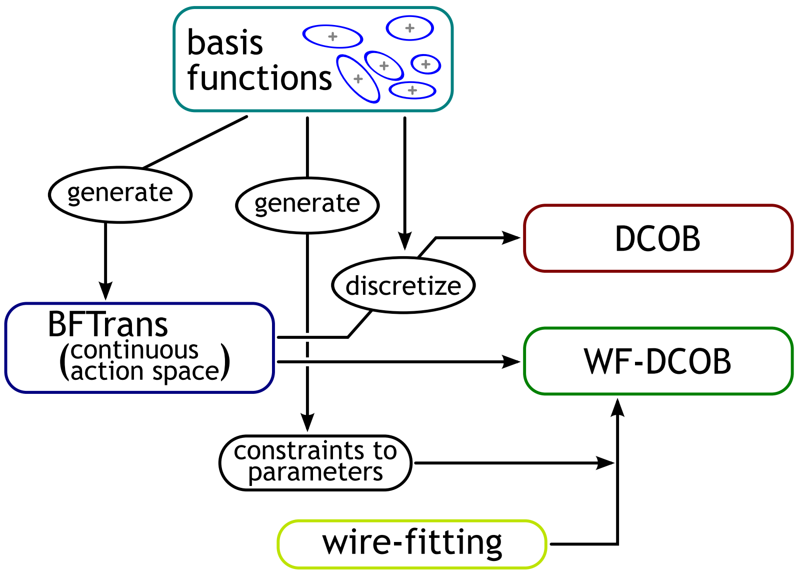


Figure 4.1 Relation among the basis functions (BFs), the wire-fitting, and the proposed methods, BFTrans, DCOB, WF-DCOB. The continuous action space BFTrans is generated from the BFs. The discrete action set DCOB is obtained by discretizing the BFTrans based on the BFs. WF-DCOB directly learns within BFTrans using the wire-fitting where the constraints of the parameters of the wire-fitting are generated from the BFs.

(C) A low-level controller $\tilde{u}(t) = Ctrl(x(t), q^{\mathrm{d}}(t + \delta\mathrm{t}))$ is given to follow a trajectory $q^{\mathrm{d}}(t)$ (e.g. a PD-controller), where $\tilde{u}$ is in $\tilde{\mathcal{U}}$ and $\delta\mathrm{t}$ denotes a control time-step.

We use NGnet in this thesis which satisfies (A). Radial BFs are alternative to NGnet. The reason of assuming that the centers are fixed is to ensure the convergence of learning.

4.2.2 Overview

When an action $u_n = (g, q^{\mathrm{trg}})$ is input to BFTrans at step n (at time t_n), the conversion to a command sequence is performed as described in Algorithm 1 (see also Figure 4.2).

BFTrans has following features as an action space:

(F1) The dynamics of the actions is suitable for the resolution of BFs.

(F2) The actions can reflect the range of the state space, such as joint angle limitations.

(F1) is satisfied by abbreviating the trajectory. The original trajectory is a curve segment of two points in the state space. On average, the length of a curve segment is comparatively long (see Figure 4.2 "Reference Trajectory"). If a motion

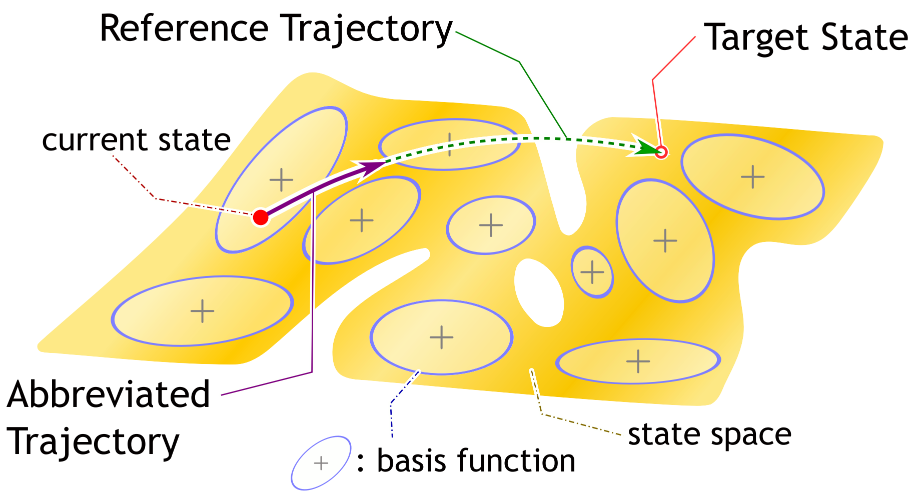


Figure 4.2 Illustration of how an action in the BFTrans is executed. First, a reference trajectory is generated, then it is abbreviated. The reference trajectory may change the state greatly, so the obtained motion is coarse. To make the motion fine, the trajectory is abbreviated.

Algorithm 1: Executing an action in the BFTrans

```
Input : Action $u_n = (g, q^{\mathrm{trg}}) \in \mathbb{R} \times \mathcal{Q} = \mathcal{U}_{\mathrm{BFTrans}}$,
        starting state $x_n = x(t_n)$
1: Estimate the time interval $T_{\mathrm{f}}$ of the trajectory from $g, x_n, q^{\mathrm{trg}}$
2: Generating a reference trajectory with which the state changes from $x_n$ to $q^{\mathrm{trg}}$ in $T_{\mathrm{f}}$:
   $q^{\mathrm{d}}(t_n + t_a),\ t_a \in [0, T_{\mathrm{f}}]$
3: Abbreviating the trajectory to $t_a \in [0, T_{\mathrm{n}}] \subseteq [0, T_{\mathrm{f}}]\ (T_{\mathrm{n}} \leqslant T_{\mathrm{f}})$
4: Following the trajectory with the low-level controller which outputs a command
   sequence: $\tilde{u}(t_n + t_a) = Ctrl(x(t_n + t_a), q^{\mathrm{d}}(t_n + t_a + \delta t)),\ t_a \in [0, T_{\mathrm{n}})$
5: The action $u_n$ is finished; $n \leftarrow n + 1$
```

of the robot was represented by a sequence of such curve segments, the whole trajectory of the motion would become coarse. To make a motion fine, each curve segment is abbreviated (Figure 4.2 "Abbreviated Trajectory"). Note that this abbreviation does not reduce the variety of actions, since the original trajectory is performed by repeating the action of the same target q^{trg} several times.

On the other hand, making each trajectory too short may cause long learning time. A moderate length is one between a BF and the nearest BF from it, since the representable fineness of a policy over the state space is almost the same as the resolution of the BFs set. Thus, the original trajectory is abbreviated to the length between the starting point and the nearest BF.

For (F2), if we select a target point inside the range of the state space, the action rarely exceeds the range. This feature is kept for discretizing target point space by the BFs set, since in many cases, the BFs are allocated inside the range of the state space.

4.2.3 Generating Trajectory

The reference trajectory $q^{\mathrm{d}}(t_n + t_a)$, $t_a \in [0, T_{\mathrm{f}}]$ is designed so that the state changes from the starting state $x_n = x(t_n)$ to the target q^{trg} in the time interval T_{f}. We represent the trajectory with a cubic function,

$$q^{\mathrm{d}}(t_n + t_a) = c_0 + c_1 t_a + c_2 t_a^2 + c_3 t_a^3. \tag{4.1}$$

The coefficients are calculated with the boundary conditions,

$$\begin{aligned} &q^{\mathrm{d}}(t_n) = C_{\mathrm{p}}(x_n), \quad q^{\mathrm{d}}(t_n + T_{\mathrm{f}}) = q^{\mathrm{trg}}, \\ &\dot{q}^{\mathrm{d}}(t_n + T_{\mathrm{f}}) = \mathbf{0}, \quad \ddot{q}^{\mathrm{d}}(t_n + T_{\mathrm{f}}) = \mathbf{0}, \end{aligned} \tag{4.2}$$

where $\mathbf{0}$ denotes a zero vector.

The motivation of introducing the interval factor g instead of T_{f} is to purely represent the speed of the action. Such a parameter is suitable to explore in the motion-speed space and is easily discretized. The speed of the action depends on both T_{f} and the distance between $C_{\mathrm{p}}(x_n)$ and q^{trg}, thus, we define g as the normalized value of T_{f} by the maximum norm. Namely, we calculate T_{f} with

$$T_{\mathrm{f}} = g \left\| q^{\mathrm{trg}} - C_{\mathrm{p}}(x_n) \right\|_{\infty} \tag{4.3}$$

where $\| \cdot \|_{\infty}$ denotes a maximum norm[1].

4.2.4 Abbreviating Trajectory

Next, we abbreviate the reference trajectory as $q^{\mathrm{d}}(t_n + t_a)$, $t_a \in [0, T_{\mathrm{n}}]$ where $0 < T_{\mathrm{n}} \leqslant T_{\mathrm{f}}$ to make the action suitable for the resolution of BFs. The abbreviation is performed by cutting off the trajectory at $t_a = T_{\mathrm{n}}$ where T_{n} is decided so that the state moves into the nearest BF from the starting state [2].

The abbreviation is performed as follows: (1) estimate $D_{\mathrm{n}}(x_n)$ indicating the distance between two neighboring BFs around the start state x_n, (2) calculate T_{n}

[1]For a vector $x = (x_1, \dots, x_D)$, the maximum norm is defined as $\|x\|_{\infty} = \max_n |x_n|$.

[2]We do not abbreviate the trajectory by observing the output of BFs since it is complicated to treat a case: the output of BFs does not change around range of the state space. BFTrans with such an abbreviation may remove the Markov property from a task.

from the ratio of $D_{\mathrm{n}}(x_n)$ and the distance between x_n and q^{trg}. Here, we employ a maximum norm as a distance rather than the L^2-norm since the trajectory of each joint is calculated independently.

To define $D_{\mathrm{n}}(x_n)$, for each BF k, we first calculate $d_{\mathrm{n}}(k)$ as the distance between its center μ_k and the center of the nearest BF from k. Then, we estimate $D_{\mathrm{n}}(x_n)$ by interpolating $\{d_{\mathrm{n}}(k)|k \in \mathcal{K}\}$ with the output of the BFs at x_n.

$d_{\mathrm{n}}(k)$ is calculated by

$$k_{\mathrm{n}}(k) = \underset{k' \in \mathcal{K}, k' \neq k}{\arg\min} \|C_{\mathrm{p}}(\mu_{k'}) - C_{\mathrm{p}}(\mu_k)\|_\infty, \tag{4.4}$$

$$d_{\mathrm{n}}(k) = \max\left(\|C_{\mathrm{p}}(\mu_{k_{\mathrm{n}}(k)}) - C_{\mathrm{p}}(\mu_k)\|_\infty,\, \mathrm{d}_{\min k}\right), \tag{4.5}$$

where $\mathrm{d}_{\min k} \in \mathbb{R}$ is a positive constant to adjust $d_{\mathrm{n}}(k)$ for too small $\|C_{\mathrm{p}}(\mu_{k_{\mathrm{n}}(k)}) - C_{\mathrm{p}}(\mu_k)\|_\infty$. For NGnet, we define it as $\mathrm{d}_{\min k} = \sqrt{\lambda_k^{\mathcal{Q}}}$ where $\lambda_k^{\mathcal{Q}}$ is the maximum eigenvalue of the covariance matrix $\Sigma_k^{\mathcal{Q}}$ on the $\mathcal{Q}$ space[3]. Note that we can precompute $\{d_{\mathrm{n}}(k)|k \in \mathcal{K}\}$ for fixed BFs.

Using the output of BFs $\phi(x_n)$, $D_{\mathrm{n}}(x_n)$ is estimated by

$$d_{\mathrm{n}} \triangleq (d_{\mathrm{n}}(1),\, d_{\mathrm{n}}(2),\, \ldots,\, d_{\mathrm{n}}(|\mathcal{K}|))^\top, \tag{4.6}$$

$$D_{\mathrm{n}}(x_n) = d_{\mathrm{n}}^\top \phi(x_n). \tag{4.7}$$

Finally, T_{n} is defined by

$$T_{\mathrm{n}}(x_n, u_n) = \begin{cases} \frac{\mathrm{F}_{\mathrm{abbrv}} D_{\mathrm{n}}(x_n)}{\|q^{\mathrm{trg}} - C_{\mathrm{p}}(x_n)\|_\infty} T_{\mathrm{f}} & (D_{\mathrm{n}}(x_n) < \|q^{\mathrm{trg}} - C_{\mathrm{p}}(x_n)\|_\infty) \\ T_{\mathrm{f}} & (\text{otherwise}) \end{cases} \tag{4.8}$$

where $\mathrm{F}_{\mathrm{abbrv}}$ denotes a scaling factor which typically takes 1.

4.2.5 Following Trajectory

The abbreviated trajectory $q^{\mathrm{d}}(t = t_n + t_a)$, $t_a \in [0, T_{\mathrm{n}})$ is followed by the low-level as $\tilde{u}(t) = \mathit{Ctrl}(x(t), q^{\mathrm{d}}(t + \delta\mathrm{t}))$, $t \in [t_n, t_n + T_{\mathrm{n}})$. For the simulated small-humanoid benchmarks, we use a simple PD-controller. For the Bioloid benchmarks, we use controllers embedded on each joint actuator.

[3] $\Sigma_k^{\mathcal{Q}}$ is calculated from the original covariance matrix Σ_k (on the $\mathcal{X}$ space) as follows. For ease of calculation, let $C_{\mathrm{p}}(x) = \hat{C}_{\mathrm{p}} x$ where $\hat{C}_{\mathrm{p}}$ is a constant matrix. The converted covariance matrix is $\Sigma_k^{\mathcal{Q}} = \hat{C}_{\mathrm{p}} \Sigma_k \hat{C}_{\mathrm{p}}^\top$.

4.3 DCOB

A discrete action set DCOB is defined by discretizing BFTrans. Recall that an action in BFTrans is denoted by $u = (g, q^{\text{trg}}) \in \mathbb{R} \times \mathcal{Q}$. The interval factor space is discretized by a small set of real numbers $\mathcal{I}$, and $\mathcal{Q}$ is discretized by a set of the centers of the BFs $\{C_{\text{p}}(\mu_k) \mid k \in \mathcal{K}\}$. Let $\mathcal{A}_{\text{DCOB}}$ denote DCOB: $\mathcal{A}_{\text{DCOB}} = \mathcal{I} \times \mathcal{K}$. An action a_n in DCOB selected at step n (at time t_n) is executed as follows:

Algorithm 2: Executing an action in DCOB

```
Input : Action $a_n = (g, k) \in \mathcal{I} \times \mathcal{K} = \mathcal{A}_{\text{DCOB}}$,
        starting state $x_n = x(t_n)$
1: $u_n \leftarrow (g, C_{\text{p}}(\mu_k))$
2: Execute BFTrans with $u_n$ (Algorithm 1)
```

The size of DCOB is $|\mathcal{I}||\mathcal{K}|$. Since we use a small set $|\mathcal{I}|$, the size of DCOB is a few times the number of BFs. Thus, if the number of BFs is reduced (recall Section 2.3), the size of DCOB is also reduced.

4.4 WF-DCOB

WF-DCOB directly learns in the continuous action space BFTrans with wire-fitting. However, as described in Section 4.1, learning in a continuous action space has issues in initializing parameters and learning stability. WF-DCOB tries to solve these issues by restricting the exploration around the actions in DCOB; namely, each fixed point in BFTrans that is an action of DCOB is allowed to move inside a region around the point (Figure 4.3).

To do this, we prepare the control wires $\mathcal{W}$ whose number is the same as the size of DCOB. Then, each control wire $i \in \mathcal{W}$ is initialized so that U_i indicates the corresponding action of DCOB (recall Section 2.2.2). During learning, each control wire is kept inside the constraint region. Instead of the set of interval factors $\mathcal{I}$, a set of interval factor *ranges* is defined for WF-DCOB. A set of ranges $\mathcal{I}_{\mathcal{R}}$ is defined as

$$\mathcal{I}_{\mathcal{R}} \triangleq \{(\mathrm{g}^{\mathrm{s}}_m, \mathrm{g}^{\mathrm{e}}_m) \mid 0 < \mathrm{g}^{\mathrm{s}}_m \leqslant \mathrm{g}^{\mathrm{e}}_m, m = 1, 2, \ldots\}, \tag{4.9}$$

where $|\mathcal{I}| = |\mathcal{I}_{\mathcal{R}}|$.

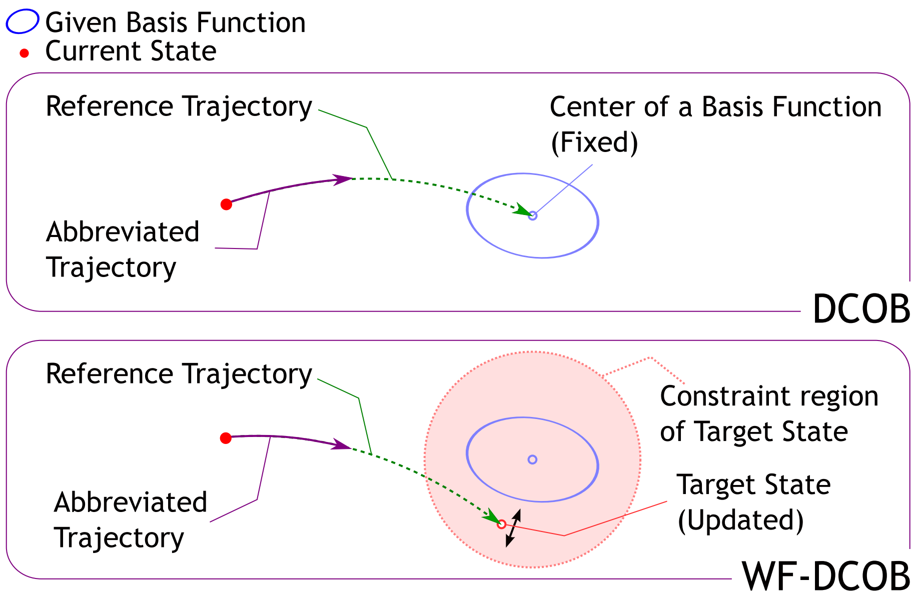


Figure 4.3 Illustration of the comparison of DCOB (top) and WF-DCOB (bottom). In both methods, the trajectory is calculated in the same manner as the BFTrans. The difference is that in DCOB, the target state is the fixed center of a selected basis function, while in WF-DCOB, the target state can change but is constrained around a corresponding basis function.

Initializing Wire-Fitting Parameters

For a control wire $i \in \mathcal{W}$, we use a_i^{dcob} to denote the corresponding action in DCOB: $a_i^{\text{dcob}} = (g_i^{\text{dcob}}, k_i^{\text{dcob}})$. Let $(\mathrm{g}_i^{\mathrm{s}}, \mathrm{g}_i^{\mathrm{e}})$ denote the range of the interval factor which includes g_i^{dcob}. For each control wire $i \in \mathcal{W}$, its parameter is defined as $U_i = (g_i, q_i^{\text{trg}})$ and is initialized by

$$g_i \leftarrow \frac{\mathrm{g}_i^{\mathrm{s}} + \mathrm{g}_i^{\mathrm{e}}}{2}, \tag{4.10a}$$

$$q_i^{\text{trg}} \leftarrow C_{\mathrm{p}}(\mu_{k_i^{\text{dcob}}}). \tag{4.10b}$$

The other parameters of the control wires $\{\theta_i | i \in \mathcal{W}\}$ are initialized by zero, since in a learning-from-scratch case, we do not have prior knowledge about the action values.

Constraints on Wire-Fitting Parameters

For $U_i = (g_i, q_i^{\text{trg}})$, the interval factor g_i is constrained inside $(\mathrm{g}_i^{\mathrm{s}}, \mathrm{g}_i^{\mathrm{e}})$, and the target point q_i^{trg} is constrained inside a hypersphere of radius $d_{\mathrm{n}}(k_i^{\text{dcob}})$ centered at $C_{\mathrm{p}}(\mu_{k_i^{\text{dcob}}})$. Here, $d_{\mathrm{n}}(k_i^{\text{dcob}})$ denotes the distance to the nearest BF from k_i^{dcob} defined in eq. (4.5). Specifically, the parameter $U_i = (g_i, q_i^{\text{trg}})$ of each control wire

$i \in \mathcal{W}$ is constrained by

$$
\begin{aligned}
&\texttt{if}\ g_i < \mathrm{g}_i^{\mathrm{s}}\ \texttt{then}\ g_i \leftarrow \mathrm{g}_i^{\mathrm{s}},\\
&\texttt{if}\ g_i > \mathrm{g}_i^{\mathrm{e}}\ \texttt{then}\ g_i \leftarrow \mathrm{g}_i^{\mathrm{e}},\\
&\texttt{if}\ \|q_i^{\mathrm{trg}} - C_{\mathrm{p}}(\mu_{k_i^{\mathrm{dcob}}})\|_\infty > d_{\mathrm{n}}(k_i^{\mathrm{dcob}})\ \texttt{then}\\
&\qquad q_i^{\mathrm{trg}} \leftarrow C_{\mathrm{p}}(\mu_{k_i^{\mathrm{dcob}}}) + d_{\mathrm{n}}(k_i^{\mathrm{dcob}}) \frac{(q_i^{\mathrm{trg}} - C_{\mathrm{p}}(\mu_{k_i^{\mathrm{dcob}}}))}{\|q_i^{\mathrm{trg}} - C_{\mathrm{p}}(\mu_{k_i^{\mathrm{dcob}}})\|_\infty}.
\end{aligned}
\tag{4.11}
$$

This constraints are applied each after update of an RL algorithm.

4.5 Experiments

This section demonstrates experimental comparisons of the proposed methods and the conventional methods. The tasks and the RL methods are defined in Chapter 2.

4.5.1 Maze2D

Experiments of the task Maze2D are demonstrated. The compared methods are DCOB with LFA-NGnet, WF-DCOB, a conventional discrete action set with LFA-NGnet, and wire-fitting.

Configurations of RL Method

As an RL method, we use Peng's Q(λ)-learning for every conditions. The parameters of the Q(λ)-learning are also similar for every conditions: $\gamma = 0.9$, $\lambda = 0.9$. We use a decreasing step-size parameter $\alpha = \alpha_0 \exp(-\delta_\alpha N_{\mathrm{eps}})$ for learning, and a decreasing temperature parameter $\tau = \tau_0 \exp(-\delta_\tau N_{\mathrm{eps}})$ for Boltzmann selection where N_{eps} denotes the episode number. These parameters are determined through preliminary experiments: $\alpha_0 = 0.3$, $\delta_\alpha = 0.002$, $\tau_0 = 0.1$, $\delta_\tau = 0.005$.

We use the default BFs defined for Maze2D, which are allocated around a 8×8 grid. This set of BFs is commonly used to approximate an action value function, namely LFA-NGnet and wire-fitting, and to generate DCOB and WF-DCOB.

Configurations of Action Spaces and Function Approximators

The following conditions are compared.

DCOB : The parameters of DCOB are as follows:

$$C_{\mathrm{p}}(x) = x = (x_1, x_2)^\top, \tag{4.12}$$
$$C_{\mathrm{d}}(x) = (0, 0)^\top, \tag{4.13}$$
$$Ctrl(x(t), q^{\mathrm{d}}(t + \delta\mathrm{t})) = q^{\mathrm{d}}(t + \delta\mathrm{t}) - C_{\mathrm{p}}(x(t)), \tag{4.14}$$
$$\mathcal{I} = \{0.3\}, \tag{4.15}$$
$$\mathrm{F}_{\mathrm{abbrv}} = 1, \tag{4.16}$$

The size of the action set is $|\mathcal{A}_{\mathrm{DCOB}}| = |\mathcal{K}||\mathcal{I}| = 64$. As a function approximator, LFA-NGnet is used.

WF-DCOB : The WF-DCOB's parameters C_{p}, C_{d}, $Ctrl$ and $\mathrm{F}_{\mathrm{abbrv}}$ are the same as DCOB. The other parameter is

$$\mathcal{I}_{\mathcal{R}} = \{(0.1, 0.5)\}. \tag{4.17}$$

Wire-fitting is used as a function approximator. The number of the control wires is the same as the size of DCOB.

Radial Action Set (R3, R4, R8, R16, R32, R64) : As a discrete action set, a "radial action set" $\mathcal{A}_{\mathrm{R}}$ is defined. Each action changes the state radially from the current state. The norm of the displacement is constant and the direction is chosen from a discrete set. Specifically,

$$\begin{aligned} &\Delta\varphi = 2\pi/\mathrm{N}_{\mathrm{dir}} \\ &\mathcal{A}_{\mathrm{R}} = \{dir_a \mid dir_a = (-\sin(a\Delta\varphi), \cos(a\Delta\varphi))^\top, \\ &\qquad a = 0, \ldots, \mathrm{N}_{\mathrm{dir}} - 1\} \end{aligned} \tag{4.18}$$

where $\mathrm{N}_{\mathrm{dir}}$ denotes the number of division of directions. Each action dir_a is converted into a sequence of command input as follows:

$$\tilde{u}(t) = \tilde{\mathrm{u}}_{\mathrm{max}} dir_a, \quad t \in [t_n, t_n + \mathrm{T}_{\mathrm{R}}) \tag{4.19}$$

where $\tilde{\mathrm{u}}_{\mathrm{max}} = 0.03$ denotes the maximum norm of an input, and $\mathrm{T}_{\mathrm{R}} = 0.1$ denotes the duration of the action. The size of $\mathcal{A}_{\mathrm{R}}$ is $\mathrm{N}_{\mathrm{dir}}$. In the following experiments, $\mathrm{N}_{\mathrm{dir}} = \{3, 4, 8, 16, 32, 64\}$ are used, which are represented as R3, ..., R64 respectively. As a function approximator, LFA-NGnet is used.

Wire-Fitting (WF4, WF8, WF16, WF32, WF64) : The parameters of control wires $\{U_i | i \in \mathcal{W}\}$ are initialized by the elements of the radial action set $\{dir_a\}$ defined above. Similarly, $\mathrm{N}_{\mathrm{dir}} = \{4, 8, 16, 32, 64\}$ are used, which are represented as

WF4, . . . , WF64 respectively. For an action, the command sequence is computed by

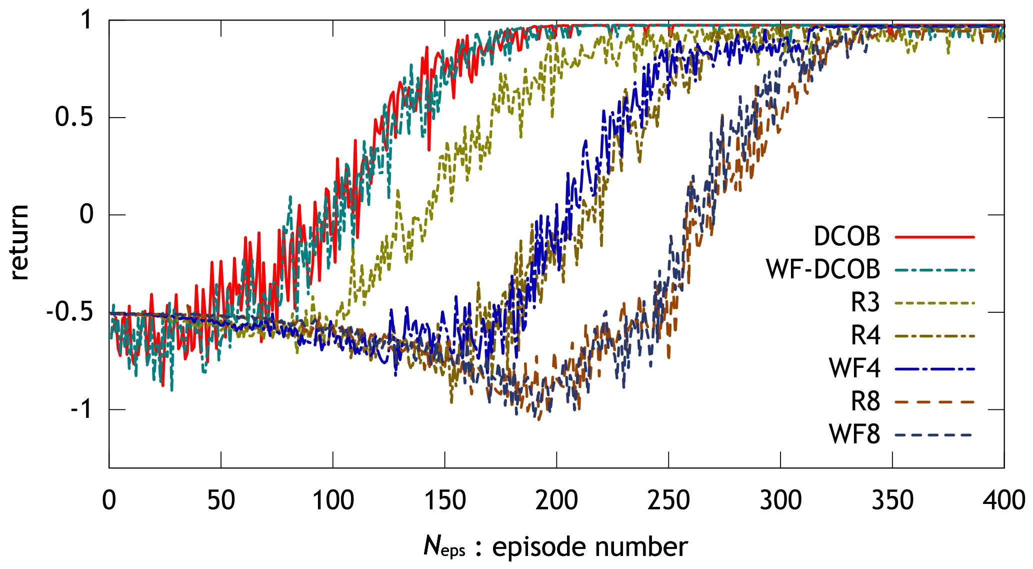
$$\tilde{u}(t) = \tilde{\mathrm{u}}_{\mathrm{max}} u(x_n). \tag{4.20}$$

The duration of an action is $\mathrm{T}_{\mathrm{WF}} = 0.1[\mathrm{s}]$.

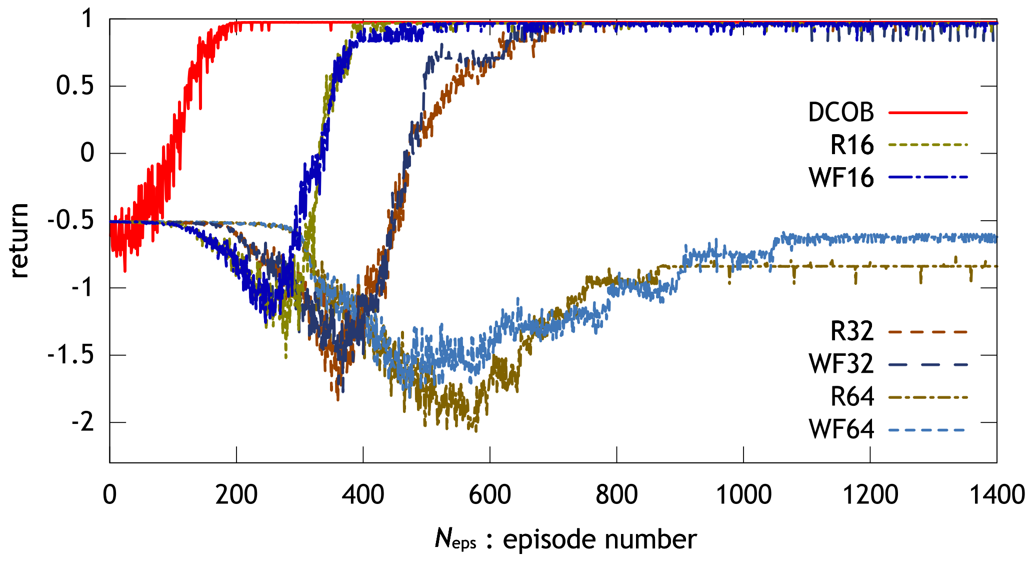
Results

Figure 4.4 shows the learning curves of the Maze2D task (the mean of the return over 25 runs is plotted per episode). The radial action set and wire-fitting have a tendency that the learning speed decreases with increasing $\mathrm{N}_{\mathrm{dir}}$. However, the learning speed of DCOB and WF-DCOB is faster compared to the others, in spite of $|\mathcal{A}_{\mathrm{DCOB}}| = 64$. It is considered to be a main factor that DCOB has few actions with which the robot moves out of the plane $\mathcal{X}_{\mathrm{pl}}$ since few BFs are allocated out of $\mathcal{X}_{\mathrm{pl}}$, as is WF-DCOB. Thus, the robot with DCOB and WF-DCOB learned a path to the *goal* faster than the robot with the other action spaces.

On the other hand, there is no major difference between DCOB and WF-DCOB, or a radial action set and the corresponding wire-fitting. Recall that WF-DCOB or wire-fitting can potentially represent better policy than DCOB or the corresponding radial action set, while WF-DCOB or wire-fitting has more learning parameters. However, from the results of Figure 4.4, it is considered that extending a discrete action to a continuous one is not beneficial for this task. Rather than this extension, the size of an action set is the dominant factor in the learning speed.



(a) Learning curves converging faster.



(b) Learning curves converging slower.

Figure 4.4 Resulting learning curves of the Maze2D task. Each curve shows the mean of the return per episode over 25 runs. The learning curves converging faster and the ones converging slower are displayed in these two graphs. Only the curve of DCOB is displayed in both graphs.

4.5.2 HumanoidML-crawling, turning

Experiments of the HumanoidML tasks are demonstrated here. The compared methods are DCOB with LFA-NGnet, WF-DCOB, a conventional discrete action set with LFA-NGnet, and wire-fitting.

Configurations of RL Method

As an RL method, we use Peng's Q(λ)-learning for every conditions. The parameters of the Q(λ)-learning are also consistent for every conditions: $\gamma = 0.9$, $\lambda = 0.9$. We use a decreasing step-size parameter $\alpha = \alpha_0 \exp(-\delta_\alpha N_{\text{eps}})$ for learning, and a decreasing temperature parameter $\tau = \tau_0 \exp(-\delta_\tau N_{\text{eps}})$ for Boltzmann selection where N_{eps} denotes the episode number. These parameters are determined through preliminary experiments: $\alpha_0 = 0.3$, $\delta_\alpha = 0.002$, $\tau_0 = 5$, $\delta_\tau = 0.004$.

DoF Configurations and BF Allocations

Several conditions are compared in order to investigate the influence of a DoF configuration and a BF allocation upon DCOB. For the HumanoidML-crawling task, the following conditions are used:

3-DoF-Grid : The 3-DoF configuration with the default BF allocation (the grid allocation on a $5 \times 5 \times 5$ grid). The number of the BFs is 125.

4-DoF-Grid : The 4-DoF configuration with the default BF allocation (the grid allocation on a $4 \times 4 \times 4 \times 4$ grid). The number of the BFs is 256.

5-DoF-Dyn : The 5-DoF configuration with the default BF allocation (the dynamics-based allocation of 202 BFs).

5-DoF-Grid : The 5-DoF configuration with the grid allocation. The BFs are allocated on a $3 \times 3 \times 3 \times 3 \times 3$ grid over the joint angle space. The number of the BFs is 243.

5-DoF-SprDmp : The 5-DoF configuration with the spring-damper allocation. Specifically, 300 BFs are allocated over the reduced joint angle space.

6-DoF-SprDmp : The 6-DoF configuration with the default BF allocation (the spring-damper allocation of 300 BFs).

7-DoF-SprDmp : The 7-DoF configuration with the default BF allocation (the spring-damper allocation of 600 BFs).

For the HumanoidML-turning task, the following condition is used:

4-DoF-Grid : The same as the 4-DoF-Grid of the HumanoidML-crawling task.

Here, 'Dyn' denotes the dynamics-based allocation, and 'SprDmp' denotes the spring-damper allocation. Note that in each condition, the set of BFs is commonly used to approximate an action value function, namely LFA-NGnet and wire-fitting, and to generate DCOB and WF-DCOB.

Configurations of Action Spaces and Function Approximators

The following conditions are compared.

