Thesis for the Degree of Doctor of Philosophy

DEEP HIERARCHICAL REINFORCEMENT LEARNING ALGORITHMS IN PARTIALLY OBSERVABLE MARKOV DECISION PROCESSES

Le Pham Tuyen

Department of Computer Science and Engineering
Graduate School
Kyung Hee University
Republic of Korea

November 2018

DEEP HIERARCHICAL REINFORCEMENT LEARNING ALGORITHMS IN PARTIALLY OBSERVABLE MARKOV DECISION PROCESSES

Le Pham Tuyen

Department of Computer Science and Engineering
Graduate School
Kyung Hee University
Republic of Korea

November 2018

Abstract

Reinforcement **L**earning (RL) is a branch of machine learning and is a powerful approach to teach a machine that behaves like an expert. The machine under RL improves its policy (a.k.a controller) through a sequence of tries and errors. It means it can learn from both successes and failures. As a result, RL is able to apply in a wide range of applications such as robotics, networking control, resource management, process control, and traffic control. In recent years, RL has achieved remarkable success due to the growing adoption of deep learning techniques and the rapid growth of computing power. The ability of AI learned by RL has surpassed the human brain in some domains such as computer games, card games, and board games, and is expected to reach human intelligence in the near future. However, in order to achieve the expectation, RL needs to tackle several challenges in advance.

The first challenge studied in this dissertation is reported on flat RL algorithms such as DQN, DRQN and Actor Critic. These algorithms often have trouble learning and are even data-efficient with respect to domains having hierarchical structures, e.g. those consisting of multiple subdomains. **H**ierarchical **R**einforcement **L**earning (HRL) is a principled approach that can tackle such challenging tasks. Basically, HRL splits a domain into a hierarchy of subdomains and let an agent (e.g. a learned agent) solve subdomains instead of the domain. As a result, the framework of a HRL algorithm is modified to be able to suitable with the hierarchy of subdomains.

In contrast, the second challenge relies on the fact that many real-world domains usually have only partial observability in which state measurements are often imperfect and partially observable. For example, a robot only knows a part of the environment due to the limitation of its sensors or an autonomous car uses a limited information from sensors to take the actions (e.g. brake, accelerate, turn left, turn right). Taking an action under partial observability is a challenge of a RL

problem.

In this dissertation, we propose **h**ierarchical **D**eep **R**ecurrent **Q**-learning algorithms (**hDRQNs**) in order to handle hierarchical domains in either full observation or partial observation. Particularly, we develop hDRQNv1 algorithm which learns a framework of hierarchical controllers to adapt with hierarchical domains. The framework is constructed by two policies of hierarchy: meta-controller and sub-controller. Meta-controller is an upper policy which plans the agent to complete a sequence of subdomains. Meanwhile, sub-controller performs primitive actions to let the agent complete a subdomain. The policies are formed by deep neural networks such as convolutional neural networks and multilayer perception, which are expected to represent highly complex problems. In addition, the policies integrated recurrent neural layers are expected to overcome the challenges under partial observability. hDRQNv2 is another proposed algorithm which is an improvement of hDRQNv1. It changes the way RNNs integrated into the framework, thus, is expected to have better performance.

Both algorithms are evaluated using various challenging hierarchical POMDPs such as multiple goals in gridworld, multiple goals in four-rooms and Montezuma's Revenge. Especially, Montezuma's Revenge is a hard game in Atari 2600 which is reported a score of zero when learning using DQN. It is hard due to the hierarchy of subdomains in the game. To the best of our knowledge, this research is the first to study Montezuma's Revenge under partial observability. We compare our proposed algorithms with some baseline algorithms such as flat RL algorithms (DQN, DRQN) and a HRL algorithm (hDQN). The comparison is performed based on some metrics such as average reward, number of steps and success ratio. In addition, hDRQNs algorithms are further investigated under various parameter configurations to validate its performance in terms of reward and number of steps. From the experimental results, the proposed algorithms are capable to learn in hierarchical domains under partial observability. Under partial observability, the proposed algorithms outperform several existing state-of-the-art algorithms including flat RL algorithm and HRL algorithm.

Acknowledgment

First and foremost, I gratefully acknowledge Professor TaeChoong Chung, my supervisor, for his warm-hearted encouragement, guidance, and support during my study life at Kyung Hee University. He is not only a teacher providing useful advices to improve my skills in research, but as a father who always stands by his son, protects his son, supports his son with all his heart.

I would like to thank Professor Ngo Anh Vien, my advisor, who has instructed me to pursue the area of Reinforcement Learning. Without his instruction, I was fallen in the mesh of researches and was hard to find a good topic to focus on.

I am grateful to my dissertation committee: Professor 채옥삼, Professor 김동한, Professor 배성호, and Professor 안치득, for their useful criticisms, comments, suggestions during the period of the dissertation defense. They help me a lot to enrich the quality of my dissertation.

I would like to thank all of my current and former members in Artificial Intelligence Lab, for their help, cooperation, and friendship over the past years. Especially, SeungYoon Choi who helped me get through a lot of thing in the department.

I would like to thank my friends, Le Xuan Hung, Huynh Thi Ngoc Duyen, Nguyen Van Nhan, Le Thanh Dat, Dinh Dong Luong, Ho Thien Luan, Nguyen Huu Nhat Minh, Pham Xuan Thanh, and all the Vietnamese friends at Global Campus of Kyung Hee University. I passed a lot of pleasant moments with them during my study here.

I am extremely grateful and express deep gratitude to my parents, my sister, and my brother-in-law. My dissertation will never be completed without the constant support, encouragement, and unconditional love from them.

Finally, I lovingly thank my beloved and reliable partner, Pham Ngoc Diep, for her love, patient and support.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Overview

Reinforcement **L**earning (RL) [3] is an area of machine learning concerned with how software agents take actions in an environment so as to maximize cumulative reward [4]. The action selection is modeled as a map called policy (a.k.a controller) which is improved through a sequence of tries and errors.

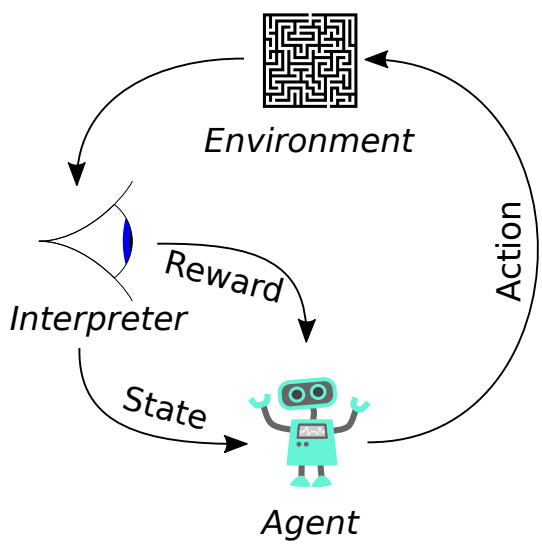


Figure 1.1: A reinforcement learning problem. An agent interacts with an environment by taking an action. The environment transits to next state. The agent observes the state and a reward from the environment

RL, supervised learning, and unsupervised learning are subclasses of machine learning under the technical perspective. Particularly, supervised learning provides a full set of labeled dataset for training whose each sample is tagged with the answer. Meanwhile, unsupervised learning provides

a dataset without answers tagged. Looking for the answers is the role of unsupervised learning. RL is a hybrid approach in which we do not provide the dataset of correct samples, but we provide a method to quantify its performance in the form of a reward signal. From a practical perspective, RL is one of the four major areas of machine learning which includes regression, classification, clustering, and reinforcement learning. Regression algorithms can predict results related to quantity, amount such as the temperature tomorrow, the energy cost next week or the number of new users will visit next month. Classification algorithms focus on predicting which category the data should belong to. For example, identifying an email is a spam email or a normal email, identifying a dog appear in an image or not, or categorizing news stories as finance, weather, entertainment, sports... Clustering algorithms try to group data based on their similarity on some properties. For example, grouping customers who have the same favorite product, grouping documents which have the same topic or grouping visitors who like the same movie. Meanwhile, reinforcement learning algorithms focus on learning an agent so that it can take a sequence of appropriate actions in an environment to maximize cumulative reward. For example, should an autonomous car break or accelerate when the pedestrian is ahead? Or should a home agent rise or lower the temperature of a room when the human is inside? Due to its characteristic, RL is able to apply in a wide range of applications. Figure 1.2 shows many application areas published on IEEE magazines [1], such as robotics [5] [6] [7] [8], networking [9] [10] [11], resource management [12] [13], and finance [14] [15] [16].

Almost RL domains (a.k.a problems, tasks) are formalized as **Markov decision process** (MDP) which is a sequence of states, actions, next states, transition functions, and rewards. However, MDP is not enough power to represent the domains which consist of multiple subdomains. As a result, **hierarchical reinforcement learning** (HRL) [17] was proposed to tackle this problem. HRL relies on a theory of **semi-Markov decision process** (SMDP) which is an extension of MDP to deal with hierarchical domains. HRL decomposes a RL domain into a hierarchy of subdomains, each of which can itself be a RL domain. Identical subdomains can be gathered into one group and are controlled by the same policy. As a result, hierarchical decomposition represents the domain in a compact form and reduces the computational complexity. Various approaches to decompose the RL problem have been proposed such as options [17], HAMs [18] [19], MAXQ [20], Bayesian

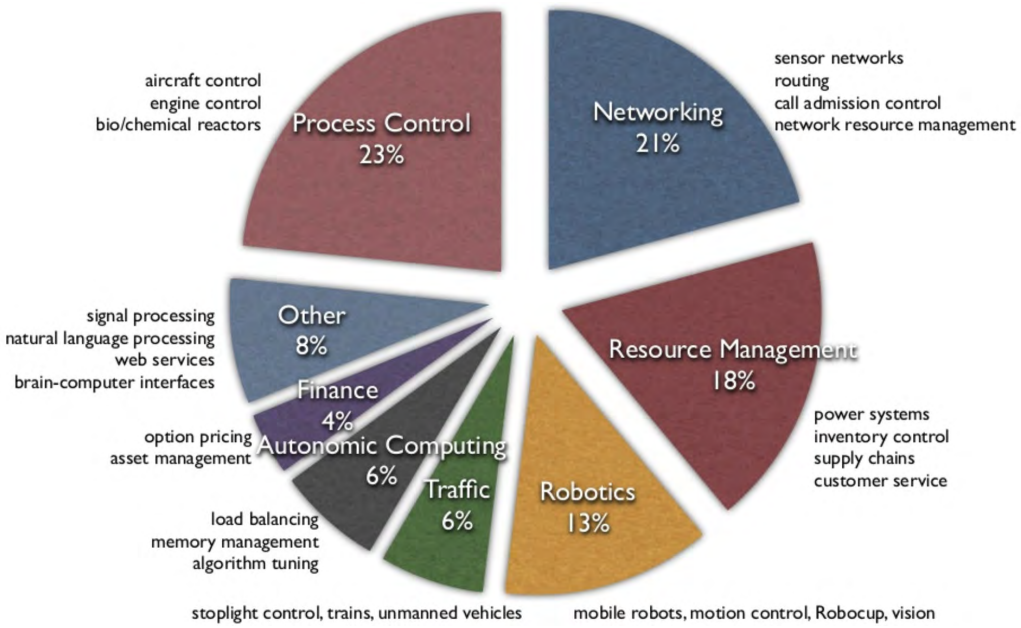


Figure 1.2: Application areas of reinforcement learning [1]

HRL [21–23] and some other advanced techniques [24].

In another aspect, most of the studies rely on an assumption of full observability, where a learning agent can observe the environment states fully. In other words, the environment is represented as a MDP. This assumption does not reflect the nature of real-world applications in which the agent only observes partial information of the environment states. Therefore, the environment, in this case, is represented as a **partial observation Markov decision process** (POMDP). In order for the agent to learn in such a POMDP environment, more advanced techniques are required to obtain good prediction over environment responses such as maintaining a belief distribution over unobservable states; or alternatively using a recurrent neural network (RNN) [25] to summarize an observable history.

1.2 Motivation

In recent years, RL has achieved remarkable success due to the growing adoption of **deep learning** techniques and the rapid growth of computing power. The ability of AI learned by RL has surpassed the human brain in some domains such as computer games [26] [27] [28], card games [29]

and board games [30] [31], and is expected to reach human intelligence in the near future.

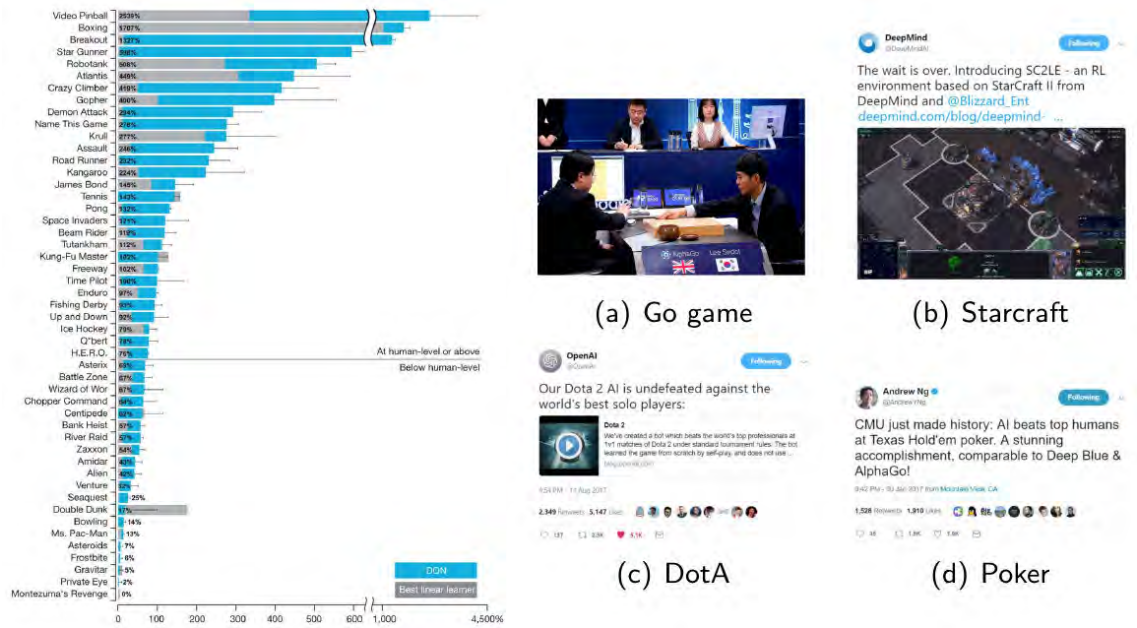


Figure 1.3: Applications of deep reinforcement learning

In deep reinforcement learning, deep learning techniques significantly affect due to their big advantages including powerful data representation, efficiently parameters updating, and stable learning.

Powerful data representation Before the era of deep learning, the RL uses shallow networks such as tabular, linear function or radial basis function, to store the Q values or represent the policy [32]. The shallow networks contained several or a hundred parameters, are not enough power to represent highly non-linear domains. After the deep learning comes, it makes a big change not only for reinforcement learning area but also for all areas of machine learning. A deep neural network consists of multiple hidden layers such as convolutional layer (CONV), fully connected layer (FC) and rectified linear unit layer (ReLU). Each layer connects with other layers to form a complex neural network of a million parameters. The complexity of deep neural networks can represent highly nonlinear domains.

Stable learning Before DQN was proposed, using a nonlinear function approximator to represent Q values such as neural network, is unstable and diverge. This instability has two causes: the correlations present in the sequence of observations and the correlations between the Q values

and the target Q values. Two ideas from DQN was proposed to make the learning process stable. First, DQN uses a biologically inspired mechanism termed *experience replay* that randomizes over the data, thereby removing correlations in the observation sequence and smoothing over changes in the data distribution. Second, DQN used an iterative update that adjusts the Q values towards target Q values that are only periodically updated, thereby reducing correlations with the target. In the practical implementation, target values are generated by a separate neural network, called target Q network, which is a copy of the Q network. The target Q network is periodically updated and is decoupled with the updating of Q network.

Efficiently parameters updating The parameters of neural networks can be updated using a batch of data samples which expected to reduce the variance in the parameter updates. In addition, most of the deep learning frameworks allow employing many optimization algorithms such as AdaGrad [33], Adam [34] and RMSProp [35], which give us many choices to train the networks.

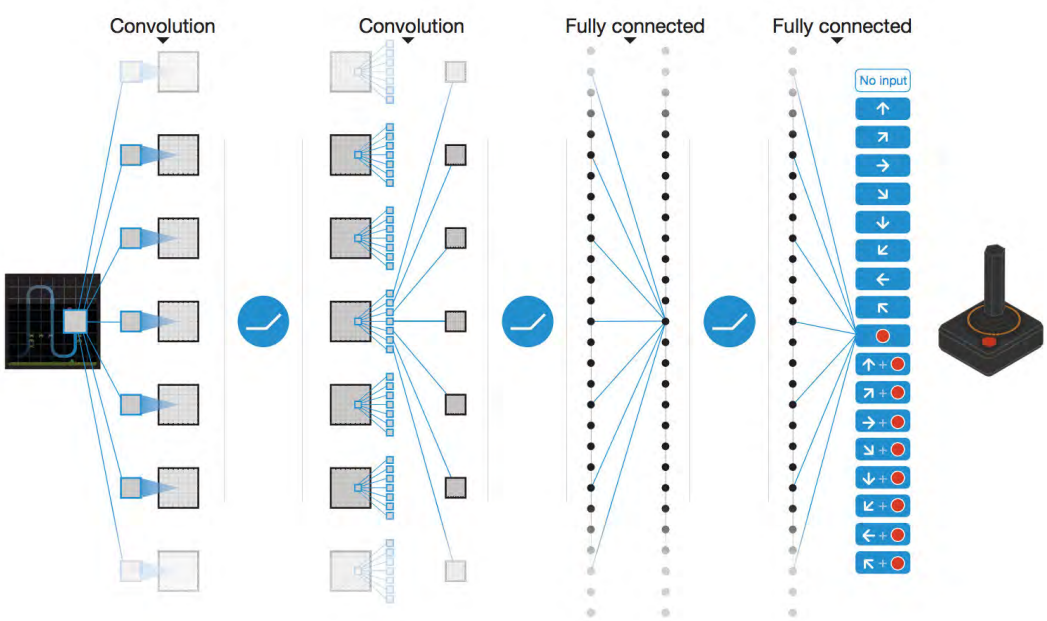


Figure 1.4: Deep Q Network (DQN) is a combination of convolutional layers, fully connected layers, ReLU layers,...

Traditional HRL can also combine with deep neural networks in different ways to improve the performance on hierarchical tasks [36] [37] [38] [39] [40] [41] [42]. Bacon et al. [37] proposed an option-critic architecture, which has a fixed number of intra-options, each followed by a "deep"

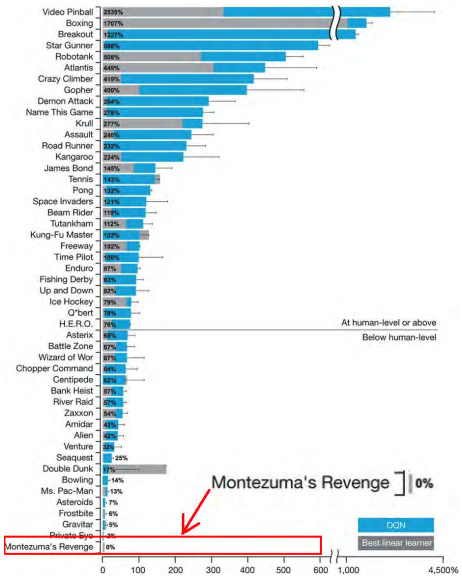
policy. At each time step, only one option is activated and is selected by another policy, which is called "policy over options". DeepMind lab [40] also proposed a deep hierarchical framework inspired by a classical framework called feudal reinforcement learning [43]. Similarly, Kulkarni et al. [44] proposed a deep hierarchical framework of two levels in which the high-level controller produces a subgoal and the low-level controller performs primitive actions to obtain the subgoal. This framework is useful to solve a problem with multiple subgoals such as Montezuma's Revenge [26] and games in Mazebase [45]. Other studies have tried to tackle more challenging problems in HRL such as discovering subgoals [46] and adaptively finding a number of options [47].

1.3 Problem Statement

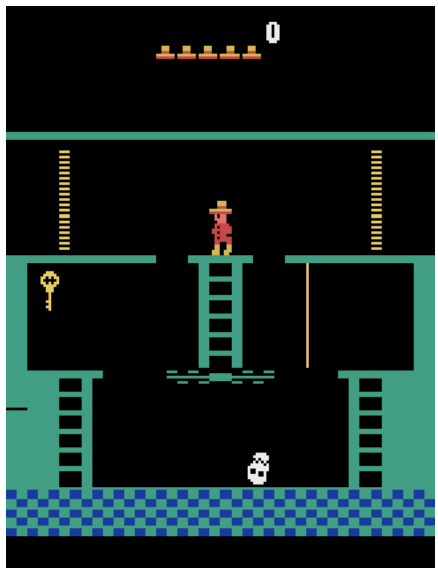
Although state-of-the-art deep reinforcement learning algorithms achieve remarkable success in the fields of computer games, robotics, . . . RL needs to tackle some challenges which are addressed as follows:

- **Challenges in solving domains having hierarchical structures** The first challenge studied in this dissertation is reported on flat RL algorithms such as DQN, DRQN and Actor Critic. These algorithms often have trouble learning and are even data-efficient with respect to domains having hierarchical structures, e.g. those consisting of multiple subdomains. Figure 1.5 shows the performance of DQN on a series of Atari 2600. On this game series, DQN reports a score of zero on the Montezuma's Revenge game which is recognized as a domain having hierarchical structures. The first screen of this game is shown in Figure 1.5b. The agent in the middle of the screen needs to get the key (in the left of the screen) to open the doors (left door or right door). The game only has score if it can pass the doors or can get the key. However, in order to reach the key, the agent needs to do a lot of steps which lead to a long and reward-delayed episode.

- **Challenges when solving domains under partial observability** The second challenge relies on the fact that many real-world domains usually have only partial observability in which state measurements are often imperfect and partially observable. For example, a robot only knows a part of the environment due to the limitation of its sensors or an autonomous



(a) DQN reports a score of zero on Montezuma's Revenge



(b) Montezuma's Revenge in ATARI 2600

Figure 1.5: Flat DRL algorithms fail to solve Montezuma's Revenge which is a hierarchical domain

car uses a limited information from sensors to take the actions (e.g. brake, accelerate, turn left, turn right).

In a summarized sentence, the challenges under our investigation are: "how to solve domains having hierarchical structures under the partial observability and how to strengthen proposed frameworks by utilizing the power of state-of-the-art deep learning methods".

1.4 Proposed Concept

In this dissertation, we want to propose deep HRL algorithms for solving hierarchical tasks under partial observability. The concept of the proposed algorithms is shown in Figure 1.7. In this concept, a planner will plan the agent to complete subdomains until the whole domain is completed. In order to predict the subgoal, the planner only receives a part of an environment (around the agent) instead of a whole environment. After receiving a part of the environment, the planer predicts a subgoal which the agent should obtain to complete the subdomain. After

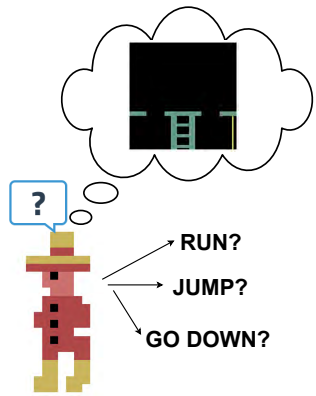


Figure 1.6: The agent is hard to take an action under partial observability

selecting the subgoal, the agent controlled by a RL policy will interact with the environment by receiving an observation, receiving a reward value, returning an action, and transiting to next state. The controller is executed until the subgoal is obtained (E.g. the subdomain is completed). Then, the agent is controlled by the planner and is planned to complete another subdomain. A sequence of operations is repeated until the agent can complete the whole domain. In this proposed concept, we want to develop the controllers utilizing the power of deep neural networks. Especially, the controllers can give the predictions from incomplete states rather than full states. In summary, our proposed algorithms include three steps: (1) the planner select the subdomain which the agent should complete first, (2) the agent follows the policy to complete a subdomain and (3) the agent repeats (1) and (2) until completing the whole domain.

