A Deep Hierarchical Reinforcement Learning Algorithm in Partially Observable Markov Decision Processes

TUYEN P. LE [ID]1, NGO ANH VIEN2, AND TAECHOONG CHUNG1

1Artificial Intelligence Laboratory, Computer Science and Engineering Department, Kyung Hee University, Global Campus, Yongin 446-701, South Korea
2EEECS/ECIT, Queen's University Belfast, Belfast BT7 1NN, U.K.

Corresponding author: TaeChoong Chung (tcchung@khu.ac.kr)

ABSTRACT In recent years, reinforcement learning (RL) has achieved remarkable success due to the growing adoption of deep learning techniques and the rapid growth of computing power. Nevertheless, it is well-known that flat reinforcement learning algorithms are often have trouble learning and are even data-efficient with respect to tasks having hierarchical structures, e.g., those consisting of multiple subtasks. Hierarchical reinforcement learning is a principled approach that can tackle such challenging tasks. On the other hand, many real-world tasks usually have only partial observability in which state measurements are often imperfect and partially observable. The problems of RL in such settings can be formulated as a partially observable Markov decision process (POMDP). In this paper, we study hierarchical RL in a POMDP in which the tasks have only partial observability and possess hierarchical properties. We propose a hierarchical deep reinforcement learning approach for learning in hierarchical POMDP. The deep hierarchical RL algorithm is proposed for domains to both MDP and POMDP learning. We evaluate the proposed algorithm using various challenging hierarchical POMDPs.

INDEX TERMS Hierarchical deep reinforcement learning, partially observable MDP (POMDP), semi-MDP, partially observable semi-MDP (POSMDP).

I. INTRODUCTION

Reinforcement Learning (RL) [1] is a subfield of machine learning focused on learning a policy in order to maximize the total cumulative reward in an unknown environment. RL has been applied to robotics [2]–[5], economics [6]–[8], computer games [9]–[11] and other applications [12]–[14]. RL is divided into two approaches: value-based approach and policy-based approach [15]. A typical value-based approach tries to obtain an optimal policy by finding optimal value functions. The value functions are updated using the immediate reward and the discounted value of the next state. Some methods based on this approach are Q-learning, SARSA, and TD-learning [1]. In contrast, a policy-based approach directly learns a parameterized policy that maximizes the cumulative discounted reward. Some techniques used to search for optimal parameters of the policy include policy gradient [16], expectation maximization [17], and evolutionary algorithm [18]. In addition, hybrid approaches of value-based approach and policy-based approach are known as actor-critic approaches [19]. Recently, RL algorithms integrated with a deep neural network (DNN) [10] have achieved good performance, even better than human performance at some tasks such as Atari games [10] and the game Go [11]. However, obtaining good performance on a task consisting of multiple subtasks, such as Mazebase [20] and Montezuma's Revenge [9], is still a major challenge for flat RL algorithms. Hierarchical reinforcement learning (HRL) [21] was developed to tackle such problems.

HRL decomposes a RL problem into a hierarchy of subproblems (subtasks), each of which can itself be a RL problem. Identical subtasks can be gathered into one group and are controlled by the same policy. As a result, hierarchical decomposition represents the problem in a compact form and reduces the computational complexity. Various approaches to decompose the RL problem have been proposed, such as options [21], HAMs [22], [23], MAXQ [24], Bayesian HRL [25]–[27] and some other advanced techniques [28]. All approaches commonly consider semi-Markov decision

process [21] as a base theory. Recently, many studies have combined HRL with deep neural networks in different ways to improve the performance on hierarchical tasks [29]–[35]. Bacon *et al.* [30] 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 [33] also proposed a deep hierarchical framework inspired by a classical framework called feudal reinforcement learning [36]. Similarly, Kulkarni *et al.* [37] 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 [9] and games in Mazebase [20]. Other studies have tried to tackle more challenging problems in HRL such as discovering subgoals [38] and adaptively finding a number of options [39].

Though many studies have made great efforts in this topic, most of them 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 Markov decision process (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 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; alternatively using a recurrent neural network (RNN) [40] to summarize an observable history. Recently, there have been some studies that use deep RNNs to deal with learning in POMDP environments [41], [42].

In this paper, we develop a deep HRL algorithm for POMDP problems inspired by the deep HRL framework [37]. The agent in this framework makes decisions through a hierarchical policy of two levels. The top policy determines the subgoal to be achieved, while the lower-level policy performs primitive actions to achieve the selected subgoal. To learn in POMDP, we represent all policies using RNNs. The RNN for lower-level policies constructs an internal state to remember the whole states observed during the course of interaction with the environment. The top policy is a RNN to encode a sequence of observations during execution of a selected subgoal. We highlight our contributions as follows. First, we exploit the advantages of RNNs to integrate with hierarchical RL in order to handle learning for challenging and complex tasks in POMDP. Second, this integration leads to a new hierarchical POMDP learning framework that is more scalable and data-efficient.