DCOB : For each N_{D}-DoF configuration, the parameters of DCOB are as follows:

$$C_{\mathrm{p}}(x) = \mathbf{q}_{N_{\mathrm{D}}}, \tag{4.21}$$

$$C_{\mathrm{d}}(x) = \dot{\mathbf{q}}_{N_{\mathrm{D}}}, \tag{4.22}$$

$$\mathit{Ctrl}(x(t), q^{\mathrm{d}}(t+\delta\mathrm{t})) = \mathrm{K}_{\mathrm{p}}\{q^{\mathrm{d}}(t+\delta\mathrm{t}) - C_{\mathrm{p}}(x(t))\} - \mathrm{K}_{\mathrm{d}} C_{\mathrm{d}}(x(t)), \tag{4.23}$$

$$\mathcal{I} = \{0.075,\ 0.1,\ 0.2\}, \tag{4.24}$$

$$\mathrm{F}_{\mathrm{abbrv}} = \begin{cases} 0.5 & (N_{\mathrm{D}} = 4), \\ 1 & (\text{otherwise}), \end{cases} \tag{4.25}$$

where $\mathbf{q}_{N_{\mathrm{D}}}$ denotes the joint angle vector, $\dot{\mathbf{q}}_{N_{\mathrm{D}}}$ denotes the joint angular velocities, $\mathrm{K}_{\mathrm{p}} = 5.0[\mathrm{Nm/rad}]$, and $\mathrm{K}_{\mathrm{d}} = 1.6[\mathrm{Nms/rad}]$. $\mathrm{F}_{\mathrm{abbrv}} = 0.5$ for $N_{\mathrm{D}} = 4$ is determined through preliminary experiments. For each DoF configuration and BF allocation, the size of DCOB $|\mathcal{A}_{\mathrm{DCOB}}| = |\mathcal{I}||\mathcal{K}|$ is as follows. 3-DoF-Grid: $|\mathcal{A}_{\mathrm{DCOB}}| = 375$, 5-DoF-Dyn: $|\mathcal{A}_{\mathrm{DCOB}}| = 606$, 5-DoF-Grid: $|\mathcal{A}_{\mathrm{DCOB}}| = 729$, 6-DoF-SprDmp: $|\mathcal{A}_{\mathrm{DCOB}}| = 900$, 7-DoF-SprDmp: $|\mathcal{A}_{\mathrm{DCOB}}| = 1800$, 4-DoF-Grid: $|\mathcal{A}_{\mathrm{DCOB}}| = 768$. As a function approximator, LFA-NGnet is used.

WF-DCOB : For each N_{D}-DoF configuration, the WF-DCOB's parameters C_{p}, C_{d}, *Ctrl* and $\mathrm{F}_{\mathrm{abbrv}}$ are the same as DCOB. The other parameter is

$$\mathcal{I}_{\mathcal{R}} = \{(0.05, 0.1), (0.1, 0.2), (0.2, 0.3)\}. \tag{4.26}$$

Wire-fitting is used as a function approximator. For each N_{D}-DoF configuration, the number of the control wires is the same as the size of DCOB.

Grid Action Set (Grid3, Grid5) : A "grid action set" $\mathcal{A}_{\mathrm{G}}$ is defined as an action set where the displacement of a target joint angle is discretized by a grid. For each N_{D}-DoF configuration, $\mathcal{A}_{\mathrm{G}}$ is defined as follows:

$$\begin{aligned} \mathcal{A}_{\mathrm{G}} = \{\Delta q \mid \Delta q = (\delta q_1, \dots, \delta q_{N_{\mathrm{D}}})^{\top}, \\ \delta q_{1,\dots,N_{\mathrm{D}}} \in \{0, \pm\Delta\varphi, \dots, \pm\tfrac{\mathrm{N}_{\mathrm{grid}}-1}{2}\Delta\varphi\}\} \end{aligned} \tag{4.27}$$

where $\mathrm{N}_{\mathrm{grid}}$ denotes the number of division of each joint angle, and $\Delta\varphi = \pi/12$ is a unit of the displacement. The size of $\mathcal{A}_{\mathrm{G}}$ is $|\mathcal{A}_{\mathrm{G}}| = (\mathrm{N}_{\mathrm{grid}})^{N_{\mathrm{D}}}$. In the following experiments, $\mathrm{N}_{\mathrm{grid}} = 3$ and 5 are used, which are represented as Grid3 and

Grid5 respectively. Each element $\Delta q \in \mathcal{A}_G$ is converted to a command sequence as follows:

$$q^{\mathrm{d}}(t) = C_{\mathrm{p}}(x(t)) + \Delta q \tag{4.28}$$

$$\tilde{u}(t) = \mathrm{K}_{\mathrm{p}}\{q^{\mathrm{d}}(t) - C_{\mathrm{p}}(x(t))\} - \mathrm{K}_{\mathrm{d}}C_{\mathrm{d}}(x(t)) \tag{4.29}$$

where $t = t_n + t_a$, $t_a \in [0, \mathrm{T_G})$, and $\mathrm{T_G} = 0.1[\mathrm{s}]$ denotes a duration of an action. As a function approximator, LFA-NGnet is used.

Wire-Fitting (WF3, WF5) : The parameters of control wires $\{U_i | i \in \mathcal{W}\}$ are initialized by the elements of the grid action set defined above. Similarly, $\mathrm{N_{grid}} = 3$ and 5 are used, which are represented as WF3 and WF5 respectively. For an action, the sequence of target joint angles is computed by

$$q^{\mathrm{d}}(t) \triangleq C_{\mathrm{p}}(x(t)) + u(x_n), \tag{4.30}$$

and is converted into a command sequence with the same manner as the grid action set. The duration of an action is $\mathrm{T_{WF}} = 0.1[\mathrm{s}]$.

Table 4.1 shows the number of BFs, actions, and control wires for each DoF configuration.

Task Setup of HumanoidML-crawling for 4-DoF

The tasks HumanoidML-crawling and turning are defined in Chapter 2 except for HumanoidML-crawling in the 4-DoF configuration. Since the robot is not symmetrically constrained in only the 4-DoF configuration, a penalty for rotational movement should be added. Similarly, reward for forward movement is changed. Concretely, we use the following reward definition only for the 4-DoF case:

$$r(t) = r_{\mathrm{mv}}(t) - r_{\mathrm{rt}}(t) - r_{\mathrm{sc}}(t) - r_{\mathrm{fd}}(t) \tag{4.31}$$

$$r_{\mathrm{mv}}(t) = 50(\dot{c}_{0x}(t)e_{z1}(t) + \dot{c}_{0y}(t)e_{z2}(t)) \tag{4.32}$$

$$r_{\mathrm{rt}}(t) = 5|\omega_z(t)| \tag{4.33}$$

$$r_{\mathrm{sc}}(t) = 2 \times 10^{-5}\|\tilde{u}(t)\| \tag{4.34}$$

where $r_{\mathrm{mv}}(t)$ is the reward for forward movement, $(e_{z1}, e_{z2}, e_{z3})^{\top}$ is the z-component of the rotation matrix of the body link, r_{rt} is the penalty for rotation, $r_{\mathrm{sc}}(t)$ is the step cost, $r_{\mathrm{fd}}(t)$ is the penalty for falling down. $r_{\mathrm{fd}}(t)$ takes 4 if the body or the head link touches the ground, otherwise it takes 0. The penalty for falling down is given once in each action. Each episode begins with the initial state where the robot lies down and stationary, and ends if $t > 20[\mathrm{s}]$ or the sum of reward is less than -40.

Results

Figure 4.5, 4.6, 4.7, 4.8, 4.9, 4.10, 4.11, and 4.12 show the learning curves of the HumanoidML tasks; in each graph, the mean of the return over 15 runs is plotted per episode. In all results, DCOB and WF-DCOB acquire a motion of outstanding performance than the other methods, and the learning speed of DCOB and WF-DCOB is almost the fastest. The possible reasons are considered as follows:

(R1) BFTrans provides a suitable action space for RL methods as mentioned in Section 4.2.2.

(R2) Utilization of BFs for the action space discretization (DCOB) or for the parameter initialization and constraints (WF-DCOB) reduces the learning time or the learning instability.

(R1) enables the robot to acquire higher performance; in Figure 4.10 and 4.11, the size of DCOB and Grid3 are almost the same, but DCOB outperforms Grid3. Similarly, in Figure 4.6, the size of DCOB and Grid5 are almost the same, but DCOB outperforms Grid5. We can verify (R2) in Figure 4.7 where the number of BFs is reduced by the dynamics-based allocation. The size of DCOB is smaller than Grid5, which enables the fast convergence. Still, the acquired performance of DCOB is almost the highest.

Next, we compare the results of DCOB and WF-DCOB. In lower DoFs (3, 4, and 5), DCOB and WF-DCOB are almost the same, or DCOB acquires better performance than WF-DCOB (Figure 4.6, 4.9). In addition, DCOB learns faster than WF-DCOB in Figure 4.6. On the other hand, in higher DoFs (6 and 7), WF-DCOB is superior to DCOB; WF-DCOB learns faster in Figure 4.10 and 4.11, WF-DCOB acquires better performance in Figure 4.11. A possible reason is considered as follows. In higher DoFs, the number of actions in DCOB is relatively small for the DoF, which leads a coarse exploration. WF-DCOB explores continuous actions around the actions in DCOB, which makes exploration finer. Thus, WF-DCOB performed better than DCOB in the higher DoF cases. Meanwhile in the lower DoFs, the number of actions in DCOB was considered to be enough for a fine exploration. In such cases, searching continuous actions improves minimally the performance. Moreover, due to the instability of learning in a continuous action space, WF-DCOB is inferior to DCOB in some cases. This consideration can also explain the results of 5-DoF-Dyn, 5-DoF-Grid, and 5-DoF-SprDmp (Figure 4.7, 4.8, and 4.9).

Figure 4.13 and 4.14 show the snapshots of acquired motions with DCOB. Though the same set of movable joints are used in 3-DoF and 5-DoF, the difference of joint-coupling or BF allocation changes the behavior of the robot (Figure 4.13(a), 4.13(c), 4.13(d), 4.14(a)). The behavior is also changed by increasing movable joints as seen in 6 and 7-DoFs (Figure 4.14(b), 4.14(c)). The crawling behavior with 4-DoF configuration looks slightly turning (Figure 4.13(b)); this is because the joint-coupling is not symmetric. Comparing the crawling behavior of 4-DoF configuration with the turning behavior of the same configuration, we can find that the difference of reward function generates the behavior (Figure 4.13(b), 4.14(d)).

Table 4.1 Number of BFs, actions, and control wires.

DoF and BF allocation	BFs	DCOB	WF-DCOB	Grid3	WF3	Grid5	WF5
3-DoF-Grid	125	375	375	27	27	125	125
4-DoF-Grid	256	768	768	81	81	625	625
5-DoF-Dyn	202	606	606	243	243	3125	3125
5-DoF-Grid	243	729	729	243	243	—	—
5-DoF-SprDmp	300	900	900	243	243	—	—
6-DoF-SprDmp	300	900	900	729	729	—	—
7-DoF-SprDmp	600	1800	1800	2187	2187	—	—

BFs: number of BFs, DCOB/Grid3/Grid5: number of actions, WF-DCOB/WF3/WF5: number of control wires. Dash (—) denotes that the action space is not used in the DoF.

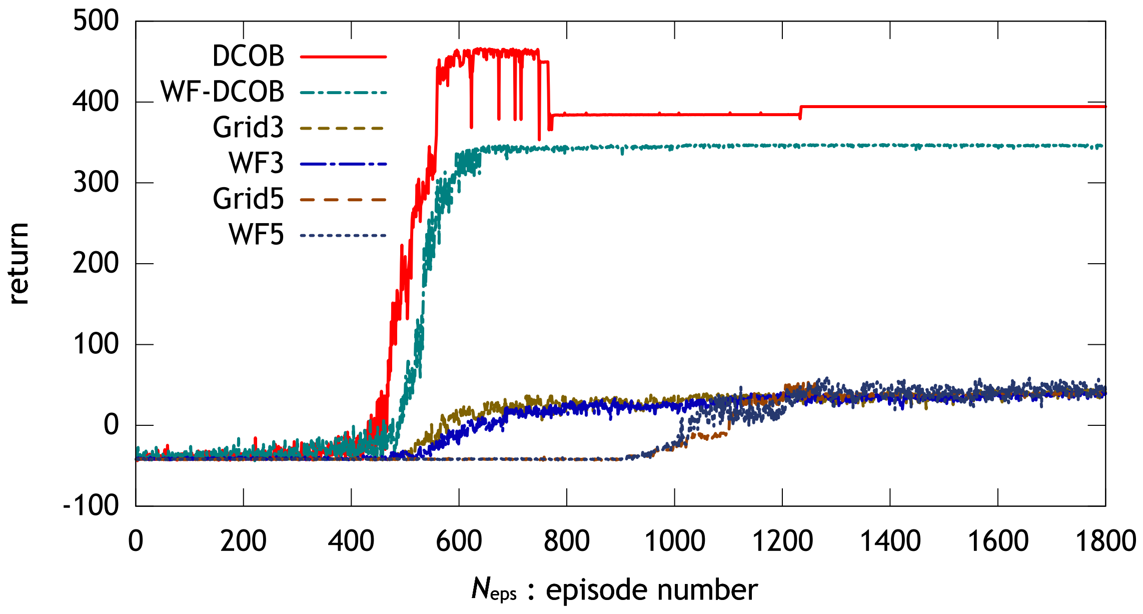


Figure 4.5 Resulting learning curves of the HumanoidML-crawling task with 3-DoF-Grid. Each curve shows the mean of the return over 15 runs per episode.

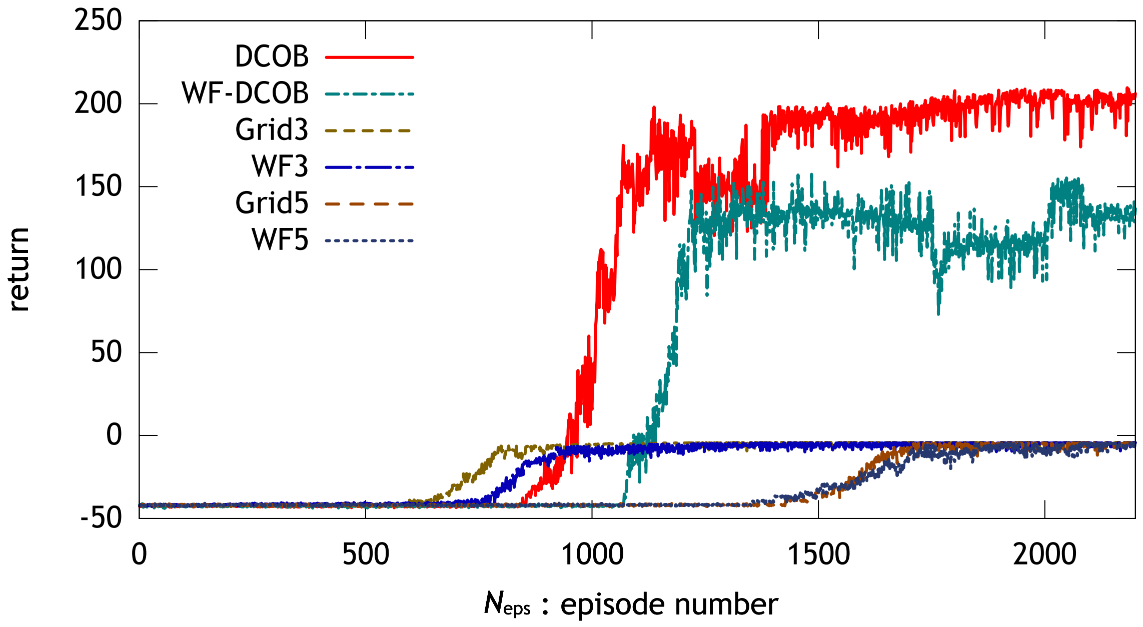


Figure 4.6 Resulting learning curves of the HumanoidML-crawling task with 4-DoF-Grid. Each curve shows the mean of the return over 15 runs per episode.

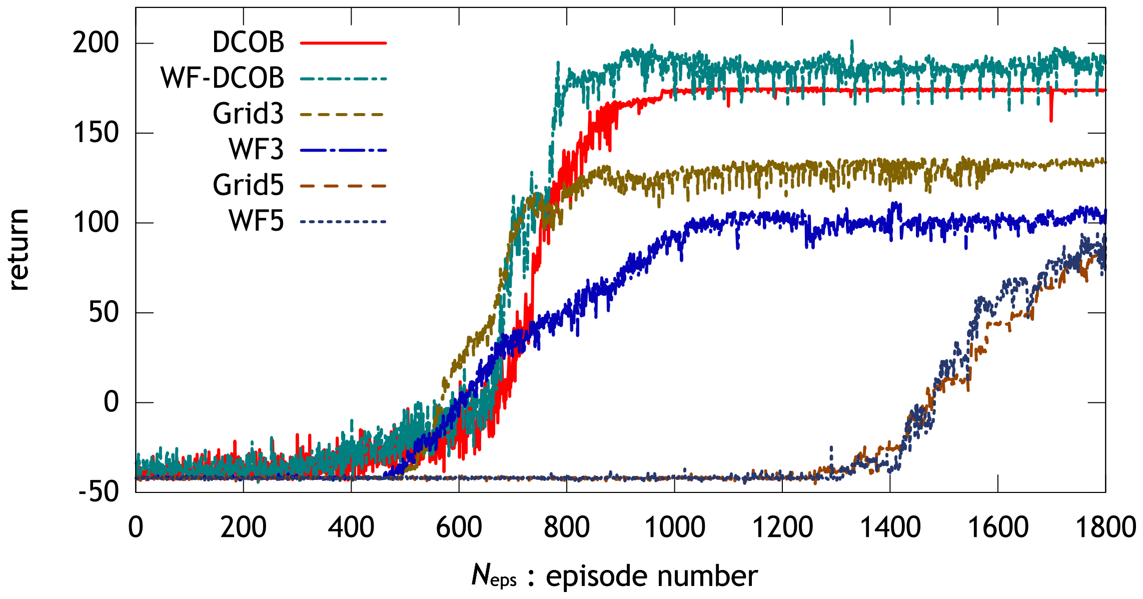


Figure 4.7 Resulting learning curves of the HumanoidML-crawling task with 5-DoF-Dyn. Each curve shows the mean of the return over 15 runs per episode.

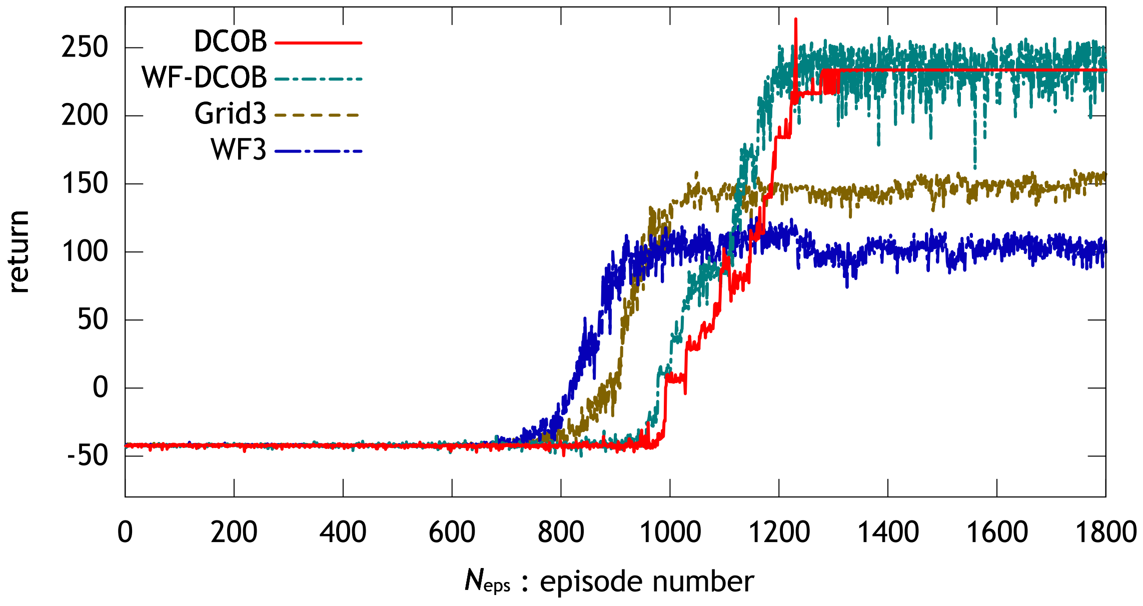


Figure 4.8 Resulting learning curves of the HumanoidML-crawling task with 5-DoF-Grid. Each curve shows the mean of the return over 15 runs per episode.

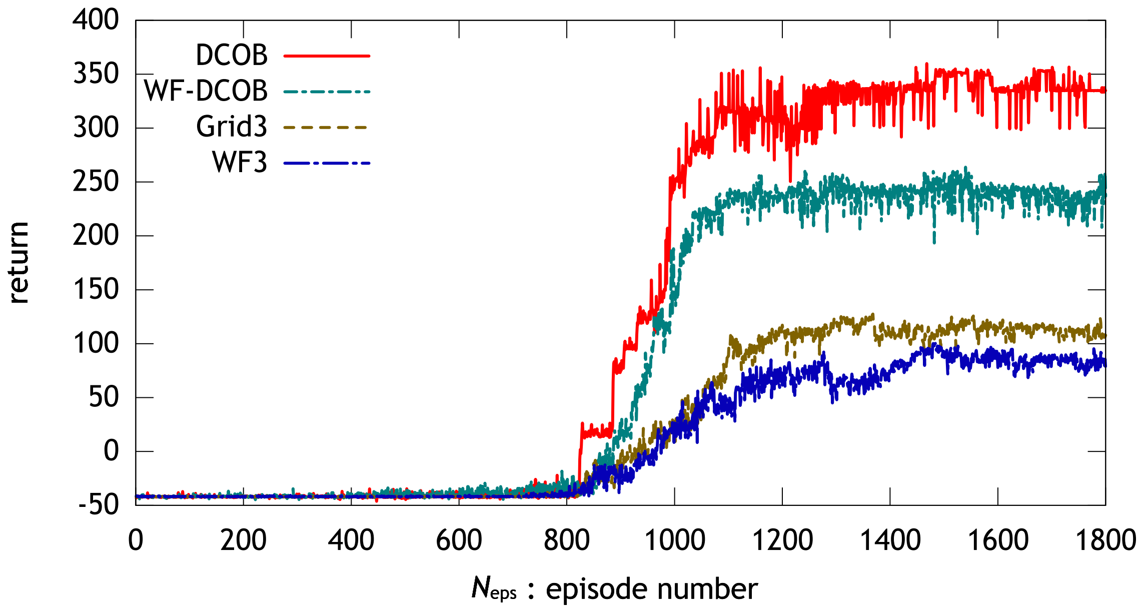


Figure 4.9 Resulting learning curves of the HumanoidML-crawling task with 5-DoF-SprDmp. Each curve shows the mean of the return over 15 runs per episode.

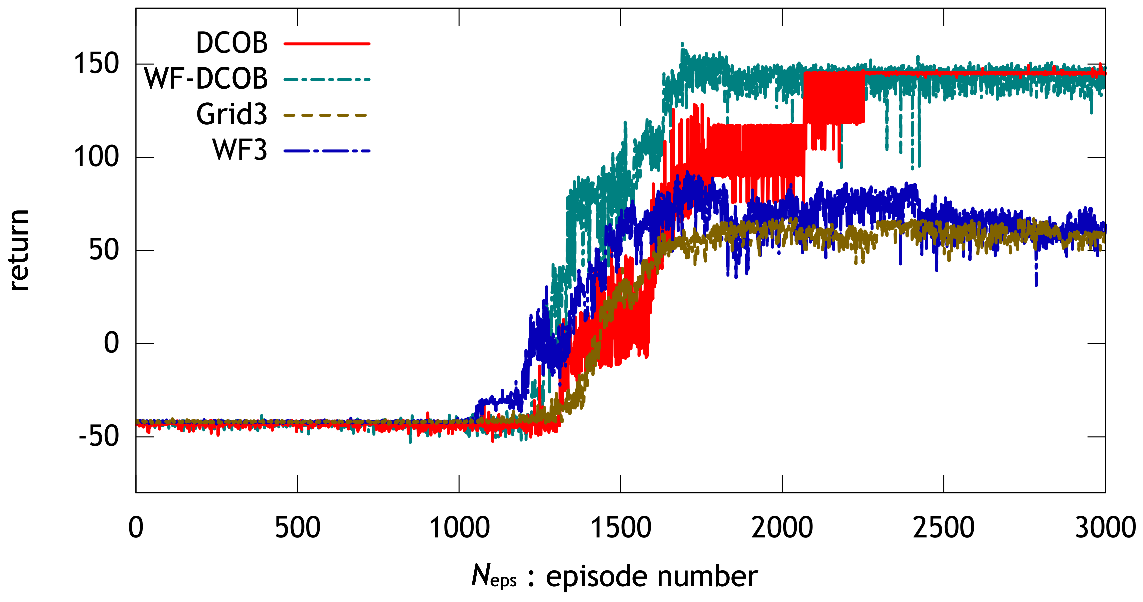


Figure 4.10 Resulting learning curves of the HumanoidML-crawling task with 6-DoF-SprDmp. Each curve shows the mean of the return over 15 runs per episode.

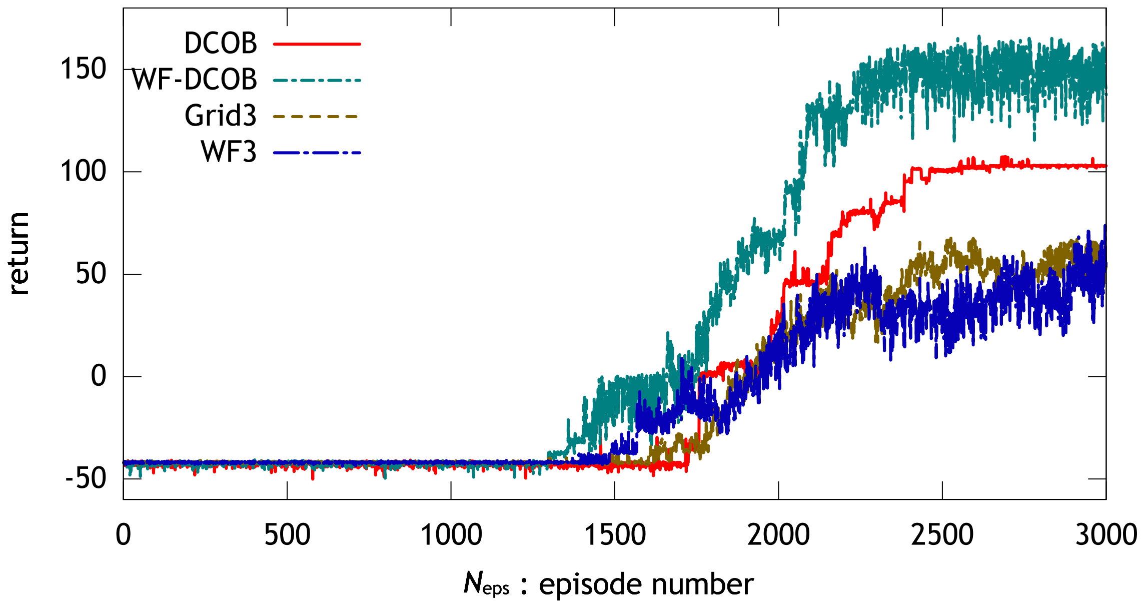


Figure 4.11 Resulting learning curves of the HumanoidML-crawling task with 7-DoF-SprDmp. Each curve shows the mean of the return over 15 runs per episode.

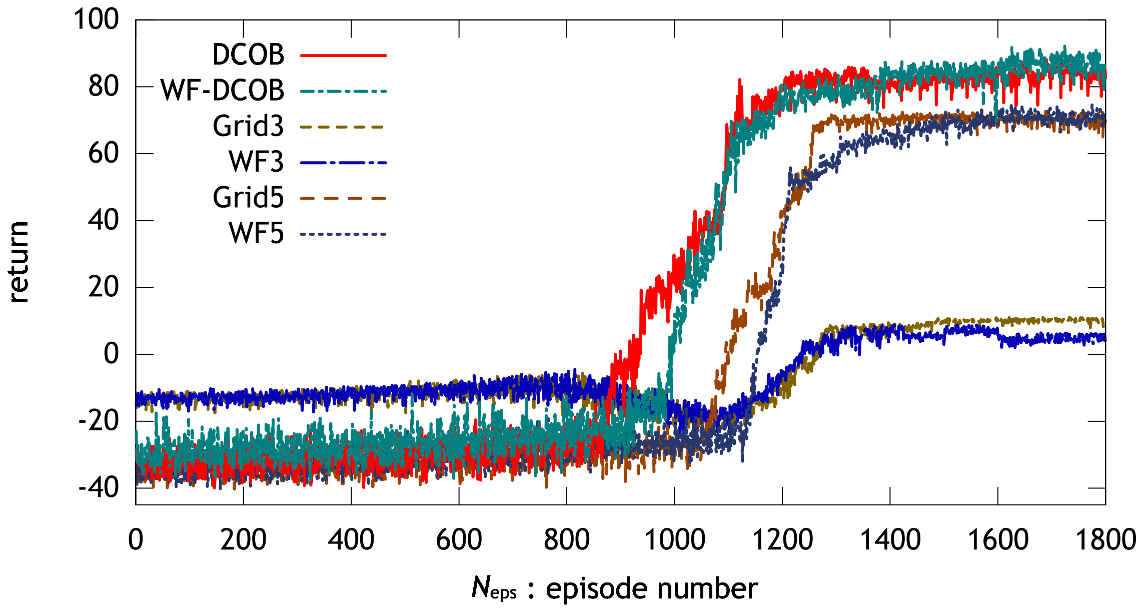
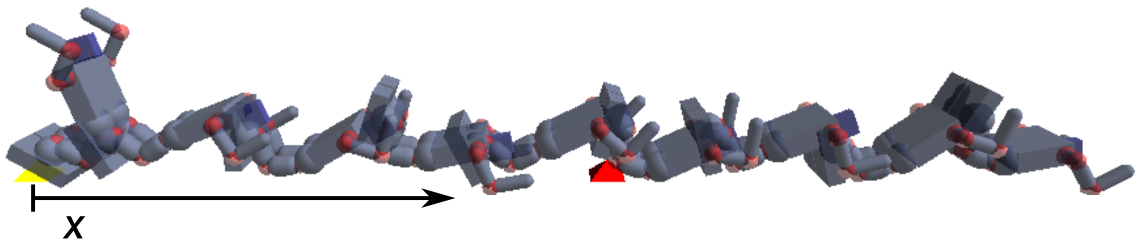
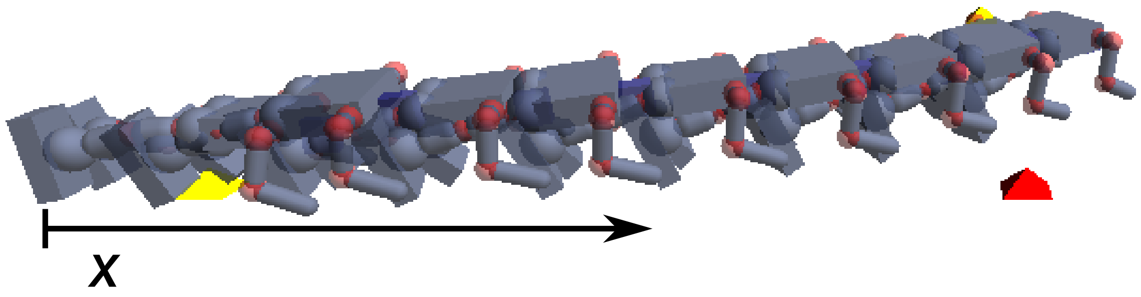


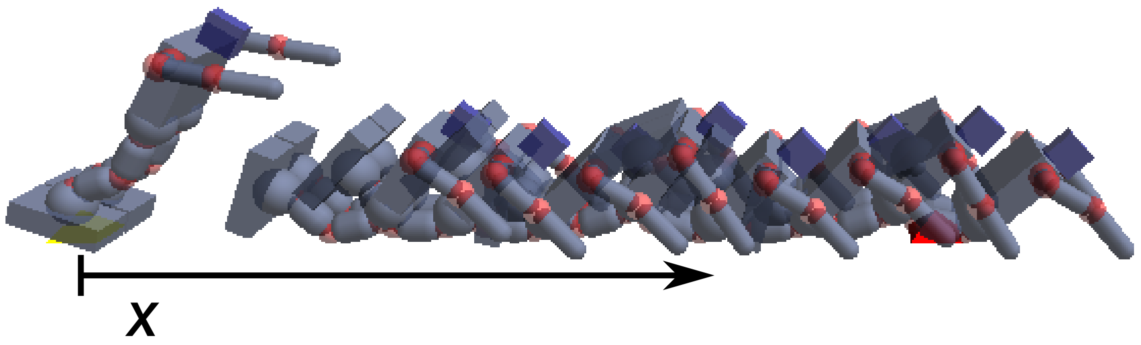
Figure 4.12 Resulting learning curves of the HumanoidML-turning task with 4-DoF-Grid. Each curve shows the mean of the return over 15 runs per episode.



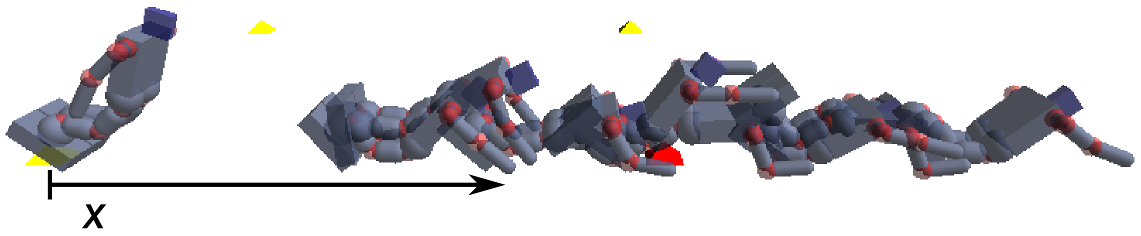
(a) Snapshots of the crawling task with 3-DoF-Grid, taken in 3-FPS.



(b) Snapshots of the crawling task with 4-DoF-Grid, taken in 3-FPS.