1.5 Key Contributions

In order to solve domains having hierarchical structures, we utilize a hierarchical reinforcement learning algorithm called hierarchical deep Q-learning algorithm [44]. This algorithm considers a domain is a hierarchy of subdomains and let an agent (e.g. a learned agent) solve subdomains instead of the domain. The algorithm learns a framework consisting of two controllers: a meta-controller and a controller. The meta-controller selects a subgoal equivalent to a subdomain and the controller performs primitive actions to obtain the subgoals. Both controllers are built based on DQN networks whose input is four contiguous full images. To deal with the partial observability, we propose frameworks which controllers are developed based on DRQNs (deep recurrent

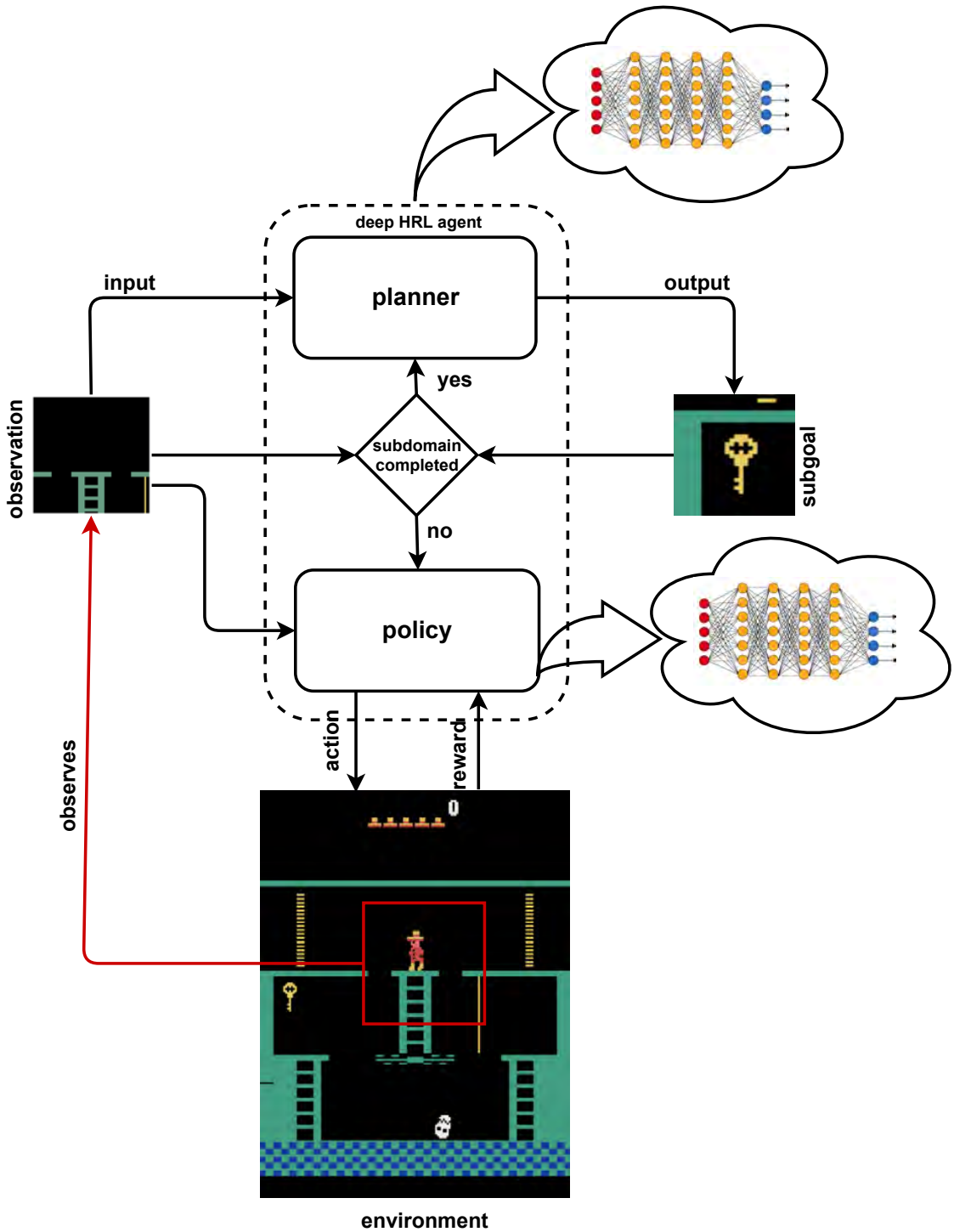


Figure 1.7: The concept of our proposed algorithms which use to solve hierarchical domain under partial observability

Q networks) instead of DQN. The RNNs in DRQNs have ability to construct internal state from incomplete states, thus, is expected to work in domains of partial observability.

Particularly, we develop a deep HRL algorithm for POMDP problems inspired by the deep HRL framework [44]. The agent in this framework makes decisions through a hierarchical policy of two levels. The upper policy determines a subgoal to be achieved, while the lower policy performs primitive actions to achieve the selected subgoal. To learn in POMDP, we represent all policies using RNNs. The RNN of lower-level policies constructs an internal state to remember the whole states observed during the course of interaction with the environment. The upper policy is a RNN to encode a sequence of observations during the execution of the selected subgoal.

We highlight our contributions as follows. We develop **h**ierarchical **D**eep **R**ecurrent **Q**-learning algorithms (**hDRQNs**) in order to handle hierarchical domains in either full observability or partial observability. Particularly, we first develop hDRQNv1 algorithm which learns a framework of hierarchical controllers to adapt with hierarchical domains. The framework is constructed by two policies of hierarchy: meta-controller and sub-controller. Meta-controller is the upper policy which plans the agent to complete a sequence of subdomains. Meanwhile, sub-controller is the lower controller which performs primitive actions to let the agent complete subdomains. In addition, the policies are formed by deep neural networks such as convolutional neural networks and multilayer perceptron, which are expected to represent highly nonlinear domains. Especially, the policies integrated recurrent neural layers are expected to overcome the challenges under partial observability. Second, hDRQNv2 is another proposed algorithm which is an improvement of hDRQNv1. It changes the way RNNs integrated into the framework, thus, is expected to have better performance. To the best of our knowledge, this dissertation is the first study which learns Montezuma's Revenge under partial observability.

In order to demonstrate the efficiency of our proposed algorithms, we compare them with some baseline algorithms such as flat algorithms (DQN, DRQN) and HRL algorithm (hDQN). Both algorithms are evaluated using three challenging hierarchical domains: multiple goals in gridworld, multiple goals in four-rooms and Montezuma's Revenge. Both domains are tested under full observability and partial observability as well. Especially, Montezuma's Revenge is a hard game in Atari 2600 which is reported a score of zero when learning using DQN. It is hard due

to the hierarchy of subdomains in the game. The comparison is performed based on some metrics such as average reward, the number of steps and success ratio. In addition, hDRQNs algorithms are further investigated under various parameter configurations to validate its performance in terms of reward and number of steps. From the experimental results, the proposed algorithms are capable to learn in hierarchical domains under partial observability. Under partial observability, the proposed algorithms outperform existing state-of-the-art algorithms including flat RL algorithms and HRL algorithms.

1.6 Dissertation Organization

The dissertation is organized into chapters as follows:

- **Chapter 1: Introduction.** Chapter 1 provides a brief introduction to the context of our proposed algorithms. The chapter further describes several challenges in the area of reinforcement learning and the limitations of current algorithms. Finally, we briefly address our proposed algorithms and highlight the contributions at the end of the chapter.

- **Chapter 2: Background and Related Works.** Chapter 2 provides the background theories behind our proposed algorithms. They include the theory about Markov decision process, partially observable Markov decision process, semi-Markov decision process, and deep learning techniques. In addition, we also briefly introduce some remarkable studies of hierarchical deep reinforcement learning algorithm and reinforcement learning algorithm under partial observability.

- **Chapter 3: Proposed Methodology.** Chapter 3 presents our proposed algorithms called hierarchical deep recurrent Q learning (hDRQNs). The proposed algorithm can solve domains having hierarchical structures under partial observability. We explain the algorithms from the overall of proposed frameworks to the detail of components. Finally, we summarize the proposed algorithms in form of pseudo-code at the end of the chapter.

- **Chapter 4: Experimental Results and Discussions.** Chapter 4 describes domains which are used to evaluate our algorithms, provides implementation details of our proposed algo-

rithms: parameters, programming language. The performance of algorithms is compared to state-of-the-art algorithms to prove the efficiency of hDRQNs algorithms.

- **Chapter 5: Conclusion and Future Works.** Chapter 5 concludes the dissertation with some discussions of limitations and also provides future works which can extend from current works.

Chapter 2

Background and Related Works

In this section, we briefly review all underlying theories that the content of this dissertation relies on. We first present the theories behind reinforcement learning and its extensions to deal with hierarchical domains and partially observed domains. Second, we briefly review the techniques of deep neural networks. Finally, some state-of-the-art deep reinforcement learning algorithms are introduced.

2.1 Reinforcement Learning Problem

Reinforcement Learning (RL) as described in [3], is a learning framework where an agent interacts with the environment through a sequence of trials and errors. In contrast with other machine learning methods, the agent is not specified the proper actions to take. Instead, the agent explores the environment to achieve the maximum amount of future rewards. Therefore, this approach slightly differs from supervised learning and unsupervised learning. Generally, RL is concerned with choosing a sequence of actions that maximizes cumulated discounted reward. Thus, it is similar to a Markov Decision Process, or MDP. In this dissertation, we focus on MDPs in which time is modeled as a sequence of discrete units called *time steps*. The introduction of MDP is defined as follows:

2.1.1 Markov Decision Process (MDP)

A discrete MDP models a sequence of decisions of a learning agent interacting with an environment at some discrete time scale, $t = 1, 2, \ldots$ Formally, an MDP consists of a tuple of five elements $\{\mathcal{S}, \mathcal{A}, \mathcal{P}, r, \gamma\}$, where $\mathcal{S}$ is a discrete state space, $\mathcal{A}$ is a discrete action space, $\mathcal{P}(s_{t+1}, s_t, a_t) = p(s_{t+1}|s_t, a_t)$ is a transition function that measures the probability of obtaining

a next state s_{t+1} given a current state-action pair (s_t, a_t), $r\,(s_t, a_t)$ defines an immediate reward achieved at each state-action pair, and $\gamma \in (0, 1)$ denotes a discount factor. MDP relies on the Markov property that the next state only depends on the current state-action pair:

$$p(s_{t+1}|\{s_1, a_1, s_2, a_2, ..., s_t, a_t\}) = p(s_{t+1}|s_t, a_t). \tag{2.1}$$

A policy of a RL algorithm is denoted by π, which is the probability of taking an action a given a state s:

$$\pi = \mathcal{P}(a|s), \tag{2.2}$$

and the goal of an RL algorithm is to find an optimal policy π^* in order to maximize the expected discounted reward as follows:

$$J(\pi) = \mathbb{E}\Big[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)\Big]. \tag{2.3}$$

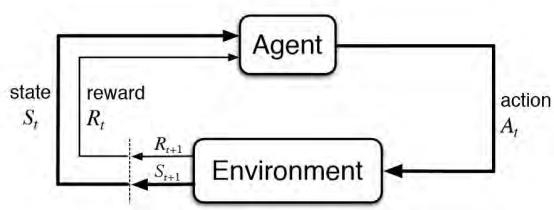


Figure 2.1: A reinforcement learning problem is modeled as a Markov decision process

2.1.2 Reinforcement Learning Approaches

RL algorithms are classified into three approaches: value-based approach, policy-based approach and actor-critic approach [32].

Value-based approach A typical value-based approach tries to obtain an optimal policy by finding optimal value functions (V values) or action value function ($\mathcal{Q}$ values). The value of state s is defined to be the expected future discounted reward which the agent can get if the agent starts in

state s and follows the policy π. Formally, this can be written as

$$\mathcal{V}^{\pi}(s) = \mathbb{E}\Big[\sum_{t=0}^{\infty} \gamma^{t} r(s_t, a_t)\Big]. \tag{2.4}$$

Equation 2.4 can be written in the recursive form which is the expected value of immediate reward and the discounted value of the next state:

$$\mathcal{V}^{\pi}(s) = \mathbb{E}[r(s,a) + \gamma \mathcal{V}^{\pi}(s')]. \tag{2.5}$$

Value function can be used to estimate the goodness of a certain state but we can only use it for action selection if we know the transition function. In contrast to value function, action value function ($\mathcal{Q}$ values) estimate the future discounted reward if the agent chooses action a at state s and follows policy π:

$$\mathcal{Q}^{\pi}(s,a) = \mathbb{E}[r(s,a) + \gamma \mathcal{V}^{\pi}(s')]. \tag{2.6}$$

Note that $\mathbb{E}[\mathcal{Q}^{\pi}(s, \pi(s))] = \mathcal{V}^{\pi}(s)$ so we can also write

$$\mathcal{Q}^{\pi}(s,a) = \mathbb{E}[r(s,a) + \gamma \mathcal{Q}^{\pi}(s', a')], \tag{2.7}$$

where a' is action taken by following policy π. A policy π_1 is better than policy π_2 if $\mathcal{V}^{\pi_1}(s) \geq \mathcal{V}^{\pi_2}(s)$ for all state $s \in \mathcal{S}$. Therefore, an optimal policy π^* will satisfy the condition

$$\mathcal{V}^{\pi^*}(s) \geq \mathcal{V}^{\pi}(s) \tag{2.8}$$

for all states and policy π. Let define $\mathcal{V}^{*}(s) = \mathcal{V}^{\pi^*}(s)$ be optimal value function, $\mathcal{Q}^{*}(s,a) = \max_{\pi} \mathcal{Q}^{\pi}(s,a)$ be optimal action value function. Replace optimal policy into Equation 2.6 we have Bellman optimality equation for $\mathcal{Q}$ values:

$$\mathcal{Q}^{*}(s,a) = \mathbb{E}[r(s,a) + \gamma \max_{a' in \mathcal{A}} \mathcal{Q}^{*}(s', a')]. \tag{2.9}$$

Similarly, we have the Bellman optimality equation for V values as follows:

$$\mathcal{V}^*(s) = \max_a \mathbb{E}[r(s,a) + \gamma \mathcal{V}^*(s')]. \tag{2.10}$$

In order to approximate V value function and $\mathcal{Q}$ value function, we can use some methods such as TD-learning, Q-learning, and SARSA [3]. TD-learning method calculate the error of the Bellman equation for one step sample $< s_t, a_t, r_t, s_{t+1} >$. At iteration k, V value has formula:

$$\mathcal{V}_{k+1}(s_t) = (1-\alpha)\mathcal{V}_k(s_t) + \alpha(r(s_t,a_t) + \gamma \mathcal{V}_k(s_{t+1}))], \tag{2.11}$$

where α is learning rate. The temporal difference (TD) error is calculated by

$$TD = r(s_t,a_t) + \gamma \mathcal{V}(s_{t+1}) - \mathcal{V}(s_t), \tag{2.12}$$

and the V value is updated along the TD error as follows:

$$\Delta \mathcal{V}(s_t) = \alpha \times TD \tag{2.13}$$

Similarly, The TD error of $\mathcal{Q}$ values is calculated by

$$TD = r(s_t,a_t) + \gamma \mathcal{Q}^\pi(s_{t+1},a_{t+1}) - \mathcal{Q}^\pi(s_t,a_t) \tag{2.14}$$

Action a' in Equation 2.14 is chosen from the policy π. Based on the choice of π, we can classify TD error into two methods: SARSA learning and Q-Learning. In SARSA learning (**S**tate **A**ction **R**eward **S**tate **A**ction learning), the agent always follows the "newest" policy (E.g. a' is chosen based on "newest" policy) and use this policy to update the $\mathcal{Q}$ values

$$TD_{SARSA} = r(s_t,a_t) + \gamma \mathcal{Q}^\pi(s_{t+1},a_{t+1}) - \mathcal{Q}^\pi(s_t,a_t) \tag{2.15}$$

Meanwhile, in Q-Learning, a' is chosen to maximize the $\mathcal{Q}$ value. The action a' can be generated from past policies.

$$TD_{Q-Learning} = r(s_t, a_t) + \gamma \max_{a'} \mathcal{Q}^{\pi}(s_{t+1}, a') - \mathcal{Q}^{\pi}(s_t, a_t) \tag{2.16}$$

Policy-based approach A policy-based approach directly learns a parameterized policy that maximizes the cumulative discounted reward as follows:

$$J(\pi^{\theta}) = \mathbb{E}[R(\xi)] = \int p(\xi|\pi^{\theta}) R(\xi) d\xi, \tag{2.17}$$

where $\xi = \{\xi_1, \xi_2, ..., \xi_t\}$ is a trajectory (also called an episode, history or roll-out), $p(\xi|\pi^{\theta})$ is the probability of generating a trajectory ξ given the policy π^{θ} and θ is parameters of policy π^{θ}. Some methods used to search for optimal parameters of the policy including policy gradient [48] [49] [50], expectation maximization [51] [52] [53], evolutionary algorithm [54] [55], or using a non-parametric policy [56] [57] [58]. Policy gradient updates θ along the gradient direction of the expected return as

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \alpha \nabla_{\boldsymbol{\theta}} J(\pi_k^{\boldsymbol{\theta}}), \tag{2.18}$$

where α denotes a learning rate and k is the current update number. Policy gradient methods guarantee convergence to an optimum (local optimum). The main computational challenge of a policy gradient method is to estimate the gradient from given trajectories such that its variance is small. Expectation maximization (EM) method models RL as Maximum Likelihood Problem. Particularly, logarit of the cumulative discounted reward can be transformed as follows,

$$\begin{aligned} \log(J(\pi^{\theta})) &= \log \int p(\xi|\pi^{\theta}) R(\xi) d\xi \\ &= \int p(\xi|\pi^{\theta'}) \log \frac{p(\xi|\pi^{\theta'}) R(\xi)}{p(\xi|\pi^{\theta'})} d\xi + KL(p(\xi|\pi^{\theta'})||p(\xi|R, \pi^{\theta'})), \end{aligned} \tag{2.19}$$

where $KL(p(\xi|\pi^{\theta'})||p(\xi|R, \pi^{\theta'}))$ is the Kullback-Leibler divergence and non-negative. So, in order to maximize $J(\pi^{\theta})$, first, we need to minimize the KL term (E-step) by choosing a distribution

$p(\xi|\pi^{\theta'}) \propto p(\xi|R, \pi^{\theta'})$. Second, we replace the result of E-step to the equation and maximize the expected complete data log-likelihood (M-step).

Actor-critic approach Actor-critic approach [2] [59] [60] proposed a framework which mixes between value-based approach and policy-based approach. Particularly, the framework is divided into two parts: actor part and critic part. The actor part receives a state from an environment and uses a policy to estimate an action. The critic part receives the state and the reward from the environment to evaluate how good at that state. The policy of actor part is updated during the learning process based on the TD error from the critic part. The actor-critic framework is demonstrated in Figure 2.2.

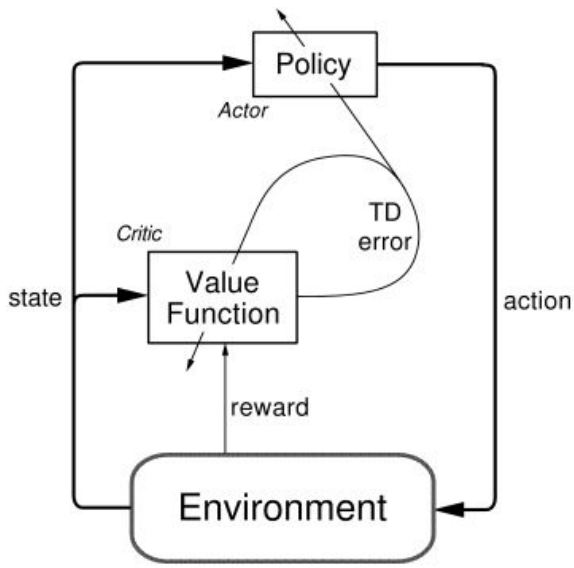


Figure 2.2: Actor-critic framework with two main elements: actor part and critic part [2]

In this dissertation, we introduce proposed algorithms which based on value-based approach. They are variations of Q-learning algorithm under the strength of deep learning techniques.

2.1.3 Semi-Markov Decision Process (SMDP)

Learning hierarchical tasks is one of challenges of reinforcement learning. The semi-Markov decision process (SMDP) [17], which is as an extension of MDP, was developed to deal with this challenge (Figure 2.3).

In this theory, the concept of “options” is introduced as a type of temporal abstraction. An

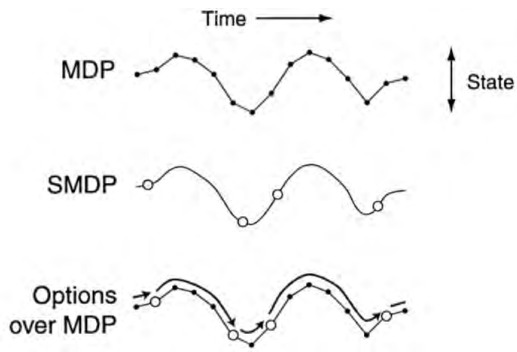


Figure 2.3: SMDP is modeled as a sequence of options over MDP

option $\xi \in \Xi$ is defined by three elements: an option's policy π, a termination condition β, and an initiation set $\mathcal{I} \subseteq \mathcal{S}$ denoted as the set of states in the option. In addition, a *policy over options* $\mu(\xi|s)$ is a policy which is used to select the current option. Figure 2.4 shows an example of SMDP's executions. First, when an option's policy π^ξ is taken, the action a_t is selected based on π^ξ. The environment then transits to state s_{t+1}. The option either terminates or continues according to a termination probability $\beta^\xi(s_{t+1})$. When the option terminates (at state s_{t+3}), the agent selects next option ξ_{t+3} based on $\mu(\xi|s_{t+1})$.

The total discounted reward received by executing option ξ is defined as follows:

$$\mathcal{R}_s^\xi = \mathbb{E}\Big[\sum_{i=t}^{t+k-1} \gamma^{i-t} r^\xi(s_i, a_i)\Big]. \tag{2.20}$$

The multi-time state transition model of option ξ [61], which is initiated in state s and terminates at state s' after k steps, has the formula:

$$\mathcal{P}_{ss'}^\xi = \sum_{k=1}^{\infty} \mathcal{P}^\xi(s', k|s, \xi)\gamma^k. \tag{2.21}$$

where $\mathcal{P}^\xi(s', k|s, \xi)$ is the joint probability of ending up at s' after k steps if taking option ξ at state s. Given this, we can write the Bellman equation for the value function of policy over options

μ as follows:

$$\mathcal{V}^{\mu}(s) = \sum_{\xi \in \Xi} \mu(\xi|s) \left[\mathcal{R}_s^{\xi} + \sum_{s'} \mathcal{P}_{ss'}^{\xi} \mathcal{V}^{\mu}(s') \right] \tag{2.22}$$

and the option-value function:

$$\mathcal{Q}^{\mu}(s, \xi) = \mathcal{R}_s^{\xi} + \sum_{s'} \mathcal{P}_{ss'}^{\xi} \sum_{\xi' \in \Xi} \mu(\xi'|s') \mathcal{Q}^{\mu}(s', \xi'). \tag{2.23}$$

Similarly, the corresponding optimal Bellman equations are as follows:

$$\mathcal{V}^{*}(s) = \max_{\xi \in \Xi} \left[\mathcal{R}_s^{\xi} + \sum_{s'} \mathcal{P}_{ss'}^{\xi} \mathcal{V}^{*}(s') \right] \tag{2.24}$$

$$\mathcal{Q}^{*}(s, \xi) = \mathcal{R}_s^{\xi} + \sum_{s'} \mathcal{P}_{ss'}^{\xi} \max_{\xi' \in \Xi} \mathcal{Q}^{*}(s', \xi'). \tag{2.25}$$

The optimal Bellman equation can be computed using synchronous value iteration (SVI) [17], which iterates the following steps for every state:

$$\mathcal{V}(s) = \max_{\xi \in \Xi} \left[\mathcal{R}_s^{\xi} + \sum_{s'} \mathcal{P}_{ss'}^{\xi} \mathcal{V}(s', \xi') \right]. \tag{2.26}$$

When the option model is unknown, $\mathcal{Q}_t(s, \xi)$ can be estimated using a Q-learning algorithm with an estimation formula:

$$\mathcal{Q}(s, \xi) \leftarrow \mathcal{Q}(s, \xi) + \alpha \left[r(s, \xi(s)) + \gamma^k \max_{\xi' \in \Xi} \mathcal{Q}(s', \xi') - \mathcal{Q}(s, \xi) \right], \tag{2.27}$$

where α denotes the learning rate, k denotes the number of time steps elapsing between s and s' and r denotes an intermediate reward if $\xi(s)$ is a primitive action, otherwise r is the total reward when executing option $\xi(s)$.