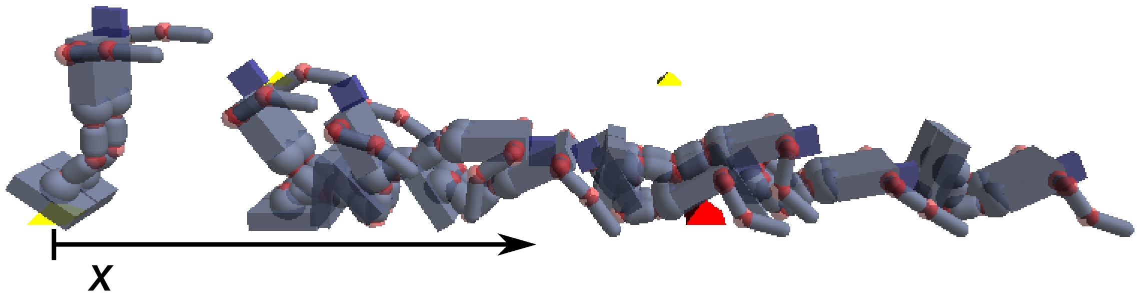


(c) Snapshots of the crawling task with 5-DoF-Dyn, taken in 3-FPS.

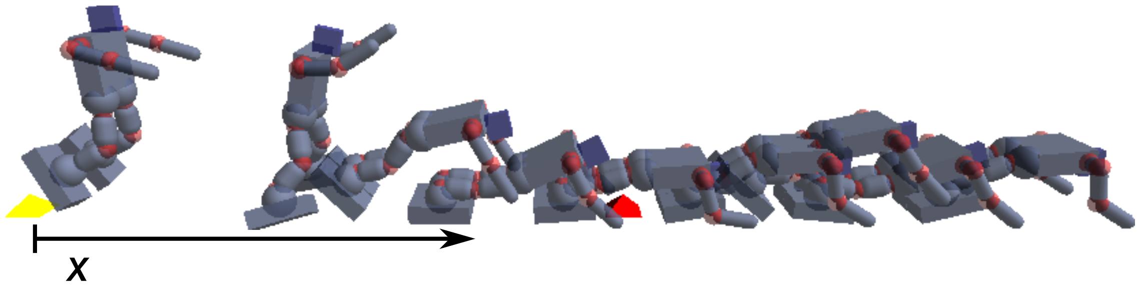


(d) Snapshots of the crawling task with 5-DoF-Grid, taken in 3-FPS.

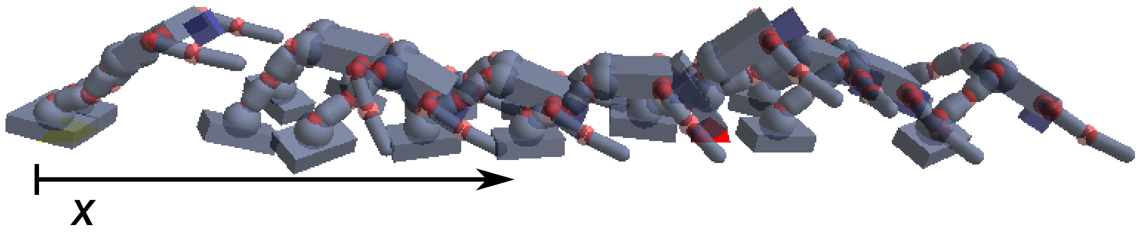
Figure 4.13 Snapshots of an acquired motion with DCOB (1).



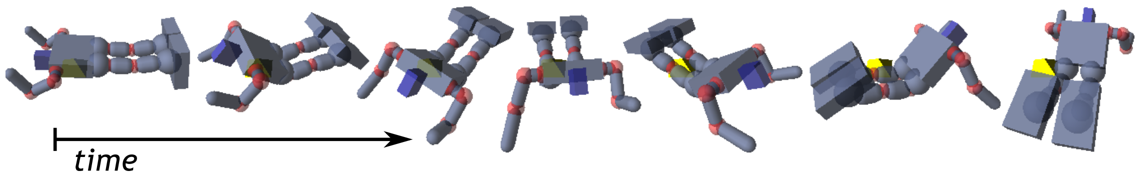
(a) Snapshots of the crawling task with 5-DoF-SprDmp, taken in 3-FPS.



(b) Snapshots of the crawling task with 6-DoF-SprDmp, taken in 3-FPS.



(c) Snapshots of the crawling task with 7-DoF-SprDmp, taken in 3-FPS.



(d) Snapshots of the turning task with 4-DoF-Grid, taken at 0[s], 0.75[s], 1.1[s], 1.35[s], 1.65[s], 2.05[s], 2.35[s].

Figure 4.14 Snapshots of an acquired motion with DCOB (2).

4.5.3 BioloidML

As an application to a real robot, experiments of the BioloidML-crawling task are demonstrated. The robot used here is a Bioloid King Spider model.

Configurations of RL Method

As an RL method, we use Peng's Q(λ)-learning for every conditions. The parameters of the Q(λ)-learning are also consistent for every conditions: $\gamma = 0.9$, $\lambda = 0.9$. We use a decreasing step-size parameter $\alpha = \alpha_0 \exp(-\delta_\alpha N_\text{eps})$ for learning, and a decreasing temperature parameter $\tau = \tau_0 \exp(-\delta_\tau N_\text{eps})$ for Boltzmann selection where N_eps denotes the episode number. These parameters are determined through preliminary experiments: $\alpha_0 = 0.3$, $\delta_\alpha = 0.002$, $\tau_0 = 0.1$, $\delta_\tau = 0.004$.

DoF Configurations and BF Allocations

We use the 5-DoF configuration and its default BF allocation, namely, the grid allocation on a $3 \times 3 \times 3 \times 3 \times 3$ grid.

Configurations of Action Spaces and Function Approximators

The following conditions are compared.

DCOB : The parameters of DCOB are as follows:

$$C_\text{p}(x) = \mathbf{q}_{N_\text{D}}, \tag{4.35}$$
$$C_\text{d}(x) = \mathbf{0}^{N_\text{D}}, \tag{4.36}$$
$$\mathcal{I} = \{0.5\}, \tag{4.37}$$
$$\text{F}_\text{abbrv} = 1, \tag{4.38}$$

where N_D=5, $\mathbf{q}_{N_\text{D}}$ denotes the joint angle vector, and $\mathbf{0}^{N_\text{D}}$ denotes N_D-dimensional zero vector. Note that the command input of the robot is the target joint angles, thus, *Ctrl* is considered to be embedded on the robot. Namely, we use $q^\text{d}(t + \delta\text{t})$ as a command input. As a function approximator, LFA-NGnet is used.

Grid Action Set (Grid3) : The grid action set $\mathcal{A}_\text{G}$ is employed that is defined in Section 4.5.2. The parameters are $\Delta\varphi = \pi/12$, and $\text{T}_\text{G} = 0.1[\text{s}]$. As the number of division, $\text{N}_\text{grid} = 3$ is used, which is referred to as Grid3. LFA-NGnet is used as a function approximator.

Results

For each conditions DCOB and Grid3, 4 runs are performed. Figure 4.15 shows obtained learning-curves. Figure 4.16 shows the mean and ±1 standard deviation of 4 runs in 0th to 77th episode where every runs are overlapping. Figure 4.17 shows the mean and ±1 standard deviation in the last 10 episodes. Figure 4.18 demonstrates a motion acquired with DCOB. Figure 4.16 shows that with DCOB, a higher return is obtained around 20th to 30th episode, which is faster than Grid3. Meanwhile, Figure 4.17 indicates the performance of the acquired motions; the graph shows that DCOB acquires motions of higher performance than Grid3. Therefore, DCOB also outperforms Grid3 both in learning speed and acquired motion performance.

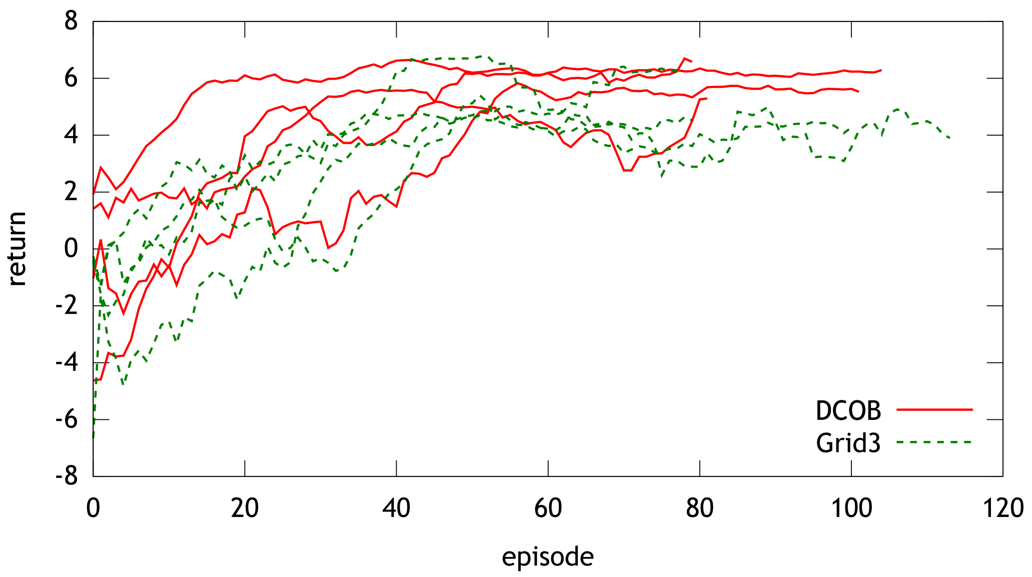


Figure 4.15 Resulting learning curves of the crawling task. Each curve shows the return per episode in a run. To see the tendency of each curve, a low-pass filter with a time constant of 10 episode is applied.

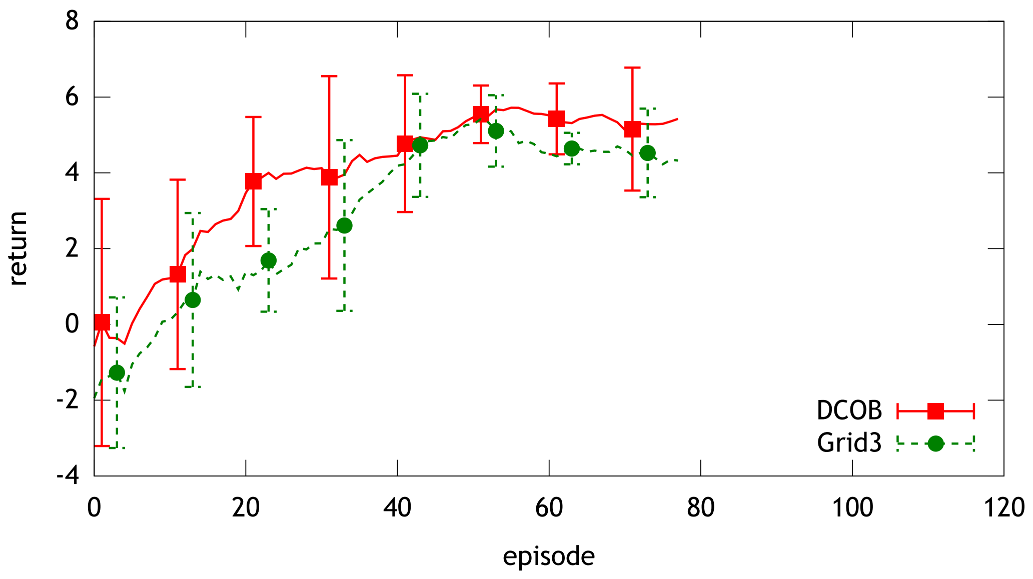


Figure 4.16 Averaged learning curves of the crawling task. Each curve shows the mean of the return over 4 runs in 0th to 77th episode. Error bar denotes ±1 standard deviation. A low-pass filter with a time constant of 10 episode is also applied.

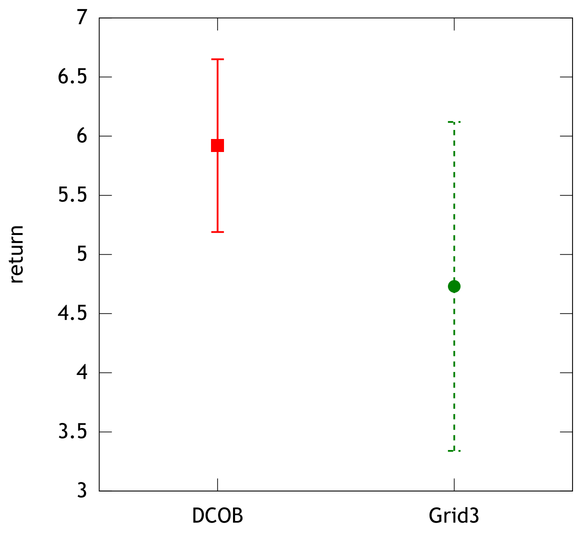


Figure 4.17 Performance of the acquired motion: the average and the ±1 standard deviation of the return over the last 10 episodes.

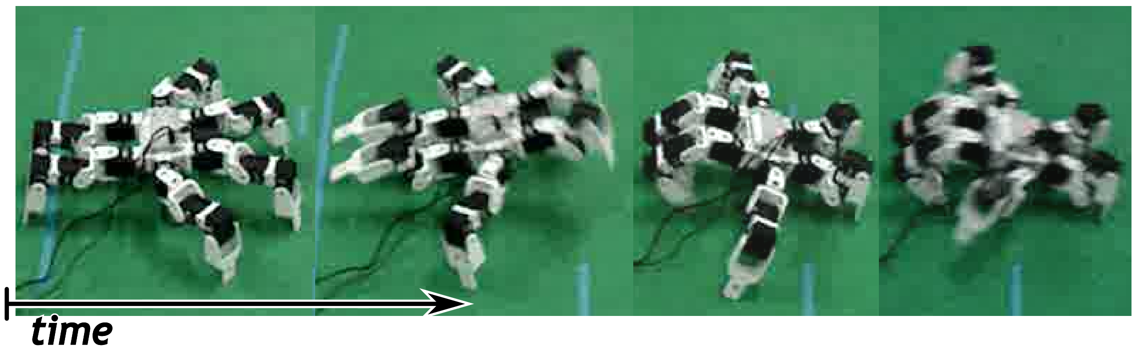


Figure 4.18 Snapshots of acquired crawling motion of the King Spider (4.8[s], 5.4[s], 6.0[s], 7.2[s]).

Demonstration of Dinosaur

Next, DCOB is applied to a crawling task of a Bioloid of the Dinosaur model. In this case, the task setup is almost the same as the King Spider case except for the number of the actuators. The differences are (1) the reward is multiplied by 5 since the Dinosaur is slower than the King Spider because of its shorter legs, (2) a penalty is given when the robot falls down (which the operator determines), and (3) the IR sensor is mounted on the head.

Figure 4.19 shows the resulting learning curves in 3 runs, and Figure 4.20 shows snapshots of an acquired crawling motion. Learning speed is slower than that of the King Spider's case. The possible reason is that the Dinosaur sometimes falls down, which makes the task more difficult than in the King Spider's case.

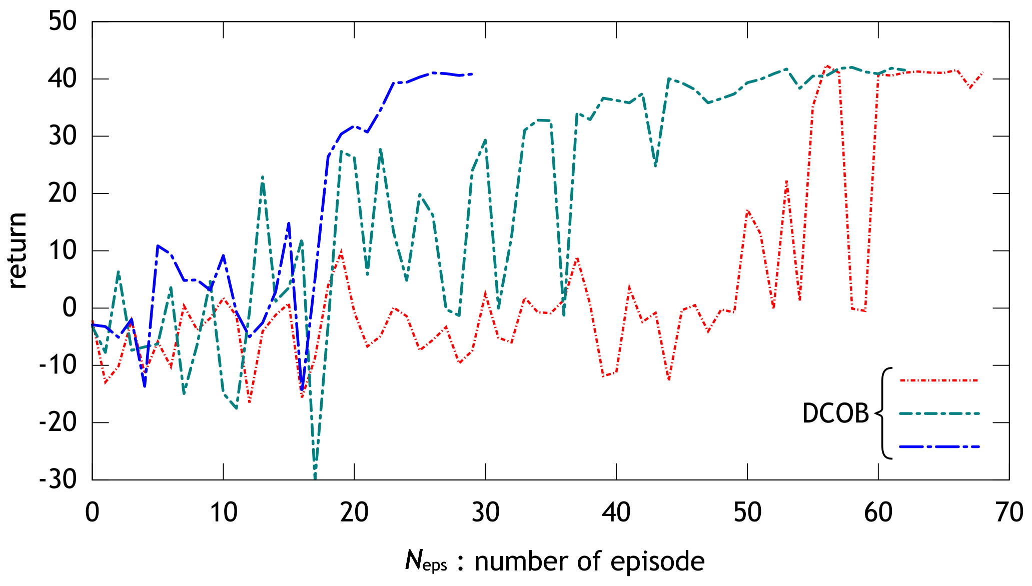


Figure 4.19 Resulting learning curves of the crawling task of the dinosaur robot. Each curve shows the return per episode in a run.

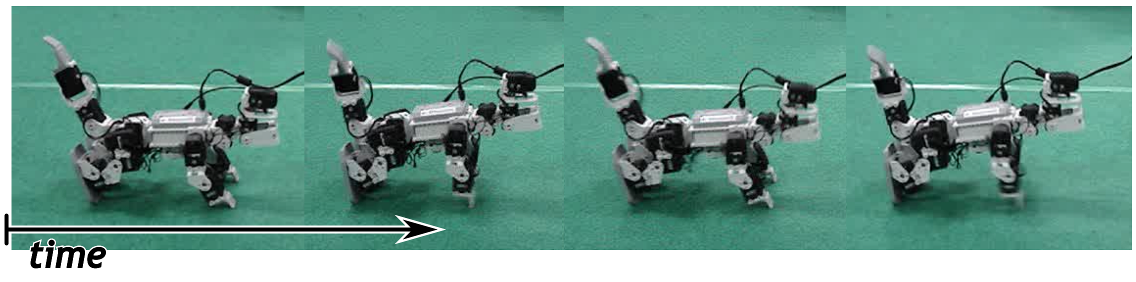


Figure 4.20 Snapshots of an acquired crawling motion of the Dinosaur (7.2[s], 7.6[s], 8.0[s], 8.6[s]).

4.6 Discussion

4.6.1 Convergence of RL with BFTrans

This section discusses the convergence of RL methods with BFTrans. The convergence of some RL methods depends on the Markov property of the task (e.g. (Tsitsiklis and Roy 1996; Tsitsiklis and Roy 1997)), thus, we clarify the Markov property of the task using BFTrans.

BFTrans converts the control command space $\tilde{u} \in \tilde{\mathcal{U}}$ to the action space $u \in \mathcal{U}$. With this conversion, the time sequence, the state transition probability, and the reward function change. The time sequence changes from a continuous time to a discrete time associated with the action sequence. It is defined by

$$t_0 = 0, \quad t_n = \sum_{n'=0}^{n-1} T_{\mathrm{n}n'}, \tag{4.39}$$

where $T_{\mathrm{n}n}$ denotes the execution time of the action u_n at step n which depends on u_n and the state $x_n = x(t_n)$. We define the new reward function as

$$R_n = \int_{t_n}^{t_n + T_{\mathrm{n}n}} r(t)\mathrm{d}t \tag{4.40}$$

where $r(t)$ is the reward at time t. If the original state transition probability depends only on the current state and the control input, i.e. $P(x'|x, \tilde{u})$, the new state transition probability depends only on the current state and the current action; $P(x_{n+1}|x_n, u_n)$. This is because the reference trajectory of the action in BFTrans is determined by only x_n and u_n, thus, the command sequence $\tilde{u}(t)$, $t \in [t_n, t_n + T_{\mathrm{n}n'})$ depends only on x_n and u_n. Therefore, if the original task has the Markov property, the converted task also has the Markov property.

Note that DCOB which discretizes BFTrans also has the same property, but, this is not the case with WF-DCOB because of its nonlinearity. That is, even if the task has the Markov property and we employ an RL algorithm that guarantees the convergence with a linear function approximator, this RL algorithm with the nonlinear function approximator does not guarantee the convergence (Tsitsiklis and Roy 1997).

4.6.2 Computational Cost of BFTrans

The computational cost of the generating trajectory step and the following trajectory step is $\mathcal{O}(\dim(\mathcal{Q}))$. The abbreviating trajectory step requires $\mathcal{O}(|\mathcal{K}|^2)$

to compute $\{d_{\mathrm{n}}(k)|k \in \mathcal{K}\}$, but d_{n} stays constant during learning for fixed BFs. Thus, we can pre-compute $\{d_{\mathrm{n}}(k)\}$. Eventually, the abbreviating trajectory step requires $\mathcal{O}(|\mathcal{K}|)$, which is the total computational cost of each BFTrans action (in general, $\dim(\mathcal{Q}) < |\mathcal{K}|$). Note that this cost is the same as evaluating all BFs, which is required in each action selection.

4.6.3 Available Types of Basis Functions

In this thesis, the BFs are assumed to be normalized Gaussian network (NGnet), but, the proposed methods (BFTrans, DCOB, and WF-DCOB) work with the other types of BFs if assumption (A) described in Section 4.2.1 is satisfied. In such cases, a required modification is to define $\mathrm{d}_{\mathrm{min}\,k}$ in eq. (4.5). Concretely, available types of BFs are a radial basis function (RBF) and the one used by Takahashi *et al.* (Takahashi, Takeda, and Asada 1999).

Note that the major factor influencing the performance of a motion is the allocation of the centers of the BFs rather than the type of BFs, since only the centers are used to calculate an action in BFTrans.

4.6.4 Related Works

Options

Sutton *et al.* proposed *options* which are generalized actions of primitive and macro actions under the RL framework (Sutton, Precup, and Singh 1999). Our DCOB can be regarded as a kind of the options specialized for robot control. There are some researches about finding options or subgoals automatically (Mcgovern and Barto 2001; Menache, Mannor, and Shimkin 2002; Stolle 2004), however the discovery of the optimal options is still an open problem. DCOB is considered to be a practical solution to it.

Parti-game Algorithm

Moore and Atkeson proposed *parti-game algorithm* as an RL method for continuous state-action spaces (Moore and Atkeson 1995). This method assumes a local controller to move to a near state, and searches an optimal policy in partitions on the state space where each action is a transition to a neighboring partition. Its feature is that partitioning of the state space is also optimized, which is superior to DCOB where the allocation of the given BFs is not changed. However, the applicable tasks of parti-game algorithm are limited; a task should have a goal state which should be explicitly given. That is, it is not applicable to the crawling task used in this thesis. DCOB can be treated as an usual action space, which does not restrict the reward function.

The Other Action Spaces

Using an RL-compatible action space instead of the command space is a common approach in RL applications. A typical way is using a PD controller and training an RL agent to learn its target value (e.g. (Morimoto and Doya 1998)). Using central pattern generator (CPG) and letting an RL agent learn its parameters is effective to learn rhythmic motions, such as walking (e.g. (Nakamura, Mori, Sato, and Ishii 2007)). The proposed methods (DCOB and WF-DCOB) are applicable to episodic tasks, such as jumping, that are difficult to learn with CPG.

Ijspeert and Nakanishi *et al.* proposed nonlinear dynamic motor primitives for robot control (Ijspeert, Nakanishi, and Schaal 2002). Later, Peters *et al.* developed an RL method to optimize the primitives' parameters (Peters, Vijayakumar, and Schaal 2003). This framework is similar to WF-DCOB since both methods construct RL-compatible action spaces and apply RL methods for a continuous action space. However, the primitive based approach assumes that the parameters are initialized through an imitation learning framework. Thus, in learning in a large domain from scratch, the performance of the acquired motion will converge to a lower local maximum. Meanwhile, WF-DCOB provides a proper parameter initialization even for large domains. Miyamoto *et al.* proposed an RL method with via-point representation (Miyamoto, Morimoto, Doya, and Kawato 2004). This method is also regarded as an action space construction method, however it also has an issue of lacking proper parameter initialization.

4.7 Conclusion

This chapter proposed a discrete action set DCOB for RL methods to handle domains of higher dimensional control input space. DCOB is generated from a set of BFs given to approximate a value function. DCOB is a discrete set but it has an ability to acquire high performance motions. Moreover, using the dynamics-based BF allocation or the spring-damper BF allocation reduces the size of DCOB, which improves the learning speed. In addition, a method called WF-DCOB was proposed to enhance the performance, where wire-fitting is employed to search continuous actions around each discrete action of DCOB. In WF-DCOB, to relax the learning instability of wire-fitting, a parameter initialization and a constraint method are proposed.

The proposed methods were applied to simulation tasks (Maze2D, HumanoidML-crawling and turning) and real robot tasks (BioloidML-crawling). In the simulation tasks, DCOB and WF-DCOB were compared with conventional action spaces. Moreover, in HumanoidML, the methods were compared in different DoF configurations and different BF-allocation methods. In every experiments, DCOB and WF-DCOB outperformed the other methods. However, though WF-DCOB was superior to DCOB in some tasks, there were cases where DCOB was better. A possible reason is that WF-DCOB can represent wider policy than DCOB, but, suffers from the learning instability of wire-fitting. From the results of HumanoidML, guidelines for choosing DCOB or WF-DCOB are as follows: use DCOB if a relatively sufficient number of BFs can be allocated in a state space (e.g., lower DoF cases); use WF-DCOB if it is difficult to allocate the enough number of BFs (e.g., higher DoF cases).

A disadvantage of the proposed methods is the assumption of fixed BFs. Optimizing the parameters of BFs sometimes can avoid the effects of the curse of dimensionality. For instance, learning the nonlinear parameters of the sigmoid functions in a neural network significantly reduces the approximation error compared to learning only the linear weights of the network (Barron 1993). DCOB assumes that the given BFs have already avoided the curse of dimensionality, but this assumption is not always satisfied. Some RL researches reported that updating not only the linear weights of NGnet but also the means and the covariance matrices can improve learning (Morimoto and Doya 1998; Kondo and Ito 2004). Thus, removing the assumption of fixed BFs from DCOB and WF-DCOB is considered to be important for wider applicability.

Chapter 5

Dynamics and Reward Models

Learning a dynamics and a reward model during reinforcement learning is useful since the agent can also update its value function by using the models. This chapter proposes a general dynamics model that is a composition of the feature space dynamics model and the state space dynamics model. This method enables to obtain good generalization ability from a small number of samples because of the linearity of the state space dynamics, while it does not lose accuracy because of the feature space dynamics model. Moreover, in some cases, reusability can be inherited from the generalization ability of the state space dynamics model. We apply a Dyna algorithm with our dynamics model to a navigation task, which demonstrates the advantage of our dynamics model.

5.1 Introduction

Using a dynamics or a reward model is a way to utilize prior knowledge about the task. The so-called model-based RL is an RL framework where an agent has a dynamics model and a reward model, and computes a policy with the models. These models are sometimes incomplete and will be updated while learning. A well-known architecture is the Dyna proposed by Sutton (Sutton 1990). The convergence of the Dyna with a linear function approximator is proven by Sutton *et al.* (Sutton, Szepesvári, Geramifard, and Bowling 2008). In the Dyna architecture, a normal (i.e. model-free) RL, learning models, and a model-based RL (i.e. planning) are performed simultaneously. Although the Dyna is a general architecture where both a model-based and a model-free RL are combined, there are some similar approaches that utilize models (Rottmann and Burgard 2009; Farahmand, Shademan, Jägersand, and Szepesvári 2009; Park, Kim, and Song 2007).

The important abilities of models used in the Dyna architecture or the other model-based RLs are: (1) the model accuracy, (2) the model generalization ability, and (3) the model *portability*. Generalization means acquiring accuracy from less data. Model portability means both the ability to reuse model parameters learned from other tasks and the ability to encode prior knowledge.

This chapter proposes a composite dynamics model of the feature space dynamics and the state space dynamics which improves these abilities. Here, the feature is defined as the output of the basis functions (that we assume to be local models) that are used to approximate a value function. Usually, the dynamics model is represented by a transition matrix of the feature. This dynamics model in the *feature space* is accurate, but has poor generalization ability and poor reusability. Instead, the dynamics model in the *state space* may provide good generalization ability, since in many practical applications, the dynamics of the environment (including the robot) is nearly linear in the state space. Typical examples are navigation tasks and robotic manipulations. Moreover, in some cases (e.g. navigation tasks), reusability can be inherited from the generalization ability of the state space dynamics model. Our proposed composite dynamics model, which we call MixFS dynamics model, can exploit the advantages of both dynamics models. Thus, the MixFS dynamics model acquires the improved abilities mentioned above.

The main contributions are (1) combining the two dynamics models and making it work with the Dyna algorithms, and (2) deriving an on-line learning method for the developed model. Another contribution is (3) providing a method to reuse the MixFS dynamics model parameters, and a method to encode prior knowledge to the reward model. We apply a Dyna with MixFS dynamics model to several conditions of a navigation task: (a) two types of the mazes, (b) three types of prior knowledge about rewards, and (c) reused/scratch dynamics model parameters. The simulation results demonstrate the advantages of MixFS which come from the abilities mentioned above.

The rest of this chapter is organized as follows. Section 5.2 introduces a Dyna architecture with a linear function approximator. Section 5.3 proposes a composite dynamics model of the feature space dynamics and the state space dynamics. Section 5.4 describes a method to reuse the learned dynamics model parameters, and a method to embed prior knowledge on the reward model parameters. Section 5.5 demonstrates some experiments. Finally, Section 5.6 concludes the chapter.

5.2 Dyna with Linear Function Approximator

This section introduces a Dyna architecture with a linear function approximator.

5.2.1 Linear Dyna with 'MG' Prioritized Sweeping

McMahan and Gordon proposed the Improved Prioritized Sweeping algorithm that is a fast planning algorithm in a Markov Decision Process (MDP) (McMahan and Gordon 2005). Sutton *et al.* called the algorithm *MG prioritized sweeping*, and developed a Dyna using it, *Dyna-MG*, which was faster than Dyna algorithms with the other prioritized sweeping methods (Sutton, Szepesvári, Geramifard, and Bowling 2008). Thus, we use Dyna-MG.

The algorithm of Dyna-MG is given as Algorithm 3. Here, $\{F_a, b_a | a \in \mathcal{A}\}$ denotes the parameters of the dynamics model and the reward model defined as follows:

$$\phi' \approx F_a \phi(x), \tag{5.1}$$

$$R \approx b_a^\top \phi(x). \tag{5.2}$$

$N_{\text{pln}} \in \mathbb{N}$ denotes a planning depth (a given constant), *PQueue* denotes a priority queue whose $pop()$ operator removes and returns the queue element of the highest priority, $Th_1 \in [0, 1]$ is a threshold that decides to include the *PQueue*, $\mathbf{e}_j \in \mathbb{R}^{|\mathcal{K}| \times 1}$ denotes a vector whose j-th element is 1 and the rest are 0, $\boxdot_{[i]}$ denotes the i-th element of a vector $\boxdot$, $\odot_{[i,j]}$ denotes the (i, j)-th element of a matrix $\odot$. $pred(i)$ denotes the set of all pairs of a feature index and an action (j, a) such that taking action a from the feature index j has a positive chance to reach the feature index i. In this thesis, we define the $\{(j, a)\} = pred(i)$ as whole pairs that satisfy all of the following conditions:

$$j \in \{1, \ldots, |\mathcal{K}|\}, \qquad i \neq j, \tag{5.3}$$

$$a = \arg\max_{a'} F_{a'[i,j]}, \qquad F_{a[i,j]} > Th_2, \tag{5.4}$$

where $Th_2 \in [0, 1]$ is a threshold. Note that $F_{a[i,j]}$ denotes the transition probability from the feature index j to i by the action a.

Algorithm 3: Dyna-MG

```
Input : State space $\mathcal{X}$ (continuous), action set $\mathcal{A}$ (discrete),
        basis functions $\phi : \mathcal{X} \to \mathbb{R}^{|\mathcal{K}|\times 1}$
Output : Coefficient vector $\theta_a$ such that $\tilde{Q}(x,a) \approx \theta_a^\top \phi(x)$
 1: Initialize: $\theta_a, F_a, b_a$
 2: for $N_{\text{eps}} = 1, 2, \ldots$ do /* $N_{\text{eps}}$: episode */
 3:   $n \leftarrow 1$ /* time index */
 4:   Choose a start state $x_n \in \mathcal{X}$
 5:   $\phi_n \leftarrow \phi(x_n)$
 6:   while not is-end-of-episode$(x_n)$ do
 7:     Carry out an action $a_n$ according to current policy, producing a reward $R_n$ and
        next state $x_{n+1}$
 8:     $\phi_{n+1} \leftarrow \phi(x_{n+1})$
 9:     Update policy by Q(0)-learning:
10:       $\delta_n \leftarrow R_n + \gamma \max_a \theta_a^\top \phi_{n+1} - \theta_{a_n}^\top \phi_n$ /* TD error */
11:       $\theta_{a_n} \leftarrow \theta_{a_n} + \alpha \phi_n \delta_n$
12:     Update models:
13:       $F_{a_n} \leftarrow F_{a_n} + \alpha(\phi_{n+1} - F_{a_n}\phi_n)\phi_n^\top$
14:       $b_{a_n} \leftarrow b_{a_n} + \alpha(R_n - b_{a_n}^\top \phi_n)\phi_n$
15:     for $i = 1, \ldots, |\mathcal{K}|$ do
16:       if $\phi_{n[i]} > Th_1$ then
17:         Put $i$ on the PQueue with priority $|\delta_n \phi_{n[i]}|$
18:     for $1, \ldots, N_{\text{pln}}$ do
19:       if PQueue is empty then break
20:       $i \leftarrow$ PQueue.pop()
21:       for all $(j, a) \in pred(i)$ do
22:         $\delta \leftarrow b_{a[j]} + \gamma \max_{a'} \theta_{a'}^\top (F_a \mathbf{e}_j) - \theta_{a[j]}$
23:         $\theta_{a[j]} \leftarrow \theta_{a[j]} + \alpha\delta$
24:         Put $j$ on the PQueue with priority $|\delta F_{a[i,j]}|$
25:     $n \leftarrow n + 1$
```

5.3 Composing Dynamics Models

As mentioned in Section 5.1, the dynamics model of the feature space is accurate, while the dynamics model of the state space has good generalization ability. Thus, we develop a composite dynamics model of the feature space dynamics and the state space dynamics to exploit both advantages. We call this model the *MixFS dynamics model*.

5.3.1 MixFS Dynamics Model

To use a dynamics model in the prioritized sweeping, a transition probability from a feature index i to j by an action a should be calculated. Hence, we define MixFS dynamics model as

$$\phi' \approx \tilde{f}_a(\phi(x), x), \tag{5.5}$$

where $\tilde{f}_a$ denotes a function approximator that estimates the succeeding feature vector ϕ' from a state x and a corresponding feature $\phi(x)$ by an action a. The parameters of this model are learned within the Dyna algorithm. Then we compute the feature transition matrix; for all feature index pairs $i, j \in \{1, \ldots, |\mathcal{K}|\}$ and action $a \in \mathcal{A}$,

$$F_{a[j,i]} \leftarrow \tilde{f}_a(\mathbf{e}_i, \mu_i)_{[j]} \tag{5.6}$$

where $\mathbf{e}_i \in \mathbb{R}^{|\mathcal{K}|\times 1}$ is defined in Section 5.2.1. If the dynamics model $\tilde{f}_a$ is the conventional one, i.e. eq. (5.1), the equation above correctly extracts F_a of eq. (5.1) by using $\mathbf{e}_i$. Otherwise, the equation above approximates the transition probability by representing x with the center of the basis function μ_i. The obtained feature transition matrix is directly used in the prioritized sweeping part of the Dyna algorithm.

For the function approximator $\tilde{f}_a$, we choose simple linear models for two dynamics models, and combine them linearly. Thus, $\tilde{f}_a$ is defined by

$$\tilde{f}_a(\phi(x), x) = \delta F_a \phi(x) + \phi(x + \delta\tilde{x}_a(x)) \tag{5.7}$$

$$\delta\tilde{x}_a(x) = A_a x + B_a \phi(x) + d_a \tag{5.8}$$

where $\delta F_a \in \mathbb{R}^{|\mathcal{K}|\times|\mathcal{K}|}$, $A_a \in \mathbb{R}^{\dim(\mathcal{X})\times\dim(\mathcal{X})}$, $B_a \in \mathbb{R}^{\dim(\mathcal{X})\times|\mathcal{K}|}$, $d_a \in \mathbb{R}^{\dim(\mathcal{X})\times 1}$ are the model parameters. To further understand, let us think about a model, $\tilde{f}'_a(\phi(x), x) = \phi(x + \delta\tilde{x}_a(x))$. Here, $x + \delta\tilde{x}_a(x)$ is a linear state space dynamics model, that is, it can estimate the succeeding state from the state x by the action a. Thus, the model $\tilde{f}'_a(\phi(x), x)$ estimates the succeeding feature vector from x by a. Clearly, MixFS dynamics model eq. (5.7) is a composite dynamics model.

5.3.2 Learning MixFS Dynamics Model

Next, we derive an on-line learning algorithm for the model parameters of MixFS dynamics model, with which the *Update models* part of the Dyna algorithm is replaced. Specifically, the learning algorithm updates δF_a, A_a, B_a, and d_a so that $\tilde{f}_a(\phi(x), x)$ can estimate the succeeding feature vector ϕ' from the observation data x_n, ϕ_n, x_{n+1}, ϕ_{n+1}. We simply use an on-line gradient descent algorithm (Bishop 2006).

However, it is difficult to update A_a, B_a and d_a in a straightforward manner, since these parameters are enveloped by the basis functions $\phi(x)$ that makes it complex to calculate the gradient. To overcome this difficulty, we separate the learning problem into two steps. First, the model parameters of the state space dynamics model A_a, B_a, d_a are updated so that $x + \delta\tilde{x}_a(x)$ can estimate the succeeding state x'. Second, the model parameter of the feature space dynamics model δF_a is updated so that $\tilde{f}_a(\phi(x), x)$ can estimate the succeeding feature vector ϕ'. Thus, the model parameters of MixFS dynamics model are updated with Algorithm 4.

Algorithm 4: Update MixFS dynamics model parameters

```
Input : Current and succeeding state $x, x' \in \mathcal{X}$, action $a \in \mathcal{A}$,
        basis functions $\phi : \mathcal{X} \to \mathbb{R}^{|\mathcal{K}| \times 1}$, step size $\alpha$,
        current model parameters $\delta F_a$, $A_a$, $B_a$, $d_a$
Output : Updated model parameters $\delta F'_a$, $A'_a$, $B'_a$, $d'_a$
1: $\tilde{x}' \leftarrow x + A_a x + B_a \phi(x) + d_a$
2: $\tilde{\phi}' \leftarrow \delta F_a \phi(x) + \phi(\tilde{x}')$
3: $A'_a \leftarrow A_a + \alpha(x' - \tilde{x}')x^\top$
4: $B'_a \leftarrow B_a + \alpha(x' - \tilde{x}')\phi(x)^\top$
5: $d'_a \leftarrow d_a + \alpha(x' - \tilde{x}')$
6: $\delta F'_a \leftarrow \delta F_a + \alpha(\phi(x') - \tilde{\phi}')\phi(x)^\top$
```

5.3.3 Computational Techniques

Constraint for Numerical Stability

The feature transition matrix F_a is assumed to be encoding transition probabilities of the MDP. However, due to the estimation error, F_a sometimes takes an irregular value, which makes the planning unstable. Thus, we constrain the F_a as follows:

```
for all $i, j \in \{1, \ldots, |\mathcal{K}|\}$:
    if $F_{a[j,i]} < 0$ then: $F_{a[j,i]} \leftarrow 0$
    if $F_{a[j,i]} > 1$ then: $F_{a[j,i]} \leftarrow 1$
for all $i \in \{1, \ldots, |\mathcal{K}|\}$:
    if $\sum_{j'} F_{a[j',i]} > 1$ then:
        for all $j \in \{1, \ldots, |\mathcal{K}|\}$: $F_{a[j,i]} \leftarrow \frac{F_{a[j,i]}}{\sum_{j'} F_{a[j',i]}}$
```

The first part constrains the range of the $F_{a[j,i]}$ to $[0, 1]$ since it represents a transition probability in the MDP. The second part constrains the sum of the transition probability w.r.t. the succeeding feature index j' from i by an action a, which should be 1.

Fast Computation

After updating the model parameters of MixFS, the feature transition matrix F_a should be computed by eq. (5.6), but it requires computational cost. Since the feature transition matrix is required only when the planning is executed in Dyna-MG, i.e. $\mathit{PQueue} \neq \emptyset$, the feature transition matrix has to be calculated only when it is demanded. To do this, $\mathit{flag}_a \in \{\texttt{true}, \texttt{false}\}$ is prepared for each action $a \in \mathcal{A}$ to judge whether F_a was already calculated for the latest model parameters.

5.4 Embedding Prior Knowledge on Models

In this section, we describe a method to reuse the learned dynamics model parameters, and a method to embed prior knowledge on the reward model parameters.

5.4.1 Reusing Dynamics Model Parameters

Let us consider how the dynamics model parameters learned in a task are reused in other tasks. Here, we assume reusing between the tasks that have the same state/action spaces. Even in such cases, the dynamics may be different; for example, navigation tasks that have different mazes. Factors to be taken into account in reusing are (1) reusing the task-specific parameters makes incorrect planning, and (2) the set of the basis functions $\mathcal{K}$ is generally defined for each task. Thus, the model parameters related to ϕ (in MixFS case, δF_a and B_a) are omitted, and the others (A_a and d_a) are reused. The dynamics $A_a x + d_a$ can be regarded as a linearized state space dynamics. Thus, for instance, it is possible for the parameters A_a and d_a to encode how a humanoid robot moves by a walking action in a navigation task where the effect of the walls are omitted.

5.4.2 Embedding Reward Sources

If the state and the reward of the goal are known, we can embed them on the reward model as prior knowledge. A model-based RL method can exploit this kind of information by planning. A general way to embed this kind of information, i.e. the *reward sources*, into the reward model is formulated and solved as follows:

For given reward sources $\{(x_i, R_i) | i = 1, \dots, N_{\mathrm{rsrc}}\}$, estimate $b_a, a \in \mathcal{A}$ to satisfy

$$\begin{bmatrix} R_1 \\ \vdots \\ R_{N_{\mathrm{rsrc}}} \end{bmatrix} = \begin{bmatrix} \phi(x_1) & \cdots & \phi(x_{N_{\mathrm{rsrc}}}) \end{bmatrix}^\top b_a. \tag{5.9}$$

Letting this equation $R = \Phi b_a$, the solution is $b_a = \Phi^\sharp R$ where $\Phi^\sharp$ denotes the pseudo-inverse of the matrix Φ.

Note that the reward model obtained by this method is slightly different from the correct one since the goal reward is given when the *succeeding* state is the goal state, but the reward model above states that the goal reward is given for any action from the goal state. This problem is addressed through learning.

5.5 Experiments

5.5.1 Accuracy of Dynamics Models

First, we compare the approximation accuracy of the two dynamics models in an environment of simple dynamics. One is the proposed MixFS dynamics model. The other is a traditional linear dynamics model of the feature space used in Dyna-MG mentioned in Section 5.2.1, which we refer to as the *Simple dynamics model*.

We employ a robot that has a 1-dimensional state space $\mathcal{X} \subset \mathbb{R}$ and only 1-element action set $\mathcal{A} = \{a\}$. The dynamics of the robot is defined as

$$x' = f_a(x) = \max(\min(1.2x + 0.5, 2.0), -1) \tag{5.10}$$

where x' denotes the succeeding state from the state x by the action a. This dynamics represents a part of a maze surrounded by walls. The experiment is performed as follows: (1) repeat N_{smpl} times: {generate x from uniform random distribution $[-2.5, 2.5]$, compute $x' = f_a(x)$, and train the models in an on-line manner}, (2) evaluate the RMS with $(x, \phi') \in \{(x_n, \phi'_n) | x_n = -2.5, -2.48, \ldots, 2.5; \phi'_n = \phi(f_a(x_n))\}$. We allocated 5 NGnet in $\mathcal{X}$ whose parameters are $\{(\mu_k, \Sigma_k) | \mu_k = -2, -1, 0, 1, 2; \Sigma_k = \frac{1}{9}\}$. We set $\alpha = 0.1$ as the step size parameter, and tested N_{smpl} in $\{10, 20, \ldots, 2000\}$.

Figure 5.1 shows the approximation accuracy of each dynamics models (the mean of the RMS over 10 runs is plotted per N_{smpl}). MixFS dynamics model has higher accuracy both in small N_{smpl} and large N_{smpl} than that of the Simple dynamics model. A possible reason is as follows. The dynamics of the robot is nearly linear in many regions of the state space, so the linear component of the state space dynamics of MixFS, $A_a x + d_a$, leads to good generalization from small samples. On the other hand, the composition of two dynamics models enhances the approximation capability, thus MixFS dynamics model also obtains a higher accuracy from a large number of samples than the Simple dynamics model.

5.5.2 Maze2D

Next, we evaluate the dynamics models with Dyna-MG algorithm in the navigation task of an omniwheel mobile robot on a 2-dimensional plane.

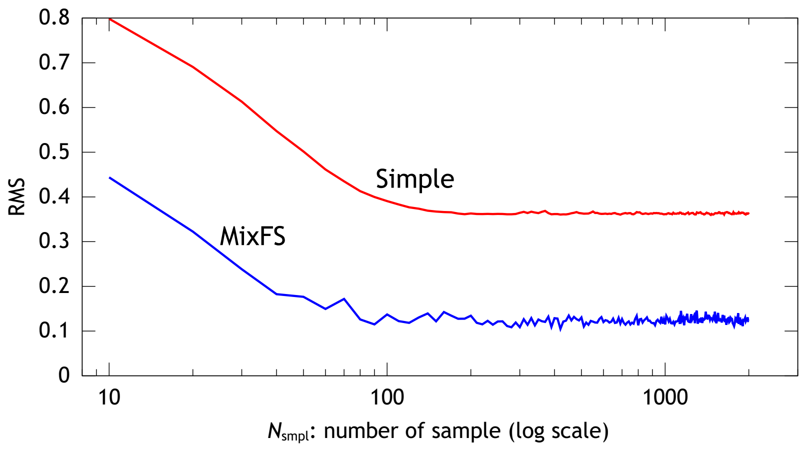


Figure 5.1 Estimation errors of the two dynamics models, MixFS and Simple, per number of samples N_{smpl} in 1-dimensional environment.

Experimental Setup

We use two types of mazes, (a): easy and (b): hard (see Figure 2.2). The state of the robot can be expressed as $x = (x_1, x_2)^\top$, and the possible actions are $\mathcal{A} = \{\mathsf{up}, \mathsf{left}, \mathsf{down}, \mathsf{right}\}$. This task is almost the same as Maze2D defined in Chapter 2; the only difference is the lack of *wind* in Figure 2.1.

We use an NGnet with 64 BFs to approximate the action value function. The same allocations are used in the two types of maze. These BFs are allocated on a 8×8 grid with added random noise to each center and covariance.

The configuration of Dyna-MG is $\gamma = 0.9$, $\alpha = 0.1$, $N_{\mathrm{pln}} = 5$, $Th_1 = 0.2$, and $Th_2 = 0.3$. For exploring actions, we use the Boltzmann policy selection with the temperature $\tau = 0.01$ for the easy maze and $\tau = 0.04$ for the hard maze. These parameters and coefficients are chosen through preliminary experiments.

Algorithm Setup

We compare Q(0)-learning (let $N_{\mathrm{pln}} = 0$ in the Dyna-MG algorithm), and the three Dyna-MG algorithms that have different dynamics model configurations:

Dyna-MG (Simple) : Dyna-MG with the Simple dynamics model (same as Algorithm 3).

Dyna-MG (MixFS) : Dyna-MG with MixFS dynamics model.

Dyna-MG (MixFS, reuse) : Dyna-MG with MixFS dynamics model whose parameters are reused (see Section 5.4.1) from learning within a maze that has no walls.

Furthermore, each of Dyna-MG algorithms is used with three reward sources (see Section 5.4.2) conditions:

NO_RWD_SRC : No reward source. The reward model parameters are initialized to zero.
GOAL_RWD_SRC : The goal reward is embedded into the reward model parameters. Specifically, we set $\{(x_i, R_i)\} = \{(x_\mathrm{g}, 1)\}$ where x_g denotes the goal state.
GOAL_OUT_RWD_SRC : The goal reward and the penalty for going out of the plane are embedded into the reward model parameters. Specifically, we set $\{(x_i, R_i)\} = \{(x_\mathrm{g}, 1)\} \cup \{(x_\ell, -0.5) | x_\ell \notin \mathcal{X}_\mathrm{pl}\}$ where the latter is the penalty sources. $\{x_\ell\}$ are sampled from the outer border of the plane $\mathcal{X}_\mathrm{pl}$, i.e., $\{x_\ell\} = \{(-1.25, -1.25), (-1, -1.25), (-0.75, -1.25), \dots\}$.

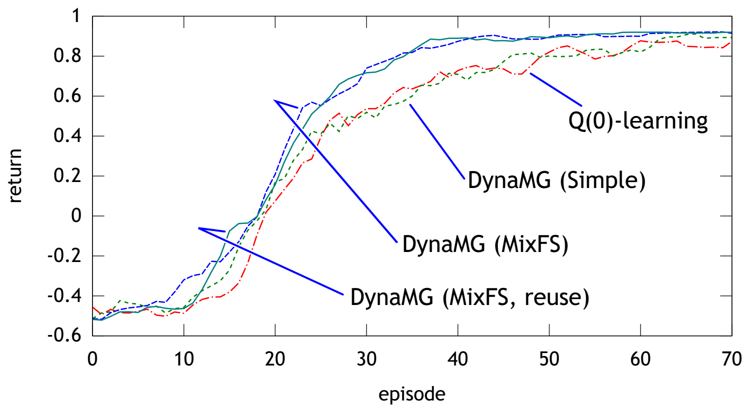
Thus, we compare the $1 + 3 \times 3 = 10$ learning conditions.

Results

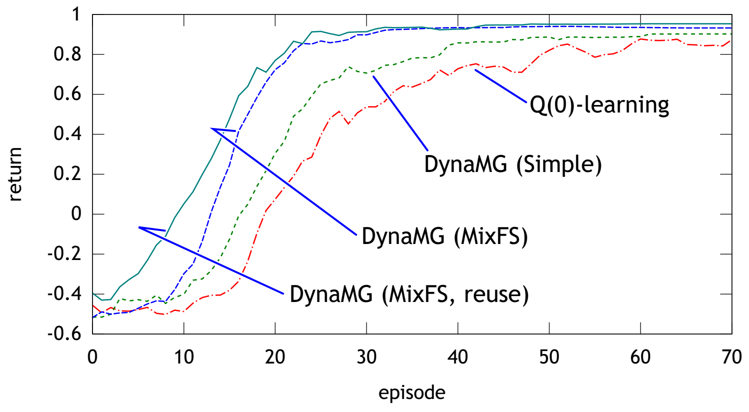
Figure 5.2 shows the learning curves of the navigation task in the easy maze (the mean of the return over 25 runs is plotted per episode). These curves result from the 10 learning conditions that are organized by the reward source type into Figure 5.2(a) NO_RWD_SRC, 5.2(b) GOAL_RWD_SRC, and 5.2(c) GOAL_OUT_RWD_SRC. The learning curve of Q(0)-learning is the same in these three graphs.

In every reward source type, the order of the learning speed are approximately the same: Dyna-MG (MixFS, reuse) (fastest), Dyna-MG (MixFS), Dyna-MG (Simple), Q(0)-learning (slowest). This result is considered to be an effect of the planning. Especially, Dyna-MG algorithms with MixFS dynamics model are faster than Dyna-MG with the Simple dynamics model. This result is possible because MixFS dynamics model can obtain a more accurate estimation than the Simple one even from a small number of samples. Thus, the planning in Dyna-MG becomes more precise which makes the learning faster.

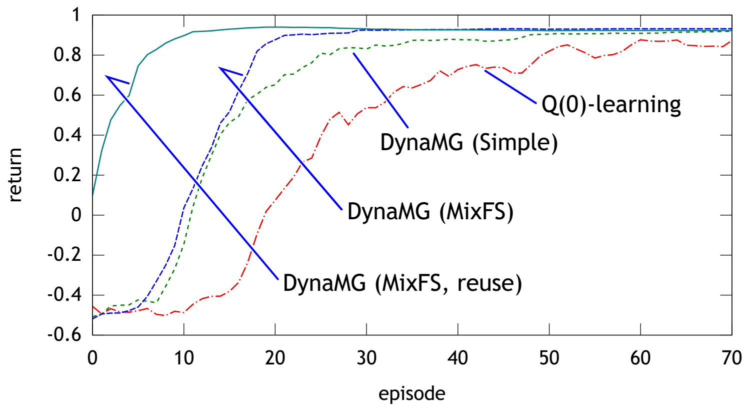
In each Dyna-MG configuration, the learning speed is positively correlated with the number of the reward sources, i.e., the amount of prior knowledge. This result shows that Dyna-MG can effectively utilize such information by planning. Especially, Dyna-MG (MixFS, reuse) of GOAL_OUT_RWD_SRC is significantly faster than the others (Figure 5.2(c)). A possible reason is that because of reusing the some dynamics parameters and embedding the reward sources, the dynamics and the reward models of the agent are fairly accurate from the beginning of the learning. Actually, if the agent only learns the dynamics related to the walls, it acquires nearly the complete dynamics model. Thus, the agent can obtain an optimal path in a small number of episodes.



(a) Learning curves of NO_RWD_SRC.



(b) Learning curves of GOAL_RWD_SRC.



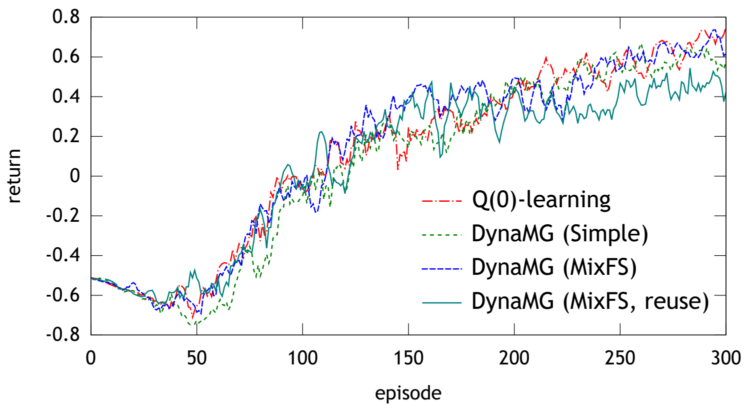
(c) Learning curves of GOAL_OUT_RWD_SRC.

Figure 5.2 Resulting learning curves of the robot navigation task in the easy maze. Each curve shows the mean of the return per episode over 25 runs. To see the tendency of each curve, a low-pass filter with a time constant of 5 episode is applied. The learning curves are organized into (a), (b), and (c) by the reward source type. The learning curve of Q(0)-learning is the same in these three graphs.

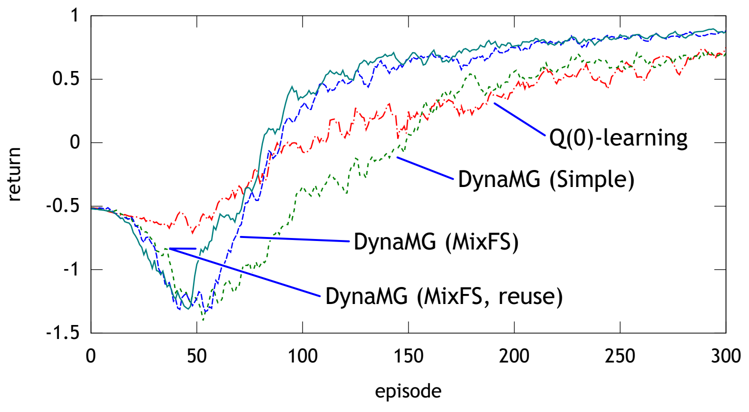
Figure 5.3 shows the learning curves of the navigation task in the hard maze (the mean of the return over 25 runs is plotted per episode). The learning conditions in Figure 5.3(a), 5.3(b), 5.3(c) are the same as the Figure 5.2.

In this result, overall, Dyna-MG (MixFS) and Dyna-MG (MixFS, reuse) also seem to be superior to Dyna-MG (Simple) and Q(0)-learning, but the differences are smaller than that of the easy maze. A possible reason is that since the hard maze has a complex dynamics, exploring the maze with learning the dynamics model and the policy are the dominant factor, while the effect of the planning using the models is relatively small.

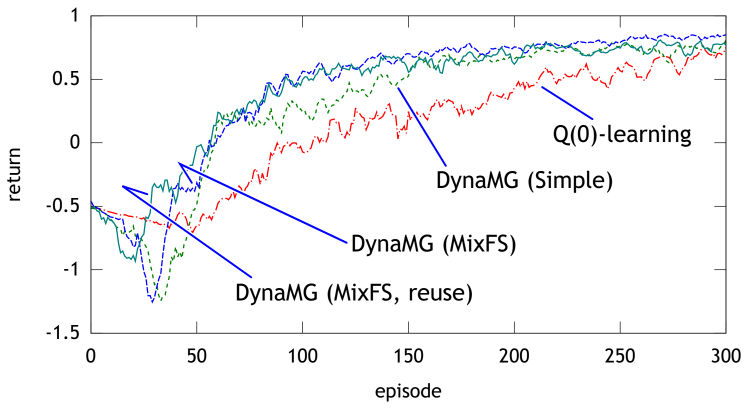
In Figure 5.3(b) and 5.3(c), the learning curves of the Dyna-MG algorithms decrease once. A possible reason is that planning with the reward model initialized by the goal source and the *incomplete* dynamics model computes the policy with which the robot attempts to go through a wall toward the goal. Actually, the robot cannot get to the goal with such a policy, so the cumulative step cost becomes dominant. Since the Dyna-MG algorithms with MixFS can estimate the accurate dynamics with fewer samples, these Dyna-MG algorithms can correct the error of the planning faster than Dyna-MG (Simple). Moreover, the exploration of Dyna-MG (MixFS, reuse) is less than that of Dyna-MG (MixFS) because of reusing the dynamics model parameters.



(a) Learning curves of NO_RWD_SRC.



(b) Learning curves of GOAL_RWD_SRC.



(c) Learning curves of GOAL_OUT_RWD_SRC.

Figure 5.3 Resulting learning curves of the robot navigation task in the hard maze. Each curve shows the mean of the return per episode over 25 runs. To see the tendency of each curve, a low-pass filter with a time constant of 5 episode is applied. The learning curves are organized into (a), (b), (c) by the reward source type. The learning curve of Q(0)-learning is the same in these three graphs.

5.5.3 Humanoid Navigation

We apply Dyna-MG with MixFS dynamics model to the navigation task of a humanoid robot, and compare it to the other methods. Though we use an action set coded by hand, the dynamics becomes more complex than that of the task in the previous section. Thus, we can investigate the applicability of MixFS dynamics model in a more practical situation.

Experimental Setup

We employ a 17-DoF humanoid robot (Figure 5.4) whose possible actions are $\mathcal{A} = \{\mathsf{walk}, \mathsf{turn\text{-}left}, \mathsf{turn\text{-}right}\}$. In this experiment, these actions are coded by hand for simplicity. The state of the robot is $x = (c_x, c_y, \phi, \cos\phi, \sin\phi)$ where c_x, c_y, ϕ denotes the x-position, the y-position, and the yaw angle of the body link respectively. The navigation task is almost the same as the previous section where the maze type is the easy (Figure 2.2). The dynamics is simulated on a dynamics simulator, ODE (Open Dynamics Engine (Smith 2006)).

The elements, $\cos\phi$ and $\sin\phi$, in the state definition are a trick to improve the generalization ability of the linear state space dynamics model. It is difficult for the linear state space dynamics model (eq. (5.8)) over the state definition $x = (c_x, c_y, \phi)$ to completely estimate the dynamics of the action walk of the humanoid robot, since the changes of the c_x and the c_y caused by the walk are proportional to $\cos\phi$ and $\sin\phi$. Extending the state definition to $x = (c_x, c_y, \phi, \cos\phi, \sin\phi)$ enables to estimate the dynamics of the walk with a linear model $A_a x$. The dynamics of the turn-left/right can also be estimated with a linear model $A_a x + d_a$.

We also use an NGnet with BFs allocated on a $8 \times 8 \times 9 \times 1 \times 1$ grid. Moreover, we let the diagonal elements of the covariance correspond to $\cos\phi$ and $\sin\phi$ large value (here, 10^6). Then the output of the BFs is the same as the output

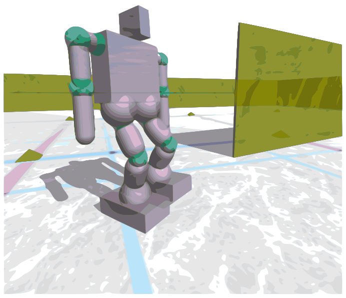


Figure 5.4 The humanoid robot employed in the navigation task. Its possible actions are walk, turn-left, and turn-right, which are coded by hand.

from the BFs where the state is defined as $x = (c_x, c_y, \phi)$ and the BFs are allocated on the $8 \times 8 \times 9$ grid. Thus, we can fairly use the same state definition and the same set of the BFs $\mathcal{K}$ for Q(0)-learning, Dyna-MG (Simple), and Dyna-MG (MixFS).

Results

We compared Q(0)-learning, Dyna-MG (Simple), Dyna-MG (MixFS), and Dyna-MG (MixFS, reuse) with a goal reward source condition (GOAL_RWD_SRC). Figure 5.5 shows the resulting learning curves (the mean of the return over 25 runs is plotted per episode). The Dyna-MG algorithms are faster than Q(0)-learning. Moreover the Dyna-MG algorithms with MixFS are faster than Dyna-MG (Simple). On the other hand, there is little difference between Dyna-MG (MixFS) and Dyna-MG (MixFS, reuse). A possible reason is that in the beginning of the learning, Dyna-MG (MixFS) quickly learns the dynamics model as accurate as the one reused in Dyna-MG (MixFS, reuse).

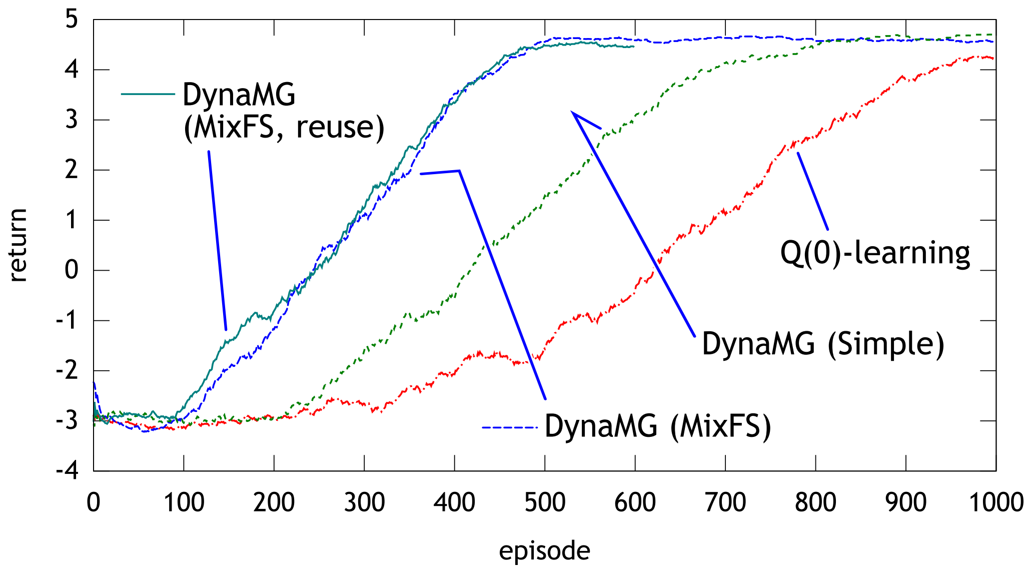


Figure 5.5 Resulting learning curves of the navigation task by the humanoid robot in the easy maze. Each curve shows the mean of the return per episode over 10 runs. To see the tendency of each curve, a low-pass filter with a time constant of 50 episode is applied.

5.6 Conclusion

This chapter proposed a composite dynamics model of the feature space dynamics and the state space dynamics, called MixFS dynamics model. MixFS can exploit the advantages of both dynamics models, that is, the generalization ability of the state space dynamics model and the accuracy of the feature space dynamics model. Moreover, MixFS also has reusability, since in some cases, including the navigation task, reusability can be inherited from the generalization ability of the state space dynamics model. We contribute to combine the two dynamics models and make it work with the Dyna algorithm, and to derive an on-line learning method of the developed model.

The simulation result demonstrates that MixFS can exploit the advantages of both dynamics models as we expected. Concretely, MixFS can estimate the dynamics more precisely from either a small number or a large number of samples than the conventional linear dynamics model of a feature space. Reusing the dynamics model parameters can utilize prior knowledge encoded into the reward model. As a result, Dyna-MG with MixFS is faster than that with the conventional dynamics model.

Chapter 6

Learning Strategy Fusion

This chapter proposes a method to fuse learning strategies (LSs) in reinforcement learning (RL) framework. In conventional RL methods, we need to choose a suitable LS for each task respectively. In contrast, the proposed method automates this selection by fusing LSs. Actually, LS fusion applies multiple LSs for a single task multiple times; a proper ordering of the LSs is automatically decided. The LSs defined in this chapter include a transfer learning, a hierarchical RL, and a model-based RL.

6.1 Introduction

This chapter attempts to integrate some learning strategies (LSs) to suit for learning of high DoF robots. Imagine learning a tennis swing. We first swing a racket slowly. After a good swing form is learned, we speed up the swing. Other example is found in an infant's learning walking model (Taga, Takaya, and Konishi 1999) where the infant starts from a lower DoF (freezing), then learns in a higher DoF (freeing).

This chapter proposes a method to fuse LSs named *LS fusion*, in which multiple LSs are applied to a single task multiple times. The most distinct feature is that when one of policies almost converges, an LS, such as freeing the DoF (Degree of Freedom), generates a new policy from the converged one. The new policy is additionally learned, which may improve the performance. Meanwhile, the old policy is still stored; if the new one has failed to improve the policy, the old one is used again. Actually, the system has many policies (*behavior modules*) for a single task, and they are increased by the LSs. Behavior modules that have potentially high performance are trained preferentially; namely, a behavior module of better performance is trained more. For this prioritization, Boltzmann selection method with an upper confidence bound (UCB) is employed. We refer to this selection method as UCB-Boltzmann selection.

This chapter also defines the following LSs:

LS-scratch generates a behavior module that learns a task from scratch using WF-DCOB (Chapter 4).

LS-accelerating generates a behavior module by accelerating the motion of a source behavior module.

LS-freeing generates a behavior module by increasing the DoF of a source behavior module.

LS-planning generates a behavior module that uses a dynamics and a reward model module to execute planning.

LS-model generates a dynamics and a reward model module (Chapter 5).

LS-hierarchy generates a hierarchical action space module.

Here, the LS-accelerating and the LS-freeing are *transfer learning* methods (Torrey and Shavlik 2009). A distinct feature of these transfer LSs is that the LSs transfer not only policy parameters, but also physical limitation of the policy. Specifically, the LS-freeing changes the DoF. The LS-accelerating changes not only the speed parameters, but the constraints of the speed parameters. Thus, these LSs modifies the task domain, which has a probability to increase performance.

The rest of this chapter is organized as follows. Section 6.2 discusses the related works. Section 6.3 describes an overview of LS fusion, Section 6.4 gives the algorithm of LS fusion, and Section 6.5 defines the LSs. Section 6.6 demonstrates some experimental results. Finally, Section 6.7 concludes this chapter.

6.2 Related Works

LS fusion is considered to be a learning architecture that consists of *multiple RL modules* for a single robot and a single task, and allows *behavior transfer*. The most distinct feature compared to the other works is that the LSs transfer not only policy parameters, but also physical limitation of the policy.

Uchibe *et al.* (Uchibe and Doya 2004) proposed the Cooperative-Competitive-Concurrent Learning with Importance Sampling (CLIS) architecture, where multiple RL modules sharing the same sensory-motor system learn for the same task simultaneously by using importance sampling. The similarity to LS fusion is employing multiple modules for a single task. CLIS architecture is considered to be transferring *samples* obtained from a module to the other modules. Thus, the architecture allows concurrent learning. On the other hand, LS fusion transfers a *policy* of a behavior module to a new module where some motion parameters, such as DoF and speed parameters, are changed. Thus, the new behavior module has a probability to increase the performance of the old module.

Fernández *et al.* (Fernández and Veloso 2006) proposed a method to probabilistically reuse the policies learned in the other tasks. LS fusion is considered to be reusing policies, so these two methods are similar. The difference is that LS fusion modifies the motion parameters including the physical limitation, while the method in (Fernández and Veloso 2006) does not change the state-action space and the dynamics. This difference also appears in the other transfer learning methods (e.g. (Zhang and Rössler 2004)).