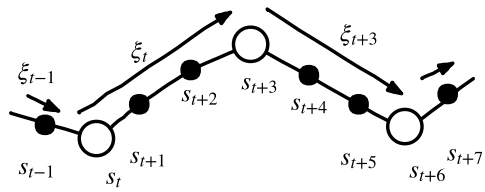


Figure 2.4: An example of SMDP. Black circle illustrates a state, and white circle illustrates the state where the option is terminated and transit to another option

2.1.4 Partially Observable Markov Decision Process (POMDP)

In many real-world tasks, the agent might not have full observability over the environment. In principle, those tasks can be formulated as a POMDP defined as a tuple of six components $\{\mathcal{S}, \mathcal{A}, \mathcal{P}, r, \Omega, \mathcal{Z}\}$, where $\mathcal{S}, \mathcal{A}, \mathcal{P}, r$ are the state space, action space, transition function and reward function, respectively, as in a Markov decision process. Ω and $\mathcal{Z}$ are the observation space and observation model, respectively. POMDPs lack the Markov property, the next state depends not only on the current state and action but it can also depend on the whole history. We can see a POMDP as an MDP with belief states b_t. The belief states define the probability of being in state s according to its history of actions and observations. Given a new action and observation, belief states are updated using the Bayes rule [62] as follows:

$$b'(s') \propto \mathcal{Z}(o|s', a) \sum_{s \in \mathcal{S}} \mathcal{P}(s'|s, a) b(s). \tag{2.28}$$

Intuitively, the belief states maintain a distribution of what the agent believes about the state that agent cannot be observable. However, updating belief states requires a high computational cost and expensive memory [25], thus, it is not applicable. Another approach is using a Q-learning agent with function approximation, which uses the Q-learning algorithm for updating the policy. Because updating the Q-value from an observation can be less accurate than estimating from a complete state, a better way would be that a POMDP agent using Q-learning uses the last k observations as input to the policy. Nevertheless, the problem with using a finite number of observations is that key-event observations far in the past would be neglected in future decisions. For this reason, recurrent neural networks (RNNs) are used to maintain a long-term state as in [63] [64]. In this dissertation, our proposed algorithms using RNNs at different levels of the hierarchical policies is

expected to be advantageous in POMDP environments.

2.2 Deep Neural Networks

Recent advances in deep learning [65] are widely applied to reinforcement learning to form deep reinforcement learning. A few years ago, reinforcement learning still used "shallow" policies such as tabular, linear function, radial basis function, or neural networks with few layers [5]. The shallow policies contain many limitations, especially in representing highly complex behaviors and the computational cost of updating parameters. In contrast, deep neural networks in deep reinforcement learning can extract more information from the raw inputs by pushing them through neural network layers such as convolutional layers (CONVs), fully connected layer (FCs) or recurrent neural layers (RNNs). Multiple layers in DNNs can have a lot of parameters, allowing them to represent highly non-linear problems. Formally, deep neural networks are parametric models consist of multiple stacked layers of artificial neurons. They are a powerful tool for approximating value functions. Basically, a deep neural network computes a nested function of the form:

$$\hat{y} = f_k(W_k \dots f_1(W_1 x + b_1) \dots + b_k). \tag{2.29}$$

Equation 2.29 can be explained as follows: output prediction $\hat{y}$ is produced from input x through k network layers. Each layer transforms its input, first multiplying by a weight matrix W_i, adding on a vector of biases b_i, and finally applying a nonlinear function f_i. The final layer of the network represents output prediction $\hat{y}$. The neural networks are trained to minimize the loss function $L = |\hat{y} - y|$ using backpropagation algorithm [66]. Backpropagation provides gradients $\Delta\theta(L)$ which specify how the parameters of the network should be changed in order to reduce the loss. θ denotes the vector of all the parameters in the model $\theta = (W_1, b_1, W_2, b_2, ..., W_k, b_k)$.

The choice of nonlinear activation function f_i significantly affects the depth and training time of a neural network. Popular activation functions used in deep neural networks are sigmoid function, rectified linear unit (ReLU). The sigmoid activation function has the form: $f(x) = 1/(1 + e^{-x})$ and ReLU has the form: $f(x) = max(0, x)$. Recently, ReLU largely uses in deep neural network due to its efficiency in training time

In addition, advances in deep learning also came in the form of more advanced optimization methods such as stochastic gradient descent (SGD), or adaptive learning rate optimization methods (AdaGrad [33], Adam [34] and RMSProp [35], AdaDelta [67]). Throughout this dissertation, we use the Adam optimizer.

The sections below briefly introduce popular neural networks which are commonly used in deep reinforcement learning such as convolutional neural networks, multilayer perceptron, and recurrent neural networks.

2.2.1 Multilayer Perceptron (MLP)

A multilayer perceptron (MLP) is a classical neural network [68] which consists of three or more layers: an input layer, an output layer, and one or more hidden layers. When MLPs are fully connected, each node in one layer connects with a certain weight W_{ij} to every node in the following layer. Thus, when the number of the node of each layer increase, the number of weights exponentially increased. It is hard to deal with the input like images which have a lot of pixels. In the dissertation, we use some fully-connected layers combined with convolutional layers to better extract features from raw images.

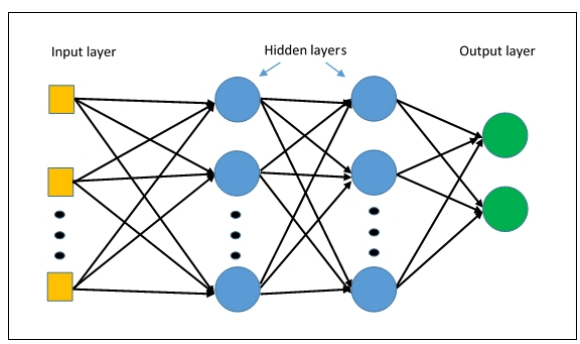


Figure 2.5: A multilayer perceptron example which has two hidden layers

2.2.2 Convolutional Neural Networks (CNN)

Convolutional neural networks (CNNs) [69] were a key innovation for learning from images. At a high level, CNNs are a method of sharing weights across pixel space. Instead of learning a full weight matrix, a CNN instead learns low-dimensional filters which are convolved across the input

pixels. These filters have been shown to learn hierarchical detectors for edges, image parts, and full objects. CNNs have proven highly successful at static image recognition problems such as the MNIST, CIFAR, and ImageNet Large-Scale Visual Recognition Challenge [70] [71] [72]. By using a hierarchy of trainable filters and feature pooling operations, CNNs are capable of automatically learning complex features required for visual object recognition tasks. In this dissertation, we use convolutional layers because the input of evaluated domains is represented by images.

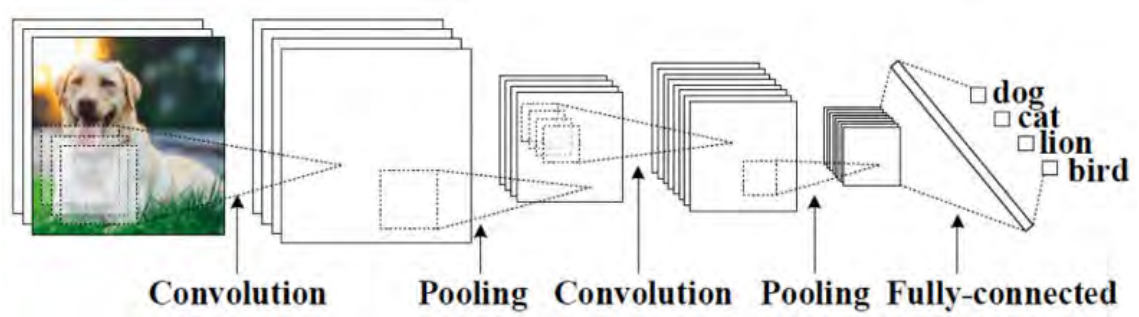


Figure 2.6: A convolutional neural network which efficiently extracts features from images

2.2.3 Recurrent Neural Networks (RNN)

Recurrent neural networks (RNNs) are capable of learning features and long-term dependencies from sequential and time-series data [73]. The structure of hidden states work as the memory of the network and state of the hidden layer at a time is conditioned on its previous state [74]. This structure enables the RNNs to store, remember, and process past complex signals for long time periods. RNNs come in many variants and some well-known RNNs used in real-world applications such as Long Short Term Memory (LSTM) [75] and Gated Recurrent Unit (GRU) [76]. Recently, researchers have used RNNs in reinforcement learning to deal with POMDP domains. RNNs have been successfully applied to domains, such as natural language processing and speech recognition, and are expected to be advantageous in the POMDP domain, which needs to process a sequence of observations rather than a single input. In this dissertation, we utilize RNNs, especially LSTM, to tackle the challenge of domains under partial observability. LSTM (Figure 2.7) has the potential to construct a complete state from a sequence of partial states.

2.3 Deep Reinforcement Learning Algorithms

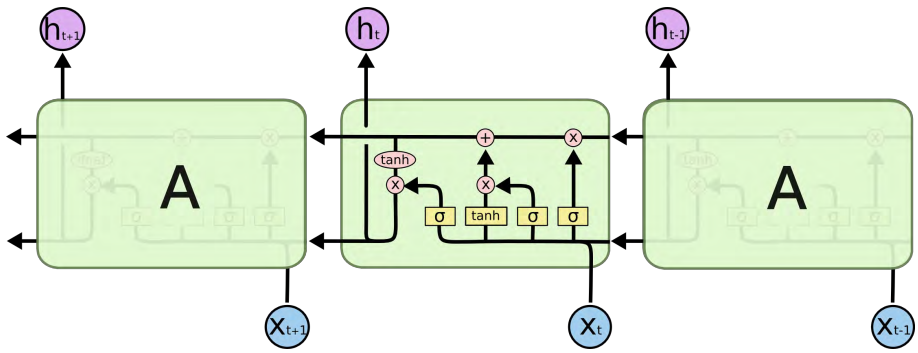


Figure 2.7: Long short term memory architecture

This section presents some DRL algorithms which belong to the value-based approach.

2.3.1 Deep Q-Learning (DQN)

Deep Q-Network (DQN) has been proposed recently by Google Deepmind [27] and has opened a new era of deep reinforcement learning. It has influenced most later studies in deep reinforcement learning. In term of the architecture, a Q-network parameterized by θ, e.g., $\mathcal{Q}(s, a|\theta)$ is built on a convolutional neural network (CNN) that receives an input of four images of size 84×84 and is processed by three hidden CONV layers. The final hidden layer is a MLP layer with 512 rectifier units. The output layer is a FC layer, which the number of outputs equal the number of actions. In term of the learning algorithm, DQN learns Q-value functions iteratively by updating the Q-value estimation via the temporal difference error:

$$\mathcal{Q}(s, a) := \mathcal{Q}(s, a) + \alpha(r + \gamma \max_{a'} \mathcal{Q}(s', a') - \mathcal{Q}(s, a)). \tag{2.30}$$

In addition, the stability of DQN also relies on two mechanisms. The first mechanism is *experience replay* memory, which stores transition data in the form of tuples $\{s, a, s', r\}$. It allows the agent to uniformly sample from and train on previous data (off-policy) in batch, thus reducing the variance of learning updates and breaking the temporal correlation among data samples. The second mechanism is the target Q-network, which is parameterized by θ', e.g., $\hat{\mathcal{Q}}(s, a|\theta')$, and is a

copy of the main Q-network. The target Q-network is used to estimate the loss function as follows:

$$\mathcal{L} = \mathbb{E}\Big[(y^{DQN} - \mathcal{Q}(s, a|\theta))^2\Big], \tag{2.31}$$

where $y^{DQN} = r + \gamma \max_{a'} \hat{\mathcal{Q}}(s', a'|\theta')$. Initially, the parameters of the target Q-network are the same as in the main Q-network. However, during the learning iteration, they are only updated at a specified time step. This update rule causes the target Q-network to decouple from the main Q-network and improves the stability of the learning process.

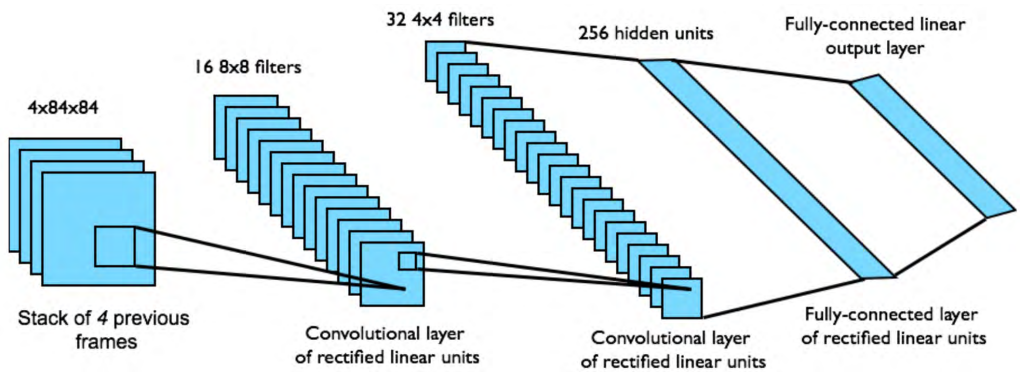


Figure 2.8: Deep Q Network which is applied to solve games in ATARI 2600

Many other models based on DQNs have been developed, such as Double DQN [77], Dueling DQN [78], and Priority Experiment Replay [79]. On the other hand, deep neural networks can be integrated into other methods rather than estimating Q-values, such as representing the policy in policy search algorithms [8], estimating the advantage function [80], or a mixed actor network and critic network [81].

2.3.2 Double DQN (DDQN)

The idea of Double Q-learning [77] is to reduce overestimations by decomposing the max operation in the target into action selection and action evaluation. Particularly, we use Q network to select the best action to take for the next state (E.g. the action with the highest $\mathcal{Q}$ value) and use target Q network to calculate the target $\mathcal{Q}$ value of taking the selected action at the next state. The

formula is the same as DQN, but replacing the target y^{DQN} with $y^{DoubleDQN}$ as follows

$$y^{DoubleDQN} = r + \gamma \hat{\mathcal{Q}}(s', \underset{a}{\operatorname{argmax}}\, \mathcal{Q}(s', a|\theta)|\theta'). \tag{2.32}$$

2.3.3 Dueling network

The dueling architecture [78] consists of two streams that represent the value and advantage functions while sharing a common convolutional feature learning module. The two streams are combined via a special aggregating layer to produce an estimate of the state-action value function Q. This dueling network should be understood as a single Q network with two streams that replaces the popular single-stream Q network in existing algorithms such as DQN. Intuitively, the dueling architecture can learn which states are (or are not) valuable, without having to learn the effect of each action for each state. This is particularly useful in states where its actions do not affect the environment in any relevant way.

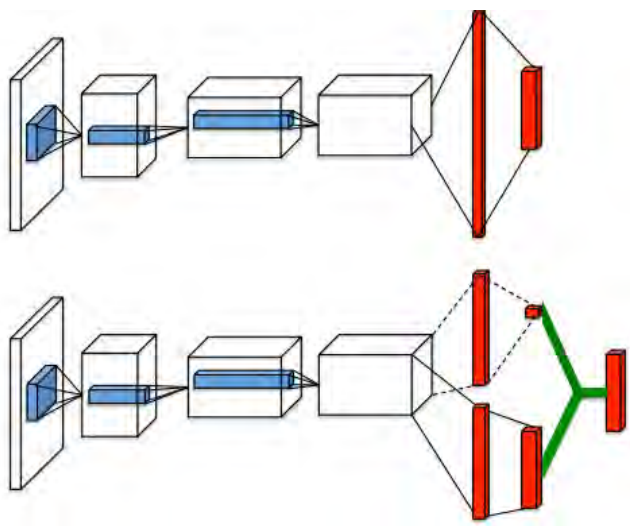


Figure 2.9: Deep Q Network (top) and Dueling network (bottom)

2.3.4 Deep Recurrent Q-Learning (DRQN)

The limitation of DQN is that its performance decreases when observed incomplete input state (POMDP). Peter Stone [63] proposed the deep recurrent Q-network (DRQN) which leverages

advances in recurrent neural networks to better deal with POMDPs. Particularly, DRQN minimally modifies the architecture of DQN, replacing only its first fully connected layer with a recurrent LSTM layer of the same size (Figure 2.10). The architecture of DRQN takes a single 84×84 preprocessed image. This image is processed by three convolutional layers and the outputs are fed to the fully connected LSTM layer. Finally, a linear layer outputs a $\mathcal{Q}$ value for each action. During training, the parameters for both the convolutional and recurrent portions of the network are learned jointly from scratch. In order to update the network contain recurrent layer, we need to use a sequence in trajectories in two strategies: bootstrapped sequential updates and bootstrapped random updates. DRQN uses flickering Atari games to evaluate its performance. The game screens of the flickering games are either fully revealed or fully obscured with probability $p = 0.5$, thus, shows the manner of partial observability. In this dissertation, we use cropped game-screen rather than the full game-screen, to show the manner of partial observability.

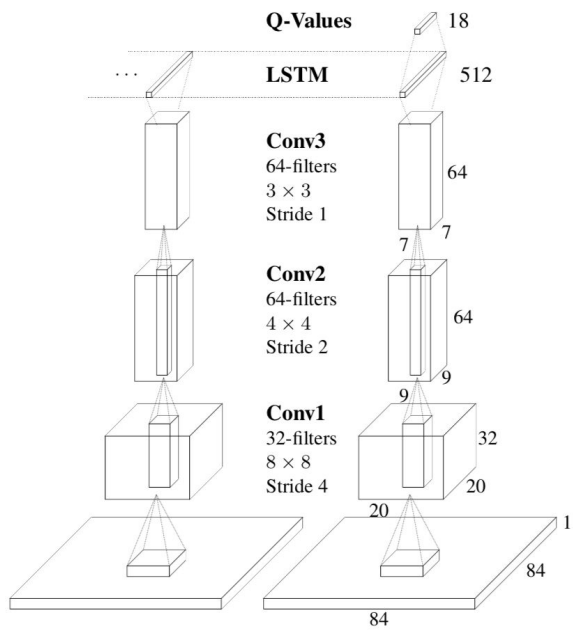


Figure 2.10: Deep recurrent Q network which integrates a long short term memory into deep Q network

2.3.5 Hierarchical Deep Q-Learning (hDQN)

Hierarchical Deep Q-learning [44] is a framework for learning hierarchical domains. The model takes decisions over two levels of hierarchy: (a) the upper level policy called *meta-controller*, takes the state and estimates a new goal, (b) the lower-level policies called *controller*, uses both the state and the chosen goal to select actions either until the goal is reached or the episode is terminated. (c) The meta-controller then chooses another goal and steps (a-b) repeat. The model is trained using stochastic gradient descent at different temporal scales to optimize expected future intrinsic (controller) and extrinsic rewards (meta-controller). The framework uses notions of intrinsic reward and extrinsic reward instead of a classical reward. Intrinsic reward and extrinsic reward are used for evaluating the controller and the meta-controller, respectively. Both controllers are built from deep $\mathcal{Q}$ networks and based on the theory of options. The framework is efficient to solve problems with a long-range delayed reward such as Montezuma's Revenge game. However, it is only deal with domains of full MDP

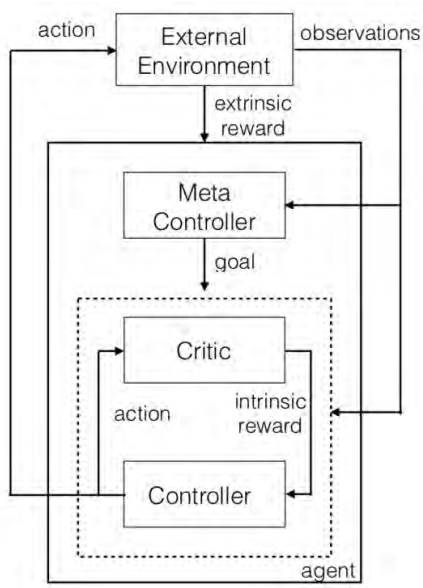


Figure 2.11: Hierarchical deep Q learning framework employs two deep Q networks to solve hierarchical tasks

Chapter 3

Proposed Methodology

In this section, we introduce our proposed framework called hierarchical deep recurrent Q-Learning algorithm (hDRQN). First, we describe the underlying theory of hDRQN called POSMDP. Second, we introduce two versions of our frameworks and explain the method for learning these frameworks. Third, we describe some components of our algorithms such as sampling strategies, subgoal definition, and extrinsic and intrinsic reward functions. We rely on partially observable semi-Markov decision process (POSMDP) settings, where the agent follows a hierarchical policy to solve the POMDP domains.

3.1 Partially Observable Semi-Markov Decision Process (POSMDP)

The setting of POSMDP [61] [22] is described as follows. The domain is decomposed into multiple subdomains. Each subdomain is equivalent to an option ξ in the SMDP framework and has a subgoal $g \in \Omega$, that needs to be achieved before switching to another option. Within an option, the agent observes a partial state $o \in \Omega$, takes an action $a \in \mathcal{A}$, receives an extrinsic reward $r^{ex} \in \mathbb{R}$, and transits to another state s' (but the agent only observes a part of the state $o' \in \Omega$). The agent executes option ξ until it is terminated (either the subgoal g is obtained or the termination signal is raised). Afterward, the agent selects and executes another option. The sequence of options is repeated until the agent reaches the final goal. In addition, to achieve subgoal g in each option, an intrinsic reward function $r^{in} \in \mathbb{R}$ is maintained. While the objective of a subdomain is to maximize the cumulative discounted intrinsic reward, the objective of the whole domain is to maximize the cumulative discounted extrinsic reward.

Specifically, the belief states are updated given a taken option ξ, observation o and current

belief states b is defined as

$$b'(s') \propto \sum_{k=1}^{\infty} \gamma^{k-1} \sum_{s} \mathcal{P}(s', o, k|s, \xi) b(s), \tag{3.1}$$

where $\mathcal{P}(s', o, k|s, \xi)$ is a joint transition and observation function of the underlying POSMDP model on the environment.

We adopt a similar notation from the two frameworks MAX-Q [20] and Options [17] to describe our problem. We denote $\mathcal{Q}^M(o_t, \xi|\theta^M)$ as the Q-value function of the meta-controller at state o_t, θ^M (in which we use an RNN to encode the past observation that has been encoded via its weights θ^M) and option (macro-action) ξ (assuming ξ has a corresponding subgoal g_t). We note that the pair $\{o_t, \theta^M\}$ represents the belief state at time t, b_t (we will use them interchangeably: $\mathcal{Q}^M(b, \xi)$ or $\mathcal{Q}^M(o, \xi|\theta^M)$).

The multi-time observation model of option ξ [61], which is initiated in belief state b and observe o, has the following formula:

$$\mathcal{P}(o|b, \xi) = \sum_{k=1}^{\infty} \sum_{s'} \sum_{s} \gamma^k \mathcal{P}(s', o, k|s, \xi) b(s). \tag{3.2}$$

Using the above formula, we can write the Bellman equation for the value function of the meta-controller μ (E.g. policy over options) as follows:

$$\mathcal{V}^M(b) = \sum_{\xi} \mu(\xi|b) \left[\mathcal{R}_b^{\xi} + \sum_{o'} \mathcal{P}(o'|b, \xi) \mathcal{V}^M(b') \right], \tag{3.3}$$

and the option-value function as

$$\mathcal{Q}^M(b, \xi) = \mathcal{R}_b^{\xi} + \sum_{o'} \mathcal{P}(o'|b, \xi) \sum_{\xi' \in \Xi} \mu(\xi'|b') \mathcal{Q}^M(b', \xi'). \tag{3.4}$$

In case, the reward term R_b^{ξ} is the total reward collected by executing the option ξ, it is equivalent to an extrinsic reward (E.g. r^{ex}) in our model. $\mathcal{Q}^M(b, \xi)$ can be estimated using one step Q-learning

algorithm as follows,

$$\mathcal{Q}^M(b,\xi) \leftarrow \mathcal{Q}^M(b,\xi) + \alpha\Big[r^{ex} + \gamma^k \max_{\xi' \in \Xi} \mathcal{Q}^M(b',\xi') - \mathcal{Q}^M(b,\xi)\Big], \tag{3.5}$$

In case the reward term R_b^ξ is an immediate reward, it is equivalent to an intrinsic reward (E.g. r^{in}) and defined as $\mathcal{V}^S(b)$. Its corresponding Q-value function is defined as $\mathcal{Q}^S(b,a)$ (the value function for sub-controllers). In the use of RNN, we also denote $\mathcal{Q}^S(b,a)$ as $\mathcal{Q}^S(\{o_t,g_t\},a|\theta^S)$, in which θ^S is the weights of the sub-controller network that encodes previous observations, and $\{o_t,g_t\}$ denote the observations input to the sub-controller. Similarly, $\mathcal{Q}^S(b,a)$ can be estimated using one step Q-learning algorithm as follows,

$$\mathcal{Q}^S(b,a) \leftarrow \mathcal{Q}^S(b,a) + \alpha\Big[r^{in} + \gamma \max_{a' \in \mathcal{A}} \mathcal{Q}^S(b',a') - \mathcal{Q}^S(b,a)\Big]. \tag{3.6}$$

3.2 Proposed frameworks

In this section, we introduce two proposed frameworks of two algorithms: hDRQNv1 and hDRQNv2.