6.3 LS Fusion Overview

LS fusion has two key elements: (1) the LSs, and (2) UCB-Boltzmann selection. This thesis assumes that there are two types in LSs; *behavior* LSs that generate new behavior modules, and *supplementary* LSs that generate supplementary modules other than behaviors, such as model modules and hierarchical-action-space modules. Let $\mathcal{LS}_{\mathrm{bhv}}$ denote the set of the behavior LSs, $\mathcal{LS}_{\mathrm{spl}}$ denote the set of the supplementary LSs. UCB-Boltzmann selection method chooses a behavior from both the existing behavior modules and the new ones generated by the behavior LSs. The selected behavior module is used to actually control the robot, and the module is updated its policy from samples.

We design LS fusion so that the behavior and the supplementary modules are generated and learned through the following flow:

(1) The supplementary LSs (the LS-hierarchy and the LS-model) generate modules if applicable.
(2) The behavior LSs generate new behavior modules. Specifically, the LS-scratch generates new behaviors which may have different DoF configurations. If there are behaviors trained enough, the LS-freeing and the LS-accelerating generate new behavior modules by transferring the trained behaviors. If there are a dynamics and a reward model module, the LS-planning generates a new behavior module using a model-based RL method.
(3) UCB-Boltzmann selection method chooses a behavior module from both the existing behavior modules and the new ones generated in (2).
(4) Several episodes are performed using the selected behavior module, and the policy of the behavior module is updated from samples. The supplementary modules are also updated if possible.
(5) (1)...(4) are repeated.

Here, the UCB (upper confidence bound) uses both the mean of a reward summation $\overline{R}_B$ and its deviation $\overline{\sigma}_B$. The deviation $\overline{\sigma}_B$ can estimate the potential improvement of the performance. The deviation $\overline{\sigma}_B$ is also used to judge if a behavior module is trained enough.

6.4 LS Fusion Algorithm

We assume that several pairs of a control command space and a state space $\{(\tilde{\mathcal{U}}, \mathcal{X})\}$ are predefined; they have different DoF configurations. Here, defining $\tilde{\mathcal{U}}$ and $\mathcal{X}$ means giving conversions between $(\tilde{\mathcal{U}}, \mathcal{X})$ and the overall (full DoF) command and state spaces $(\tilde{\mathcal{U}}_{\mathrm{w}}, \mathcal{X}_{\mathrm{w}})$. Specifically, we assume linear conversions with constant matrices $\mathrm{C}_{\tilde{\mathcal{U}}}$ and $\mathrm{C}_{\mathcal{X}}$ such that $\tilde{u}_{\mathrm{w}} = \mathrm{C}_{\tilde{\mathcal{U}}}\tilde{u}$, $x = \mathrm{C}_{\mathcal{X}} x_{\mathrm{w}}$ where $\tilde{u} \in \tilde{\mathcal{U}}$, $\tilde{u}_{\mathrm{w}} \in \tilde{\mathcal{U}}_{\mathrm{w}}$, $x \in \mathcal{X}$, and $x_{\mathrm{w}} \in \mathcal{X}_{\mathrm{w}}$.

Each behavior learning strategy *LS* is defined as a function $\mathsf{GEN}_{\mathrm{bhv}}(LS, \mathcal{U}, \mathcal{X}, \mathit{Task})$ that generates behavior modules, and each supplementary learning strategy is defined as a function $\mathsf{GEN}_{\mathrm{spl}}(LS, \mathit{Task})$ that generates supplementary modules.

The LS fusion algorithm is defined for an episodic task. Algorithm 5 shows the overall algorithm[1]. Here, $\mathrm{N}_{\mathrm{LSSp}}$ is an interval of executing the supplementary LSs (LSSp means LS Supplementary), $\mathrm{N}_{\mathrm{LSBh}}$ is an interval of executing the behavior LSs (LSBh means LS Behavior). $\mathrm{N}_{\mathrm{LSBh}} > 1$ is needed to compute the valid reward statistics (we choose $\mathrm{N}_{\mathrm{LSSp}} = 20$ and $\mathrm{N}_{\mathrm{LSBh}} = 10$ in the experiments of this chapter). UCB-Boltzmann selection method chooses a behavior module from both the existing $\mathcal{B}$ and new behavior modules generated by the behavior LSs. Note that only the selected new behavior module is added into $\mathcal{B}$.

Thus, the key elements of LS fusion are each LS ($\mathsf{GEN}_{\mathrm{bhv}}$, $\mathsf{GEN}_{\mathrm{spl}}$) and UCB-Boltzmann selection method. Note that LS fusion works with any LS for that $\mathsf{GEN}_{\mathrm{bhv}}$ or $\mathsf{GEN}_{\mathrm{spl}}$ is defined. The rest of this section describes the reward statistics and UCB-Boltzmann selection method. In the next section, we specify the LSs used in this thesis.

6.4.1 Reward Statistics

We evaluate the performance of a behavior module by $R \triangleq \frac{\sum_t r_t}{T}$, where $\{r_t | t = 1, 2, \ldots\}$ denotes the observed reward sequence in an episode, and T denotes to-

[1]In implementing this algorithm, the size of $\mathcal{B}$ is limited to 20 per a task to prevent the large memory usage. If the size exceeds the limit, a behavior module that has the minimum UCB of R_{UCB} except for B_{next} is removed from $\mathcal{B}$.

Algorithm 5: Learning strategy fusion

```
Input : Task $Task$, behavior modules $\mathcal{B}$,
        state-space modules $\{\mathcal{X}\}$, action-space modules: $\{\mathcal{U}\}$,
        dynamics-model modules $\{M_{\text{dyn}}\}$, reward-model modules $\{M_{\text{rwd}}\}$
        /* $\{M_{\text{dyn}}\}$ and $\{M_{\text{rwd}}\}$ may be empty */
 1: for $N_{\text{eps}} = 1, 2, \ldots$ do /* $N_{\text{eps}}$: episode number */
 2:   if $N_{\text{eps}} \bmod \mathrm{N}_{\text{LSSp}} = 0$ then
 3:     for each $LS \in \mathcal{LS}_{\text{spl}}$ do
 4:       $\{\mathcal{X}\}, \{\mathcal{U}\}, \{M_{\text{dyn}}\}, \{M_{\text{rwd}}\} \leftarrow \mathsf{GEN}_{\text{spl}}(LS, Task)$
 5:   if $N_{\text{eps}} \bmod \mathrm{N}_{\text{LSBh}} = 0$ then
 6:     /* select a behavior module: */
 7:     $\mathcal{B}_{\text{new}} \leftarrow \{\}$
 8:     for each $(\mathcal{U}, \mathcal{X})$ do
 9:       for each $LS \in \mathcal{LS}_{\text{bhv}}$ do
10:         $\mathcal{B}_{\text{new}} \leftarrow \mathcal{B}_{\text{new}} \cup \mathsf{GEN}_{\text{bhv}}(LS, \mathcal{U}, \mathcal{X}, Task)$
11:     Select $B_{\text{next}}$ from $\mathcal{B} \cup \mathcal{B}_{\text{new}}$ by UCB-Boltzmann selection
12:     if $B_{\text{next}} \in \mathcal{B}_{\text{new}}$ then $\mathcal{B}' \leftarrow \mathcal{B} \cup \{B_{\text{next}}\}$ else $\mathcal{B}' \leftarrow \mathcal{B}$
13:     return $B_{\text{next}}, \mathcal{B}'$
14:   Perform the episode with $B_{\text{next}}$:
       $B_{\text{next}}$ is updated by its own learning algorithm
       $\{M_{\text{dyn}}\}, \{M_{\text{rwd}}\}$ are updated if possible
15:   Update the reward statistics $\overline{R}_{B_{\text{next}}}, \overline{R^2}_{B_{\text{next}}}, \overline{\sigma}_{\max B_{\text{next}}}$
```

tal time in the episode. The definition of R depends on a task. In general, a sum of reward (return) may be used, but in our crawling task, this definition is suitable to select a better behavior module, especially in the early stage of learning.

Since each behavior module is updated to obtain better policy, its performance changes with each episode. Thus, we compute the mean and the standard deviation of R while forgetting the old data, and use them to select a behavior module. Let $R_{N_{\text{eps}}}$ the observation at an N_{eps}-th episode. The reward statistics $\overline{R}_B$, $\overline{R^2}_B$ are updated by

$$\overline{R}_B \leftarrow \alpha_{\text{R}} R_{N_{\text{eps}}} + (1 - \alpha_{\text{R}})\overline{R}_B, \tag{6.1}$$

$$\overline{R^2}_B \leftarrow \alpha_{\text{R}} R^2_{N_{\text{eps}}} + (1 - \alpha_{\text{R}})\overline{R^2}_B, \tag{6.2}$$

where α_{R} is a learning rate. The standard deviation of R can be obtained by $\overline{\sigma}_B = (\overline{R^2}_B - \overline{R}^2_B)^{1/2}$. In addition, at the end of each episode, $\overline{\sigma}_{\max B}$ is updated which is used in some LSs.

The reward statistics $\overline{R}_B$, $\overline{R^2}_B$ are initialized by zero if B is generated by the LS-scratch or the LS-planning. If B is generated by the LS-accelerating or the LS-freeing, the statistics are initialized by the source behavior module's values.

6.4.2 UCB-Boltzmann Selection

We employ an UCB of R to evaluate the priority of search. In addition, we use Boltzmann selection to probabilistically select a behavior module.

The UCB of R is defined by

$$R_{\mathrm{UCB}B} \triangleq \overline{R}_B + \mathrm{f}_{\mathrm{UCB}}\overline{\sigma}_B \tag{6.3}$$

where $\mathrm{f}_{\mathrm{UCB}}$ is a real constant value that decides the weight of expected improvement (typically 1 or 2).

According to Boltzmann selection, the probability to select B is defined as

$$\pi(B) \propto \exp(\frac{1}{\tau_{\mathrm{lsd}}} R_{\mathrm{UCB}B}) \tag{6.4}$$

where τ_{lsd} is a temperature parameter to adjust randomness. We decrease τ_{lsd} with $\tau_{\mathrm{lsd}} = \tau_{\mathrm{lsd0}} \exp(-\delta_{\tau_{\mathrm{lsd}}} N_{\mathrm{eps}})$.

6.5 Learning Strategies

This section defines the LSs, namely, defines a function $\mathsf{GEN}_{\mathrm{bhv}}(LS, \mathcal{U}, \mathcal{X}, Task)$ or $\mathsf{GEN}_{\mathrm{spl}}(LS, Task)$ for each learning strategy LS. Every behavior module B has information, $Task^{(B)}$: a task that B is learning, $LS^{(B)}$: a learning strategy with which the behavior module is generated, $\mathcal{U}^{(B)}$: an action space and $\mathcal{X}^{(B)}$: a state space where the behavior module learns, and $\mathcal{K}^{(B)}$: a set of BFs.

For the LS-scratch, we consider that using an RL method with a discrete action set is suitable, since a policy is learned from scratch. In contrast, for LS-accelerating and the LS-freeing, it is better to use an RL method with a continuous action space. This is because a new behavior module generated by these LSs searches a policy around the source behavior's policy, which is exploration in a narrow area.

For these reasons, we utilize WF-DCOB (Chapter 4) for the LSs, since WF-DCOB has features like the discrete action set DCOB but explores continuous actions around each discrete action of DCOB. In addition, WF-DCOB explicitly has parameters related to a speed of motion and target joint angles. Thus, WF-DCOB is suitable not only for the LS-scratch, but for the LS-accelerating and the LS-freeing.

6.5.1 LS-Scratch

The function $\mathsf{GEN}_{\mathrm{bhv}}(\mathsf{LS\text{-}scr}, \mathcal{U}, \mathcal{X}, \mathit{Task})$ generates a behavior module if $\mathcal{U}$ is a command space $\tilde{\mathcal{U}}$ and there is no behavior module of the same setup. Namely, a module is generated if $\mathcal{B}$ does not include a behavior module B such that $LS^{(B)} = \mathsf{LS\text{-}scr}$, $\mathcal{U}^{(B)} = \tilde{\mathcal{U}}$, and $\mathcal{X}^{(B)} = \mathcal{X}$. The reason of preventing a generation of the same setup is that the probability that the new behavior module obtains better performance than the existing one is not high. A new behavior module B_{new} uses WF-DCOB with Q(λ)-learning, where a default (predefined) set of BFs is employed. If a default set of BFs is not predefined for $\mathcal{X}$, a new behavior module is not generated[2]. WF-DCOB's configurations C_{p}, C_{d}, Ctrl, $\mathcal{I}_{\mathcal{R}}$, and $\mathrm{F}_{\mathrm{abbrv}}$ are assumed to be predefined for each $(\tilde{\mathcal{U}}, \mathcal{X})$. Since this behavior module learns from scratch, the parameters of wire-fitting are initialized by the WF-DCOB's manner as follows:

$$\theta_i^{(B_{\mathrm{new}})} = 0, \tag{6.5}$$

$$U_i^{(B_{\mathrm{new}})} = (g_i^{(B_{\mathrm{new}})}, q_i^{\mathrm{trg}(B_{\mathrm{new}})}) \tag{6.6}$$

$$= (\frac{\mathrm{g}_i^{\mathrm{s}} + \mathrm{g}_i^{\mathrm{e}}}{2}, C_{\mathrm{p}}(\mu_{k_i})), \tag{6.7}$$

for each $i \in \mathcal{W}$.

6.5.2 LS-Accelerating

The LS-accelerating generates behavior modules from source behavior modules that have the same command and state spaces. Also, the generation with the same setup is prevented, and this LS works with only a command space. The acceleration is performed by multiplying the interval factor of the source module's WF-DCOB by a real constant value $\mathrm{f}_{\mathrm{accel}} < 1$.

Specifically, in the function $\mathsf{GEN}_{\mathrm{bhv}}(\mathsf{LS\text{-}accel}, \tilde{\mathcal{U}}, \mathcal{X}, \mathit{Task})$, for each B_{src} such that $\mathcal{U}^{(B_{\mathrm{src}})} = \tilde{\mathcal{U}}$ and $\mathcal{X}^{(B_{\mathrm{src}})} = \mathcal{X}$, a new behavior module B_{new} is generated if the conditions are satisfied:

(1) $\mathcal{B}$ does not include a behavior module B such that $LS^{(B)} = \mathsf{LS\text{-}accel}$ and $Src^{(B)} = B_{\mathrm{src}}$,

(2) $\overline{\sigma}_{B_{\mathrm{src}}} / \overline{\sigma}_{\max B_{\mathrm{src}}} < \sigma_{\mathrm{th}}$,

where $Src^{(B)}$ denotes the source behavior module of B, $\overline{\sigma}_B$ denotes a deviation of B's return, and $\overline{\sigma}_{\max B}$ denotes its maximum. Thus, condition (2) checks if B_{src} almost converged, namely, is trained enough. σ_{th} is a threshold.

[2]In the following experiments, $\mathcal{X}_{16}$ is the case.

B_{new} uses the same BFs with B_{src}, namely $\mathcal{K}^{(B_{\text{new}})} = \mathcal{K}^{(B_{\text{src}})}$. The parameters of wire-fitting of B_{new} are copied from B_{src} except for $\{U_i^{(B_{\text{new}})}\}$ that is multiplied by $\mathrm{f}_{\text{accel}}$. Specifically, for each $i \in \mathcal{W}^{(B_{\text{src}})}$,

$$\theta_i^{(B_{\text{new}})} = \theta_i^{(B_{\text{src}})}, \tag{6.8}$$

$$U_i^{(B_{\text{new}})} = (g_i^{(B_{\text{new}})}, q_i^{\text{trg}(B_{\text{new}})}) \tag{6.9}$$

$$= (\mathrm{f}_{\text{accel}} g_i^{(B_{\text{src}})}, q_i^{\text{trg}(B_{\text{src}})}). \tag{6.10}$$

In addition, the set of constraint range $\mathcal{I}_{\mathcal{R}} = \{(\mathrm{g}_i^{\text{s}}, \mathrm{g}_i^{\text{e}}) | i = 1, 2, \dots\}$ is also modified:

$$\mathrm{g}_i^{\text{s}(B_{\text{new}})} = \mathrm{f}_{\text{accel}} \mathrm{g}_i^{\text{s}(B_{\text{src}})}, \tag{6.11}$$

$$\mathrm{g}_i^{\text{e}(B_{\text{new}})} = \mathrm{f}_{\text{accel}} \mathrm{g}_i^{\text{e}(B_{\text{src}})}. \tag{6.12}$$

Thus, the LS-accelerating transfers not only the policy parameters, but also the limitation of the policy.

6.5.3 LS-Freeing

The LS-freeing frees the DoF of a source behavior module to larger DoF based on a predefined freeing direction F. Each freeing direction F includes information, $\tilde{\mathcal{U}}_{\text{src}}^{(F)}, \mathcal{X}_{\text{src}}^{(F)}, \tilde{\mathcal{U}}_{\text{dest}}^{(F)}$, and $\mathcal{X}_{\text{dest}}^{(F)}$ which denote the spaces of a source behavior module and the spaces of a destination behavior module. This LS works with only a command space.

In the function $\mathsf{GEN}_{\text{bhv}}(\mathsf{LS\text{-}free}, \tilde{\mathcal{U}}, \mathcal{X}, \mathit{Task})$, for each pair of (F, B_{src}) such that $\tilde{\mathcal{U}}_{\text{dest}}^{(F)} = \tilde{\mathcal{U}}$, $\mathcal{X}_{\text{dest}}^{(F)} = \mathcal{X}$, $\mathcal{U}^{(B_{\text{src}})} = \tilde{\mathcal{U}}_{\text{src}}^{(F)}$, and $\mathcal{X}^{(B_{\text{src}})} = \mathcal{X}_{\text{src}}^{(F)}$, a new behavior module B_{new} is generated if the conditions are satisfied:

(1) $\mathcal{B}$ does not include a behavior module B such that $LS^{(B)} = \mathsf{LS\text{-}free}$, $\tilde{\mathcal{U}}^{(B)} = \tilde{\mathcal{U}}$, $\mathcal{X}^{(B)} = \mathcal{X}$, and $Src^{(B)} = B_{\text{src}}$,

(2) The same as condition (2) of the LS-accelerating.

Condition (1) is to prevent the generation with the same setup.

B_{new} is initialized so that its action value function is almost the same as that of B_{src}. To do this, first, the freeing matrices $D_{\tilde{\mathcal{U}}}$ and $D_{\mathcal{X}}$ are calculated so that the conversions $\tilde{u}_{\text{dest}} = D_{\tilde{\mathcal{U}}} \tilde{u}_{\text{src}}$, $x_{\text{dest}} = D_{\mathcal{X}} x_{\text{src}}$ are performed where $\tilde{u}_{\text{dest}} \in \tilde{\mathcal{U}}_{\text{dest}}^{(F)}$, $\tilde{u}_{\text{src}} \in \tilde{\mathcal{U}}_{\text{src}}^{(F)}$, $x_{\text{dest}} \in \mathcal{X}_{\text{dest}}^{(F)}$, and $x_{\text{src}} \in \mathcal{X}_{\text{src}}^{(F)}$. We define these matrices as

$$D_{\tilde{\mathcal{U}}} = \mathrm{C}^{\sharp}_{\tilde{\mathcal{U}}_{\text{dest}}^{(F)}} \mathrm{C}_{\tilde{\mathcal{U}}_{\text{src}}^{(F)}}, \tag{6.13}$$

$$D_{\mathcal{X}} = \mathrm{C}_{\mathcal{X}_{\text{dest}}^{(F)}} \mathrm{C}^{\sharp}_{\mathcal{X}_{\text{src}}^{(F)}}, \tag{6.14}$$

where $\sharp$ denotes a pseudo-inverse. The parameters of the B_{new}'s BFs are calculated as

$$\mu_k^{(B_{\mathrm{new}})} = D_{\mathcal{X}} \mu_k^{(B_{\mathrm{src}})}, \tag{6.15}$$

$$\Sigma_k^{(B_{\mathrm{new}})} = D_{\mathcal{X}} \Sigma_k^{(B_{\mathrm{src}})} D_{\mathcal{X}}^{\top}, \tag{6.16}$$

for each $k \in \mathcal{K}^{(B_{\mathrm{src}})}$. The parameters of wire-fitting are initialized as

$$\theta_i^{(B_{\mathrm{new}})} = \theta_i^{(B_{\mathrm{src}})}, \tag{6.17}$$

$$U_i^{(B_{\mathrm{new}})} = (g_i^{(B_{\mathrm{new}})}, q_i^{\mathrm{trg}(B_{\mathrm{new}})}) \tag{6.18}$$

$$= (g_i^{(B_{\mathrm{src}})}, D_{\tilde{\mathcal{U}}} q_i^{\mathrm{trg}(B_{\mathrm{src}})}), \tag{6.19}$$

for each $i \in \mathcal{W}^{(B_{\mathrm{src}})}$. In this case, $\mathcal{I}_{\mathcal{R}}$ is not changed.

6.5.4 LS-Planning

The function $\mathsf{GEN}_{\mathrm{bhv}}(\mathsf{LS\text{-}pln}, \mathcal{U}, \mathcal{X}, \mathit{Task})$ generates a behavior module if $\mathcal{U}$ is a discrete action space, and a dynamics and a reward model module for $\mathcal{U}$, $\mathcal{X}$, and *Task* are available. Also, the generation with the same setup is prevented.

A new behavior module uses Dyna-MG where we use Q(λ)-learning instead of Q(0)-learning (see Chapter 5 for the detail). A default set of BFs is employed.

6.5.5 LS-Model

The function $\mathsf{GEN}_{\mathrm{spl}}(\mathsf{LS\text{-}model}, \mathit{Task})$ generates a dynamics and a reward model module for each combination of $(\mathcal{U}, \mathcal{X})$ and *Task* if $\mathcal{U}$ is a discrete action space. MixFS dynamics model is used as the dynamics model, and a simple reward model is employed (see Chapter 5 for the models). The parameters of these models are $\{\delta F_a, A_a, B_a, d_a | a \in \mathcal{U}\}$ and $\{b_a | a \in \mathcal{U}\}$ respectively. After the model module is generated, its parameters are updated when an action in $\mathcal{U}$ is executed even if any behavior module does not use the model module.

If there are some dynamics modules learned in the other tasks, the new dynamics module is initialized using the parameters of the existing dynamics modules. Specifically, A_a and d_a are copied from the existing modules. This transfer is performed in the manner mentioned in Chapter 5. If the state and the reward of a goal (or a forbidden region) are known, they are embedded into the reward model as prior knowledge. The method to embed reward sources is also described in Chapter 5.

6.5.6 LS-Hierarchy

The function $\mathsf{GEN}_{\mathrm{spl}}(\mathsf{LS\text{-}hier}, \mathit{Task})$ generates a hierarchical action space $\mathcal{H}$. The LS-hierarchy assumes that each task has a category label. A hierarchical action space is generated as a set of subtasks that have a same category label. Thus, the LS-hierarchy does not construct a hierarchical action space in a fully automatic manner. When an action in $\mathcal{H}$ is selected, the LS fusion algorithm is also used to execute the subtask. If a subtask is a continuing task (the HumanoidML-crawling and the HumanoidML-turning tasks are the case), we need to configure the duration of the subtask.

6.6 Experiments

6.6.1 Maze2D

To demonstrate the LS-model and the LS-planning work as expected, LS fusion is applied to the Maze2D task. Though the task is as mentioned in Chapter 2, the type of the maze changes at 400th episode which means that two tasks are learned in sequence. Concretely, in 0th to 399th episode, Maze2D-middle is learned, then Maze2D-hard is learned.

In this learning, each LS is expected to be used in the following scenario:

(1) The LS-model generates a dynamics and a reward model module for Maze2D-middle.
(2) The LS-planning generates a behavior module for Maze2D-middle, where the model modules are used.
(3) The modules above are learned during 0th to 399th episode.
(4) At the beginning of the 400th episode, the LS-model generates a dynamics and a reward model module for Maze2D-hard. In this case, the dynamics model for Maze2D-middle is used to initialize the parameters of the new model.
(5) The LS-planning generates a behavior module for Maze2D-hard, where the model modules are used.
(6) The modules are learned.

Figure 6.1 shows the resulting learning curves of this task (the mean of the return over 25 runs is plotted per episode). Here, the following methods are compared; LSF (MixFS): LS fusion where the LS-model uses MixFS dynamics model, LSF (Simple): LS fusion where the LS-model uses the Simple dynamics

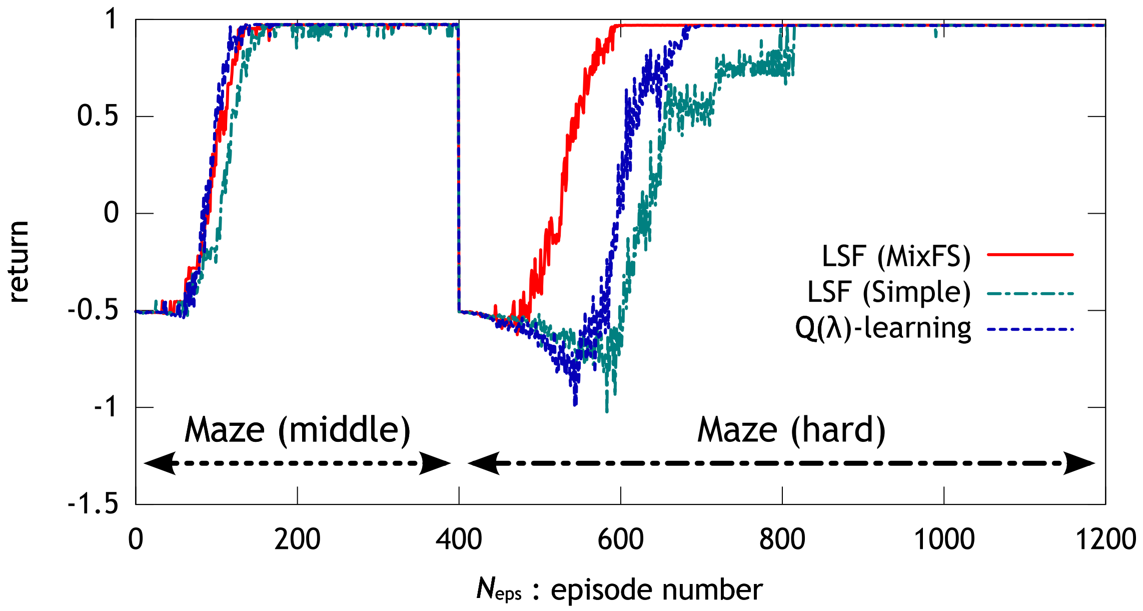


Figure 6.1 Resulting learning curves of the Maze2D task where the maze changes at 400th episode. Each curve shows the mean of the return over 15 runs per episode.

model, and Q(λ)-learning: normal RL methods for the two tasks. In learning Maze2D-middle, the learning speed of these three are almost the same. In learning Maze2D-hard, LSF (MixFS) is faster than the others. A possible reason is that LSF (MixFS) uses the prior knowledge obtained in Maze2D-middle by transferring the dynamics model. These results indicate that LS fusion works as expected.

6.6.2 HumanoidML-crawling

Next, we apply LS fusion to learn HumanoidML-crawling task. In this experiment, we investigate the effect of the transfer LSs, the LS-accelerating and the LS-freeing. The task HumanoidML-crawling is almost the same as defined in Chapter 2 except for the reward function. We use eq. (4.34) as the reward function since the DoF configurations include 4-DoF which is not symmetric (see also Section 4.5.2).

Space Configurations

We use six sets of DoF configurations: 3-DoF $(\tilde{\mathcal{U}}_3, \mathcal{X}_3)$, 4-DoF $(\tilde{\mathcal{U}}_4, \mathcal{X}_4)$, 5-DoF $(\tilde{\mathcal{U}}_5, \mathcal{X}_5)$, 6-DoF $(\tilde{\mathcal{U}}_6, \mathcal{X}_6)$, 7-DoF $(\tilde{\mathcal{U}}_7, \mathcal{X}_7)$, and 16-DoF $(\tilde{\mathcal{U}}_{16}, \mathcal{X}_{16})$. These configurations are defined in Chapter 2. In N_{D}-DoF configuration, its command input space is a N_{D}-dimensional vector space that represents target joint angles. Its state space is

$$\begin{aligned} x = (&c_{0z}, q_w, q_x, q_y, q_z, \mathbf{q}_{N_{\mathrm{D}}}^{\top}, \\ &\dot{c}_{0x}, \dot{c}_{0y}, \dot{c}_{0z}, \omega_x, \omega_y, \omega_z, \dot{\mathbf{q}}_{N_{\mathrm{D}}}^{\top})^{\top} \end{aligned} \tag{6.20}$$

where (c_{0x}, c_{0y}, c_{0z}) denotes the position of the center-of-mass of the body link, (q_w, q_x, q_y, q_z) denotes the rotation of the body link in quaternion, $(\omega_x, \omega_y, \omega_z)$ denotes the rotational velocity of the body link, and $\mathbf{q}_{N_{\mathrm{D}}}$ denotes the joint angle vector of the N_{D}-DoF. The reason for the absence of c_{0x} and c_{0y} from state x is that a policy for the crawling task does not have to depend on the global location of the robot. The default BFs are allocated as mentioned in Chapter 2.

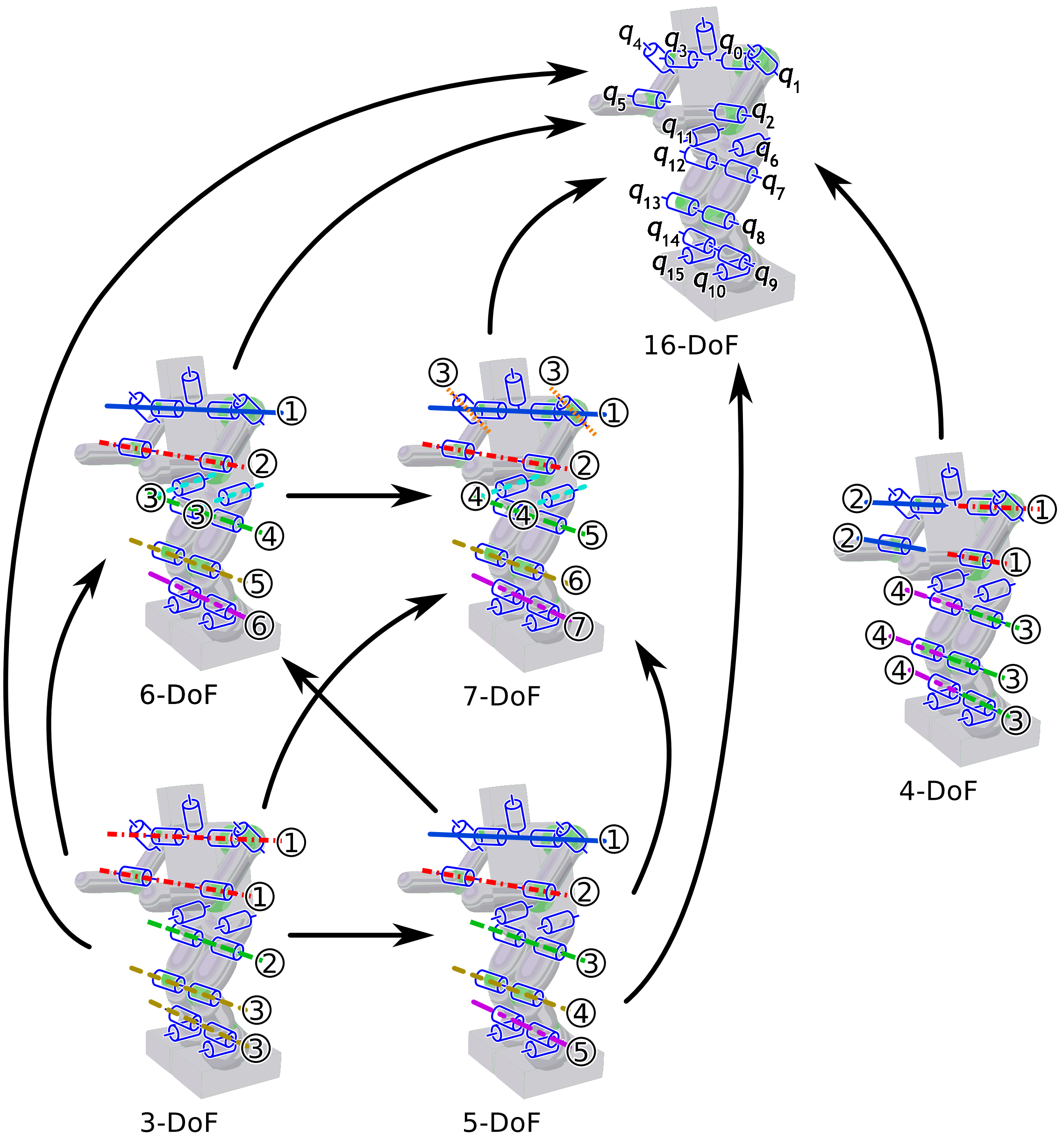
The possible freeing directions between these DoF configurations are defined as shown in Figure 6.2. Each arrow shows that freeing is possible in this direction.

Learning Method Configurations

We choose the parameters of LS fusion (denoted as LSF in the experiments) as follows: $\mathrm{f}_{\mathrm{UCB}} = 2$, $\alpha_{\mathrm{R}} = 0.05$, $\mathrm{N}_{\mathrm{LSBh}} = 10$, $\mathrm{N}_{\mathrm{LSSp}} = 20$, $\tau_{\mathrm{lsd0}} = 20$, $\delta_{\tau_{\mathrm{lsd}}} = 0.004$, $\sigma_{\mathrm{th}} = 0.2$, and $\mathrm{f}_{\mathrm{accel}} = 0.95$. The LS-scratch generates a behavior module whose parameters are set as $\tau_0 = 2$, $\delta_\tau = 0.02$ for Boltzmann selection of the decreasing temperature parameter $\tau = \tau_0 \exp(-\delta_\tau N_{\mathrm{eps}B})$ where $N_{\mathrm{eps}B}$ denotes a number of episodes performed by the behavior module B. The parameters of WF-DCOB are as follows: $C_{\mathrm{p}}(x) = \mathbf{q}_{N_{\mathrm{D}}}$, $C_{\mathrm{d}}(x) = \dot{\mathbf{q}}_{N_{\mathrm{D}}}$ for $x \in \mathcal{X}_{N_{\mathrm{D}}}$, and $\mathcal{I}_{\mathcal{R}} = \{(0.05, 0.1), (0.1, 0.2), (0.2, 0.3)\}$ for every configurations, and $\mathrm{F}_{\mathrm{abbrv}} = 0.5$ for

4-DoF, $\mathrm{F}_{\mathrm{abbrv}} = 1$ for the other DoFs. The LS-accelerating and the LS-freeing generate a behavior module of the parameters $\tau_0 = 0.1$, $\delta_\tau = 0.02$ since the behavior module starts from an almost converged policy. Every behavior modules use Peng's Q(λ)-learning (eq. (2.1)) with $\gamma = 0.9$, $\lambda = 0.9$, and a decreasing step size parameter $\alpha = \max(0.05, 0.3\exp(-0.002N_{\mathrm{eps}B}))$.

As a comparison, some configurations of WF-DCOB are also applied. Note that WF-DCOB for the crawling task is superior to conventional methods through



q_0 q_1 q_2 q_3 q_4 q_5 q_6 q_7 q_8 q_9 q_{10} q_{11} q_{12} q_{13} q_{14} q_{15}
16-DoF
6-DoF
7-DoF
4-DoF
3-DoF
5-DoF

Figure 6.2 Possible freeing directions between the DoF configurations. Each encircled number shows an index of dimension; joints with the same number are coupled. Each arrow shows that freeing is possible in this direction.

its performance is almost the same as that of DCOB (Chapter 4). We employ five conditions that are denoted as WF-DCOB-{3,4,5,6,7}. Each number indicates a DoF. All of them use Peng's Q(λ)-learning with the same parameters as a behavior module of the LS-scratch.

Results

We execute 10 runs for each configuration. Figure 6.3 shows the learning curves of the first five runs of LSF (ex0,...,ex4). In this figure, each circle shows the return acquired by a behavior module generated by the LS-scratch (we refer to it as a scratch behavior module). Namely, the other points on the solid curve are obtained by behavior modules generated by the LS-accelerating and the LS-freeing. In four out of five runs, the learning curves converge to higher values than that of the scratch behavior modules. This result means that the transfer learning by the LS-accelerating and the LS-freeing successfully improves the performance of the motion. Meanwhile, in ex2, the performance is not improved by the transfer learning. A possible reason is that the scratch behavior module acquires a high performance motion in the early stage of the learning, and there is no room for improvement. The notable point is that LS fusion algorithm selects a suitable sequence of the LSs including selection of a DoF configuration.

Let us see a detailed learning process. Figure 6.4 shows a learning curve and a behavior module transition in a run obtained from LSF (ex0 in Figure 6.3). The returns obtained by scratch behavior modules are also plotted by circles. In the early stage of the learning (0th to 500th episode), the scratch behavior modules dominate. One of them seems to converge to a return about 230 around 350th episode. The converged module uses the 3-DoF configuration, with which a behavior module can learn the policy quickly because of the lower dimension. Then, the LS-accelerating and the LS-freeing are applied. The behavior module used in the final stage of the learning is obtained through the following LS sequence: S(3-DoF)→F(3→5-DoF)→A→F(5→16-DoF)→A→A→A, where S denotes the LS-scratch, A denotes the LS-accelerating, and F denotes the LS-freeing. The final convergent value of the return is around 370, thus, the performance is improved by the transfer learning. Figure 6.5 shows snapshots during learning (ex0 in Figure 6.3), and Table 6.1 shows the profiles of motions in Figure 6.5. These results show how the performance of motion is improved.

Figure 6.6 shows the resulting learning curves of the crawling task (the mean of the return per episode over 10 runs). Among WF-DCOBs, WF-DCOB-3 converges fastest and converges to the highest value of return. A possible reason is that in addition to the lowest dimension, the joint coupling of the 3-DoF is suitable for the crawling task. On the other hand, LSF reaches at a higher value than that of WF-DCOB-3. The reason is thought to be an effect of LS fusion; though the return value of WF-DCOB-3 is not improved after convergence, LSF improves its policy by applying the LS-accelerating and the LS-freeing.

Therefore, these results demonstrate that using LS fusion enables (1) in the early stage of learning, an agent can select a suitable DoF configuration, and (2) after a scratch behavior module converges, the LS-accelerating and the LS-freeing can improve the policy.

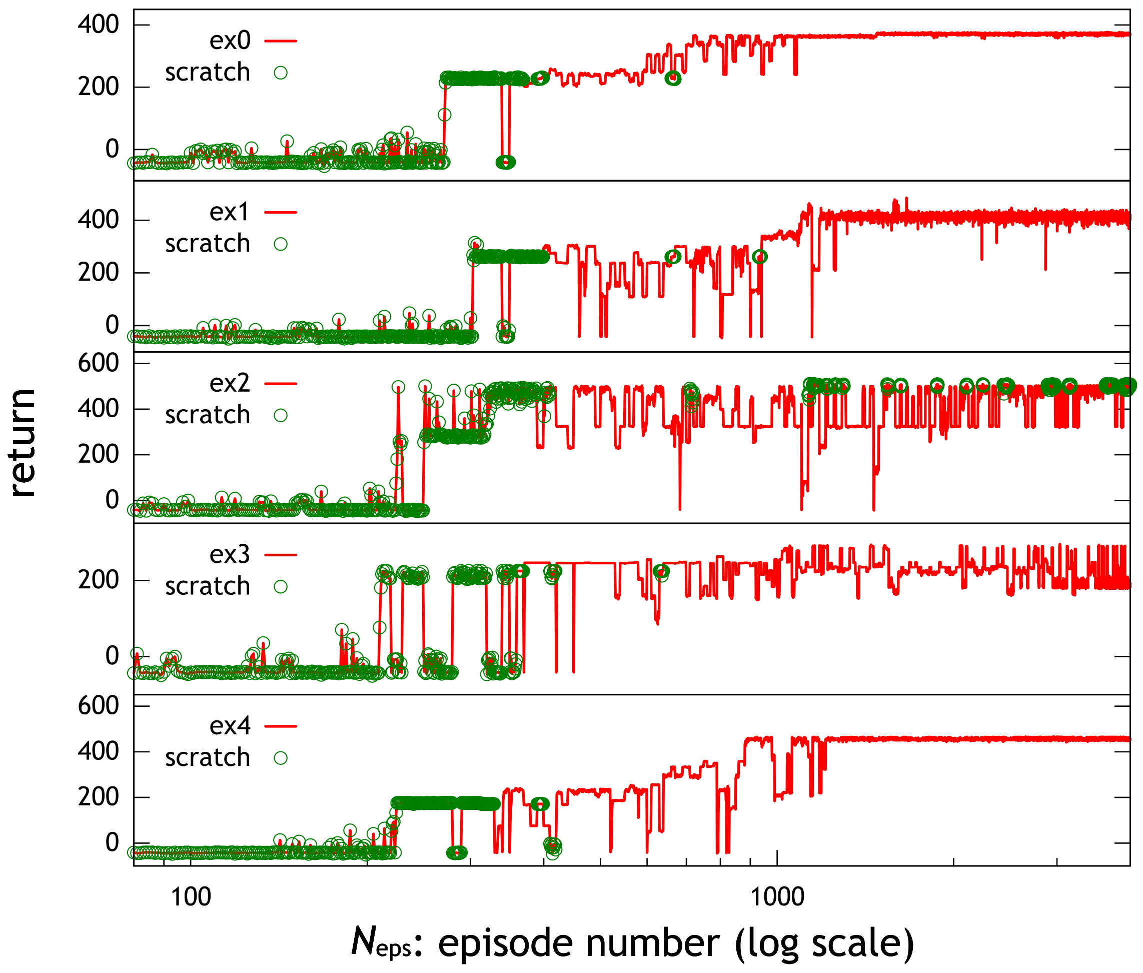


Figure 6.3 Learning curves of five runs obtained from LSF. Each solid line shows the return per episode, and each circle shows the return acquired by a behavior generated by the LS-scratch.

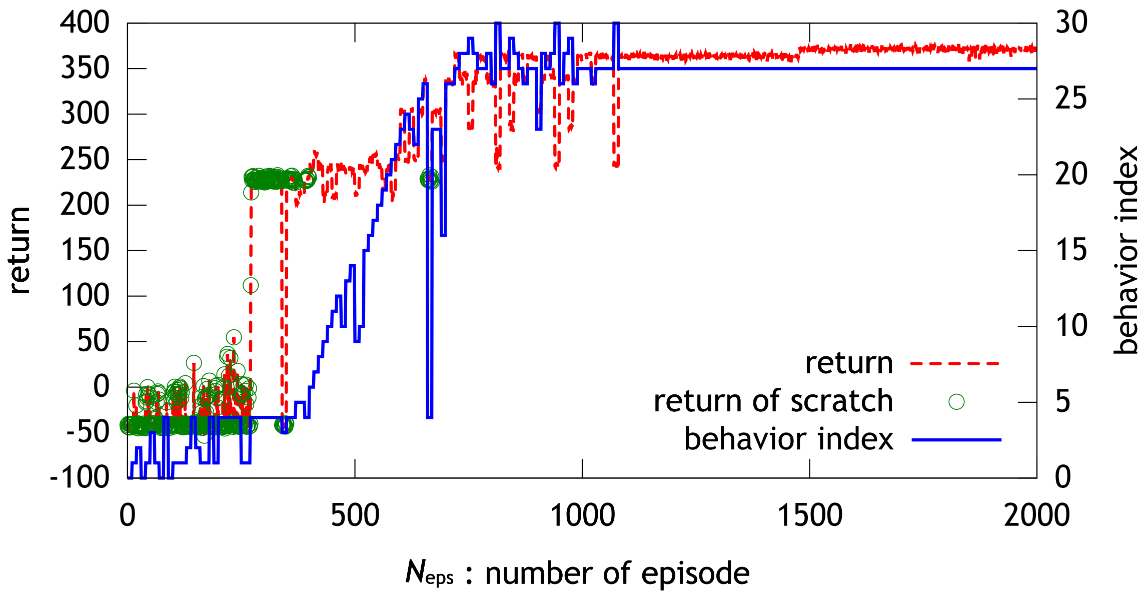


Figure 6.4 Learning curve and module transition in a run obtained from LSF (ex0 in Figure 6.3). The dotted line shows the return per episode, each circle shows the return acquired by a behavior generated by the LS-scratch, and the solid line shows the index of the selected behavior module in each episode.

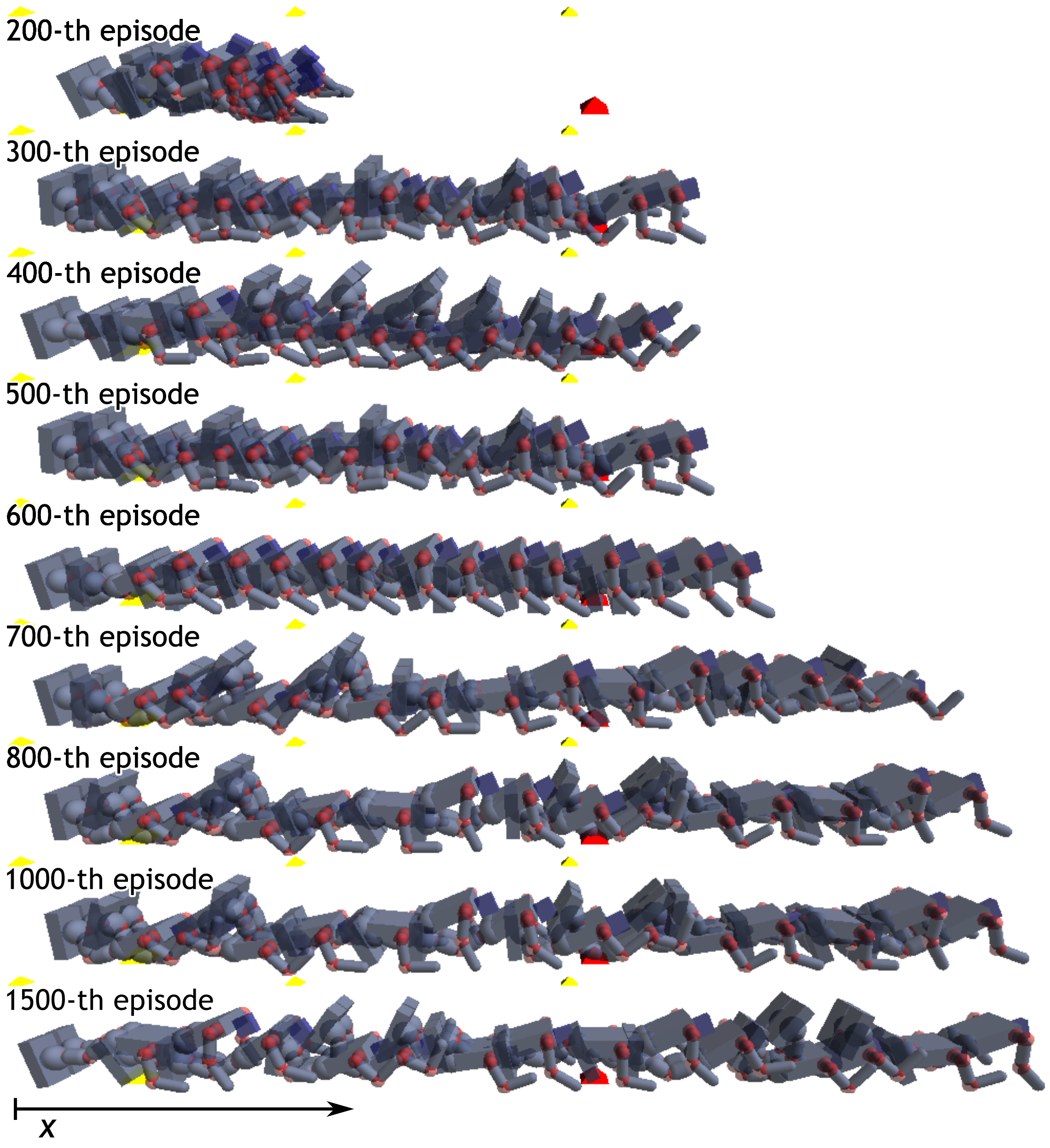


Figure 6.5 Snapshots of motions during learning (ex0 in Figure 6.3). Every snapshots are taken at 3-FPS during first 15 frames in each episode.

Table 6.1 Profiles of motions in Figure 6.5.

Episode	Return	Procedure
200	6.12	S(3)
300	231.55	S(3)
400	210.00	S(3)→F(3→16)
500	229.46	S(3)
600	248.01	S(3)→F(3→5)→A→F(5→6)→F(5→16)
700	328.49	S(3)→F(3→5)→A→F(5→16)→A→A
800	361.10	S(3)→F(3→5)→A→F(5→16)→A→A→A
1000	361.87	S(3)→F(3→5)→A→F(5→16)→A→A→A
1500	374.33	S(3)→F(3→5)→A→F(5→16)→A→A→A

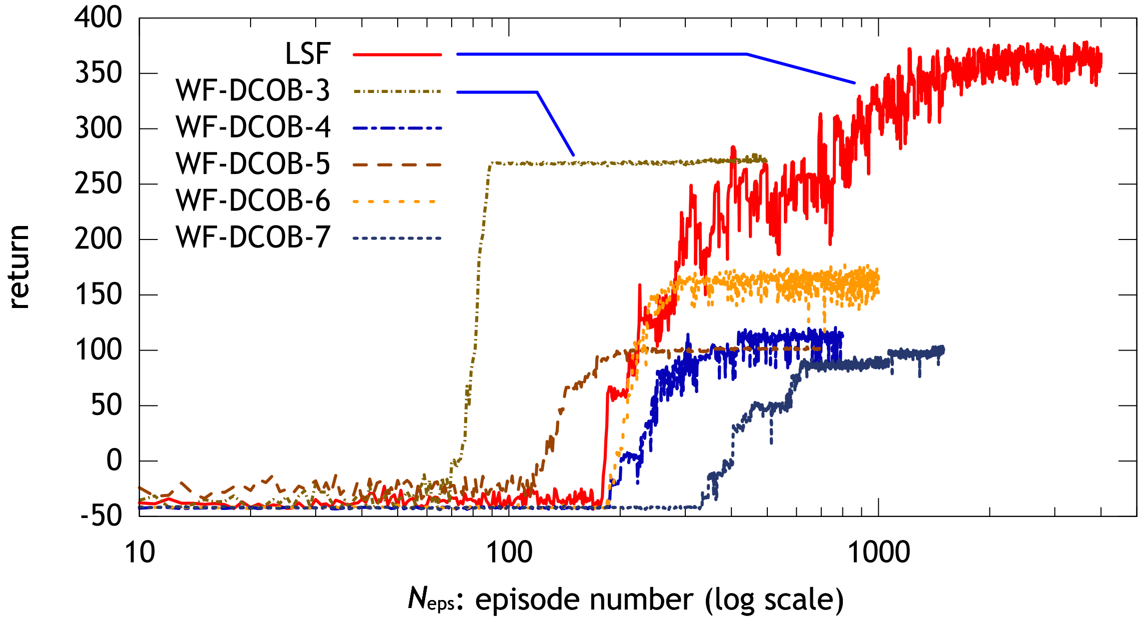


Figure 6.6 Resulting learning curves of the crawling task. Each curve shows the mean of the return per episode over 10 runs.

6.6.3 HumanoidMaze – Learning from Scratch

Next, to demonstrate the scalability, LS fusion is applied to a maze task of the simulated humanoid robot. In this task, the robot learns from scratch, which means it learns not only a path to goal, but also crawling and turning motions. This task is referred to as HumanoidMaze. In order to train the robot, we specify a task sequence; 0th to 1499th episode: the crawling task, 1500th to 2499th episode: the turning-left task, 2500th to 3499th episode: the turning-right task, and 3500th to 3999th episode: the maze task (easy2).

In this learning, each LS is expected to be used in the following scenario:

(1) The primitive tasks (crawling and turning) are learned with the LS-scratch, the LS-freeing, and the LS-accelerating.

(2) The LS-hierarchy generates a hierarchical action space in which the primitive tasks are treated as the subtasks.

(3) For the hierarchical action space, the LS-model generates a dynamics and a reward model module. Then the LS-planning generates a behavior module.

The parameter setup for LS fusion is the same as that in Section 6.6.2.

We execute 10 runs. Figure 6.7 shows the learning curves of the first five runs of LSF (ex0,...,ex4). In the learning stage of the crawling and the turning tasks, we can find that scenario (1) is achieved. Figure 6.8 shows a learning curve and a behavior module transition in a run obtained in ex0 of Figure 6.7. This graph shows how the set of behavior modules increases.

Figure 6.9 shows the resulting learning curves of the task (the mean of the return per episode over 10 runs). In learning the maze task, one out of ten runs fails to acquire a path to goal. The major factor is considered to be a poor connection between the crawling policy and the turning policy. Using such policies as low-level actions may remove the Markov property from the maze task. Thus, the robot fails in the maze task. Such failure will be avoided by mixing the primitive tasks in the primitive learning stage. This problem is not specific to LS fusion, but may also arise in the other hierarchical RL methods. Therefore, this failure does not dismiss the claim that LS fusion has scalability for complex tasks.

Figure 6.10 shows snapshots of an acquired behavior at the end of the maze task. We can see that the robot moves from start to goal by using crawling and turning motions.

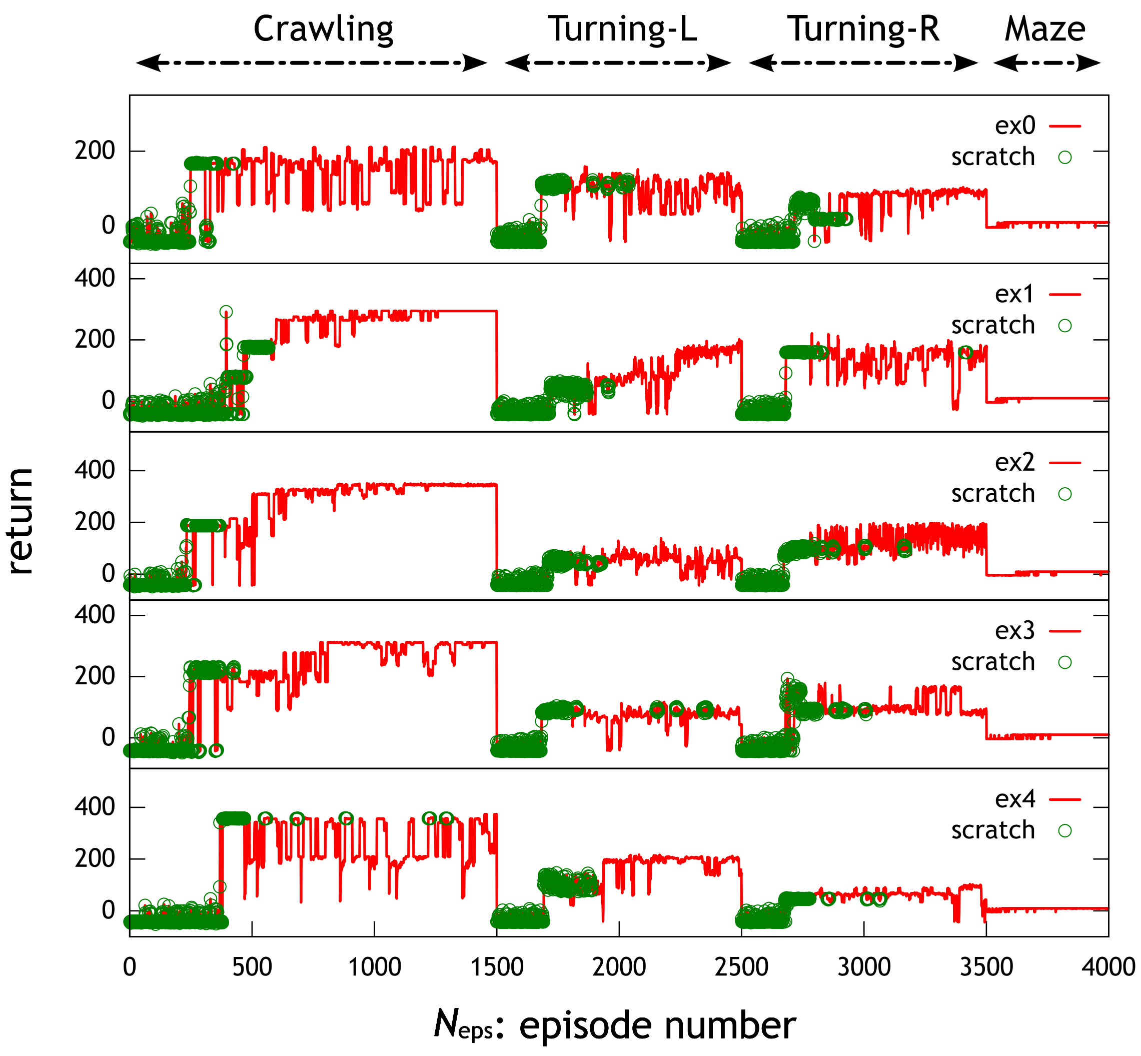


Figure 6.7 Learning curves of five runs obtained from LSF. Each solid line shows the return per episode, and each circle shows the return acquired by a behavior generated by the LS-scratch.

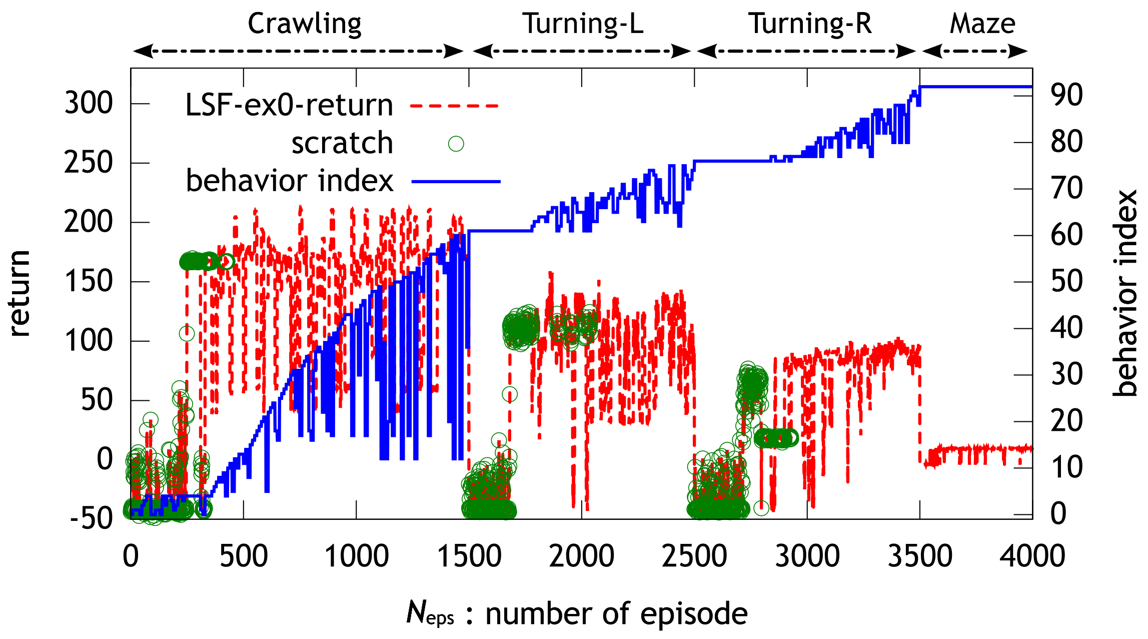


Figure 6.8 Learning curve and module transition in a run obtained from LSF (ex0 in Figure 6.7). The dotted line shows the return per episode, each circle shows the return acquired by a behavior generated by the LS-scratch, and the solid line shows the index of the selected behavior module in each episode.

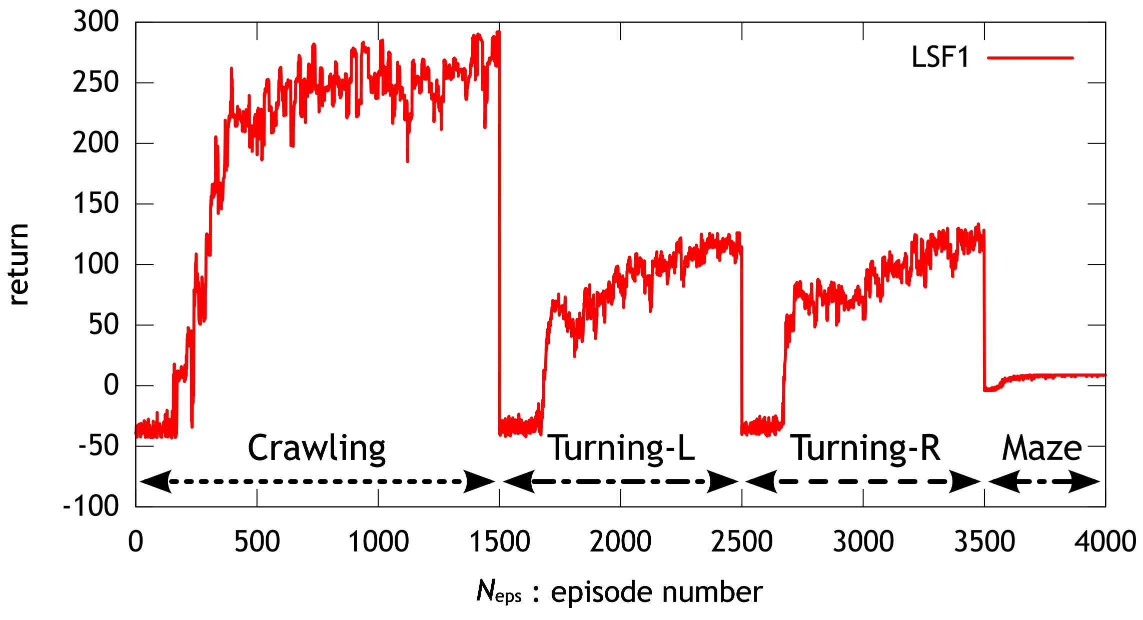


Figure 6.9 Resulting learning curves of the HumanoidMaze task (learning from scratch) where the task changes at 1500th, 2500th, and 3500th episode. Each curve shows the mean of the return over 10 runs per episode.

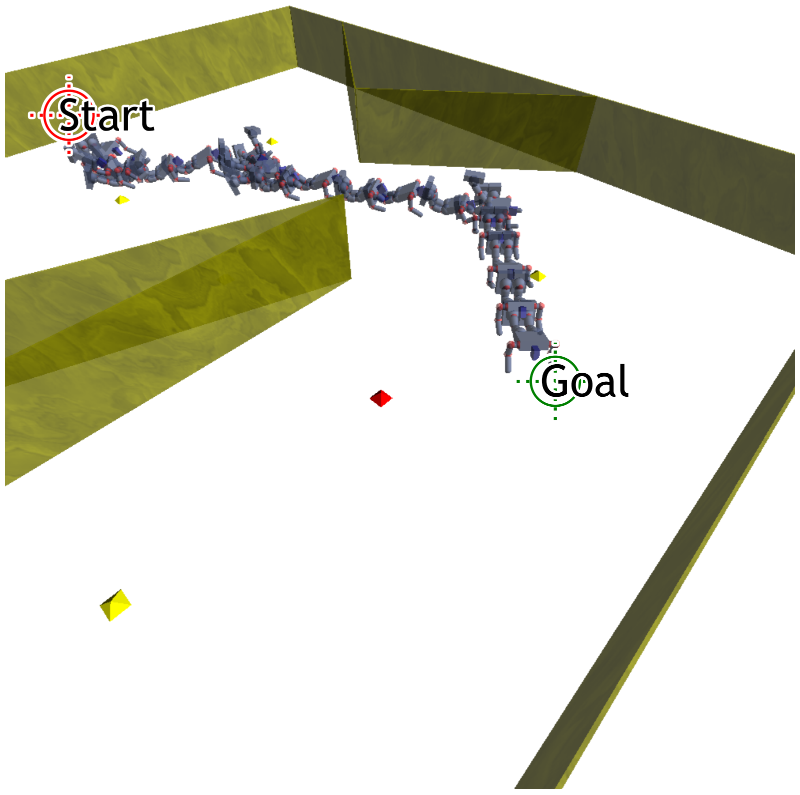


Figure 6.10 Snapshots of an acquired behavior at the end of the HumanoidMaze task (taken in 1-FPS).

6.6.4 HumanoidMaze – Model Transfer

This experiment demonstrates the availability of the LS-model and the LS-planning in the humanoid robot case. The task setup is similar to the one in Section 6.6.1. The difference is that in this experiment, we use a simulated humanoid robot that has crawling and turning policies previously learned as the primitive actions. Concretely, we use the crawling and the turning policies acquired in the experiments of Section 6.6.3 as the primitive actions. In 0th to 499th episode, a maze task (easy1) is learned, then a maze task (easy2) is learned. A similar scenario to the experiment in Section 6.6.1 is expected in this learning. The parameter setup for LS fusion is the same as that in Section 6.6.2.

Figure 6.11 shows the resulting learning curves (the mean of the return over 10 runs is plotted per episode). Here, the following methods are compared; LSF (MixFS): LS fusion where the LS-model uses MixFS dynamics model, LSF (Simple): LS fusion where the LS-model uses the Simple dynamics model, and Q(λ)-learning: normal RL methods for the two tasks. In learning the easy1 maze, the learning speed of these three are almost the same. In learning the easy2 maze, LSF (MixFS) is slightly faster than the others due to transferring the dynamics model. The reason why the improvement of the learning speed is not large is considered to be using complex primitive actions. These results indicate that LS fusion works as expected.

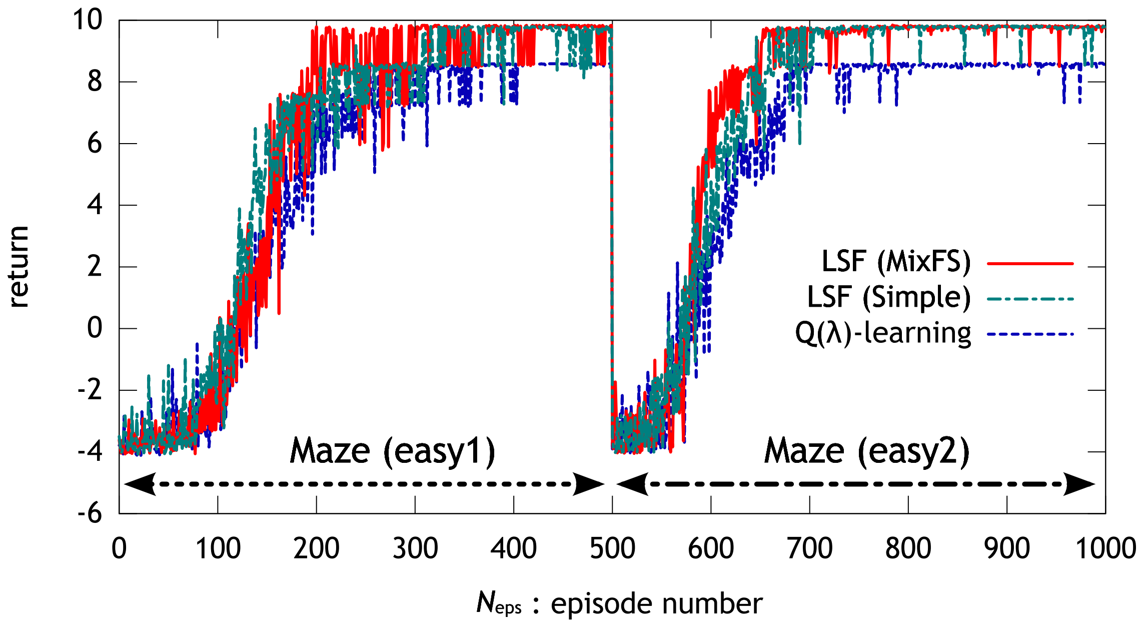


Figure 6.11 Resulting learning curves of the HumanoidMaze task (model transfer) where the maze changes at 500th episode. Each curve shows the mean of the return over 10 runs per episode.

6.7 Conclusion

This chapter proposed the learning strategy (LS) fusion method where some LSs are integrated for learning a single task by a single robot. As for LSs, we developed the LS-scratch, the LS-accelerating, the LS-freeing, the LS-planning, the LS-model, and the LS-hierarchy. In the LS fusion algorithm, an upper confidence bound (UCB) and Boltzmann selection method are employed to decide when and which LS is applied.

The simulation experiments of a crawling task of a small size humanoid robot demonstrated the advantage of LS fusion compared to learning with single learning modules. Namely, using LS fusion enables (1) in the early stage of learning, LS fusion can select a suitable DoF configuration, and (2) after a behavior module learning from scratch converges, the LS-accelerating and the LS-freeing can improve the policy.