Frameworks 1 Hierarchical frameworks 1 is briefly illustrated in Figure 3.1 and is clearly illustrated in Figure 3.2. Basically, version 1 is inspired by a related idea in [44]. The difference is that our framework is built on two deep recurrent neural policies: a meta-controller and a sub-controller. By organizing proposed algorithms in form of two hierarchical policies, they can deal with the tasks having multiple subtasks. the meta-controller has the role of a planner which plans the agent to complete a sequence of subtasks. It is equivalent to a "policy over subgoals" (in SMDP) that receives current observation o_t and determines new subgoal g_t. In contrast, the sub-controller has the role of a low-level controller which directly interacts with the environment by performing action a_t. It is equivalent to the option's policy in SMDP and receives both subgoal g_t and observation o_t as inputs to the deep neural network. Each controller contains an arrow (Figure 3.1) pointed to itself that indicates that the controller employs recurrent neural networks. In addition, an internal component called "critic" is used to determine whether the agent has achieved the subgoal or not and to produce an intrinsic reward that is used to learn the sub-controller. The

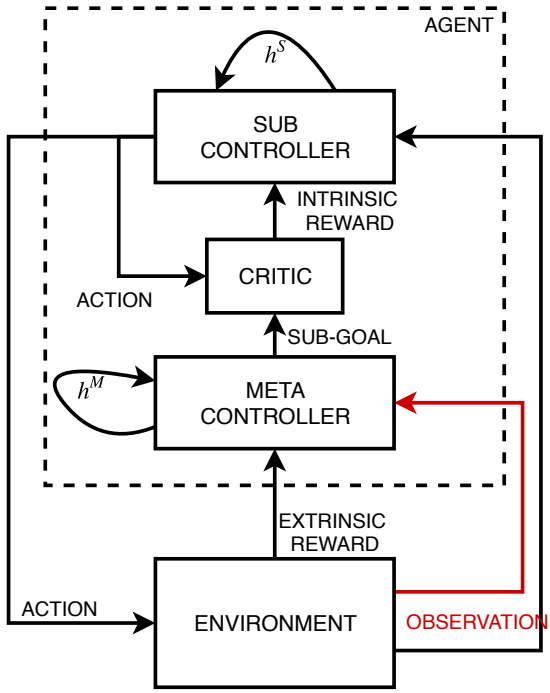


Figure 3.1: Framework 1 (Brief version)

networks for the controllers are illustrated in Figure 3.2. A sequence of four CONV layers and ReLU layers interleaved together is used to extract information from the raw observations. A recurrent layer, especially LSTM, is employed in front of the features to memorize the information from previous observations. CONVs are used to learn low-dimensional representations for image inputs, while the recurrent layer is used to capture the temporal relations among the observation data. Without using the recurrent layer, CONVs cannot accurately approximate the state features from observations in the POMDP domain. The output of the recurrent layer is split into Advantage stream $\mathcal{A}$ and Value stream $\mathcal{V}$ before being unified with the $\mathcal{Q}$-values. This architecture is inherited from the Dueling architecture [78], which effectively learns states without having to learn the effect of an action on that state. The input of the meta-controller's network is a raw image (color image) while the input of the sub-controller's network is two raw images: one for observation and one for subgoal. The output of the meta-controller's network is a vector of $\mathcal{Q}$ values called $\mathcal{Q}^M$. The number of $\mathcal{Q}^M$ values equals the number of predefined subgoals. Meanwhile, the output of sub-controller's network is a vector of $\mathcal{Q}$ values called $\mathcal{Q}^S$. The number of $\mathcal{Q}^S$ values equals the number of primitive actions.

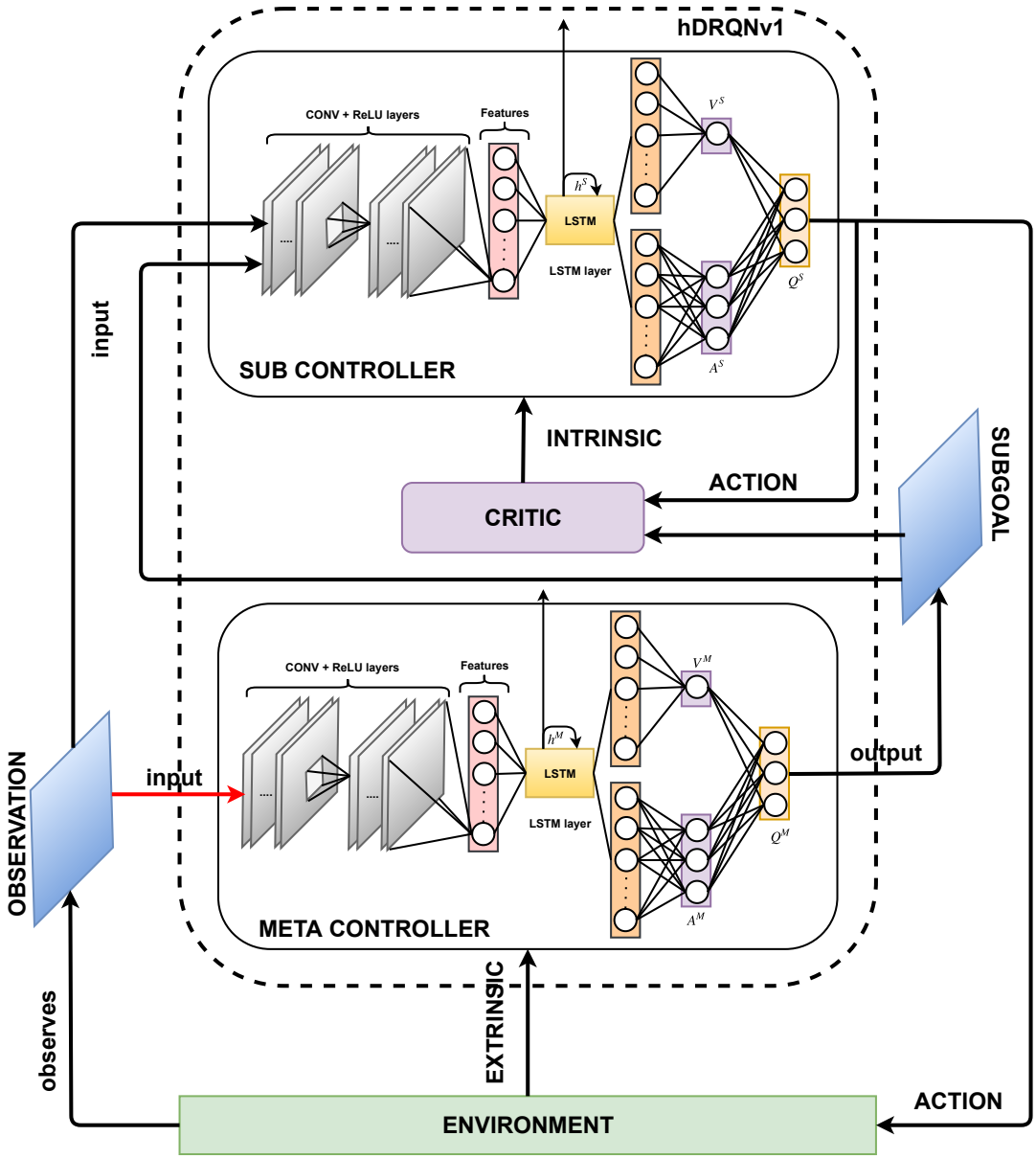


Figure 3.2: Framework 1 (Extended version)

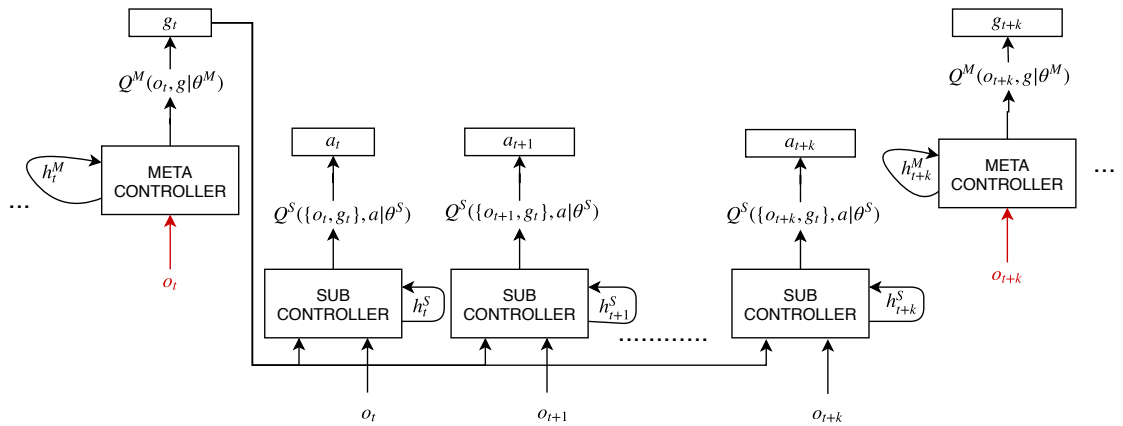


Figure 3.3: Workflow of framework 1

From the framework in Figure 3.2, we can interpret into the workflow in Figure 3.3. In this workflow, either meta-controller or sub-controller is activated at a time, and meta-controller is the first controller activated. Particularly, at step t, the meta-controller takes an observation o_t from the environment, extracts state features through some deep neural layers, internally constructs a hidden state h_t^M, and produces the $\mathcal{Q}$ subgoal values $\mathcal{Q}^M(o_t, g_t|\theta^M)$. The $\mathcal{Q}$ subgoal values then are used to determine next subgoal g_{t+k}. Similarly, the sub-controller receives both observation o_t and subgoal g_t, extracts their feature, constructs a hidden state h_t^S, and produces $\mathcal{Q}$ action values $\mathcal{Q}^S(\{o_t, g_t\}, a_t|\theta^S)$, which are used to determine next action a_{t+1}.

Framework 2 In contrast to the framework 1, framework 2 in Figure 3.5 and Figure 3.4 does not use the current observation to determine the subgoal in the meta-controller but instead uses the last hidden state h_t^S of the sub-controller. The last hidden state that is inferred from a sequence of observations of the sub-controller is expected to contribute to the meta-controller to correctly determine the next subgoal. For framework 2, the network of the sub-controller is the same as the network of the sub-controller in the framework 1. However, because the meta-controller uses the internal hidden state of the sub-controller as the input rather than a raw image, the network of the meta-controller is constructed by three fully connected layers and ReLU layers for extracting the features. The part of the network behind the features is the same as the meta-controller in the framework 1. Similarly, the framework 2 is interpreted to the workflow in Figure 3.6. The red arrow indicates the relationship between sub-controller and meta-controller where hidden states from the sub-controller are the input of the meta-controller.

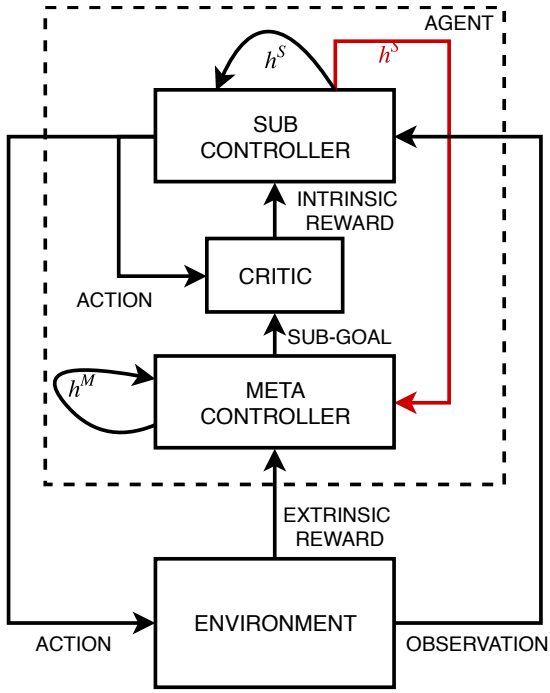


Figure 3.4: Framework 2 (Brief version)

3.3 Learning Model

As discussed from the previous section, the meta-controller receives an observation o_t (framework 1) or hidden states h^S from sub-controller (framework 2) and outputs a $\mathcal{Q}$ subgoal values $\mathcal{Q}^M(\hat{o}_t, g_t|\theta^M)$ where $\hat{o}_t$ can be o_t or h^S, and θ^M is parameters of meta-controller including weights and biases. Similarly, $\mathcal{Q}^S(\{o_t, g_t\}, a_t|\theta^S)$ is $\mathcal{Q}$ action value generated by the sub-controller where subgoal g_t and observation o_t is inputs and θ^S is parameters. Concurrently, both controllers generate hidden states called h^M and h^S because they employ recurrent layers. Formally, we have:

$$h_t^M, \mathcal{Q}^M(\hat{o}_t, g_t|\theta^M) = f_{RNN}^M(\Phi^M, h_{t-1}^M) \tag{3.7}$$

$$h_t^S, \mathcal{Q}^S(\{o_t, g_t\}, a_t|\theta^S) = f_{RNN}^S(\Phi^S, h_{t-1}^S), \tag{3.8}$$

where f_{RNN}^M and f_{RNN}^S are the respective recurrent networks of the meta-controller and the sub-controller, which receive the state features and previous hidden states and then provide the next

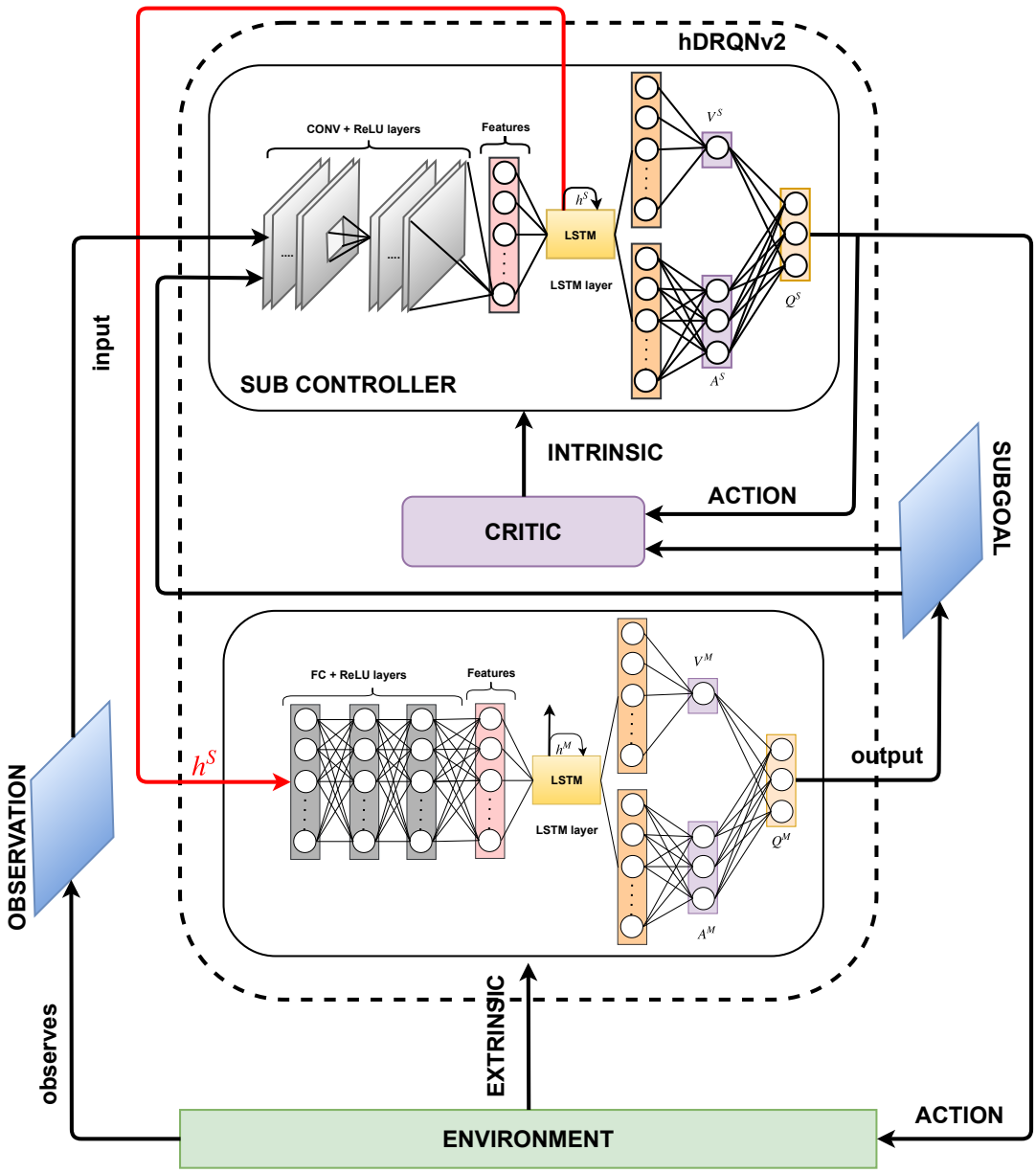


Figure 3.5: Framework 2 (Extended version)

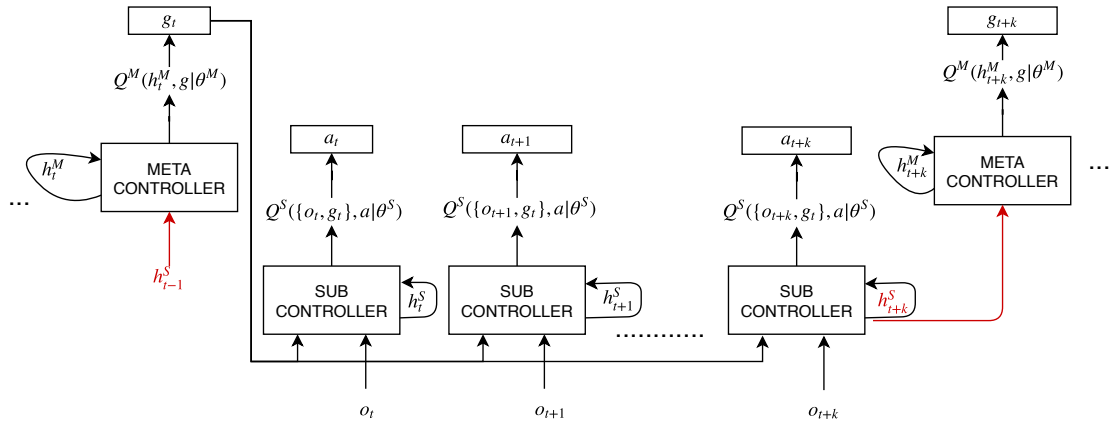


Figure 3.6: Workflow of framework 2

hidden states and $\mathcal{Q}$ values. Φ^M and Φ^S are the features after the extraction process of the meta-controller and the sub-controller, respectively. The formula of Φ^M and Φ^S in formal form is as follows:

$$\Phi^M = \begin{cases} f^M_{extract}(o_t) & \text{framework 1} \\ f^M_{extract}(h^S) & \text{framework 2} \end{cases} \tag{3.9}$$

$$\Phi^S = f^S_{extract}(o_t, g_t), \tag{3.10}$$

where $f^M_{extract}$ and $f^S_{extract}$ infer the network part to extract features from raw images (E.g. CONV layers and FC layers)

In order to learn the parameters of the network, we use a state-of-the-art Double DQN method. In contrast to DQN, which uses the maximum operator for both selecting and evaluating an action, Double DQN separately uses the main $\mathcal{Q}$ network to greedily estimate the next action and the target $\mathcal{Q}$ network to estimate the value function. This method has been shown to achieve better performance than DQN regarding Atari games [77]. The method is explained in details as follows.

Learning in meta-controller When a subdomain completed (E.g. A subgoal obtained), a tuple of $< \hat{o}, g, \hat{o}', r^{ex} >$ is stored to an experience replay $\mathcal{M}^M$ (Figure 3.7). At each time step, a batch of samples is randomly selected from the experience replay. These samples are used to update the parameters of the network. Particularly, $\mathcal{Q}^M$ is trained by minimizing the loss function $\mathcal{L}^M$ as

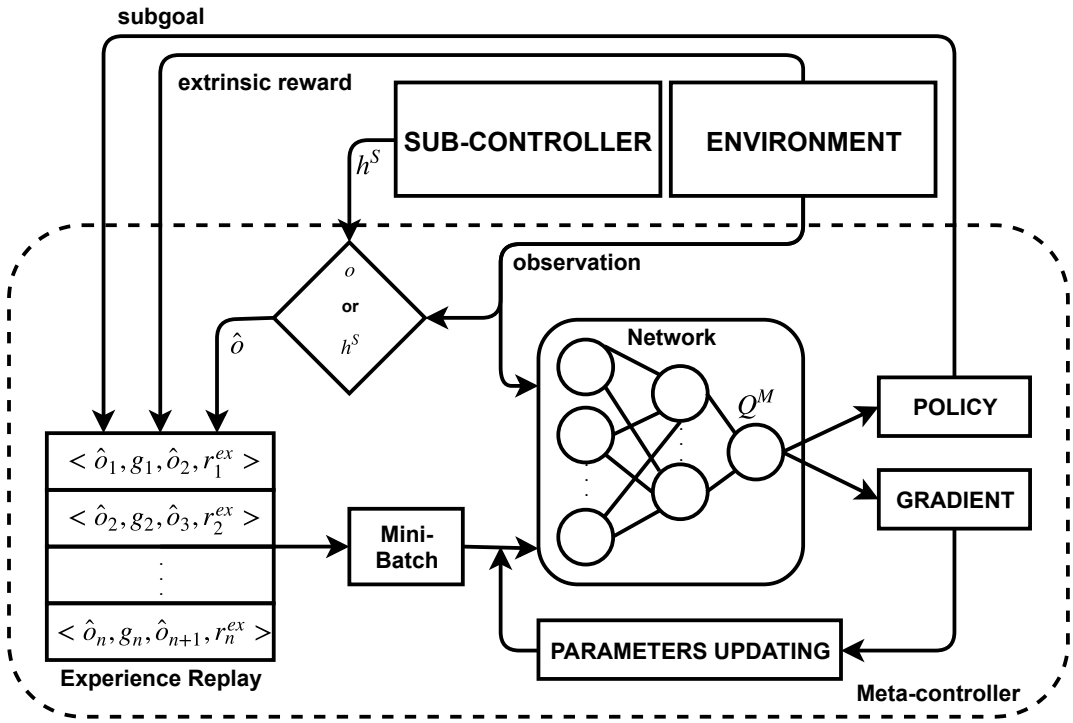


Figure 3.7: Learning in meta-controller

follows:

$$\mathcal{L}^M = \mathbb{E}_{(o,g,o',g',r^{ex})\sim\mathcal{M}^{\mathcal{M}}}\left[y_i^M - \mathcal{Q}^M(o,g|\theta^M)\right], \tag{3.11}$$

where $\mathbb{E}$ denotes the expectation over a batch of data which is uniformly sampled from experience replay $\mathcal{M}^M$, i is the iteration number in the batch of samples and y_i^M is the target value of $\mathcal{Q}^M$. y_i^M is calculated based on Double DQN technique as follows,

$$y_i^M = r^{ex} + \gamma\mathcal{Q}^{M'}(o', \underset{g'}{\operatorname{argmax}}\, \mathcal{Q}^M(o', g'|\theta^M)|\theta^{M'}). \tag{3.12}$$

where and $\mathcal{Q}^{M'}$ is the target network of $\mathcal{Q}^M$

Learning in sub-controller Similarly, at each time step, a tuple of $< \{o, g\}, a, r^{in}, \{o', g'\} >$ is stored to an experience replay $\mathcal{M}^S$ (Figure 3.8) and a batch of samples is randomly selected to

update parameters of $\mathcal{Q}^S$. The parameters of $\mathcal{Q}^S$ is updated to minimize the loss function $\mathcal{L}^S$

$$\mathcal{L}^S = \mathbb{E}_{(o,g,a,r^{in})\sim\mathcal{M}^S}\left[y_i^S - \mathcal{Q}^S(\{o,g\},a|\theta^S)\right], \tag{3.13}$$

where

$$y_i^S = r^{in} + \gamma\mathcal{Q}^{S'}(\{o',g\}, \underset{a'}{\operatorname{argmax}}\, \mathcal{Q}^S(\{o',g\},a'|\theta^S)|\theta^{S'}). \tag{3.14}$$

and $\mathcal{Q}^{S'}$ is the target network of $\mathcal{Q}^S$.