The scalability of LS fusion was verified in a maze task of the simulated humanoid robot. In this task, the robot learned a path to a goal using crawling and turning motions acquired in previous learning sessions.

Our LS fusion architecture does not share the samples among the behavior modules. Introducing a kind of importance sampling to share the samples like CLIS (Uchibe and Doya 2004) may improve the learning speed, which is one of our future tasks.

Chapter 7

Application to Humanoid Locomotion

This chapter studies a new scheme for learning locomotion by a humanoid robot: a robot is embedded with a primitive balancing controller during learning. With this scheme, the robot learns locomotion in safety, and the size of a state-action space can be reduced. Specifically, this chapter investigates two possible approaches: (A) learning a foot-placement policy for locomotion where the robot has two stance modes, and (B) applying learning strategy (LS) fusion (Chapter 6) for learning locomotion. In approach (A), we consider several RL methods for this switching-stance-mode domain. In approach (B), we expect that the robot starts to learn with a lower degree of freedom, and incrementally improves the policy by the LSs.

7.1 Introduction

Past research on RL applications to locomotion utilizes central pattern generators (CPGs) (Matsubara, Morimoto, Nakanishi, Sato, and Doya 2006; Righetti and Ijspeert 2006; Nakamura, Mori, Sato, and Ishii 2007), or a property of passive dynamic walking (Tedrake, Zhang, and Seung 2004; Hitomi, Shibata, Nakamura, and Ishii 2006). These methods restrict the behavior of the robot to certain patterns or dynamics. But such restriction is desirable for walking, which reduces the learning time greatly.

In contrast, we study a new scheme for learning walking by a humanoid robot: a robot is embedded with a primitive balancing controller during learning as illustrated in Figure 7.1. Here, we employ a balancing controller proposed by Hyon *et al.* (Hyon, Hale, and Cheng 2007; Hyon 2009). This scheme has two advantages:

(1) The robot learns walking in safety.
(2) The size of a state-action space can be reduced.

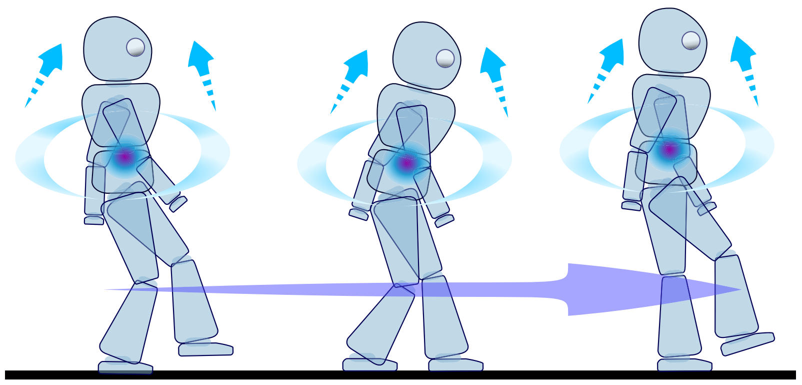


Figure 7.1 Illustration of the new learning-walking scheme. A humanoid robot is embedded with a primitive balancing controller during learning walking.

The balancing controller restricts the behavior of the robot to avoid falling down. Using the remaining DoF (degree of freedom), the robot learns to walk. In reality, the robot can still move its center of mass (CoM) and the leg joints, which enables the robot to walk. However, the dynamics of the robot with the controller becomes so complex that we cannot easily identify the dynamics model. Thus, it is difficult to design optimal walking gaits. Even so, model-free RL methods are applicable to this case.

Compared to the CPG approaches, the new scheme is thought to be safer since falling down is automatically prevented by the balancing controller. In some passive dynamic walking approaches, the falling down of the robot is avoided due to its hardware property. For instance, falling down of the robot in (Tedrake, Zhang, and Seung 2004) is a rare occasion. However, such kind of avoidance restricts the variety of robot behavior a lot. In contrast, the new scheme is a software approach. The balancing controller is implemented on a humanoid robot that has a wide variety of motions.

This chapter investigates two possible approaches:

(A) Applying an RL method to learn a foot-placement policy for walking. In this case, the robot has two stance modes: double stance and single stance. We consider several RL methods for this switching-stance-mode domain.

(B) Applying learning strategy (LS) fusion (Chapter 6) for learning humanoid locomotion. We expect that the robot starts to learn with a lower degree of freedom (DoF), and incrementally improves the policy by the LSs.

In the rest of this chapter, Section 7.2 describes approach (A), and Section 7.3 describes approach (B). Section 7.4 concludes this chapter.

7.2 RL Methods in Switching Stance Mode

This section investigates an RL method to learn a foot-placement policy for walking under the new scheme. The features of the RL problem in this case are as follows:

(F1) The robot learns from scratch to obtain better performance than hand-coded policies.

(F2) The robot uses an on-line learning method (or batch mode of small sizes) to handle (F1) and reduce the learning cost.

(F3) The robot has two modes: double stance and single stance, and the selectable action spaces (we call *sub*-action spaces) change according to the mode (Figure 7.2).

To handle (F3), we consider a hierarchical RL approach (Kirchner 1998; Barto and Mahadevan 2003; Cohen, Maimon, and Khmlenitsky 2006) and a *structured* function approximator (FA) approach. Kirchner applied a hierarchical version of Q-learning (HQL) to a similar task, the forward movement of a six-legged robot (Kirchner 1998). However, in our task, only two layers are needed to handle (F3). In this case, we can define a single FA into which the *sub*-FAs (corresponding to the sub-action spaces) are structured. Then we apply a normal (non-hierarchical) RL method to the structured FA. In general, if the available prior knowledge of a task is almost the same, a hierarchical RL method is inferior to a normal RL method since the former one has to limit its algorithm for converge. Thus, the structured FA is considered to be a better approach.

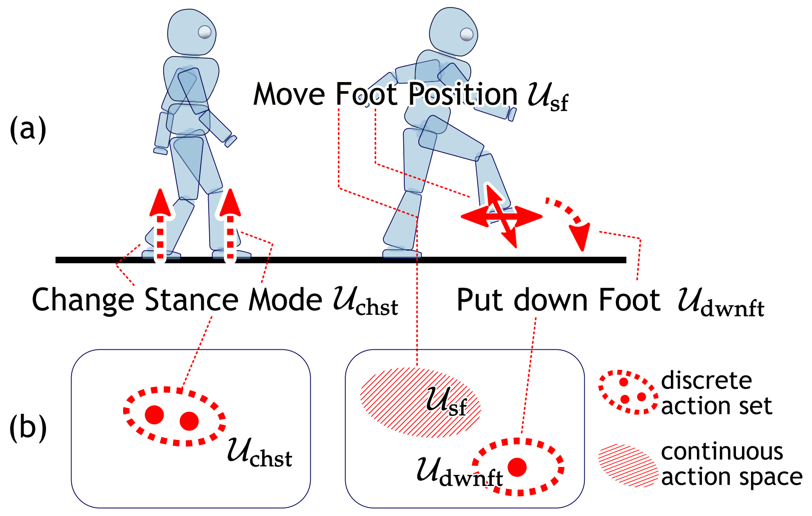

(a)
Move Foot Position $\mathcal{U}_{\text{sf}}$
Change Stance Mode $\mathcal{U}_{\text{chst}}$
Put down Foot $\mathcal{U}_{\text{dwnft}}$
(b)
$\mathcal{U}_{\text{chst}}$
$\mathcal{U}_{\text{sf}}$
$\mathcal{U}_{\text{dwnft}}$
discrete action set
continuous action space

Figure 7.2 Illustration of the switching stance mode. (a) Selectable actions in the double and the single stance modes. (b) Sub-action spaces.

Therefore, in this section, we compare Cohen's hierarchical RL (HRL) algorithm (Cohen, Maimon, and Khmlenitsky 2006) and a normal RL method that is applicable for the structured FA. As the normal RL method, we employ Peng's Q(λ)-learning (Peng and Williams 1994) and fitted Q iteration (Ernst, Geurts, and Wehenkel 2003; Ernst, Geurts, and Wehenkel 2005).

7.2.1 Cohen's Hierarchical Reinforcement Learning

As a hierarchical RL method, we use Cohen's hierarchical RL (HRL) (Cohen, Maimon, and Khmlenitsky 2006). Many hierarchical RL methods require that a task can be decomposed into sub-tasks. But, in our case, we only design a single reward function. Thus, the hierarchical RL methods requiring sub-tasks are not applicable to our case. However, Cohen's HRL uses a single reward function, which is suitable for our case. Thus, we choose to use it.

Though Cohen's HRL is developed for discrete state-action spaces, we extend this algorithm in a straightforward manner so that a FA is available for each module.

For our task, we construct a two layer modular structure. Specifically, it has one higher module and several lower modules. Each lower module has its unique sub-action space. The higher module selects a lower module as an action.

7.2.2 Structured Function Approximator

We define the structured FA for our walking task, where the action space consists of discrete sets and continuous vector spaces whose selectability depends on the state. Such an action space can be defined as a direct sum of discrete sets and vector spaces. Thus, we denote the action space as $\mathcal{U} = \bigsqcup_{p\in\mathcal{P}} \mathcal{U}_p$, where $\mathcal{U}_p$ is a sub-action space and $\mathcal{P}$ is a set of sub-space indexes. Again, see Figure 7.2 as an example of a set of sub-action spaces. In the following, we denote $u = (p, u_p) \in \mathcal{U}$ for convenience. We use $\mathcal{P}(x) \subseteq \mathcal{P}$ to express the selectable sub-action spaces at a state x.

We simply define a FA over the action space $\mathcal{U}$ by structuring (combining) sub-action value functions. First of all, we define each sub-action value function $Q_p(x, u_p)$ over $\mathcal{X} \times \mathcal{U}_p$. We choose the LFA-NGnet for a discrete set $\mathcal{U}_p$, and wire-fitting for a continuous space $\mathcal{U}_p$. Then we define the overall Q as

$$Q(x, u) \triangleq \sum_{p'\in\mathcal{P}} \delta_{pp'} Q_{p'}(x, u_{p'}) = Q_p(x, u_p), \tag{7.1}$$

where $u = (p, u_p)$. We let θ_p the parameter of a sub-action value function Q_p. The parameter vector of the Q can be defined as $\theta = (\theta_1^\top, \dots, \theta_{|\mathcal{P}|}^\top)^\top$. The derivative of Q w.r.t. θ is given by

$$\nabla_\theta Q(x,u)^\top = \big(\delta_{p1}\nabla_{\theta_1} Q_1(x,u_1)^\top, \\ \dots, \delta_{p|\mathcal{P}|}\nabla_{\theta_{|\mathcal{P}|}} Q_{|\mathcal{P}|}(x,u_{|\mathcal{P}|})^\top\big), \tag{7.2}$$

where $u = (p, u_p)$.

The greedy action at x can be given by

$$u^\star = \operatorname*{arg\,max}_{u \in \bigsqcup_{p\in\mathcal{P}(x)} \mathcal{U}_p} Q(x,u). \tag{7.3}$$

This can be evaluated as follows: (1) calculating $\hat{u}_p = \arg\max_{u_p \in \mathcal{U}_p} Q_p(x,u_p)$ for all $p \in \mathcal{P}(x)$, (2) calculating $u^\star = \arg\max_{u\in\{\hat{u}_p\}} Q(x,u)$.

As an exploration policy, we define a two-stage action selection method so that the RL agent can explore as broadly as possible and it has a scalability for any kind of sub-FAs. In the first stage, for each $p \in \mathcal{P}(x)$, select a sub-action $\hat{u}_p$ from $\mathcal{U}_p$ based on $Q_p(x,u_p)$. Here, we use Boltzmann selection if Q_p is the LFA-NGnet and WF-Boltzmann selection if Q_p is wire-fitting.

In the second stage, select an action u from $\{\hat{u}_p | p \in \mathcal{P}(x)\}$ based on their action values $\{Q_p(x,\hat{u}_p)\}$. We use a weighted version of Boltzmann selection to consider the size of $\mathcal{U}_p$ so that the RL agent can broadly explore. That is,

$$\pi(\hat{u}_p|x) = \frac{w_p \exp(\frac{1}{\tau}Q_p(x,\hat{u}_p))}{\sum_{p'\in\mathcal{P}(x)} w_{p'} \exp(\frac{1}{\tau}Q_{p'}(x,\hat{u}_{p'}))}, \tag{7.4}$$

where w_p denotes the weight to compensate the size of $\mathcal{U}_p$. We decide the weights $\{w_p\}$ so that they are proportional to the size of the action set $\mathcal{U}_p$ if $\mathcal{U}_p$ is discrete, or to the number of the control wires of Q_p if Q_p is wire-fitting. Note that in the early stage of learning, the probability of $\hat{u}_p$ is nearly proportional to w_p, which makes the exploration appropriate for the size of $\mathcal{U}_p$.

7.2.3 Experiments

We apply the RL methods mentioned above to the walking task of a human-size biped humanoid robot shown in Figure 7.3. The humanoid robot has 50 DoF and torque controllability with hydraulic actuation (Kawato 2008). Its height is 1.58 m, and its hip height is 0.82 m when at an upright posture. It weighs 93.7 kg. Its DoF configuration is shown in Figure 7.3(b). The arms and legs each have 7

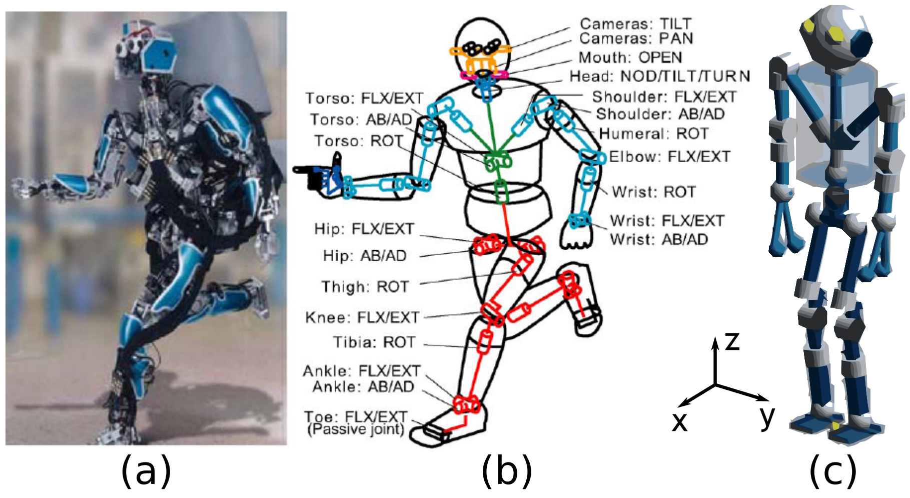


Figure 7.3 SARCOS biped humanoid robot developed by NICT/ATR (Kawato 2008). (a) Hardware. (b) DoF configuration. (c) Simulation model. The figures (a) and (b) are reprinted from (Hyon 2009).

DoFs, and the neck and torso each have 3 DoFs. In this section, we demonstrate a simulation comparison of the RL methods. In the experiments, we use a dynamics simulator with a precise model (Figure 7.3(c)) including a well-tuned contact model.

Robotic System Setup for RL

The robot is embedded with the balancing controller (Hyon 2009). It regulates the center of mass (CoM) to the center of the supporting region through the optimal force control. For biped walking case, the desired CoM, as well as the position of the swinging foot should be varied. Although the controller has compliant stabilization and terrain-adaptation capabilities, its performance is not satisfactory for dynamic situations because of sensory delays and limitations in the low-level joint controllers. Thus, we investigate which RL method is most appropriate for this situation.

The state given to the RL agent consists of the stance mode, $\{\mathsf{left}, \mathsf{double}, \mathsf{right}\}$, the position of the CoM, and the previous stance mode. We also tested the velocity of the CoM instead of the previous stance mode. But, the velocity is so sensitive to sensor noise and is therefore not suitable for the real robot case. Thus, the state x is defined as

$$x = (mode_{\mathrm{st}}, x_{\mathrm{cm}}, y_{\mathrm{cm}}, mode_{\mathrm{pst}}), \tag{7.5}$$

where x_{cm} and y_{cm} denote x- and y-position of the CoM respectively, $mode_{\mathrm{st}}$ de-

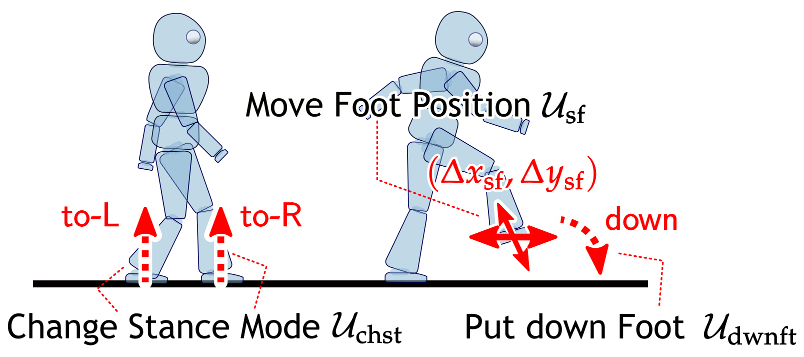

Move Foot Position $\mathcal{U}_{\text{sf}}$
$(\Delta x_{\text{sf}}, \Delta y_{\text{sf}})$
to-L
to-R
down
Change Stance Mode $\mathcal{U}_{\text{chst}}$
Put down Foot $\mathcal{U}_{\text{dwnft}}$

Figure 7.4 Illustration of the sub-action spaces of the walking task.

notes the current stance-mode, and $mode_{\text{pst}}$ denotes the past (previous) stance mode. In the following experiments, we allocate 405 Gaussians on a $3 \times 9 \times 5 \times 3$ grid as the basis functions of the NGnet.

Though the robot with the balancing controller can move the position of the CoM and the swinging foot, we pre-implement some primitive CoM movements so that the RL agent can learn the task easily. Thus, the available sub-action spaces are defined as illustrated in Figure 7.4. The available sub-action space in the double stance mode is defined as follows.

$\mathcal{U}_{\text{chst}} = \{\text{to-R}, \text{to-L}\}$: Changing the stance mode to right or left. Specifically, the position of CoM is moved to above the left/right foot, then the other foot is raised up. These actions are executed in 1.0 second respectively.

And the available sub-action spaces in the single stance mode are defined as follows.

$\mathcal{U}_{\text{dwnft}} = \{\text{down}\}$: Putting down the swinging foot. This action is executed in 1.0 second.

$\mathcal{U}_{\text{sf}} = \{(\Delta x_{\text{sf}}, \Delta y_{\text{sf}}) \mid \Delta x_{\text{sf}}, \Delta y_{\text{sf}} \in \mathbb{R}\}$: Moving x_{sf} and y_{sf} which denote the x and y positions of the swinging foot. Δx_{sf} and Δy_{sf} denote their differences. This action is executed in 0.1 second.

Namely, the action space is $\mathcal{U} = \mathcal{U}_{\text{chst}} \sqcup \mathcal{U}_{\text{dwnft}} \sqcup \mathcal{U}_{\text{sf}}$, and the selectable action spaces at a state x can be written by

$$\mathcal{P}(x) = \begin{cases} \{\text{chst}\} & \text{if } mode_{\text{st}} = \text{double}, \\ \{\text{dwnft}, \text{sf}\} & \text{if } mode_{\text{st}} = \text{left or right}. \end{cases} \tag{7.6}$$

RL Methods Configurations

For the discrete action spaces $\mathcal{U}_{\text{chst}}$ and $\mathcal{U}_{\text{dwnft}}$, we define Q_{chst} and Q_{dwnft} as the LFA-NGnet, respectively. For the continuous action space $\mathcal{U}_{\text{sf}}$, we define $Q^{\text{wf}}_{\text{sf}}$

as wire-fitting. For comparison, we also define $Q_{\mathrm{sf}}^{\mathrm{disc}}$ as the LFA-NGnet over $\mathcal{U}_{\mathrm{sf}}^{\mathrm{disc}}$ defined by discretizing $\mathcal{U}_{\mathrm{sf}}$ with a 3×3 grid. The parameters of every LFA-NGnet are initialized to zero. About $Q_{\mathrm{sf}}^{\mathrm{wf}}$, $\{\theta_i | i \in \mathcal{W}\}$ are initialized to zero, while $\{U_i | i \in \mathcal{W}\}$ are initialized with points of a 3×3 grid on $\mathcal{U}_{\mathrm{sf}}$.

In this experiments, we compare the following combinations of the RL algorithms and the sub-FAs[1].

S-WF-QL: Q(λ)-learning for Q^{wf} where Q_{chst}, Q_{dwnft}, and $Q_{\mathrm{sf}}^{\mathrm{wf}}$ are structured.

S-WF-QLFQI: The combination of Q(λ)-learning and fitted Q iteration for Q^{wf}.

S-DISC-QL: Q(λ)-learning for Q^{disc} where Q_{chst}, Q_{dwnft}, and $Q_{\mathrm{sf}}^{\mathrm{disc}}$ are structured.

S-DISC-QLFQI: The combination of Q(λ)-learning and fitted Q iteration for Q^{disc}.

S-DISC-Q0LFQI: The combination of Q(0)-learning and fitted Q iteration for Q^{disc}.

HRL: Cohen's HRL for a two layer modular structure where the lower modules learn Q_{chst}, Q_{dwnft}, and $Q_{\mathrm{sf}}^{\mathrm{disc}}$. The higher module learns the policy to select a lower module.

S-DISC-Q0LFQI is compared to verify the effect of the eligibility trace (λ). The reason why $Q_{\mathrm{sf}}^{\mathrm{disc}}$ is used in HRL rather than $Q_{\mathrm{sf}}^{\mathrm{wf}}$ is due to the stability of the linear FA.

For every RL method, we set $\gamma = 0.95$. We use a decreasing step size parameter $\alpha = \max(0.05, 0.3\exp(-0.002 N_{\mathrm{eps}}))$ for Q(λ)-learning and HRL. N_{eps} denotes a number of episodes. For fitted Q iteration, we use a constant step size parameter $\alpha = 0.05$. For Q(λ)-learning, we set $\lambda = 0.9$ and apply the *replacing trace* (Singh and Sutton 1996) to make the eligibility trace stable (see also (Tsitsiklis and Roy 1997)). For the combination of Q(λ)-learning and fitted Q iteration, we set $N_{\mathrm{FQI}} = 3$ and $N_{\mathrm{smpl}} = 10$. As the exploration policy, we use Boltzmann (or Boltzmann-like) selection, with a decreasing temperature $\tau = 1.0\exp(-0.002 N_{\mathrm{eps}})$.

Task Setup

The objective of the walking task is to move forward along the x-axis as far as possible. Though the balancing controller is embedded, the robot still has a probability of falling down. The balancing controller employed in this thesis does not consider the swinging leg motions explicitly. When the swinging leg moves too

[1]We used the RL library, SkyAI: `skyai.sourceforge.net`

fast and CoP locates near the supporting edge, then the robot can lose the stability. Thus, we design the reward function as follows:

$$r(t) = r_{\text{mv}}(t) - r_{\text{sc}}(t) - r_{\text{fd}}(t), \tag{7.7a}$$
$$r_{\text{mv}}(t) = 200 v_{\text{cm}x}(t), \tag{7.7b}$$
$$r_{\text{sc}}(t) = 1\delta \text{t}, \tag{7.7c}$$
$$r_{\text{fd}}(t) = \begin{cases} 50 & \text{if falling-down,} \\ 0 & \text{otherwise,} \end{cases} \tag{7.7d}$$

where r_{mv} means a reward for moving, r_{sc} means a step cost, and r_{fd} means a penalty for falling down[2]. The sum of $r(t)$ during an action is given to the RL agent as the reward for the action. Each episode starts with the initial state where the robot is standing up (first snapshot in Figure 7.7) and stationary, and ends if $t > 75$ s or the robot is falling down.

Result

Figure 7.5 shows the resulting learning curves of the walking task (the mean of the return over 10 runs per episode). The horizontal axis is in logarithmic scale. The horizontal line (MANUAL) shows the performance of a manually manipulated walking with a keyboard interface[3]. Figure 7.6 shows the trajectory in an episode of the CoM position of a walking gait acquired by S-WF-QLFQI, and Figure 7.7 shows the corresponding snapshots.

HRL and S-DISC-Q0LFQI are very slow, that is, they take a lot of episodes to acquire performance. A possible reason is that the update rule of HRL is similar to Q(0)-learning which may be poor for the walking task. About S-DISC-Q0LFQI, Q(0)-learning is dominant in the early stage of learning rather than fitted Q iteration since suitable samples are not obtained yet. Thus, S-DISC-Q0LFQI is considered to be as slow as HRL.

All of the methods using Q(λ)-learning are much faster than these two methods. Thus, the eligibility trace (λ) is considered to be very effective for the walking task.

The acquired performance of S-WF-QLFQI is the best, which implies that it converges to the highest value of the return. It substantially exceeds the performance of the manually manipulated walking. Compared with Q^{disc}, the FA

[2]Specifically, the falling-down is defined as $(|\phi| > 25.0^\circ) \vee (|\theta| > 25.0^\circ)$ where ϕ and θ denote the x and y-Euler angles respectively.

[3]The maximum return value in 30 trials is plotted.

Q^{wf} has an ability to acquire better performance since Q^{wf} directly approximate over the continuous action space $\mathcal{U}_{\mathrm{sf}}$ by wire-fitting. However, there is no performance difference between S-DISC-QL and S-WF-QL because of the instability of wire-fitting caused by its nonlinearity. Thus, we consider the reason for the best performance of S-WF-QLFQI is due to both using wire-fitting and updating by fitted Q iteration which is more stable than Q(λ)-learning. We should also note that S-WF-QLFQI is slightly slower than S-WF-QL. A possible reason is that Q(λ)-learning and fitted Q iteration conflict since the definition of the action value function is slightly different in the two algorithms.

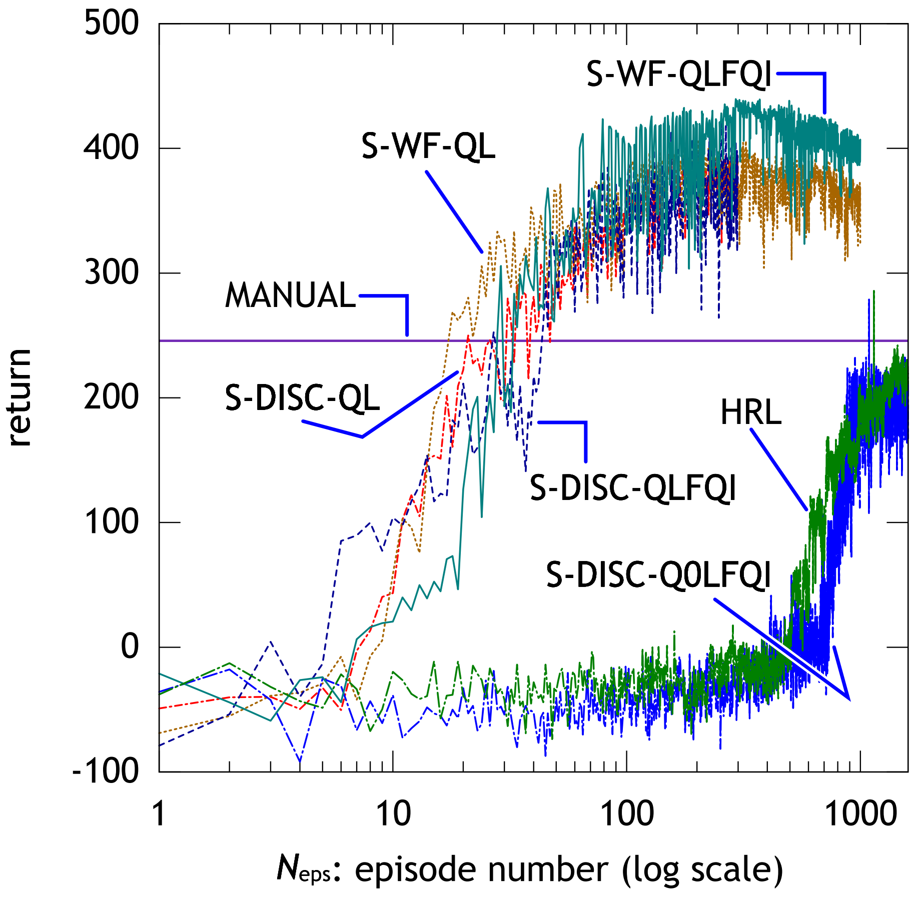


Figure 7.5 Resulting learning curves of the walking task. Each curve shows the mean of the return over 10 runs per episode. The horizontal line (MANUAL) shows the performance of a manually manipulated walking with a keyboard interface.

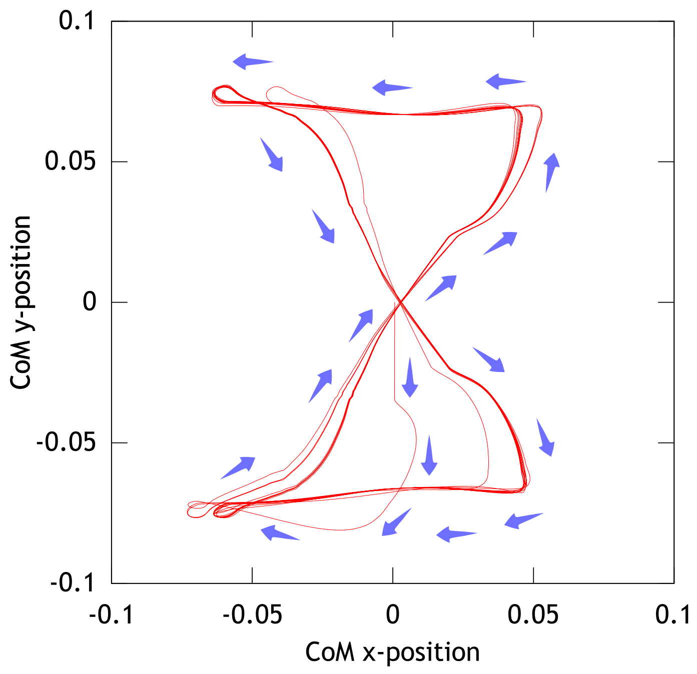


Figure 7.6 Trajectory of the CoM position. Small arrows indicate the direction of the movement. Note that the origin of the CoM is the center of the feet.

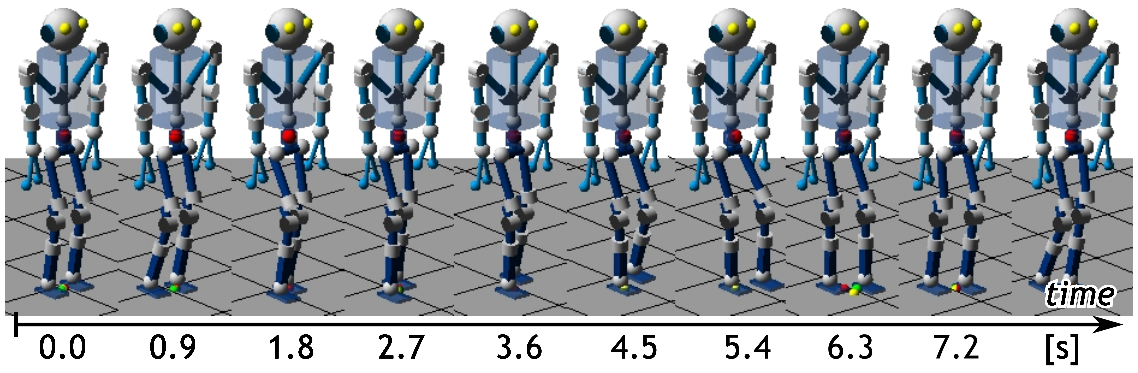


Figure 7.7 Animation snapshots.

7.3 Learning Strategy Fusion for Direct Joint Control

Next, we apply learning strategy (LS) fusion proposed in Chapter 6 to the humanoid locomotion task.

7.3.1 LS Fusion Setup

The humanoid locomotion is a motion learning task; a policy to be learned is a low-level control law. Thus, as LSs, we employ the LS-scratch, the LS-freeing, and the LS-accelerating. These LSs use WF-DCOB as defined in Chapter 6. In this task, we expect a following scenario:

(1) In the early stage of learning, a walking motion near a stepping motion at the same place is learned by using a lower DoF configuration.
(2) Then, the LS-freeing and the LS-accelerating are applied multiple times to speed up.
(3) Finally, a fast-walking policy using a higher DoF configuration is acquired.

Since an acquired policy is assumed to use a higher (near full) DoF configuration, we directly control the joint angles rather than a task space. Concretely, we control the leg joints by a simple PD-controller. This joint controller outputs torque commands, which linearly added to the balancing controller's torque commands. The total torque commands are applied to the robot.

7.3.2 Experiments

The task setup is the same as defined in the previous section except for the reward function. For these experiments, we change the penalty for falling down as follows:

$$r_{\text{fd}}(t) = \begin{cases} 20 & \text{if falling-down,} \\ 0 & \text{otherwise.} \end{cases} \tag{7.8}$$

The reason why the penalty is reduced to 20 is that the original penalty (50) is too large for the LS-freeing and the LS-accelerating to judge if a behavior module is trained enough. The whole reward function is as defined in eq. (7.7).

Space Configurations

We use four sets of DoF configurations (Figure 7.8): 2-DoF $(\tilde{\mathcal{U}}_2, \mathcal{X}_2)$, 4-DoF $(\tilde{\mathcal{U}}_4, \mathcal{X}_4)$, 8-DoF $(\tilde{\mathcal{U}}_8, \mathcal{X}_8)$, and 10-DoF $(\tilde{\mathcal{U}}_{10}, \mathcal{X}_{10})$. In every DoF configurations, the left leg and the right leg move independently and have a same joint-coupling configuration; thus, only the configurations of the left leg are illustrated in Figure 7.8. In the 2-DoF, a joint pair $\{q_1, q_2, q_4, q_6, q_7\}$ is coupled. In the 4-DoF, joint pairs $\{q_1, q_7\}$, $\{q_2, q_4, q_6\}$ are coupled respectively. In the 8-DoF, a joint pair $\{q_2, q_4\}$ is coupled, and q_1, q_6, q_7 move independently. In the 10-DoF, q_1, q_2, q_4, q_6, q_7 move independently. The joints q_3 and q_5 are fixed at neutral positions.

In N_D-DoF configuration, its command input space is a N_D-dimensional vector space that represents target joint angles. Its state space is

$$x = (\mathbf{q}_{N_\mathrm{D}}^\top, v_{\mathrm{cm}y})^\top \tag{7.9}$$

where $v_{\mathrm{cm}y}$ denotes y-velocity of the CoM, and $\mathbf{q}_{N_\mathrm{D}}$ denotes the coupled joint angle vector of the N_D-DoF.