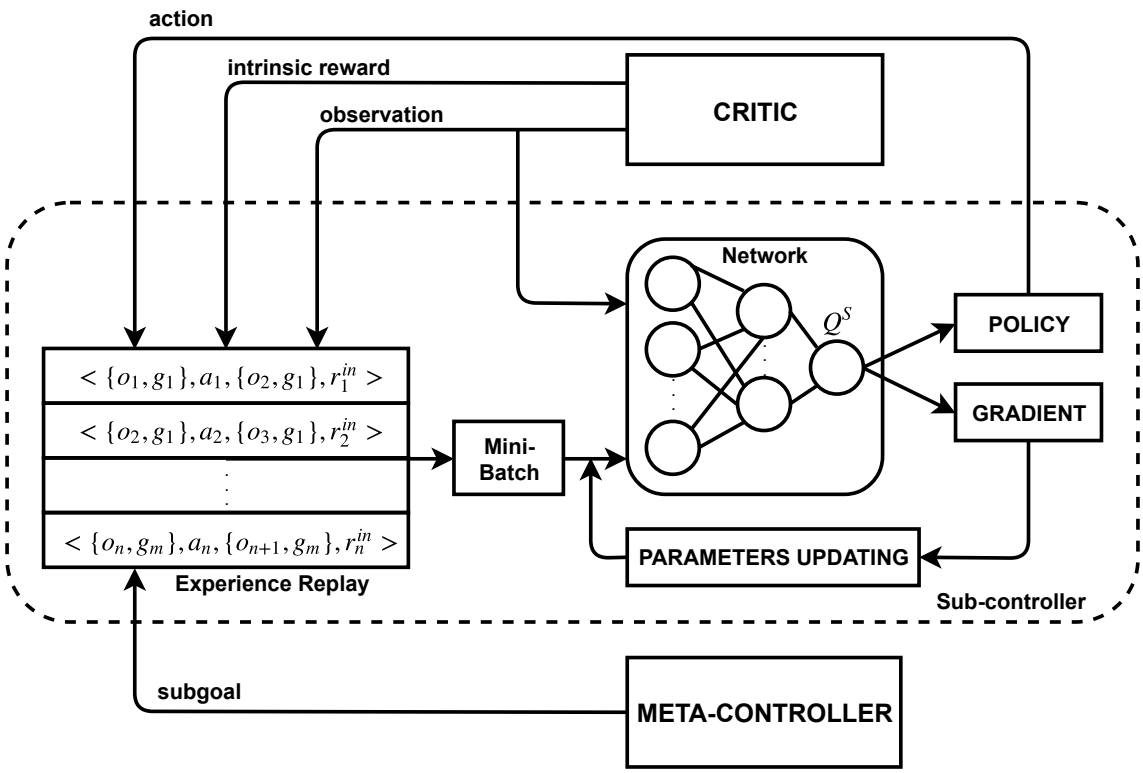


Figure 3.8: Learning in sub-controller

3.4 Minibatch Sampling Strategy

For updating RNNs in our model, we need to analyze a batch of samples. Particularly, episodes from the experience replay are uniformly sampled and processed from the beginning of the episode

to the end of the episode. This strategy called Bootstrapped Sequential Updates [63], is an ideal method to update RNNs because their hidden state can carry all information throughout an entire episode. However, this strategy is computationally expensive in a long episode, which can contain many time steps. Another approach, proposed in [63], has been evaluated to achieve the same performance as Bootstrapped Sequential Updates while also reducing the computational complexity. The strategy called Bootstrapped Random Updates, is illustrated in Figure 3.9. This strategy randomly selects a batch of episodes from the experience replay. Then, for each episode, we begin at a random index and select a sequence of n transitions. The value of n, which affects the performance of our algorithms, is analyzed in Chapter 4. For each controller, we have n^M and n^S corresponding to the length of sampled transitions in the meta-controller and the sub-controller. We apply the same procedure of Bootstrapped Random Updates to our algorithms.

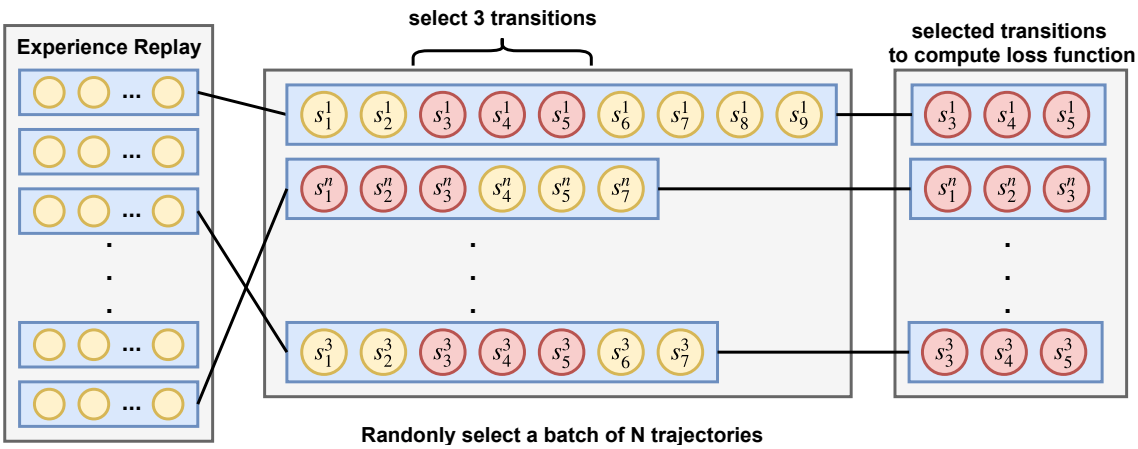


Figure 3.9: Bootstrapped random updates strategy

In addition, the mechanism explained in [82] is also applied. That study discusses a problem when updating DRQN: using the first observations in a sequence of transitions to update the $\mathcal{Q}$ value function might be inaccurate. Thus, the solution is to use the last observations to update DRQN. Particularly, our method uses the last $\frac{n}{2}$ transitions to update the Q-value.

3.5 Subgoal Definition

Our model is based on the "option" framework. Learning an option is accomplished via flat deep RL algorithms to achieve a subgoal of that option. Each subgoal is equivalent to an option termination $\beta(s)$. When the agent has the same position with the subgoal, we state that the agent has obtained the subgoal and another subgoal is assigned. However, due to the curve of dimensionality in RL, discovering subgoals among existing states in the environment is still a challenge in hierarchical reinforcement learning. To simplify the model, we assume that a set of pre-defined subgoals is provided in advance.

The pre-defined subgoals based on object-oriented MDPs [83], where entities or objects in the environment are decoded as subgoals. Besides, each entity in the environment has a relationship with other entities. For example, in the domain of two goals in four-rooms (Figure 3.10), we have five entities: agent, blue goal, green goal, first obstacle, second obstacle. Relationships between entities are as follows: agent *reaches* the blue goal, agent *reaches* the green goal, agent *hits* the first obstacle, agent *hits* the second obstacle. They are used in the internal critic to build intrinsic reward functions.

3.6 Intrinsic and Extrinsic Reward Functions

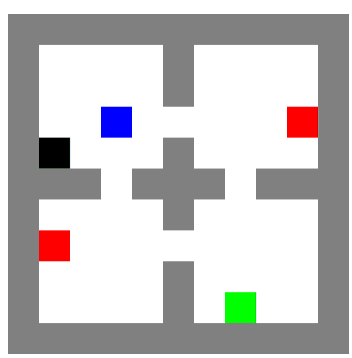


Figure 3.10: Example domain for illustrating the notions of intrinsic and extrinsic motivation

Traditional RL accumulates all reward and penalty based on a reward function, which is difficult to learn in a specified task in a complex domain. In contrast, hierarchical RL introduces the notions of an intrinsic reward function and an extrinsic reward function. Initially, intrinsic moti-

vation is based on psychology, cognitive science, and neuroscience [84] and has been applied to hierarchical RL [85] [86] [87] [88] [89] [90]. Our framework follows the model of intrinsic motivation in [86]. Particularly, within an option (or skill), the agent needs to learn an option's policy (the sub-controller in our framework) to obtain a subgoal (a salient event) under the reinforcement of an intrinsic reward while a policy over options (the meta-controller) is learned to generate a sequence of subgoals under the reinforcement of an extrinsic reward, for the overall task. Defining "good" intrinsic reward functions and extrinsic reward functions is still an open question in reinforcement learning, and it is difficult to find a reward function model that is generalized to all domains.

To demonstrate some of the notions above, Figure 3.10 describes the domain of multiple goals in four-rooms, which is used to evaluate our algorithm in Chapter 4. The four-rooms contain a number of objects: an agent (in black), two obstacles (in red) and two goals (in blue and green). These objects are randomly located on the map. At each time step, the agent has to follow one of the four actions: top, down, left or right, and has to move to the goal location in a proper order: the blue goal first and then the green goal. If the agent obtains all goals in the right order, it will receive a big reward; otherwise, it will only receive a small reward. In addition, the agent has to learn to avoid the obstacles if it does not want to be penalized. For this example, the salient event is equivalent to reaching the subgoal or hitting the obstacle. In addition, there are two skills the agent should learn. One is moving to the goals while correctly avoiding the obstacles, and the second is selecting the goal it should reach first. The intrinsic reward for each skill is generated based on the salient events encounters while exploring the environment. Particularly, to reach the goal, the intrinsic reward includes the reward for reaching the goal successfully and the penalty if the agent encounters an obstacle. To reach the goals in order, the intrinsic reward includes a big reward if the agent reaches the goals in the proper order and a small reward if the agent reaches the goal in an improper order. A detailed explanation of intrinsic and extrinsic rewards for this domain is included in Chapter 4.

Algorithm 1: hDRQN in POMDP

Require:

POMDP $M = \{\mathcal{S}, \mathcal{A}, \mathcal{P}, r, \Omega, \mathcal{Z}\}$

Meta-controller with the network $\mathcal{Q}^M$ (main) and $\mathcal{Q}^{M'}$ (target) parameterized by θ^M and $\theta^{M'}$, respectively.

Sub-controller with the network $\mathcal{Q}^S$ (main) and $\mathcal{Q}^{S'}$ (target) parameterized by θ^S and $\theta^{S'}$, respectively.

Exploration rate ϵ^M for meta-controller and ϵ^S for sub-controller.

Experience replay memories $\mathcal{M}^M$ and $\mathcal{M}^S$ of meta-controller and sub-controller, respectively.

A pre-defined set of subgoals $\mathcal{G}$.

f^M and f^S are recurrent networks of meta-controller and sub-controller, respectively.

Ensure:

Initialize:

- Experiences replay memories $\mathcal{M}^M$ and $\mathcal{M}^S$;
- Randomly initialize θ^M and θ^S;
- Assign value to the target networks $\theta^{M'} \leftarrow \theta^M$ and $\theta^{S'} \leftarrow \theta^S$;
- $\epsilon^M \leftarrow 1.0$ and decreasing to 0.1 ;
- $\epsilon^S \leftarrow 1.0$ and decreasing to 0.1 ;

for $k = 1, 2, \ldots K$ **do**
  Initialize: the environment and get the start observation o ;
  Initialize: hidden states $h^M \leftarrow \mathbf{0}$;
  while o is **not** terminal **do**
    Initialize: hidden states $h^S \leftarrow \mathbf{0}$;
    Initialize: start observations $o_0 \leftarrow \hat{o}$ where $\hat{o}$ can be observation o or hidden state h^S;
    Determine subgoal: $g, h^M \leftarrow EPS\_GREEDY(\hat{o}, h^M, \mathcal{G}, \epsilon^M, \mathcal{Q}^M, f^M)$;
    while o is **not** terminal **and** g is **not** reached **do**
      Determine action: $a, h^S \leftarrow EPS\_GREEDY(\{o, g\}, h^S, \mathcal{A}, \epsilon^S, \mathcal{Q}^S, f^S)$;
      Execute action a, receive reward r, extrinsic reward r^{ex}, intrinsic reward r^{in}, and obtain the next state s' ;
      Store transition $\{\{o, g\}, a, r^{in}, \{o', g'\}\}$ in $\mathcal{M}^S$;
      Update sub-controller $SUB\_UPDATE(\mathcal{M}^S, \mathcal{Q}^S, \mathcal{Q}^{S'})$;
      Update meta-controller $META\_UPDATE(\mathcal{M}^M, \mathcal{Q}^M, \mathcal{Q}^{M'})$;
      Transition to next observation $o \leftarrow o'$;
    end while
    Store transition $\{o_0, g, r^{ex}_{total}, \hat{o}'\}$ in $\mathcal{M}^S$ where $\hat{o}'$ can be observation o' or the last hidden state h^S;
  end while
  Anneal ϵ^M and ϵ^S;
end for

Algorithm 2: $EPS\_GREEDY(x, h, \mathcal{B}, \epsilon, \mathcal{Q}, f)$

Require:
x: input of the $\mathcal{Q}$ network
h: internal hidden states
$\mathcal{B}$: a set of outputs
ϵ: exploration rate
$\mathcal{Q}$ network and recurrent function f
Ensure:
$h \leftarrow f(x, h)$;
if $random() < \epsilon$ **then**
    $o \leftarrow$ An element from the set of output $\mathcal{B}$;
else
    $o = \text{argmax}_{m \in \mathcal{B}} \mathcal{Q}(x, m)$;
end if
Return o, h;

3.7 Pseudo-code Algorithms

In this section, our contributions are summarized through pseudo-code Algorithm 1. The algorithm learns four neural networks: two networks for the meta-controller ($\mathcal{Q}^M$ and $\mathcal{Q}^{M'}$) and two networks for the sub-controller ($\mathcal{Q}^S$ and $\mathcal{Q}^{S'}$). They are parameterized by θ^M, $\theta^{M'}$, θ^S and $\theta^{S'}$, respectively. The architectures of the networks are described in Section 3.2. In addition, the algorithm separately maintains two experience replay memories $\mathcal{M}^M$ and $\mathcal{M}^S$ to store transition data from the meta-controller and the sub-controller, respectively. Before starting the algorithm, the parameters of the main networks are randomly initialized and are copied to the target networks. ϵ^M and ϵ^S are annealed from 1.0 to 0.1, which gradually increase the control of the controllers.

The algorithm started by looping through a specified number of episodes (Line 9) and each episode is executed until the agent reaches the terminal state. To start an episode, first, a starting observation o_0 is obtained (Line 10). Next, hidden states, which are inputs for the RNNs, must be initialized with zero values (Line 11 and Line 13) and are updated during the episode (Line 15 and Line 17). Each subgoal is determined by passing observation o or hidden state h^S (depending on the framework) to the meta-controller (Line 15). By following a greedy ϵ fashion, a subgoal will be selected from the meta-controller if it is a random number greater than ϵ. Otherwise, a random subgoal will be selected (Algorithm 2). The sub-controller is taught to reach the subgoal;

Algorithm 3: $META\_UPDATE(\mathcal{M}^M, \mathcal{Q}^M, \mathcal{Q}^{M'})$

Require:

1: $\mathcal{M}^M$: experience replay memory of meta-controller

Ensure:

2: **Sample** a mini-batch of $\{o, g, r^{ex}, o'\}$ from $\mathcal{M}^M$ as the strategy explained at 3.4;

3: **Update the network** by minimizing the loss function:

$$\mathcal{L}^M = \mathbb{E}_{(o,g,o',g',r^{ex})\sim\mathcal{M}^{\mathcal{M}}}\left[y_i^M - \mathcal{Q}^M(o, g|\theta^M)\right]$$

where

$$y_i^M = r^{ex} + \gamma\mathcal{Q}^{M'}(o', \underset{g'}{\operatorname{argmax}}\, \mathcal{Q}^M(o', g'|\theta^M)|\theta^{M'})$$

4: **Update the target network:** $\theta^{M'} \leftarrow \tau\theta^M + (1-\tau)\theta^{M'}$

when the subgoal is reached, a new subgoal will be selected. The process is repeated until the final goal is obtained. The intrinsic reward is evaluated by the critic module and is stored in $\mathcal{M}^S$ (Line 19) for updating the sub-controller. Meanwhile, the extrinsic reward is directly received from the environment and is stored in $\mathcal{M}^M$ for updating the meta-controller (Line 24).

The main part of the proposed algorithm is updating the controllers at Line 20 and 21. Pseudo-code of updating the meta-controller and the sub-controller is shown in Algorithm 3 and Algorithm 4, respectively. In Algorithm 3, a batch of data is sampled from $\mathcal{M}^M$ at Line 2 and is used to update main network $\mathcal{Q}^M$ at Line 3 using Double DQN which is described in Section 3.3. Finally, we use a soft update technique to update target network $\mathcal{Q}^{M'}$ at Line 4. The soft update technique uses a small learn rate τ to update the target network, thus, decoupling from the main network. Similarly, updating the sub-controller in Algorithm 4 has the same procedure as updating the meta-controller.

Algorithm 4: $SUB\_UPDATE(\mathcal{M}^S, \mathcal{Q}^S, \mathcal{Q}^{S'})$

Require:

1: $\mathcal{M}^S$: experience replay memory of meta-controller

2: **Sample** a mini-batch of $\{\{o, g\}, a, r^{in}, \{o', g'\}\}$ from $\mathcal{M}^S$ as the strategy explained at 3.4;

3: **Update the main network** by minimizing the loss function:

$$\mathcal{L}^S = \mathbb{E}_{(o,g,a,r^{in})\sim\mathcal{M}^S}\left[y_i^S - \mathcal{Q}^S(\{o, g\}, a|\theta^S)\right],$$

where

$$y_i^S = r^{in} + \gamma\mathcal{Q}^{S'}(\{o', g\}, \underset{a'}{\operatorname{argmax}}\, \mathcal{Q}^S(\{o', g\}, a'|\theta^{S'})|\theta^{S'})$$

4: **Update the target network:** $\theta^{S'} \leftarrow \tau\theta^S + (1-\tau)\theta^{S'}$

Chapter 4

Experimental Results and Discussions

In this section, we evaluate two versions of the hierarchical deep recurrent network algorithm in terms of reward (intrinsic and extrinsic) and success ratio. hDRQNv1 is the algorithm formed by framework 1, and hDRQNv2 is the algorithm formed by framework 2. We compare them with flat algorithms (DQN, DRQN) and the state-of-the-art hierarchical RL algorithm (hDQN). The comparisons are performed on three domains. The domain of multiple goals in a gridworld is used to evaluate many aspects of the proposed algorithms. Meanwhile, the harder domain called multiple goals in four-rooms, is used to compare the proposed algorithms with baselines algorithms. Finally, one of the most challenging games in ATARI 2600 [26], called Montezuma's Revenge, is used to confirm the efficiency of our proposed framework. A sample animation of evaluated domains can be found here [1]

4.1 Experiment Settings

4.1.1 Domains

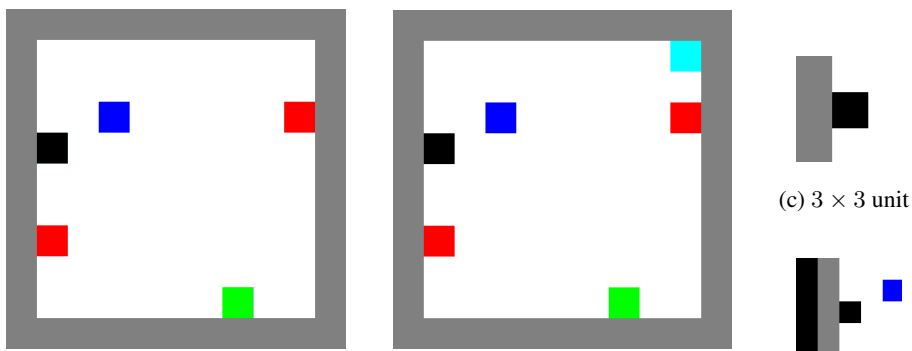
We evaluate the proposed algorithms on five domains which are summarized in Table 4.1 and are described in detail as follows:

Two Goals in Gridworld A gridworld map of size 11×11 units contains a number of objects: an agent (in black), two obstacles (in red) and two goals (in blue and green). These objects are randomly located on the map. At each time step, the agent has to follow one of the four actions: top, down, left or right, and has to move to the goal location in a proper order: the blue goal first and then the green goal. If the agent obtains all goals in the right order, it will receive a big

[1] A sample animation of evaluated domains: https://youtu.be/r2wQiOc6euE

Table 4.1: Summarization of domains

Domains	Subgoals	Obstables	Size	Remark
Two goals in gridworld	2	2	11 x 11	gridworld
Three goals in gridworld	3	2	11 x 11	gridworld
Two goals in four-rooms	2	2	11 x 11	four-rooms
Three goals in four-rooms	3	2	11 x 11	four-rooms
Montezuma's Revenge	7	1	210 x 160	complex map



(a) Two goals in the gridworld domain

(b) Three goals in the gridworld domain

(c) 3×3 unit

(d) 5×5 unit

Figure 4.1: Multiple goals in the gridworld domain

reward; otherwise, it will only receive a small reward. In addition, the agent has to learn to avoid the obstacles if it does not want to be penalized. At each time step, the agent only observes a part of the surrounding environment, either a 3×3 units (Figure 4.1c) or 5×5 units (Figure 4.1d). The agent is allowed to choose one of four actions (top, left, right, bottom), which are deterministic. The agent cannot move if the action leads it into the wall. The rewards for the agent are defined as follows. If the agent hits an obstacle, it will receive a penalty of minus one. If the agent reaches two goals in proper order, it will receive a reward of one for each goal. Otherwise, it only receives 0.01. The hDRQN algorithms use an intrinsic reward function and an extrinsic reward function, which are defined as follows:

$$r^{in} = \begin{cases} 1 & \text{obtain a goal} \\ -1 & \text{hit an obstacle} \end{cases} \tag{4.1}$$

and

$$r^{ex} = \begin{cases} 1 & \text{for each reached goal in proper order} \\ 0.01 & \text{for each reached goal in improper order} \end{cases} \tag{4.2}$$

In addition, we also maintain a classical reward which is used to compare with baseline algorithms:

$$r = \begin{cases} 1 & \text{for each reached goal in proper order} \\ 0.01 & \text{for each reached goal in improper order} \\ -1 & \text{hit an obstacle} \end{cases} \tag{4.3}$$

Three Goals in Gridworld This domain is similar to the domain of two goals in gridworld, but added one more goal (cyan). The agent similarly receives a big reward if it visits all goals in proper order: blue ⇒ green ⇒ cyan, and does not hit the obstacles. The hierarchical property of this domain is shown in Figure 4.2 in which each goal in the domain is equivalent to subgoals in the hierarchical algorithms. The meta-controller learns to obtain subgoals in proper order while the sub-controller learns to obtain the particular goal. We expect that the algorithms can adapt to the increasing of number of subgoals.

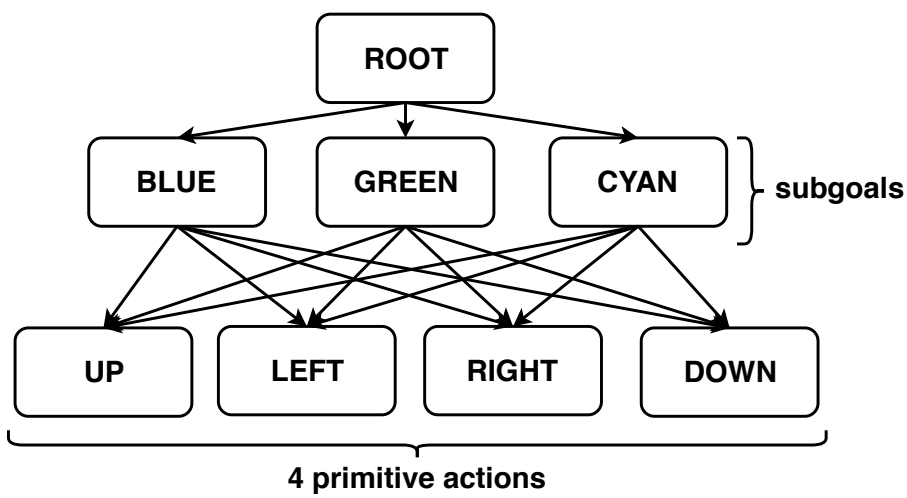
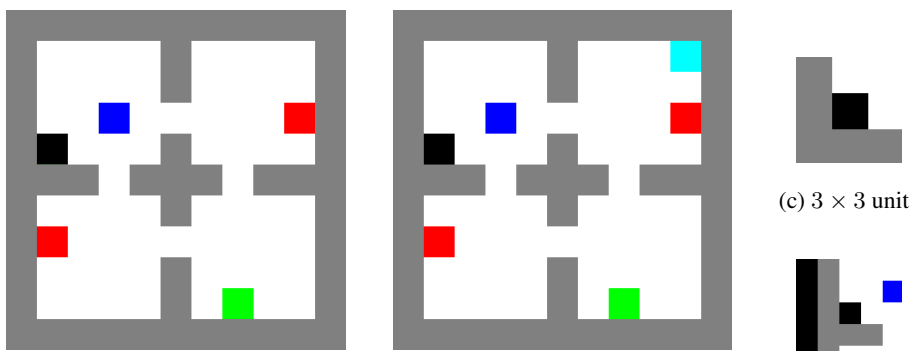


Figure 4.2: Hierarchy of the domains of multiple goals

Multiple Goals in Four-rooms In this domain, we apply the multiple goals domain to a complex map called four-rooms (Figure 4.3). The dynamics of the environment in this domain is similar to the domain of multiple goals in gridworld. The difference is that the agent in this domain must usually pass through hallways to obtain goals that are randomly located in four rooms. By using a four-rooms map, we challenge the agent and expect it can effectively learn in a more complex map. Originally, the four-rooms domain was an environment for testing a hierarchical reinforcement learning algorithm [17].



(a) Two goals in the four-rooms domain

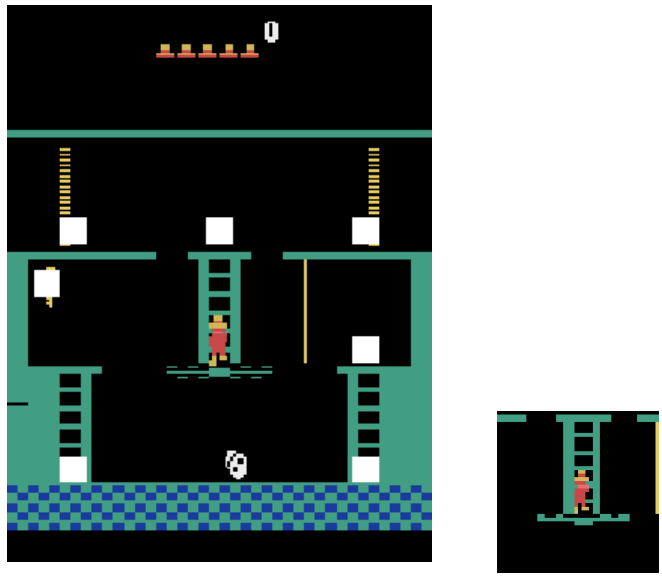
(b) Three goals in the four-rooms domain

(c) 3×3 unit

(d) 5×5 unit

Figure 4.3: Multiple goals in the four-rooms domain

Montezuma's Revenge Montezuma's Revenge is one of the hardest games in ATARI 2600, and the DQN algorithm [27] can only achieve a score of zero. We use OpenAI Gym to simulate this domain [91]. The game is hard because the agent must execute a long sequence of actions until a state with non-zero reward (delayed reward) can be visited. In addition, in order to obtain a state with larger rewards, the agent needs to reach a special state in advance. This dissertation evaluates our proposed algorithms on the first screen of the game (Figure 4.4). Particularly, the agent, which only observes a part of the environment (Figure 4.4b), needs to pass through doors (the yellow line in the top left and top right corners of the figure) to explore other screens. However, to pass through the doors, first, the agent needs to pick up the key on the left side of the screen. Thus, the agent must learn to navigate to the key's location and then navigate back to the door and transit to the next screen. The agent will earn 100 points after it obtains the key and 300 after it reaches any door (with a key obtained in advance). In total, the agent can receive 400 points for this screen.



(a) The first screen of Montezuma's Revenge game

(b) Observable area

Figure 4.4: Montezuma's Revenge game in ATARI 2600

The intrinsic reward function is defined to motivate the agent to explore the whole environment. Particularly, the agent will receive an intrinsic value of 1 if it could reach a subgoal from the other subgoals. The set of subgoals is pre-defined in Figure 4.4a (the white rectangles). In contrast to the intrinsic reward function, the extrinsic reward function is defined as a reward value of 1 when the agent obtains the key or opens the doors.

$$r^{in} = \begin{cases} 1 & \text{reach subgoal} \\ 0 & \text{otherwise} \end{cases} \tag{4.4}$$

and

$$r^{ex} = \begin{cases} 1 & \text{obtain the key or open the doors} \\ 0 & \text{otherwise} \end{cases} \tag{4.5}$$

In addition, we also maintain a classical reward which is used to compare with baseline algorithms.

Particularly, A score of 100 points is given after agent obtains the key and a score of 300 is given after agent reaches any door (with the obtained key)

$$r = \begin{cases} 100 & \text{obtain key} \\ 300 & \text{open doors} \end{cases} \tag{4.6}$$

The domain can be interpreted to a hierarchy of subdomain as Figure 4.5. The first level of hierarchy tree is seven subgoals which are defined in Table 4.2 and the second level of hierarchy tree is 18 actions which are defined in Tables 4.3.

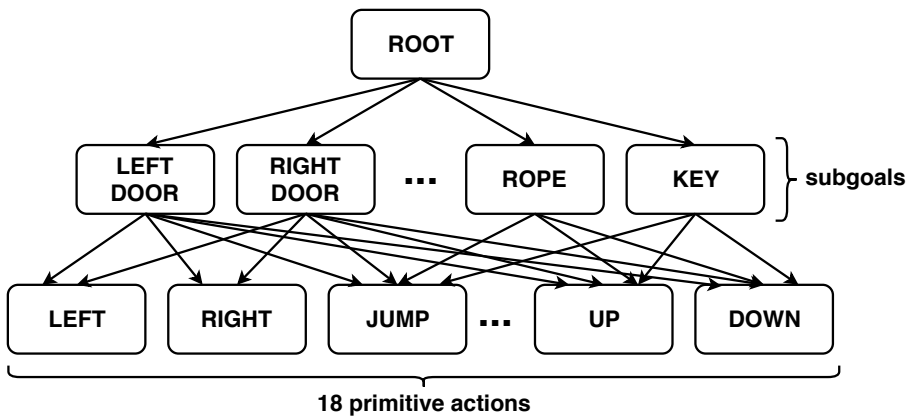


Figure 4.5: Hierarchy of Montezuma's Revenge game

Table 4.2: Seven predefined subgoals in Montezuma's Revenge

7 subgoals
left-door
top-mid-ladder
right-door
top-right-ladder
bottom-right-ladder
bottom-left-ladder
key

4.1.2 Baselines

We compare our proposed algorithms with some baseline algorithms: DQN, DRQN, hDQN and summarize them in the Table 4.4

Deep $\mathcal{Q}$-Learning algorithm (DQN) DQN described in Section 2.3.1, is a state-of-the-art algorithm. It starts the era of deep reinforcement learning. It has a remarkable achievement on almost ATARI 2600 games. However, it fails to solve the Montezuma's Revenge game and only works with fully observable domains

Table 4.3: 18 primitive actions of the agent in Montezuma's Revenge equivalent to 18 combinations of joystick's buttons

18 actions
NOOP
FIRE
UP
RIGHT
LEFT
DOWN
UPRIGHT
UPLEFT
DOWNRIGHT
DOWNLEFT
UPFIRE
RIGHTFIRE
LEFTFIRE
DOWNFIRE
UPRIGHTFIRE
UPLEFTFIRE
DOWNRIGHTFIRE
DOWNLEFTFIRE

Deep Recurrent $\mathcal{Q}$-Learning algorithm (DRQN) DRQN described in Section 2.3.4, is proposed to solve domains under partial observability. By introducing a framework which combines DQN and LSTM, DRQN has the ability to capture the correlation in a sequence of transitions under partial observability. Flickering ATARI 2600 games is used to show the efficiency of DRQN under partial observability. However, it cannot overcome the challenges raised by hierarchical domains.

Hierarchical Deep $\mathcal{Q}$-Learning (hDQN) hDQN described in Section 2.3.5, can deal with hierarchical domains by introducing a framework of two levels of policies. Upper policy plans the agent to complete subgoals and lower policies control the agent to perform primitive actions. The algorithm can pass Montezuma's Revenge game under full observability.

Table 4.4: Comparison between algorithms

Algorithms	MDP	POMDP	Hierarchical domains
DQN	o	x	x
DRQN	o	o	x
hDQN	o	x	o
hDRQNv1	o	o	o
hDRQNv2	o	o	o

4.1.3 Evaluation Metric

Some common metrics used to evaluate RL algorithms are expected cumulative reward, expected steps. On Montezuma's Revenge game, we use the success ratio to show how much the subdomain completed. In addition, we use expected cumulative extrinsic and expected cumulative intrinsic to show the performance of the meta-controller and the sub-controller.

Expected cumulative reward We sum the reward of agent for through the entire episodes and get the average reward by running the algorithm multiple times.

Expected steps The number of steps to complete the domains is used to compare algorithms. The better algorithm will complete the domain quickly.

Expected cumulative extrinsic and **expected cumulative intrinsic** They are introduced by the meta-controller and the sub-controller in the hierarchical algorithms.

Success ratio shows how much the subdomains completed

4.1.4 Algorithms Settings

The settings for each domain are different, but they have some commonalities as follows. For the hDRQNv1 algorithm, the inputs to the meta-controller and the sub-controller are an image of size $44 \times 44 \times 3$ (a color image). The input image is resized from an observation, which is

Table 4.5: Configuration of hDRQNv1's networks

Meta-Controller						
Layer	*Input*	*Filter size*	*Stride*	*Num Filter*	*Activation*	*Output*
CONV 1	44 x 44 x 3	3 x 3	4	32	ReLU	11 x 11 x 32
CONV 2	11 x 11 x 32	3 x 3	2	32	ReLU	5 x 5 x 32
CONV 3	5 x 5 x 32	3 x 3	1	32	ReLU	3 x 3 x 32
CONV 4	3 x 3 x 32	3 x 3	1	256	ReLU	1 x 1 x 256
LSTM	Features | 256 Internal state | 256					Features | 256 Internal state | 256
Advantage	128				Linear	Number of subgoal
Value	128				Linear	1
Q	Advantage and Value					Number of subgoal
Sub-controller						
Layer	*Input*	*Filter size*	*Stride*	*Num Filter*	*Activation*	*Output*
CONV 1	44 x 44 x 3	3 x 3	4	32	ReLU	11 x 11 x 32
CONV 2	11 x 11 x 32	3 x 3	2	32	ReLU	5 x 5 x 32
CONV 3	5 x 5 x 32	3 x 3	1	32	ReLU	3 x 3 x 32
CONV 4	3 x 3 x 32	3 x 3	1	256	ReLU	1 x 1 x 256
LSTM	Features | 256 Internal state | 256					Features | 256 Internal state | 256
Advantage	128				Linear	Number of action
Value	128				Linear	1
Q	Advantage and Value					Number of action

Table 4.6: Configuration of hDRQNv2's networks

Meta-Controller						
Layer	*Input*	*Filter size*	*Stride*	*Num Filter*	*Activation*	*Output*
FC1	256 (hidden state from SUB)				ReLU	200
FC2	200				ReLU	200
FC3	200				None	256
LSTM	Features | 256				None	Features | 256
	Internal state | 256					Internal state | 256
Advantage	128				Linear	Number of subgoal
Value	128				Linear	1
Q	Advantage and Value				None	Number of subgoal
Sub-controller						
Layer	*Input*	*Filter size*	*Stride*	*Num Filter*	*Activation*	*Output*
CONV 1	44 x 44 x 3	3 x 3	4	32	ReLU	11 x 11 x 32
CONV 2	11 x 11 x 32	3 x 3	2	32	ReLU	5 x 5 x 32
CONV 3	5 x 5 x 32	3 x 3	1	32	ReLU	3 x 3 x 32
CONV 4	3 x 3 x 32	3 x 3	1	256	ReLU	1 x 1 x 256
LSTM	Features | 256				None	Features | 256
	Internal state | 256					Internal state | 256
Advantage	128				Linear	Number of action
Value	128				Linear	1
Q	Advantage and Value				None	Number of action

Table 4.7: Parameters of proposed algorithms

Name	**Abbreviation**	**Values**
Optimization Method		ADAM [34]
Learning rate	α	0.0001
Discounted factor	γ	0.99
Sub-controller experience replay	$\mathcal{M}^S$	5000 episodes
Meta-controller experience replay	$\mathcal{M}^M$	1000 episodes
Soft-update target network	τ	0.001
Batch size of meta-controller	N^M	12
Batch size of sub-controller	N^S	4
Update frequence	freq	5
Exploration rate of meta-controller	ϵ^M	$1.0 \Rightarrow 0.1$
Exploration rate of sub-controller	ϵ^S	$1.0 \Rightarrow 0.1$
Length of meta-transition	n^S	variant
Length of sub-transition	n^M	variant
Initialization method		Xavier Initializer [92]

observed around the agent (either 3×3 unit or 5×5 unit). The image feature of 256 values extracted through four CONVs and ReLUs is put into a LSTM layer of 256 states to generate 256 output values, and internal hidden states of 256 values is also constructed. For the hDRQNv2 algorithm, hidden states of 256 values is put into the network of the meta-controller. The states are passed through three fully connected layers and ReLU layers instead of four CONV layers. The output is features of 256 values. The algorithm uses ADAM [34] for learning the neural network parameters with the learning rate 0.0001 for both the meta-controller and the sub-controller. The algorithms update the parameter at each five time steps and use a soft-update of 0.001 to update the parameters of target networks. The size of minibatch is 12 and 4 for meta-controller and sub-controller respectively. The algorithm uses a discount factor of 0.99. The capacity of $\mathcal{M}^S$ and $\mathcal{M}^M$ is 5000 episodes and 1000 episodes, respectively. The configuration of networks is summarized in Table 4.5 (hDRQNv1) and Table 4.6 (hDRQNv2), and the parameters of the proposed algorithms are summarized in Table 4.7. The selected parameters of the proposed algorithms are optimal values which are chosen based on the performance of algorithms and the power of the computer (Table 4.8) which is used to do the experiments. The length of sampled transitions, n^S and n^M is varying depending on the specified domains. In section 4.2, we evaluate the effects of these parameters on the performance of algorithms. In addition, the level of observation also affects to the performance of the proposed algorithms. So, we evaluate the impact of this parameter in section 4.3.

4.1.5 Environment Settings

We implement the algorithms using python (Python 2.7) code. The network part is constructed using Tensorflow v1.4 [93]. In addition, in the implementation code, we use OpenCV library to crop a full image into a cropped image ans resize the images to fit with the input of controllers. The experiments are performed on a machine which has the configuration as shown in Table 4.8

Table 4.8: Environment settings

Name	Configuration
Python	v2.7
OpenCV	v3.4.0
Tensorflow	v1.4
Platform	Ubuntu
CPU	Intel Core i7
GPU	Nvidia 1080 8GB
Cuda	8.0
RAM	64GB
Hard disk	SSD 256

4.2 Experiment 1: Comparison on Different Length of Transition

4.2.1 Description

In the first evaluation, we use the domain of two goals in gridworld to compare each proposed algorithm with different lengths of selected transitions which are discussed in Section 3.4. The agent in this evaluation can observe an area of 5×5 units. We report the performance through three runs of 20000 episodes, where each episode has 50 steps. The number of steps for each episode assures that the agent can explore any location on the map.

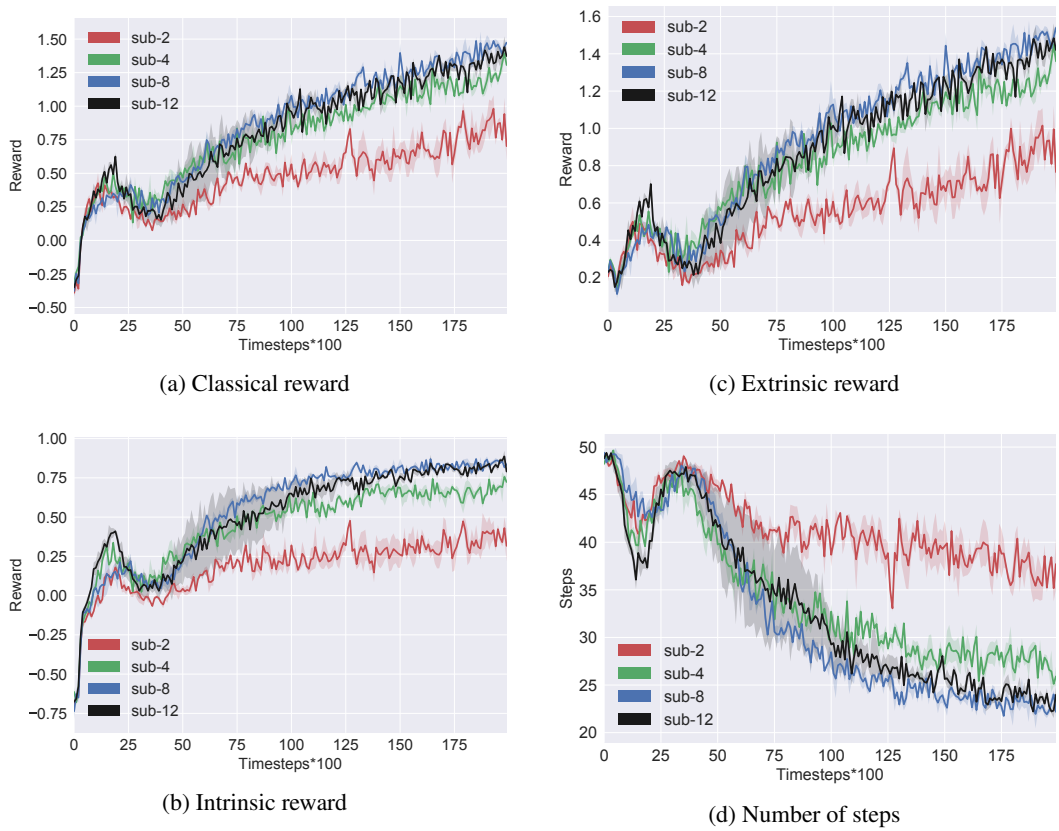
4.2.2 Results

Different lengths of sub-transitions on hDRQNv1 On hDRQNv1, we use a fixed length of meta-transitions ($n^M = 1$) and compare different lengths of sub-transitions. The report is shown in Figure 4.6. With a fixed length of meta-transitions, the algorithm performs well with a long length of sub-transition ($n^S = 8$ or $n^S = 12$). The performance decreases when the length of sub-transitions is decreased. The second observation is that with $n^S = 8$ or $n^S = 12$, there is little difference in performance. The average number of steps to obtain two goals in order is around 22.

Different lengths of sub-transitions on hDRQNv2 Similarly for hDRQNv2, we use a fixed length of meta-transitions ($n^M = 1$) and compare different lengths of sub-transitions. The report

is shown in Figure 4.7. Similarly, the algorithm performs well with a long length of sub-transition. $n^S = 8$ seems converging faster than $n^S = 12$

Different lengths of meta-transitions on hDRQNv1 and hDRQNv2 The report is shown in Figure 4.8. For a fixed length of sub-transitions ($n^S = 8$), with the hDRQNv1 algorithm, the setting with $n^M = 2$ has low performance and high variance compared to the setting with $n^M = 1$. Meanwhile, with the hDRQNv2 algorithm, the performance is the same at both settings $n^M = 1$ and $n^M = 2$. Meta-controller of hDRQNv2 has better performance than meta-controller of hDRQNv1. As a result, the performance of hDRQNv2 outperforms the performance of hDRQNv1.



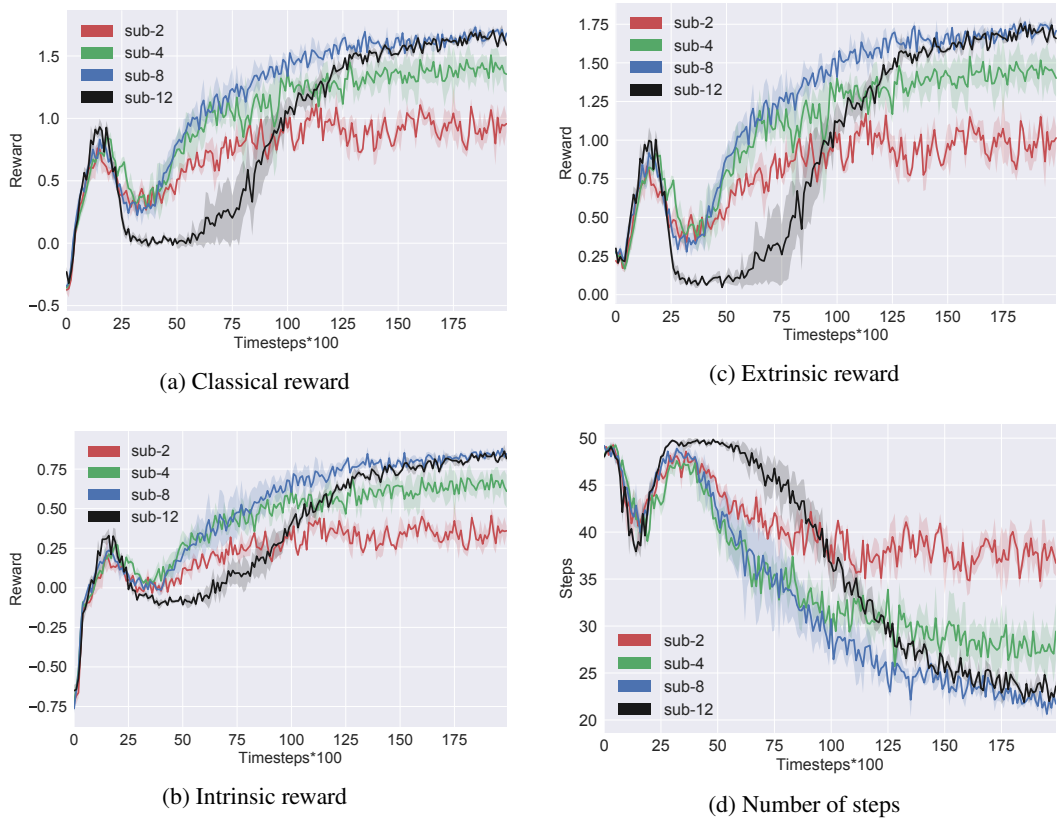
(a) Classical reward

(c) Extrinsic reward

(b) Intrinsic reward

(d) Number of steps

Figure 4.6: Evaluation of different lengths of sub-transitions on hDRQNv1. We use a fixed length of meta-transitions ($n^M = 1$)



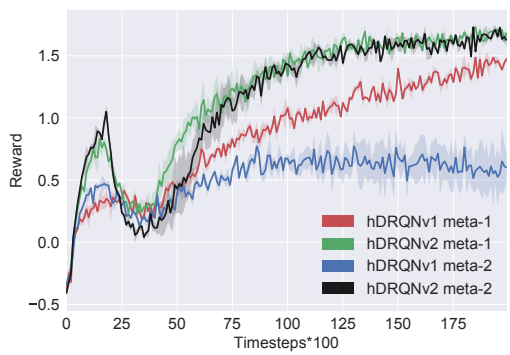
(a) Classical reward

(c) Extrinsic reward

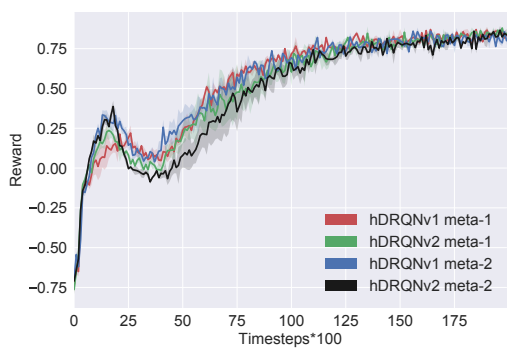
(b) Intrinsic reward

(d) Number of steps

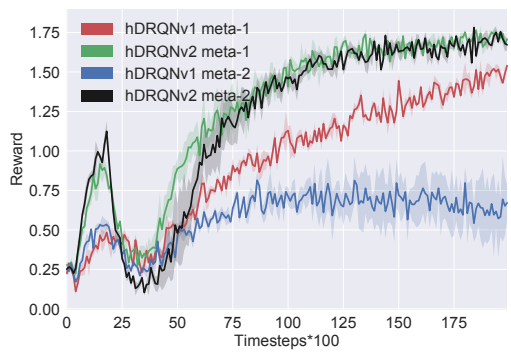
Figure 4.7: Evaluation of different lengths of sub-transitions on hDRQNv2. We use a fixed length of meta-transitions ($n^M = 1$)



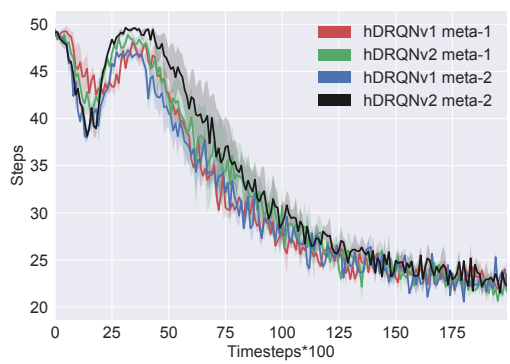
(a) Classical reward



(b) Intrinsic reward



(c) Extrinsic reward



(d) Number of steps

Figure 4.8: Evaluation of different lengths of meta-transitions on hDRQNv1 and hDRQNv2. We use a fixed length of sub-transitions ($n^S = 8$)

4.2.3 Discussion

Intuitively, the recurrent layer needs a sequence of transitions (E.g. histories) that is long enough to increase the probability that the agent will reach the subgoal within that sequence. In case of two goals in gridworld domain, they are $n^S = 8$ and $n^M = 1$. This is reasonable because only eight transitions of sub-controller are needed for the agent to reach any subgoals and one transition of meta-controller to complete a task of two subgoals. While sub-controller of hDRQNv1 and hDRQNv2 has the same performance, meta-controller of hDRQNv2 outperforms meta-controller of hDRQNv1. This indicates that the hidden states from the sub-controller are a better input to determine the subgoal rather than a raw observation.

4.3 Experiment 2: Comparison on Different Level of Observations

4.3.1 Description

In the second evaluation, we use the domain of two goals in gridworld to compare proposed algorithms at different levels of observation. Figure 4.9 shows the performance of hDRQN algorithms with a 3×3 observable agent compared with a 5×5 observable agent and a fully observable agent. We report the performance through three runs of 100000 episodes, where each episode has 50 steps.

4.3.2 Results

From the figure, the performance of a 5×5 observable agent using hDRQNv2 seems to converge faster than a fully observable agent. However, the performance of the fully observable agent surpasses the performance of 5×5 observable agent at the end. The 3×3 observable agents (both hDRQNv1 and hDRQNv2) have poor performance compare to the 5×5 observable agent and the fully observable agent. The 5×5 observable agent using hDRQNv2 has better performance than the fully observable agent using hDRQNv1.