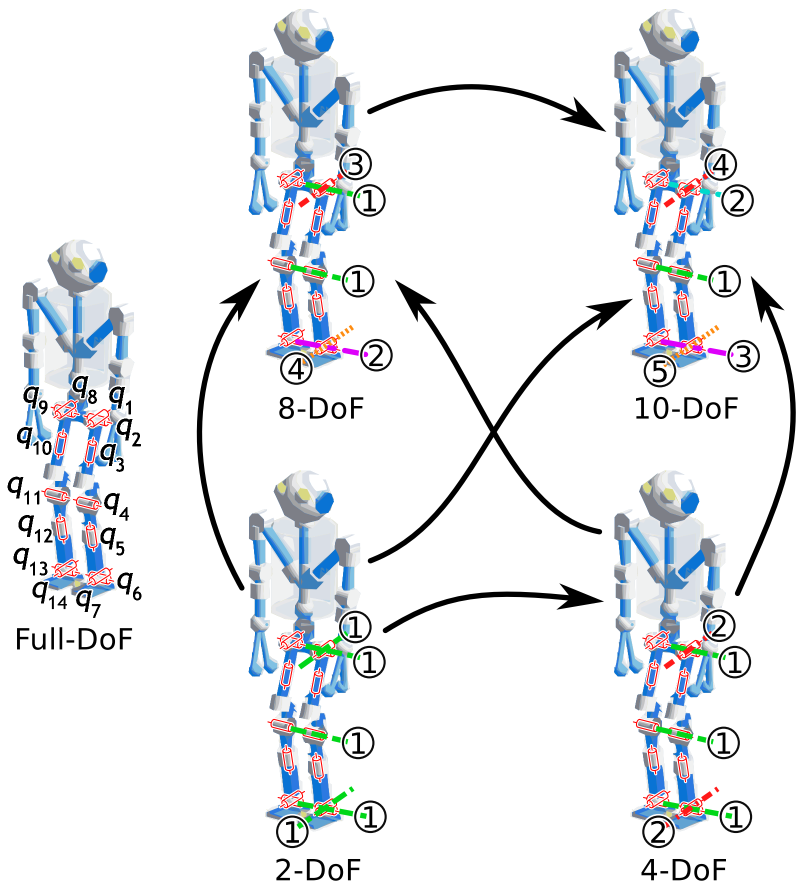


Figure 7.8 DoF configurations and possible freeing directions. The Full-DoF configuration is not used in the experiments. Each encircled number shows an index of dimension; joints with the same number are coupled. In every DoF configurations, the left leg and the right leg move independently and have a same joint-coupling configuration. Each arrow shows that freeing is possible in this direction.

In order to realize a fast learning, the default BFs are allocated only on $\mathcal{X}_2$; namely, the LS-scratch generates only a behavior module that uses the 2-DoF configuration. Specifically, we allocate 125 BFs by the spring-damper allocation method (Chapter 2).

The possible freeing directions between these DoF configurations are defined as shown in Figure 7.8. Each arrow shows that freeing is possible in this direction.

Learning Method Configurations

We choose the parameters of LS fusion (denoted as LSF in the experiments) as follows: $\mathrm{f}_{\mathrm{UCB}} = 2$, $\alpha_{\mathrm{R}} = 0.1$, $\mathrm{N}_{\mathrm{LSBh}} = 5$, $\mathrm{N}_{\mathrm{LSSp}} = 20$, $\tau_{\mathrm{lsd0}} = 2$, $\delta_{\tau_{\mathrm{lsd}}} = 0.004$, $\sigma_{\mathrm{th}} = 0.2$, and $\mathrm{f}_{\mathrm{accel}} = 0.95$. The LS-scratch generates a behavior module whose parameters are set as $\tau_0 = 2$, $\delta_\tau = 0.02$ for Boltzmann selection of the decreasing temperature parameter $\tau = \tau_0 \exp(-\delta_\tau N_{\mathrm{eps}B})$ where $N_{\mathrm{eps}B}$ denotes a number of episodes performed by the behavior module B. The parameters of WF-DCOB are as follows: $C_{\mathrm{p}}(x) = \mathbf{q}_{N_{\mathrm{D}}}$, $C_{\mathrm{d}}(x) = \mathbf{0}^{N_{\mathrm{D}}}$ for $x \in \mathcal{X}_{N_{\mathrm{D}}}$, and $\mathcal{I}_{\mathcal{R}} = \{(0.8, 1.5), (1.5, 3.0)\}$, $\mathrm{F}_{\mathrm{abbrv}} = 1$ for every DoF configurations. The LS-accelerating and the LS-freeing generate a behavior module of the parameters $\tau_0 = 0.1$, $\delta_\tau = 0.02$ since the behavior module starts from an almost converged policy. Every behavior modules use Peng's Q(λ)-learning (eq. (2.1)) with $\gamma = 0.9$, $\lambda = 0.9$, and a decreasing step size parameter $\alpha = \max(0.05, 0.1 \exp(-0.002 N_{\mathrm{eps}B}))$.

As a comparison, some configurations of WF-DCOB are also applied. We employ a condition denoted as WF-DCOB-2 that uses the 2-DoF configuration and the Peng's Q(λ)-learning with the same parameters as a behavior module of the LS-scratch.

Results

We execute 10 runs for each condition. Figure 7.9 shows the learning curves of the first five runs of LSF (ex0,...,ex4). Each circle shows the return acquired by a behavior module generated by the LS-scratch (we refer to it as a scratch behavior module). In every run, the learning curve converges to a higher value than that of the scratch behavior modules. This result means that the transfer learning by the LS-accelerating and the LS-freeing successfully improves the performance of the motion. However, the transfer LSs are applied earlier than that of Section 6.6.2; that is, the transfer LSs are applied before the scratch module converges in some runs. A possible reason is that the maximum value of the deviation is large, with which the behavior module is recognized to be trained enough[4]. However, LS fusion algorithm selects a suitable sequence of the LSs including selection of a DoF configuration.

Figure 7.10 shows the resulting learning curves of the humanoid locomotion task (the mean of the return per episode over 10 runs). LSF reaches a higher value than that of WF-DCOB-2. The reason is considered to be an effect of LS fusion; though the return value of WF-DCOB-2 is not improved after convergence, LSF improves its policy by applying the LS-accelerating and the LS-freeing.

Therefore, these results demonstrate that LS fusion is effective in the humanoid locomotion task as well as that in the HumanoidML-crawling case shown in Section 6.6.2.

[4]Specifically, $\overline{\sigma}_{\max B_{\mathrm{src}}}$ is relatively large in the generable condition of the LS-accelerating and the LS-freeing.

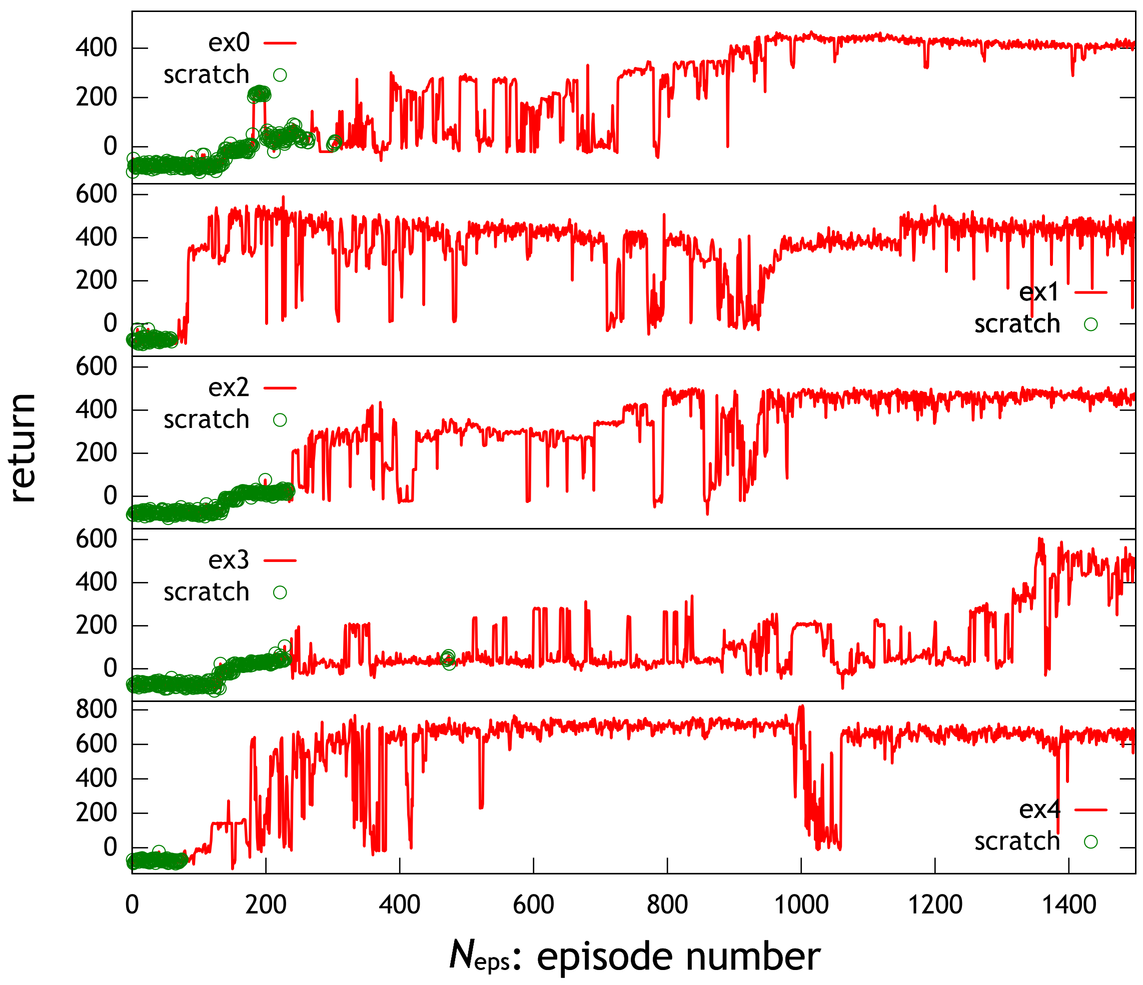


Figure 7.9 Learning curves of five runs obtained from LSF. Each solid line shows the return per episode, and each circle shows the return acquired by a behavior generated by the LS-scratch.

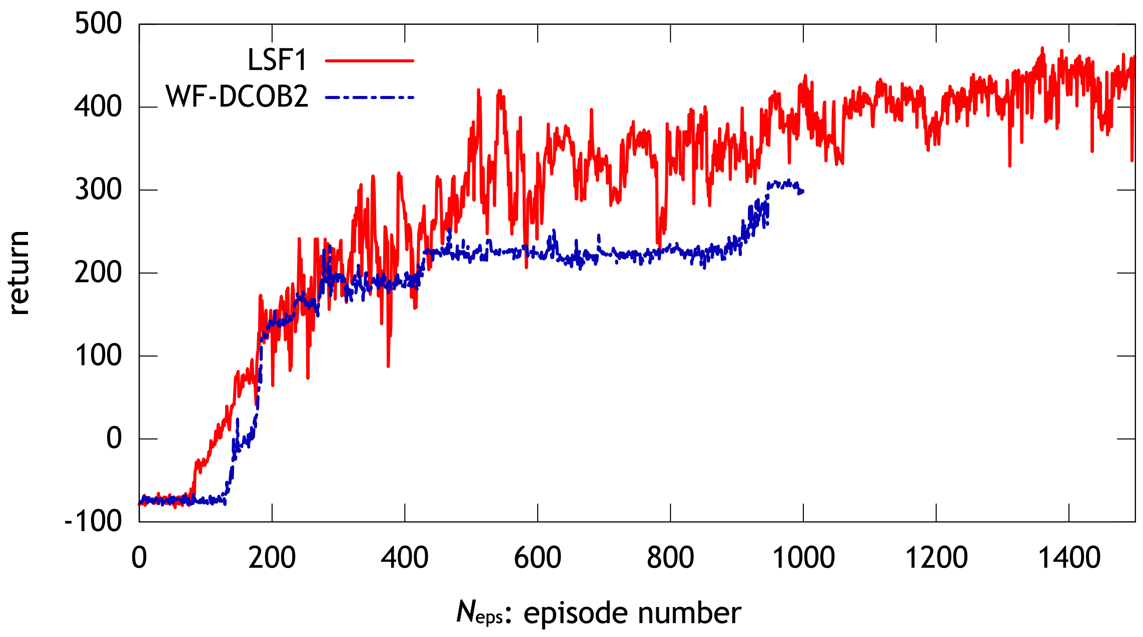


Figure 7.10 Resulting learning curves of the humanoid locomotion task. Each curve shows the mean of the return per episode over 10 runs.

7.4 Conclusion

This chapter studied a new scheme for learning locomotion by a humanoid robot, that is, a robot is embedded with a primitive balancing controller during learning. The advantages of this scheme are that a robot learns locomotion in safety, and that the size of a state-action space can be reduced. This chapter investigated two approaches: (A) learning a foot-placement policy under a switching-stance-mode domain, and (B) applying LS fusion for learning locomotion.

In approach (A), we considered a hierarchical RL approach and FA approaches, and compared them in simulation. The results demonstrated that Cohen's hierarchical RL algorithm did not work well; it took a longer learning time. On the other hand, the structured FA was defined for our situation. The RL methods based on Peng's Q(λ)-learning could obtain a suitable policy much faster than the HRL. Especially, applying the combination of fitted Q iteration and Q(λ)-learning to the structured FA acquired the best performance. The reason is considered as both the accuracy of wire-fitting and the stability of fitted Q iteration.

In approach (B), we employed the LS-scratch, the LS-freeing, and the LS-accelerating. Since an acquired policy was assumed to use a higher (near full) DoF configuration, we directly controlled the joint angles rather than a task space. Simulation results demonstrated that LS fusion is effective in the humanoid locomotion task as well as that in the HumanoidML-crawling case shown in Section 6.6.2. Namely, the robot started to learn with a lower degree of freedom, and incrementally improved the policy by the LSs.

Chapter 8

Conclusion of Thesis

This thesis proposed a *highly modularized learning system* that is suitable for the objective-based task design. The proposed system realized that multiple learning strategies (LSs) are applied to each task of a single robot multiple times, where the ordering of the LSs is automatically decided. For this realization, the system consists of four elements: (1) behavior modules, (2) fundamental module types, (3) LS modules, and (4) UCB-Boltzmann selection method.

This thesis proposed some elemental technologies to define LSs. In Chapter 4, DCOB, a method to generate a discrete action space for large domains, was proposed. DCOB was extended to WF-DCOB for learning continuous actions. Based on WF-DCOB, the LS-scratch and two transfer LSs, the LS-freeing and the LS-accelerating, were defined. In Chapter 5, a method to decompose a dynamics model was proposed, with which we can obtain a task invariant element of a dynamics model. The LS-model uses this method to learn a dynamics. Furthermore, the LS-model also use the method to transfer a dynamics model of a task to one of the other task. In addition, a highly modularized RL library SkyAI was developed to implement the proposed methods. Chapter 3 introduced SkyAI.

In Chapter 6, the core algorithm of the highly modularized learning system was proposed, which is referred to as LS fusion. LS fusion includes UCB-Boltzmann selection method to select a behavior module in each learning stage. The most distinct feature of LS fusion is that it automatically applies multiple LSs for each task multiple times. The definition of the LSs were also described in this chapter.

A lot of experiments were performed to verify the proposed methods. The DCOB and the WF-DCOB outperformed the conventional methods especially in motion learning tasks of a simulated humanoid robot. The DCOB also demonstrated the outstanding performance in a crawling task of a real spider robot. The method to decompose a dynamics model improved the learning speed of maze tasks. LS fusion improved the performance of acquired motions by WF-

DCOB. The scalability of LS fusion was verified in a maze task of the simulated humanoid robot. In this task, the robot learned a path to a goal using crawling and turning motions acquired in previous learning sessions.

In addition, an application to humanoid locomotion was studied in Chapter 7. This chapter studied a new scheme for learning locomotion by a humanoid robot: a robot is embedded with a primitive balancing controller during learning. The simulation experiments demonstrated that LS fusion is also effective in this application.

Through these experiments, this thesis concludes that the proposed highly modularized learning system is one of the most realistic approaches to realize the objective-based task design.

Appendix A

Spring-Damper Allocation

This method allocates K BFs on a state space. First, K hyperspheres that have the same radius r and different positions $p_{\mathrm{s}[1,\ldots,K]}$ are allocated on the state space so that the spheres spread as widely as possible and the radius is a maximum value without overlapping. Then, each BF is calculated from r and $p_{\mathrm{s}[k]}$.

To allocate the hyperspheres, pseudo-dynamics of a spring-damper system is calculated. Algorithm 6 shows the entire procedure. Here, the variables mean

D:	dimensionality of the state space.
r:	radius of sphere (common in every spheres).
v_{r}:	speed of radius expansion.
$p_{\mathrm{s}[1,\ldots,K]}$:	position of each sphere.
$v_{\mathrm{s}[1,\ldots,K]}$:	velocity of each sphere.
K_{sp}:	spring constant (1.0).
F_{m}:	margin ratio (0.2).
$\mathrm{x}_{\mathrm{max}}$:	upper bound of state space.
$\mathrm{x}_{\mathrm{min}}$:	lower bound of state space.
f_{ri}:	internal force of radius to expand (0.1).
D_{r}:	dumping constant of radius's movement (10.0).
D_{s}:	dumping constant of sphere's movement (2.0).
m_{r}:	radius's mass (5.0).
m_{s}:	sphere's mass (5.0).
$\delta\mathrm{t}$:	time-step (0.1).

Each parenthetic value denotes a typical value of the constant.

Using the output of the algorithm, the parameters of each BF k (Gaussian) is calculated as follows:

$$\mu_k = p_{\mathrm{s}[k]}, \tag{A.1}$$

$$\Sigma_k = (\mathrm{F}_{\Sigma} r)^2 \mathbf{1}^{\mathrm{D}}, \tag{A.2}$$

where F_{Σ} is a scaling constant (typical value is 0.6).

Algorithm 6: Spring-damper allocation

```
1: initialize $r$ (proper value), $p_{s[1,\dots,K]}$ (random), $v_r$ (zero), $v_{s[1,\dots,K]}$ (zero)
2: repeat
3:   Calculate force:
4:   $f_r \leftarrow 0$ /* force that expands radius */
5:   $f_{s[1:K]} \leftarrow \mathbf{0}^D$ /* total force of each sphere */
6:   for $i_1 = 1, 2, \dots, K$ do /* for each sphere */
7:     for $i_2 = i_1 + 1, \dots, K$ do /* contact force from the other spheres */
8:       $d \leftarrow \|p_{s[i_2]} - p_{s[i_1]}\|$
9:       if $d < 2r$ then /* force for overlapping */
10:        $F \leftarrow \frac{K_{sp}(2r-d)}{d}(p_{s[i_2]} - p_{s[i_1]})$
11:        $f_{s[i_1]} \leftarrow f_{s[i_1]} - F$
12:        $f_{s[i_2]} \leftarrow f_{s[i_2]} + F$
13:        $f_r \leftarrow \max(f_r, K_{sp}(2r-d))$
14:    for $d = 1, 2, \dots, D$ do /* contact force from boundaries */
15:      if $x_{\min[d]} \neq x_{\max[d]}$ then
16:        $f \leftarrow K_{sp}((x_{\min[d]} + F_m r) - p_{s[i_1][d]})$
17:        if $f > 0$ then
18:          $f_{s[i_1][d]} \leftarrow f_{s[i_1][d]} + f$
19:          $f_r = \max(f_r, f)$
20:        $f \leftarrow K_{sp}(p_{s[i_1][d]} - (x_{\max[d]} - F_m r))$
21:        if $f > 0$ then
22:          $f_{s[i_1][d]} \leftarrow f_{s[i_1][d]} - f$
23:          $f_r \leftarrow \max(f_r, f)$
24:    $f_{s[i_1]} \leftarrow f_{s[i_1]} - D_s v_{s[i_1]}$ /* damper for sphere's movement */
25:  $f_r \leftarrow f_{ri} - f_r$
26:  $f_r \leftarrow f_r - D_r v_r$ /* damper for radius's movement */
27:  Calculate dynamics:
28:  $r \leftarrow r + \delta t v_r$ /* update radius */
29:  $v_r \leftarrow v_r + \delta t \frac{f_r}{m_r}$ /* update radius's velocity */
30:  for $i = 1, 2, \dots, K$ do /* movement of each sphere */
31:    $p_{s[i]} \leftarrow p_{s[i]} \delta t v_{s[i]}$ /* update sphere's position */
32:    $v_{s[i]} \leftarrow v_{s[i]} \delta t \frac{f_{s[i]}}{m_s}$ /* update sphere's velocity */
33:  if $r < 0$ then /* constraint on radius */
34:    $r \leftarrow 0$
35: until converging
36: return $r$, $p_{s[1,\dots,K]}$
```

Appendix B

Dynamics of Maze2D Task

The dynamics of Maze2D environment is calculated as Algorithm 7. Here, $\tilde{\mathrm{u}}_{\mathrm{max}} = 0.03$ denotes the maximum norm of an input. The effect of the *wind* at x is denoted by $wind(x)$, specifically,

$$wind(x) = \begin{cases} 0 & (\|x\| < \rho_{\mathrm{w}1}) \\ \frac{x}{\|x\|}\mathrm{w}_1 & (\rho_{\mathrm{w}1} \leqslant \|x\| < \rho_{\mathrm{w}2}) \\ \frac{x}{\|x\|}\mathrm{w}_2 & (\rho_{\mathrm{w}2} \leqslant \|x\|) \end{cases} \tag{B.1}$$

where $\mathrm{w}_1 = 0.01$, $\mathrm{w}_2 = 0.08$, $\rho_{\mathrm{w}1} = 0.5$, $\rho_{\mathrm{w}2} = 1.0$. A wall is denoted by $wall \in Wall$ whose elements are the start point $wall.p_1$ and the end point $wall.p_2$. Figure B.1 illustrates the dynamics of a wall.

Algorithm 7: Dynamics of Maze2D

```
Input : current state $x$, control input $u$
Output : next state $x'$
1: if $\|u\| > \tilde{u}_{\max}$ then $u \leftarrow \frac{u}{\|u\|}\tilde{u}_{\max}$
2: Apply wind: $\Delta x \leftarrow u + wind(x)$
3: Apply walls:
4: if for a $wall \in Wall$, the line segment $(wall.p_1, wall.p_2)$ and the line segment
   $(x, x{+}\Delta x)$ are crossing then
5:     $u_{wall} \leftarrow \frac{wall.p_1 - wall.p_2}{\|wall.p_1 - wall.p_2\|}$
6:     $\Delta x \leftarrow (u_{wall}^{\top}\Delta x)u_{wall}$
7:     if for the other $wall' \in Wall \backslash \{wall\}$, the line segment $(wall'.p_1, wall'.p_2)$ and the
       line segment $(x, x{+}\Delta x)$ are also crossing then $\Delta x \leftarrow 0$
8: return next state: $x' \leftarrow x + \Delta x$
```

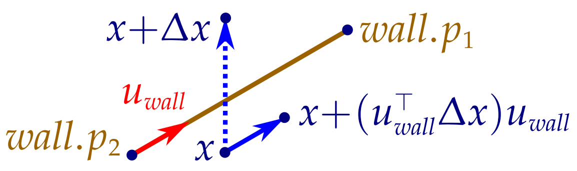


Figure B.1 Dynamics of a wall. If an action, i.e. a line segment $(x, x{+}\Delta x)$, and a wall are crossing, the action is modified to move along the wall.

Bibliography

Ando, N., T. Suehiro, and T. Kotoku (2008). A software platform for component based rt-system development: OpenRTM-Aist. *Simulation, Modeling, and Programming for Autonomous Robots 5325*, 87–98.

Baird, L. C. and A. H. Klopf (1993). Reinforcement learning with high-dimensional, continuous actions. Technical Report WL-TR-93-1147, Wright Laboratory, Wright-Patterson Air Force Base.

Barron, A. (1993, May). Universal approximation bounds for superpositions of a sigmoidal function. *IEEE Transactions on Information Theory 39*(3), 930–945.

Barto, A. G. and S. Mahadevan (2003). Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems 13*(4), 341–379.

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.

Cohen, S., O. Maimon, and E. Khmlenitsky (2006). Reinforcement learning with hierarchical decision-making. In *ISDA '06: Proceedings of the Sixth International Conference on Intelligent Systems Design and Applications*, USA, pp. 177–182. IEEE Computer Society.

Doya, K., K. Samejima, K. Katagiri, and M. Kawato (2002). Multiple model-based reinforcement learning. *Neural Computation 14*(6), 1347–1369.

Ernst, D., P. Geurts, and L. Wehenkel (2003, September). Iteratively extending time horizon reinforcement learning. In N. Lavra, L. Gamberger, and L. Todorovski (Eds.), *Proceedings of the 14th European Conference on Machine Learning*, Dubrovnik, Croatia, pp. 96–107. Springer-Verlag Heidelberg.

Ernst, D., P. Geurts, and L. Wehenkel (2005). Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research 6*, 503–556.

Farahmand, A. M., A. Shademan, M. Jägersand, and C. Szepesvári (2009, May). Model-based and model-free reinforcement learning for visual servoing. In *the IEEE Internactional Conference in Robotics and Automation (ICRA'09)*, Kobe, Japan, pp. 2917–2924.

Fernández, F. and M. Veloso (2006). Probabilistic policy reuse in a reinforcement learning agent. In *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, New York, NY, USA, pp. 720–727. ACM Press.

Gaskett, C., L. Fletcher, and A. Zelinsky (2000). Reinforcement learning for a vision based mobile robot. In *the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'00)*.

Hitomi, K., T. Shibata, Y. Nakamura, and S. Ishii (2006). Reinforcement learning for quasi-passive dynamic walking of an unstable biped robot. *Robotics and Autonomous Systems 54*(12), 982–988.

Hyon, S. (2009). Compliant terrain adaptation for biped humanoids without measuring ground surface and contact forces. *Robotics, IEEE Transactions on 25*(1), 171–178.

Hyon, S., J. Hale, and G. Cheng (2007). Full-body compliant human-humanoid interaction: Balancing in the presence of unknown external forces. *Robotics, IEEE Transactions on 23*(5), 884–898.

Ijspeert, A., J. Nakanishi, and S. Schaal (2002). Learning attractor landscapes for learning motor primitives. In S. Becker, S. Thrun, and K. Obermayer (Eds.), *Advances in Neural Information Processing Systems 15 (NIPS2002)*, pp. 1547–1554.

Kajita, S. and T. Sugihara (2009). Humanoid robots in the future. *Advanced Robotics 23*(11), 1527–1531.

Kawato, M. (2008). From 'understanding the brain by creating the brain' towards manipulative neuroscience. *Phil. Trans. R. Soc. B 363*(1500), 2201–2214.

Kimura, H., T. Yamashita, and S. Kobayashi (2001). Reinforcement learning of walking behavior for a four-legged robot. In *Proceedings of the 40th IEEE Conference on Decision and Control*.

Kirchner, F. (1998). Q-learning of complex behaviours on a six-legged walking machine. *Robotics and Autonomous Systems 25*(3-4), 253–262.

Kober, J. and J. Peters (2009). Learning motor primitives for robotics. In *the IEEE Internactional Conference in Robotics and Automation (ICRA'09)*, pp. 2509–2515.

Kondo, T. and K. Ito (2004). A reinforcement learning with evolutionary state recruitment strategy for autonomous mobile robots control. *Robotics and Autonomous Systems 46*(2), 111–124.

Matsubara, T., J. Morimoto, J. Nakanishi, S. Hyon, J. G. Hale, and G. Cheng (2007). Learning to acquire whole-body humanoid CoM movements to achieve dynamic tasks. In *the IEEE Internactional Conference in Robotics and Automation (ICRA'07)*, pp. 2688–2693.

Matsubara, T., J. Morimoto, J. Nakanishi, M. Sato, and K. Doya (2006). Learning CPG-based biped locomotion with a policy gradient method. *Robotics and Autonomous Systems 54*(11), 911–920.

Mcgovern, A. and A. G. Barto (2001). Automatic discovery of subgoals in reinforcement learning using diverse density. In *In Proceedings of the eighteenth international conference on machine learning*, pp. 361–368. Morgan Kaufmann.

McMahan, H. B. and G. J. Gordon (2005). Generalizing dijkstra's algorithm and gaussian elimination for solving mdps. Technical Report CMU-CS-05-127, Carnegie Mellon University.

Menache, I., S. Mannor, and N. Shimkin (2002). Q-cut - dynamic discovery of sub-goals in reinforcement learning. In *ECML '02: Proceedings of the 13th European Conference on Machine Learning*, London, UK, pp. 295–306. Springer-Verlag.

Miyamoto, H., J. Morimoto, K. Doya, and M. Kawato (2004). Reinforcement learning with via-point representation. *Neural Networks 17*(3), 299–305.

Moore, A. W. and C. G. Atkeson (1995). The parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces. *Mach. Learn. 21*(3), 199–233.

Morimoto, J. and K. Doya (1998). Reinforcement learning of dynamic motor sequence: Learning to stand up. In *the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'98)*, pp. 1721–1726.

Morimoto, J. and K. Doya (31 July 2001). Acquisition of stand-up behavior by a real robot using hierarchical reinforcement learning. *Robotics and Autonomous Systems 36*(1), 37–51.

Morimoto, J., S. Hyon, C. Atkeson, and G. Cheng (2008). Low-dimensional feature extraction for humanoid locomotion using kernel dimension reduction. In *the IEEE Internactional Conference in Robotics and Automation (ICRA'08)*, pp. 2711–2716.

Nakamura, Y., T. Mori, M. Sato, and S. Ishii (2007). Reinforcement learning for a biped robot based on a CPG-actor-critic method. *Neural Networks 20*(6), 723–735.

Park, J.-J., J.-H. Kim, and J.-B. Song (2007). Path planning for a robot manipulator based on probabilistic roadmap and reinforcement learning. *International Journal of Control, Automation, and Systems 5*(6), 674–680.

Peng, J. and R. J. Williams (1994). Incremental multi-step Q-learning. In *International Conference on Machine Learning*, pp. 226–232.

Peters, J., S. Vijayakumar, and S. Schaal (2003). Reinforcement learning for humanoid robotics. In *Humanoids2003, IEEE-RAS International Conference on Humanoid Robots*.

Righetti, L. and A. Ijspeert (2006). Programmable Central Pattern Generators: an application to biped locomotion control. In *the IEEE Internactional Conference in Robotics and Automation (ICRA'06)*, pp. 1585–1590.

Rottmann, A. and W. Burgard (2009). Adaptive autonomous control using online value iteration with gaussian processes. In *the IEEE Internactional Conference in Robotics and Automation (ICRA'09)*, Kobe, Japan, pp. 2106–2111.

Sato, M. and S. Ishii (2000). On-line EM algorithm for the normalized Gaussian network. *Neural Computation 12*(2), 407–432.

Schaul, T., J. Bayer, D. Wierstra, Y. Sun, M. Felder, F. Sehnke, T. Rückstieß, and J. Schmidhuber (2010). Pybrain. *Journal of Machine Learning Research 11*, 743–746.

Singh, S. P. and R. S. Sutton (1996). Reinforcement learning with replacing eligibility traces. *Machine Learning 22*(1-3), 123–158.

Smith, R. (2006). *Open dynamics engine (ODE)*. http://www.ode.org/.

Stolle, M. (2004, February). Automated discovery of options in reinforcement learning. Master's thesis, McGill University.

Sutton, R. and A. Barto (1998). *Reinforcement Learning: An Introduction*. Cambridge, MA: MIT Press.

Sutton, R. S. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *In Proceedings of the Seventh International Conference on Machine Learning*, pp. 216–224. Morgan Kaufmann.

Sutton, R. S., D. Precup, and S. Singh (1999). Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence 112*, 181–211.

Sutton, R. S., C. Szepesvári, A. Geramifard, and M. Bowling (2008). Dyna-style planning with linear function approximation and prioritized sweeping. In *Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence*, pp. 528–536.

Taga, G., R. Takaya, and Y. Konishi (1999). Analysis of general movements of infants towards understanding of developmental principle for motor control. In *Proceedings of IEEE International Conference on Systems, Man, and Cybernetics, 1999 (SMC '99)*, Volume 5, pp. 678–683.

Takahashi, Y. and M. Asada (2003). Multi-layered learning systems for vision-based behavior acquisition of a real mobile robot. In *Proceedings of SICE Annual Conference 2003*, pp. 2937–2942.

Takahashi, Y., K. Noma, and M. Asada (2008). Efficient behavior learning based on state value estimation of self and others. *Advanced Robotics 22*(12), 1379–1395.

Takahashi, Y., M. Takeda, and M. Asada (1999). Continuous valued q-learning for vision-guided behavior acquisition. In *the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'99)*.

Tanner, B. and A. White (2009). Rl-glue: Language-independent software for reinforcement-learning experiments. *Journal of Machine Learning Research 10*, 2133–2136.

Tedrake, R., T. Zhang, and H. Seung (2004). Stochastic policy gradient reinforcement learning on a simple 3d biped. In *the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'04)*, Volume 3, pp. 2849–2854.

Tham, C. K. and R. W. Prager (1994). A modular q-learning architecture for manipulator task decomposition. In *the Eleventh International Conference on Machine Learning*, pp. 309–317.

Torrey, L. and J. Shavlik (2009). Transfer learning. In E. Soria, J. Martin, R. Magdalena, M. Martinez, and A. Serrano (Eds.), *Handbook of Research on Machine Learning Applications*, Chapter 11. IGI Global.

Tsitsiklis, J. N. and B. V. Roy (1996). Feature-based methods for large scale dynamic programming. *Machine Learning 22*, 59–94.

Tsitsiklis, J. N. and B. V. Roy (1997). An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control 42*(5), 674–690.

Uchibe, E. and K. Doya (2004). Competitive-cooperative-concurrent reinforcement learning with importance sampling. In *In Proc. of International Conference on Simulation of Adaptive Behavior: From Animals and Animats*, pp. 287–296.

Wolpert, D. M. and M. Kawato (1998). Multiple paired forward and inverse models for motor control. *Neural Networks 11*(7-8), 1317–1329.

Zhang, J. and B. Rössler (2004). Self-valuing learning and generalization with application in visually guided grasping of complex objects. *Robotics and Autonomous Systems 47*(2-3), 117–127.