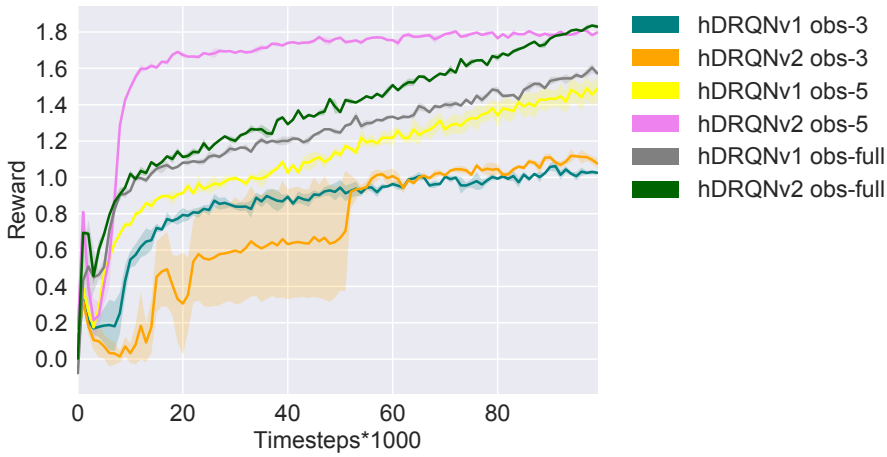


Figure 4.9: Evaluation on different levels of observation

4.3.3 Discussion

It is clear that a fully observable agent has more information around it than a 5×5 observable agent and a 3×3 observable agent; thus, the agent with a larger observation area can explore and localize the environment completely. As a result, the performance of the agent with a larger observation area is better than the agents with smaller observing abilities. However, even though a 5×5 observable agent using hDRQNv2 has a smaller observable area than a fully observable agent using hDRQNv1, the performance of hDRQNv2 based agent is better than the performance of hDRQNv1 based agent. It shows that using features constructed from a sequence of the partial observation (hDRQNv2) seems better than using features constructed from a full observation (hDRQNv1).

4.4 Experiment 3: Performance Comparison

4.4.1 Description

In the third evaluation, we compare the performance of the proposed algorithms with the well-known algorithms DQN, DRQN, and hDQN [44]. Both algorithms assume that the agent only observes an area of 5×5 units around it and use $n^M = 2$ and $n^S = 8$ for meta-transition and sub-transition, respectively. The performance of two goals in gridworld domain is averaged through

three runs of 20000 episodes and each episode has 50 time steps. Meanwhile, the performance of three goals in gridworld and two goals in four-rooms are averaged through three runs of 50000 episodes and each episode has 50 time steps. Finally, performance of three goals in four-rooms is averaged through three runs of 100000 episodes, and each episode has 100 time steps.

4.4.2 Results

The results of multiple goals in gridworld and in four-rooms are shown in Figure 4.10 and Figure 4.13, respectively. For both domains with two goals and three goals, the hDRQN algorithms outperform the other algorithms and hDRQNv2 has the best performance. The hDQN algorithm, which can operate in a hierarchical domain, is better than the flat algorithms but not better than the hDRQN algorithms.

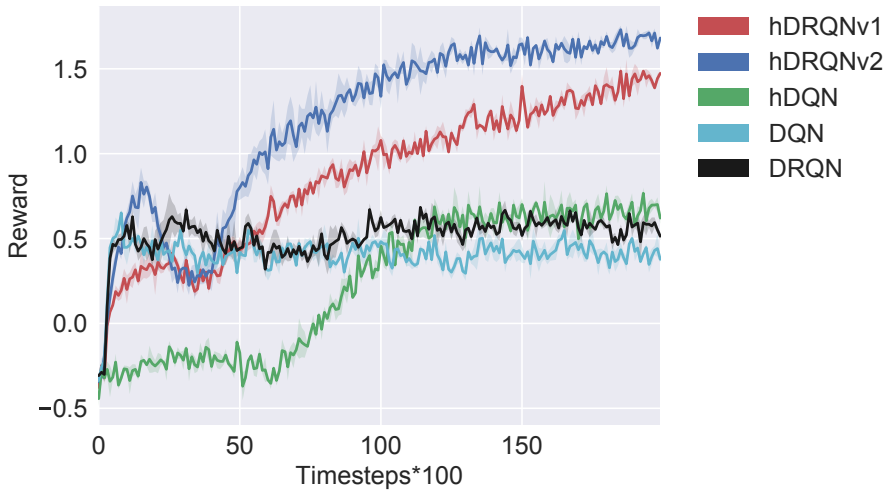
4.4.3 Discussion

The proposed algorithms have shown their abilities in solving hierarchical tasks under partial observability. Meanwhile, DQN and hDQN requires a sequence of full observation, and DRQN has problems with hierarchical tasks. There have no baselines algorithms can solve two problems (E.g. hierarchical task and partial observability) simultaneously.

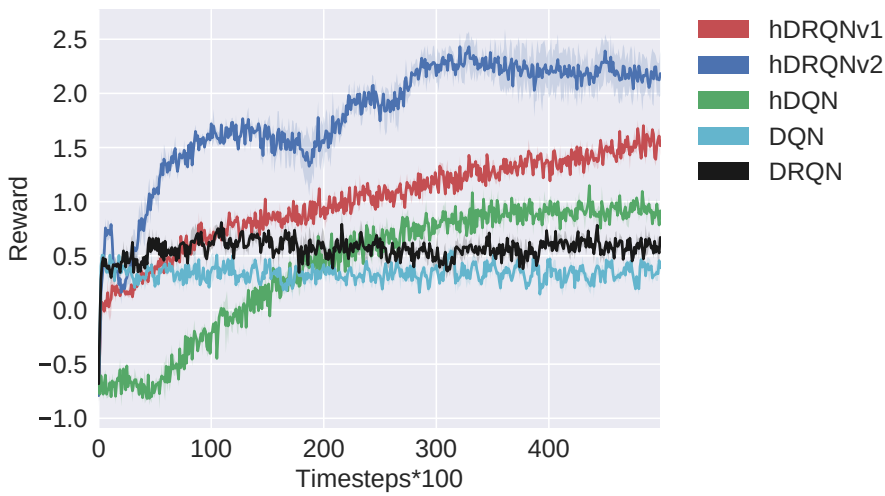
4.5 Experiment 4: Montezuma's Revenge

4.5.1 Description

The setting of sub-transition and meta-transition is 8 and 16, respectively. In this domain, the agent can observe an area of 70×70 pixels. The observation area is then resized to 44×44 to fit the input of the controller's network. Because learning the meta-controller and the sub-controllers simultaneously is highly complex and time-consuming, we separate the learning process into two phases. In the first phase, we learn the sub-controllers completely such that the agent can explore the whole environment by moving between subgoals. In the second phase, we learn the meta-controller and the sub-controllers altogether. The performance of the second phrase is averaged through three runs of 100000 episodes.

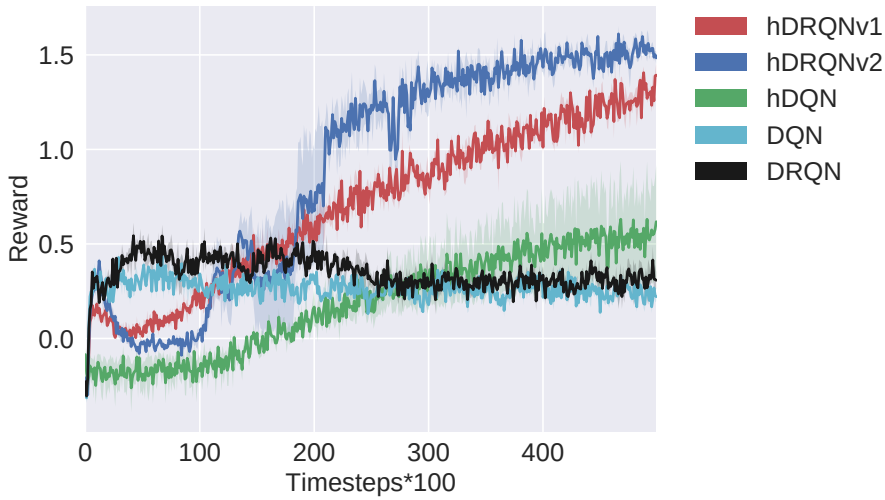


(a) Two goals in gridworld

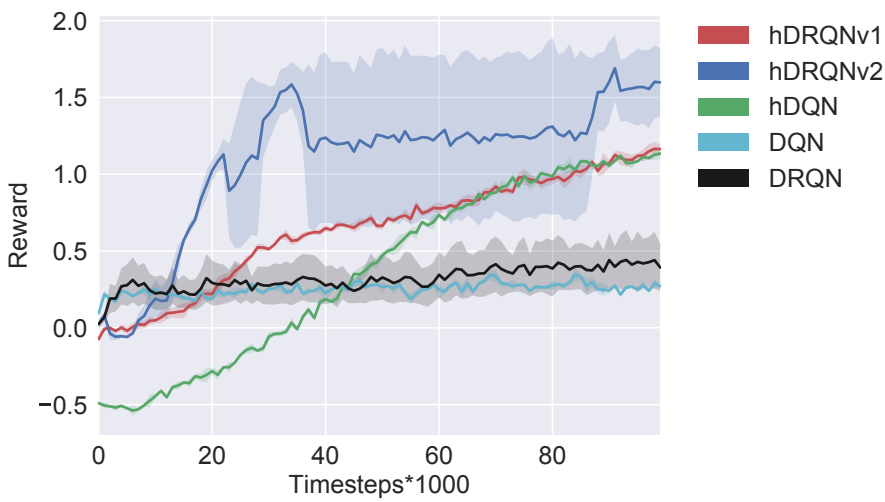


(b) Three goals in gridworld

Figure 4.10: Comparing the hDRQN algorithms with some baseline algorithms on the domains of multiple goals in gridworld



(a) Two goals in four-rooms



(b) Three goals in four-rooms

Figure 4.11: Comparing the hDRQN algorithms with some baseline algorithms on the domains of multiple goals in four-rooms

4.5.2 Results and Discussions

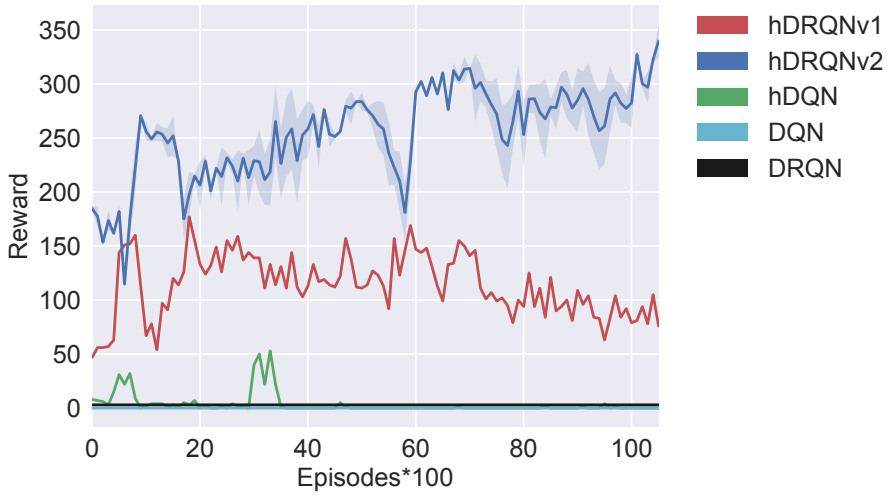
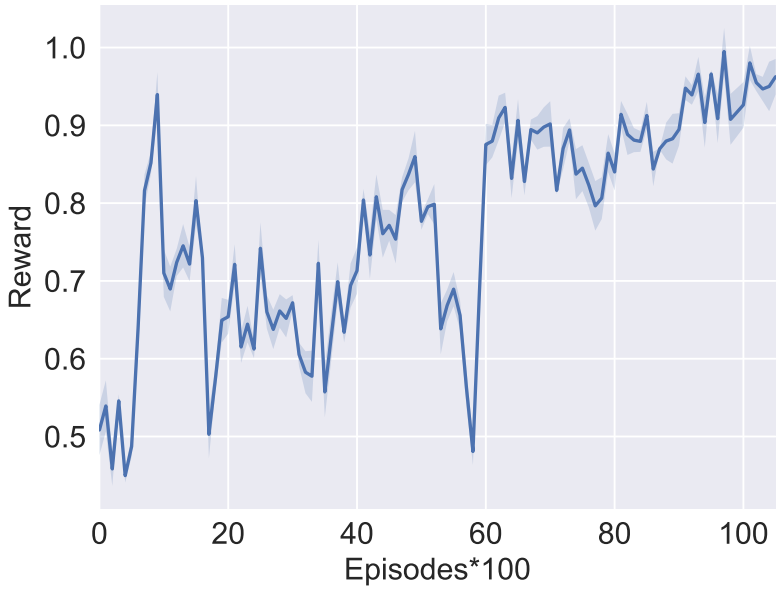


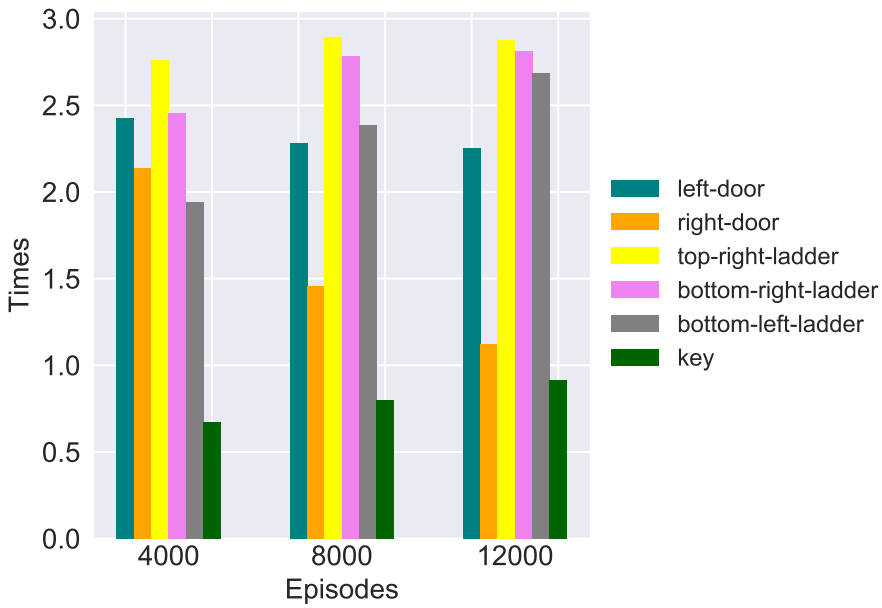
Figure 4.12: Comparing the hDRQN algorithms with some baseline algorithms for the Montezuma's Revenge game

The performance of the proposed algorithms compared to the baseline algorithms is shown in Figure 4.12. DQN reported a score of zero, which is similar to the result from [27]. DRQN, which can perform well in a partially observable environment, also achieved a score of zero because of the highly hierarchical complexity of the domain. hDQN can obtain scores on this domain. However, it cannot perform well in a partial observability setting. The performance of hDQN in a full observability setting can be found in the paper of Kulkarni [44]. Our proposed algorithms can adapt to the partial observability setting and hierarchical domains as well. The hDRQNv2 algorithm shows a better performance than hDRQNv1. It seems that the difference in the architecture of the two frameworks (described in Section 3.2) has affected their performance. Particularly, using internal states of a sub-controller as the input to the meta-controller can give more information for prediction than using only raw observations. To evaluate the effectiveness of two algorithms, we report the success ratio for reaching the goal "key" in Figure 4.13a and the number of time steps the agent explores each subgoal in Figure 4.13b. In Figure 4.13a, the agent using the hDRQNv2 algorithm almost picks up the "key" at the end of the learning process. Moreover, Figure 4.13b shows that hDRQNv2 tends to explore more often for subgoals that are on the way to reaching the "key" (E.g. top-right-ladder, bottom-right-ladder, and bottom-left-ladder)

while exploring less often for other subgoals such as the left door and the right door. Figure 4.14 demonstrates a sample gameplay of Montezuma's Revenge game and is interpreted as follows. At (1), the meta-controller picks top-right-ladder as subgoal and the sub-controller let the agent go to the current goal. After the agent reach top-right-ladder at (2), the meta-controller picks the next subgoal at (3) which is bottom-right-ladder. At (3), the sub-controller let the agent go down to the bottom-right-ladder. Similarly, the controller executes a sequence of picking subgoals and obtaining subgoals: (4)(5)(6)(7)(8)(9)(10)(11). The agent hits the skull at (11) and is died at (12), thus, the subdomain (the option) is terminated (12) and agent select another subgoal at (13). At (13), the meta-controller picks right-door as the subgoal and the sub-controller let the agent move to the right-door. The domain is completed after the agent reaches the right-door at (14).



(a) Success ratio for reaching the goal “key”



(b) The number of times the agent visits subgoals

Figure 4.13: Some metric on Montezuma’s Revenge

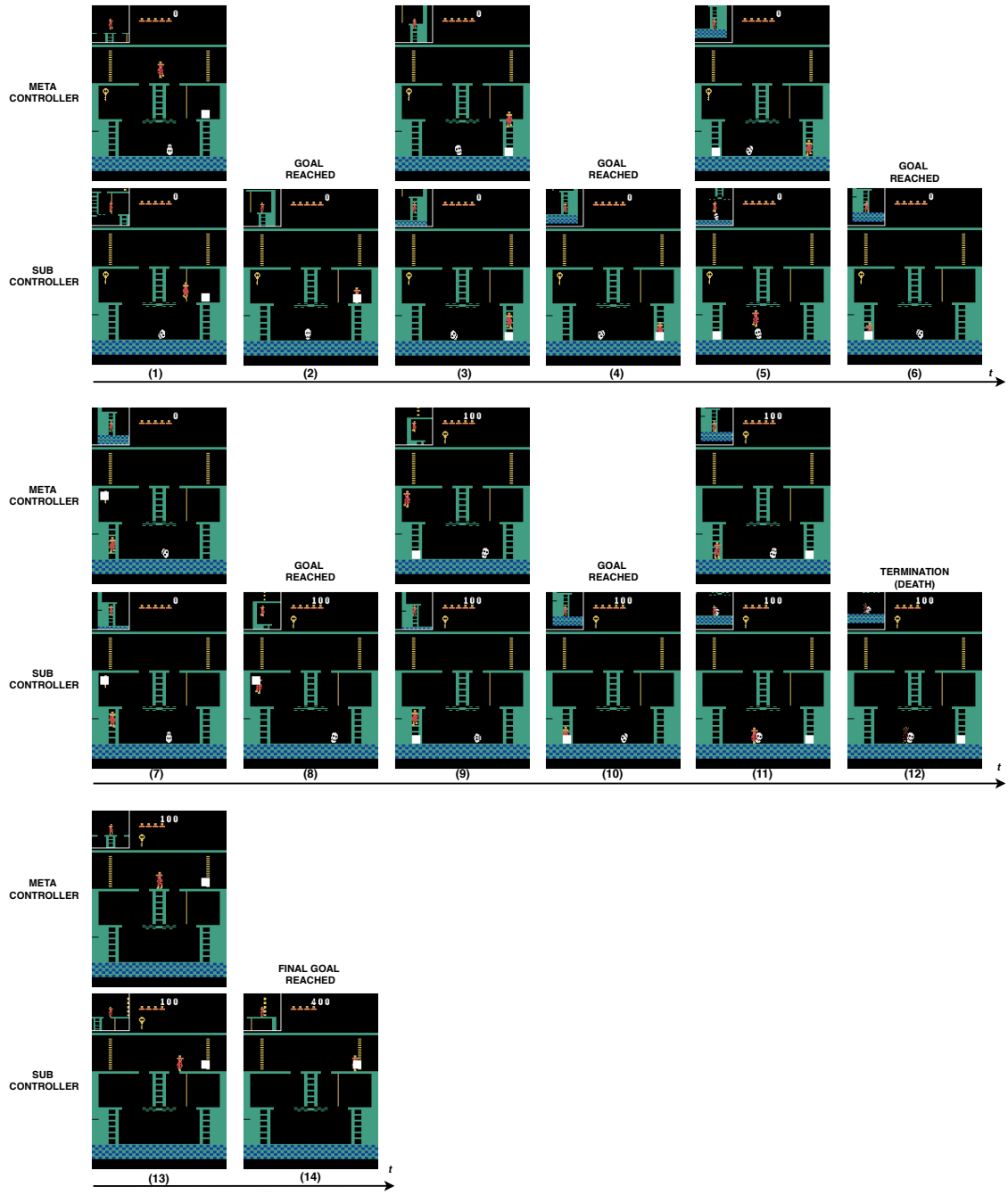


Figure 4.14: A gameplay of Montezuma's Revenge game

Chapter 5

Conclusion and Future Works

5.1 Conclusion

In the field of reinforcement learning, hierarchical reinforcement learning is still a challenging topic due to the complexity of domains. Moreover, hierarchical reinforcement learning has increased the challenge by evaluating hierarchical domains under partial observability. Over the last decade, many approaches are proposed to overcome the difficulty in HRL. However, they are still stuck on highly hierarchical domains due to simply representing policy. Recently, deep learning is applied to hierarchical reinforcement learning and got remarkable achievements. The deep neural network allowed to represent highly complex policy, is integrated to hierarchical reinforcement learning and is expected to solve highly complex domains.

In this dissertation, we introduced a new hierarchical deep reinforcement learning algorithm. The algorithms can solve Montezuma's Revenge game under partial observability. The algorithms have some features described as follows:

- Proposed algorithms learn frameworks for both full observability (MDP) and partial observability (POMDP).
- The algorithms take advantage of deep neural networks to produce hierarchical policies that can solve domains with a highly hierarchical nonlinearity
- The algorithms integrate LSTM allowed learning data efficiently and better convergence.
- The algorithms employ several advanced methods in deep reinforcement learning such as Double DQN and Dueling architecture. Their techniques have been proved to help better convergence.

We showed that the proposed frameworks perform well when learning in hierarchical POMDP environments.

5.1.1 Applicable areas of our proposed algorithms

Naturally, applicable domains of proposed algorithms appear in form of multiple subdomains. This section introduces several applicable areas of proposed algorithms as follows:

Games Computer games are used in many studies to evaluate hierarchical reinforcement learning algorithms. The context is that the agent with a limited vision must explore the environment and do some dependent tasks. Some examples such as:

- ***Minecraft***: The agent with limited vision explores the map from room 1, navigates to a block from room 2, picks up the block and place it to room 3 [94].

- ***Startcraft***: Our proposed algorithms can fit to some tasks such as CollectMineralsAndGas or BuildMarines which are defined and implemented by Goodle DeepMind [28]. The agent starts with a limited base and is rewarded for the total resources collected in a limited time. A successful agent must build more workers and expand to increase its resource collection rate.

- ***DotA 2***: The game has been actively developed for over a decade, with game logic implemented in hundreds of thousands of lines of code. This logic takes milliseconds per tick to execute, versus nanoseconds for Chess or Go engines. OpenAI is in progess to build agents which can reach the human level. They train their agent using PPO on 256 GPUs and 128,000 CPU cores [95]. This game appears in form of a complex hierarchical task which the agent (hero) does some subtasks simultaneously such as farming, navigating, and killing. So, we expect our proposed algorithms can fit with some scenarios of this games.

- ***League of Legends***: Similar to DotA 2, this game is a very complex video game. However, it can be considered as a hierarchical tasks and can be applied by hierarchical algorithms such as our proposed algorithms.

Robotics Robotics is another area to which we can apply our proposed algorithms. The robots with limited sensors only observe a limited range of a environment. Even though the robot can

observe the complete state, the noise or disturbance from physical sensor also forms the POMDP domains. Under such contexts, the agent needs to explore the environment and does some hierarchical tasks. Some real-applications are shown as bellow:

- ***Navigation Task***: the trajectorie is decomposed into multiple smaller trajectories and the robot completes smaller trajectories one-by-one until completed the whole trajectories [96].
- ***Reacher Task***: a manipulator automatically finds the best trajectory to reach a specified goal. We can use our proposed algorithms to control the manipulator so that the manipulator reach multiple subgoals which is on the way to the final goal [97].

5.2 Future Works

Even though the proposed algorithms show their efficiency with hierarchical domains under POMDP, they contain several drawbacks. Solving these drawbacks will open new directions for this topic. In addition, the proposed algorithms have the potential to be extended to other topics. The details of future works are explained in sections below:

5.2.1 Multiple Levels of Hierarchy

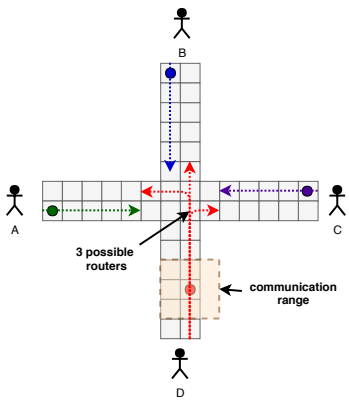
Developing a framework of two levels of controllers somehow limits the ability of the framework to solve hierarchical task having more than two levels of hierarchy. Most studies also propose the framework of two levels of policies in term of master/worker [40] [43] or policy/"policy over option" [37]. There has no study which discusses the number of levels of hierarchy. Therefore, developing a framework which can have an adaptive number of levels of controllers is expected to deal with any hierarchical domains.

5.2.2 Subgoal Discovery

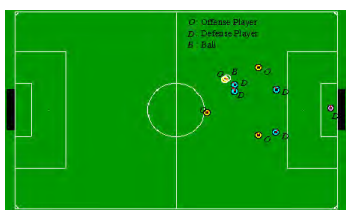
In order to simplify the learning problem in the hierarchical POMDP, we assumed that the set of subgoals is predefined and fixed because the problem of discovering a set of subgoals in POMDP is still a difficult problem. Assuming a set of predefined subgoals is not natural under partial observability because it requires a priori knowledge which breaks the assumption about partial

observability. Even though the domain is under full observability, selecting subgoals among states is still a hard problem due to the curve of dimensionality. Finding an algorithm, which allows to discovery and obtains subgoals simultaneously, is encouraged to remove the assumption about predefined subgoals. Some studies [39] [98] [46] [99] [100] have been first novel ideas the way to explore this topic.

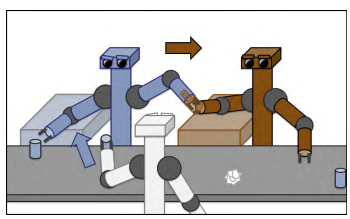
5.2.3 Hierarchical Multi-agent Domains under Partial Observability



(a) Multiple taxies co-operate to pick up and take off passengers



(b) Half Field Offense (A team of robots co-operates to score under the defense of another team)



(c) Multiple robots do a hierarchical tasks in a factory

Figure 5.1: Some hierarchical multi-agent domains

Proposed algorithms can be extended by applying it to hierarchical multi-agent problems under partial observability. For example, multiple taxies (Figure 5.1a) perform hierarchical tasks of picking up and taking off the passengers. However, each of them only observes and communicate with other taxies within a range (POMDP). We expect the hDRQNs combined with some communication strategies can help each agent efficiently cooperate with other agents to maximize the cumulative reward of the hierarchical tasks under partial observability. Some studies have focused on either a team of agent solves hierarchical tasks under full observability [98] [101] [102] [103] or a team of agent solves a single task under partial observability [104] [105] but there was no study that learns a team of agents to solve hierarchical tasks under partial observability. This is the change for us to propose great ideas.

Bibliography

[1] R. Sutton. (2009) Deconstructing reinforcement learning. [Online]. Available: http://videolectures.net/icml09_sutton_itdrl/

[2] V. R. Konda and J. N. Tsitsiklis, "Actor-critic algorithms," in *Advances in neural information processing systems (NIPS)*, 2000, pp. 1008–1014.

[3] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*, 1st ed. Cambridge, MA, USA: MIT Press, 1998.

[4] Wikipedia. (2018) Reinforcement learning. [Online]. Available: https://en.wikipedia.org/wiki/Reinforcement_learning

[5] M. P. Deisenroth, G. Neumann, J. Peters *et al.*, "A survey on policy search for robotics," *Foundations and Trends® in Robotics*, vol. 2, no. 1–2, pp. 1–142, 2013.

[6] J. Peters and S. Schaal, "Policy gradient methods for robotics," in *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*. IEEE, 2006, pp. 2219–2225.

[7] ——, "Natural actor-critic," *Neurocomputing*, vol. 71, no. 7, pp. 1180 – 1190, 2008, progress in Modeling, Theory, and Application of Computational Intelligence. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0925231208000532

[8] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, 2015, pp. 1889–1897.

[9] P. Marbach, O. Mihatsch, M. Schulte, and J. N. Tsitsiklis, "Reinforcement learning for call admission control and routing in integrated service networks," in *Advances in Neural Information Processing Systems (NIPS)*, 1998, pp. 922–928.

[10] A. Pietrabissa, "A reinforcement learning approach to call admission and call dropping control in links with variable capacity," *European Journal of Control*, vol. 17, no. 1, pp. 89–103, 2011.

[11] H. Tong and T. X. Brown, "Reinforcement learning for call admission control and routing under quality of service constraints in multimedia networks," *Machine Learning*, vol. 49, no. 2-3, pp. 111–139, 2002.

[12] D. Ernst, M. Glavic, and L. Wehenkel, "Power systems stability control: reinforcement learning framework," *IEEE transactions on power systems*, vol. 19, no. 1, pp. 427–435, 2004.

[13] M. Glavic, R. Fonteneau, and D. Ernst, "Reinforcement learning for electric power system decision and control: Past considerations and perspectives," *IFAC-PapersOnLine*, vol. 50, no. 1, pp. 6918–6927, 2017.

[14] J. W. Lee, "Stock price prediction using reinforcement learning," in *Industrial Electronics, 2001. Proceedings. ISIE 2001. IEEE International Symposium on*, vol. 1. IEEE, 2001, pp. 690–695.

[15] J. Moody and M. Saffell, "Learning to trade via direct reinforcement," *IEEE transactions on neural Networks*, vol. 12, no. 4, pp. 875–889, 2001.

[16] Y. Deng, F. Bao, Y. Kong, Z. Ren, and Q. Dai, "Deep direct reinforcement learning for financial signal representation and trading," *IEEE transactions on neural networks and learning systems*, vol. 28, no. 3, pp. 653–664, 2017.

[17] R. S. Sutton, D. Precup, and S. Singh, "Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning," *Artificial intelligence*, vol. 112, no. 1-2, pp. 181–211, 1999.

[18] R. E. Parr, *Hierarchical control and learning for Markov decision processes*. University of California, Berkeley Berkeley, CA, 1998.

[19] R. Parr and S. J. Russell, "Reinforcement learning with hierarchies of machines," in *Advances in neural information processing systems (NIPS)*, 1998, pp. 1043–1049.

[20] T. G. Dietterich, "Hierarchical reinforcement learning with the maxq value function decomposition," *Journal of Artificial Intelligence Research*, vol. 13, no. 1, pp. 227–303, 2000.

[21] N. A. Vien, H. Q. Ngo, S. Lee, and T. Chung, "Approximate planning for bayesian hierarchical reinforcement learning," *Appl. Intell.*, vol. 41, no. 3, pp. 808–819, 2014.

[22] N. A. Vien and M. Toussaint, "Hierarchical monte-carlo planning," in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA.*, 2015, pp. 3613–3619.

[23] N. A. Vien, S. Lee, and T. Chung, "Bayes-adaptive hierarchical mdps," *Appl. Intell.*, vol. 45, no. 1, pp. 112–126, 2016.

[24] A. G. Barto and S. Mahadevan, "Recent advances in hierarchical reinforcement learning," *Discrete Event Dynamic Systems*, vol. 13, no. 4, pp. 341–379, 2003.

[25] K. P. Murphy, "A survey of pomdp solution techniques," U.C. Berkeley, Tech. Rep., 2000.

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

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

[28] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, J. Agapiou, J. Schrittwieser *et al.*, "Starcraft ii: A new challenge for reinforcement learning," *arXiv preprint arXiv:1708.04782*, 2017.

[29] N. Brown and T. Sandholm, "Superhuman ai for heads-up no-limit poker: Libratus beats top professionals," *Science*, p. eaao1733, 2017.

[30] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[31] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, p. 354, 2017.

[32] J. Kober and J. Peters, *Reinforcement Learning in Robotics: A Survey*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 579–610. [Online]. Available: https://doi.org/10.1007/978-3-642-27645-3_18

[33] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2121–2159, 2011.

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

[35] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[36] M. Bellemare, S. Srinivasan, G. Ostrovski, T. Schaul, D. Saxton, and R. Munos, "Unifying count-based exploration and intrinsic motivation," in *Advances in Neural Information Processing Systems (NIPS)*, 2016, pp. 1471–1479.

[37] P.-L. Bacon, J. Harb, and D. Precup, "The option-critic architecture," in *Proceedings of the 31th Conference on Artificial Intelligence (AAAI)*, 2017, pp. 1726–1734.

[38] R. Fox, S. Krishnan, I. Stoica, and K. Goldberg, "Multi-level discovery of deep options," *arXiv preprint arXiv:1703.08294*, 2017.

[39] S. Lee, S.-W. Lee, J. Choi, D.-H. Kwak, and B.-T. Zhang, "Micro-objective learning: Accelerating deep reinforcement learning through the discovery of continuous subgoals," *arXiv preprint arXiv:1703.03933*, 2017.

[40] Vezhnevets, A. Sasha, S. Osindero, T. Schaul, N. Heess, M. Jaderberg, D. Silver, and K. Kavukcuoglu., "Feudal networks for hierarchical reinforcement learning," in *International Conference on Machine Learning (ICML)*, 2017, pp. 3540–3549.

[41] I. P. Durugkar, C. Rosenbaum, S. Dernbach, and S. Mahadevan, "Deep reinforcement learning with macro-actions," *arXiv preprint arXiv:1606.04615*, 2016.

[42] K. Arulkumaran, N. Dilokthanakul, M. Shanahan, and A. A. Bharath, "Classifying options for deep reinforcement learning," *arXiv preprint arXiv:1604.08153*, 2016.

[43] P. Dayan and G. E. Hinton, "Feudal reinforcement learning," in *Advances in neural information processing systems (NIPS)*, 1993, pp. 271–278.

[44] T. D. Kulkarni, K. Narasimhan, A. Saeedi, and J. Tenenbaum, "Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation," in *Advances in Neural Information Processing Systems (NIPS)*, 2016, pp. 3675–3683.

[45] S. Sukhbaatar, A. Szlam, G. Synnaeve, S. Chintala, and R. Fergus, "Mazebase: A sandbox for learning from games," *arXiv preprint arXiv:1511.07401*, 2015.

[46] C.-C. Chiu and V.-W. Soo, "Subgoal identifications in reinforcement learning: A survey," in *Advances in Reinforcement Learning*. InTech, 2011.

[47] M. Stolle, "Automated discovery of options in reinforcement learning," Ph.D. dissertation, McGill University, 2004.

[48] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in neural information processing systems (NIPS)*, 2000, pp. 1057–1063.

[49] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, vol. 8, no. 3-4, pp. 229–256, 1992.

[50] J. Peters and S. Schaal, "Reinforcement learning of motor skills with policy gradients," *Neural networks*, vol. 21, no. 4, pp. 682–697, 2008.

[51] P. Dayan and G. E. Hinton, "Using expectation-maximization for reinforcement learning," *Neural Computation*, vol. 9, no. 2, pp. 271–278, 1997.

[52] J. Kober and J. R. Peters, "Policy search for motor primitives in robotics," in *Advances in neural information processing systems (NIPS)*, 2009, pp. 849–856.

[53] S. Levine and V. Koltun, "Guided policy search," in *Proceedings of the 30th International Conference on Machine Learning (ICML)*, 2013, pp. 1–9.

[54] D. E. Moriarty, A. C. Schultz, and J. J. Grefenstette, "Evolutionary algorithms for reinforcement learning," *Journal of Artificial Intelligence Research*, vol. 11, pp. 241–276, 1999.

[55] S. Whiteson and P. Stone, "Evolutionary function approximation for reinforcement learning," *Journal of Machine Learning Research*, vol. 7, no. May, pp. 877–917, 2006.

[56] N. A. Vien, P. Englert, and M. Toussaint, "Policy search in reproducing kernel hilbert space." in *International Joint Conferences on Artificial Intelligence (IJCAI)*, 2016, pp. 2089–2096.

[57] T. P. Le, V. A. Ngo, P. M. Jaramillo, and T. Chung, "Importance sampling policy gradient algorithms in reproducing kernel hilbert space," *Artificial Intelligence Review*, pp. 1–21, 2017.

[58] G. Lever and R. Stafford, "Modelling policies in mdps in reproducing kernel hilbert space," in *Artificial Intelligence and Statistics (AISTATS)*, 2015, pp. 590–598.

[59] J. Peters and S. Schaal, "Natural actor-critic," *Neurocomputing*, vol. 71, no. 7-9, pp. 1180–1190, 2008.

[60] S. Bhatnagar, R. S. Sutton, M. Ghavamzadeh, and M. Lee, "Natural actor–critic algorithms," *Automatica*, vol. 45, no. 11, pp. 2471–2482, 2009.

[61] C. C. White, "Procedures for the solution of a finite-horizon, partially observed, semi-markov optimization problem," *Operations Research*, vol. 24, no. 2, pp. 348–358, 1976.

[62] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra, "Planning and acting in partially observable stochastic domains," *Artificial intelligence*, vol. 101, no. 1, pp. 99–134, 1998.

[63] M. Hausknecht and P. Stone, "Deep recurrent q-learning for partially observable mdps," in *2015 AAAI Fall Symposium Series*, 2015.

[64] M. Egorov, "Deep reinforcement learning with pomdps," Stanford Unviersity, Tech. Rep., 2015.

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

[66] P. J. Werbos, "Beyond regression: New tools for prediction and analysis in the behavioral sciences," Ph.D. dissertation, Harvard University, 1974.

[67] M. D. Zeiler, "Adadelta: an adaptive learning rate method," *arXiv preprint arXiv:1212.5701*, 2012.

[68] F. Rosenblatt, "Principles of neurodynamics. perceptrons and the theory of brain mechanisms," CORNELL AERONAUTICAL LAB INC BUFFALO NY, Tech. Rep., 1961.

[69] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[70] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems (NIPS)*, 2012, pp. 1097–1105.

[71] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2015, pp. 1–9.

[72] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *European conference on computer vision (ECCV)*. Springer, 2014, pp. 818–833.

[73] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE transactions on neural networks*, vol. 5, no. 2, pp. 157–166, 1994.

[74] T. Mikolov, A. Joulin, S. Chopra, M. Mathieu, and M. Ranzato, "Learning longer memory in recurrent neural networks," *arXiv preprint arXiv:1412.7753*, 2014.

[75] S. Hochreiter, "The vanishing gradient problem during learning recurrent neural nets and problem solutions," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 6, no. 02, pp. 107–116, 1998.

[76] K. Cho, B. Merrienboer, C. Gulcehre, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," in *2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014.

[77] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the 30th Conference on Artificial Intelligence (AAAI)*, vol. 2, 2016, p. 5.

[78] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Freitas, "Dueling network architectures for deep reinforcement learning," *arXiv preprint arXiv:1511.06581*, 2015.

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

[80] S. Gu, T. Lillicrap, I. Sutskever, and S. Levine, "Continuous deep q-learning with model-based acceleration," in *Proceedings of the 33nd International Conference on Machine Learning (ICML)*, 2016, pp. 2829–2838.

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

[82] G. Lample and D. S. Chaplot, "Playing fps games with deep reinforcement learning," in *Proceedings of the 31th Conference on Artificial Intelligence (AAAI)*, 2017, pp. 2140–2146.

[83] C. Diuk, A. Cohen, and M. L. Littman, "An object-oriented representation for efficient reinforcement learning," in *Proceedings of the 25th international conference on Machine learning (ICML)*. ACM, 2008, pp. 240–247.

[84] R. M. Ryan and E. L. Deci, "Intrinsic and extrinsic motivations: Classic definitions and new directions," *Contemporary educational psychology*, vol. 25, no. 1, pp. 54–67, 2000.

[85] A. Stout, G. D. Konidaris, and A. G. Barto, "Intrinsically motivated reinforcement learning: A promising framework for developmental robot learning," *Computer Science Department Faculty Publication Series*, no. 41, 2005. [Online]. Available: https://scholarworks.umass.edu/cs_faculty_pubs/41

[86] A. BARTO, "Intrinsically motivated learning of hierarchical collections of skills," *Proc. ICDL-2004*, pp. 112–119, 2004.

[87] S. Singh, R. L. Lewis, A. G. Barto, and J. Sorg, "Intrinsically motivated reinforcement learning: An evolutionary perspective," *IEEE Transactions on Autonomous Mental Development*, vol. 2, no. 2, pp. 70–82, 2010.

[88] M. Frank, J. Leitner, M. Stollenga, A. Förster, and J. Schmidhuber, "Curiosity driven reinforcement learning for motion planning on humanoids," *Frontiers in neurorobotics*, vol. 7, p. 25, 2014.

[89] S. Mohamed and D. J. Rezende, "Variational information maximisation for intrinsically motivated reinforcement learning," in *Advances in neural information processing systems (NIPS)*, 2015, pp. 2125–2133.

[90] J. Schmidhuber, "Formal theory of creativity, fun, and intrinsic motivation (1990–2010)," *IEEE Transactions on Autonomous Mental Development*, vol. 2, no. 3, pp. 230–247, 2010.

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

[92] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 2010, pp. 249–256.

[93] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org.

[94] C. Tessler, S. Givony, T. Zahavy, D. J. Mankowitz, and S. Mannor, "A deep hierarchical approach to lifelong learning in minecraft." 2017.

[95] OpenAI, "Openai five," https://blog.openai.com/openai-five/, 2018.

[96] B. Bischoff, D. Nguyen-Tuong, I. Lee, F. Streichert, A. Knoll *et al.*, "Hierarchical reinforcement learning for robot navigation," in *Proceedings of the European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN 2013)*, 2013.

[97] F. Schnell, "Hierarchical reinforcement learning in robot control."

[98] Lau, Q. Peter, M. L. Lee, and W. Hsu, "Coordination guided reinforcement learning." in *International Foundation for Autonomous Agents and Multiagent Systems (AAMAS)*, vol. 1, 2012, pp. 215–222.

[99] MCGOVERN, "Automatic discovery of subgoals in reinforcement learning using diverse density." in *Proc. 18th Int. Conf. on Machine Learning, Williamstown, MA*, 2001, pp. 361–368.

[100] T. D. Kulkarni, A. Saeedi, S. Gautam, and S. J. Gershman, "Deep successor reinforcement learning." *arXiv preprint arXiv:1606.02396*, 2016.

[101] Lowe, Ryan, Y. Wu, A. Tamar, J. Harb, O. P. Abbeel, and I. Mordatch, "Multi-agent actor-critic for mixed cooperative-competitive environments." in *Advances in Neural Information Processing Systems (NIPS)*, 2017, pp. 6379–6390.

[102] Gupta, J. K., M. Egorov, and M. Kochenderfer, "Cooperative multi-agent control using deep reinforcement learning." in *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. Springer, 2017, pp. 66–83.

[103] Foerster, J. N., Y. M. Assael, N. de Freitas, and S. Whiteson, "Learning to communicate to solve riddles with deep distributed recurrent q-networks," *arXiv preprint arXiv:1602.02672*, 2016.

[104] Sukhbaatar, Sainbayar, and R. Fergus, "Learning multiagent communication with backpropagation." in *Advances in Neural Information Processing Systems (NIPS)*, 2016, pp. 2244–2252.

[105] Omidshafiei, Shayegan, J. Pazis, C. Amato, J. P. How, and J. Vian, "Deep decentralized multi-task multi-agent reinforcement learning under partial observability." in *International Conference on Machine Learning (ICML)*, 2017, pp. 2681–2690.

Appendix A

List of Publications

International Journal Papers

[1] **Tuyen P. Le**, Hoang Huu Viet, Sang Hyeok An, Seung Gwan Lee, Dong-Han Kim, and Tae Choong Chung, "Univector field method-based multi-agent navigation for pursuit problem in obstacle environments," in Journal of Central South University (SCIE, IF:0.761), 24(4):1002–1012, Apr 2017.

[2] **Tuyen P. Le**, Vien Anh Ngo, P Marlith Jaramillo, and TaeChoong Chung, "Importance sampling policy gradient algorithms in reproducing kernel hilbert space," in Artificial Intelligence Review (SCI, IF:3.814), pages 1–21, Oct 2017.

[3] **Tuyen P. Le**, Ngo Anh Vien and TaeChoong Chung, "A Deep Hierarchical Reinforcement Learning Algorithm in Partially Observable Markov Decision Processes," in IEEE Access (SCIE, IF:3.557), vol. 6, pp. 49089-49102, 2018.

[4] Hoang Huu Viet, Le Hong Trang, SeungGwan Lee, **Tuyen P. Le**, and TaeChoong Chung, "A shortlist-based bidirectional local search for the stable marriage problem," in Applied Intelligence (SCIE, IF:1.983), 2018 (Major Revision)

Domestic Journal Papers

[1] 최승윤, **Tuyen P. Le**, 정태충, "Deep Deterministic Policy Gradient 알고리즘을 응용한 자전거의 자율 주행 제어," in Convergence Security Journal. Sep 2018. (Accepted)

International Conference Papers

[1] **Tuyen P. Le** and T. Chung, "Controlling bicycle using deep deterministic policy gradient

algorithm," In 2017 14th International Conference on Ubiquitous Robots and Ambient Intelligence (URAI), pages 413–417, June 2017.

[2] **Tuyen P. Le**, Abu. Layek, Ngo Anh Vien, and TaeChoong Chung, "Deep reinforcement learning algorithms for steering an underactuated ship," In 2017 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI), pages 602–607, Nov 2017.

[3] **Tuyen P. Le**, Nguyen Dang Quang, SeungYoon Choi, and TaeChoong Chung. "Learning a Self-driving Bicycle Using Deep Deterministic Policy Gradient," 18th International Conference on Control, Automation and Systems (ICCAS 2018), Oct 2018.

[4] Viet-Hung Dang, Ngo Anh Vien, **Tuyen P. Le**, and TaeChoong Chung, "A functional optimization method for continuous domains," In Industrial Networks and Intelligent Systems, pages 254–265, 2018. Springer International Publishing

[5] M. A. Layek, N. Q. Thai, M. A. Hossain, N. T. Thu, **Tuyen P. Le**, A. Talukder, T. Chung, and E. N. Huh, "Performance analysis of h.264, h.265, vp9 and av1 video encoders," In 2017 19th Asia-Pacific Network Operations and Management Symposium (APNOMS), pages 322–325, Sept 2017.

Domestic Conference Papers

[1] **Tuyen P. Le** and TaeChoong Chung, "RLVisualizer: An application for Visualizing Trajectories of Reinforcement Learning Problem," pages 13–14. The Korea Contents Society, 2017.

[2] **Tuyen P. Le** and TaeChoong Chung, "Pleasure of Learning," ICCC International Digital Design Invitation Exhibition, :131–131, 2017.

[3] **Tuyen P. Le**, Marlith Jaramillo, CholJin Jong, Seung-yoon Choi, JinSeok Kim, Md. Abu Layek, and TaeChoong Chung, "A Non-parametric policy based Algorithm in Reproducing Kernel Hilbert Space," pages 892–893. KOREA INFORMATION SCIENCE SOCIETY, 2016.

[4] **Tuyen P. Le**, Seung-yoon Choi, JinSeok Kim, Md. Abu Layek, CholJin Jong, and Tae-Choong Chung, "Reinforcement Learning of Vehicle Agent and Art work Trial using the Learning Trajectories," pages 719–721. KOREA INFORMATION SCIENCE SOCIETY, 2017.

[5] **Tuyen P. Le**, Seung-yoon Choi, Abu Layek, and TaeChoong Chung, "Gathering Objects in Four-rooms Domain under Partially Observability," pages 865–867. KOREA INFORMATION SCIENCE SOCIETY, 2017.

[6] CholJin Jong, JinSeok Kim, Md. Abu Layek, **Tuyen P. Le**, Marlith Jaramillo, Seung-yoon Choi, and TaeChoong Chung, "Study of Sound Location Tracking Mobile Robot Using Lego Mindstorms," pages 1028–1029. KOREA INFORMATION SCIENCE SOCIETY, 2016.

[7] Seung-yoon Choi, **Tuyen P. Le**, Cheoljin Jeong, Jinseok Kim, Marlith Jaramillo, Md. Abu Layek, and TaeChoong Chung, "A Study of Sequential Workspace Management Approach for Autonomous Mobile Robot in Path Planning Problem," pages 1036–1038. KOREA INFORMATION SCIENCE SOCIETY, 2016.

[8] JinSeok Kim, CholJin Jong, Md. Abu Layek, **Tuyen P. Le**, Marlith Jaramillo, Seung-yoon Choi, and TaeChoong Chung, "Selected wireless mesh network model and architecture for a communication interruption in the fixed wireless environment," pages 1265–1267. KOREA INFORMATION SCIENCE SOCIETY, 2016.

[9] Md. Abu Layek, **Tuyen P. Le**, Marlith Jaramillo, JinSeok Kim, Jeong cheol jin, Eui-Nam Huh, Seung-yoon Choi, and TaeChoong Chung, "Compression Efficiency Of Text Images In Hangul And Other Languages," pages 777–779. KOREA INFORMATION SCIENCE SOCIETY, 2016.

[10] Md Alamgir Hossain, Ngo Thien Thu, **Tuyen P. Le**, Ashis Talukder, TaeChoong Chung, Eui-Nam Huh, Md Abu Layek, Ngo Quang Thai, "Performance Analysis of AV1 for Video Coding in Very Low Bit Rates," pages 118–120. KOREA INFORMATION SCIENCE SOCIETY, 2017.

[11] Md Alamgir Hossain, Ngo Thien Thu, **Tuyen P. Le**, Ashis Talukder, TaeChoong Chung, Eui-Nam Huh, Md Abu Layek, Ngo Quang Thai, "Analysis of the Effects of Timing Presets on the Performance of H.264/AVC and H.265/HEVC Video Encoders," pages 442–443. Korea Institute Of Communication Sciences, 2017.

[12] Minh N. H. Nguyen, **Tuyen P. Le**, Nguyen H. Tran, and Choong Seon Hong. "Deep Reinforcement Learning based Smart Building Energy Management," pages 871–873. KOREA INFORMATION SCIENCE SOCIETY, 2017.