The rest of the paper is organized as follows. Section II reviews the background knowledge such as the semi-Markov decision process, partially observable Markov decision process and deep reinforcement learning. Our contributions are described in Section III, which consists of two parts. The deep model part describes all elements in our algorithm and the learning algorithm part summarizes the algorithm in a generalized form. The usefulness of the proposed algorithms is demonstrated through POMDP benchmarks in Section IV. Finally, the conclusion is given in Section V.

II. BACKGROUND
In this section, we briefly review all underlying theories that the content of this paper relies on: Markov decision process, semi-Markov decision process for hierarchical RL, partially observable Markov decision process for RL, and deep reinforcement learning.

A. MARKOV DECISION PROCESS
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:

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

A policy of a RL algorithm is denoted by π, which is the probability of taking an action a given a state s: $\pi = \mathcal{P}(a|s)$, and the goal of an RL algorithm is to find an optimal policy $\pi*$ in order to maximize the expected total discounted reward as follows:

$$J(\pi) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)\right]. \tag{1}$$

B. SEMI-MARKOV DECISION PROCESS FOR HIERARCHICAL REINFORCEMENT LEARNING
Learning over different levels of policy is the main challenge for hierarchical tasks. The semi-Markov decision process (SMDP) [21], which is as an extension of MDP, was developed to deal with this challenge. In this theory, the concept of "options" is introduced as a type of temporal abstraction. An 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 introduced to select options. An option is executed as follows. First, when an option's policy π^ξ is taken, the action a_t is selected based on π^ξ. The environment then transitions 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, the agent can select a next option based on $\mu(\xi|s_{t+1})$. The total discounted reward

received by executing option ξ is defined as

$$\mathcal{R}_s^\xi = \mathbb{E}\left[\sum_{i=t}^{t+k-1} \gamma^{i-t} r^\xi(s_i, a_i)\right]. \tag{2}$$

The multi-time state transition model of option ξ [43], which is initiated in state s and terminates at state s' after k steps, has the formula

$$\mathcal{P}_{ss'}^\xi = \sum_{k=1}^\infty \mathcal{P}^\xi(s', k|s, \xi)\gamma^k. \tag{3}$$

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 a policy μ over options:

$$\mathcal{V}^\mu(s) = \sum_{\xi \in \Xi} \mu(\xi|s)\left[\mathcal{R}_s^\xi + \sum_{s'} \mathcal{P}_{ss'}^\xi \mathcal{V}^\mu(s')\right] \tag{4}$$

and the option-value function:

$$\mathcal{Q}^\mu(s, \xi) = \mathcal{R}_s^\xi + \sum_{s'} \mathcal{P}_{ss'}^\xi \sum_{\xi' \in \Xi} \mu(\xi'|s')\mathcal{Q}^\mu(s', \xi'). \tag{5}$$

Similarly, the corresponding optimal Bellman equations are as follows:

$$\mathcal{V}^*(s) = \max_{\xi \in \Xi}\left[\mathcal{R}_s^\xi + \sum_{s'} \mathcal{P}_{ss'}^\xi \mathcal{V}^*(s')\right] \tag{6}$$

$$\mathcal{Q}^*(s, \xi) = \mathcal{R}_s^\xi + \sum_{s'} \mathcal{P}_{ss'}^\xi \max_{\xi' \in \Xi} \mathcal{Q}^*(s', \xi'). \tag{7}$$

The optimal Bellman equation can be computed using synchronous value iteration (SVI) [21], which iterates the following steps for every state:

$$\mathcal{V}(s) = \max_{\xi \in \Xi}\left[\mathcal{R}_s^\xi + \sum_{s'} \mathcal{P}_{ss'}^\xi \mathcal{V}(s', \xi')\right]. \tag{8}$$

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

$$
\begin{aligned}
\mathcal{Q}(s, \xi) \leftarrow\ & \mathcal{Q}(s, \xi) \\
& + \alpha\left[r(s, \xi(s)) + \gamma^k \max_{\xi' \in \Xi} \mathcal{Q}(s', \xi') - \mathcal{Q}(s, \xi)\right],
\end{aligned} \tag{9}
$$

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 $\xi(s)$.

C. PARTIALLY OBSERVABLE MARKOV DECISION PROCESS IN REINFORCEMENT LEARNING

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. If the POMDP model is known, the optimal approach is to maintain a hidden state b_t called the belief state. The belief defines the probability of being in state s according to its history of actions and observations. Given a new action and observation, belief updates are performed using the Bayes rule [44] defined as follows:

$$b'(s') \propto \mathcal{Z}(s', a, o) \sum_{s \in \mathcal{S}} \mathcal{P}(s, a, s')b(s). \tag{10}$$

However, exact belief updates require a high computational cost and expensive memory [40]. Another approach is using a Q-learning agent with function approximation, which uses 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, an RNN is used to maintain a long-term state, as in [41]. Our model using RNNs at different levels of the hierarchical policies is expected to be advantageous in POMDP environments.

D. DEEP REINFORCEMENT LEARNING

Recent advances in deep learning [45] 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, radial basis network, or neural networks with few layers. 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 multiple neural layers such as a multilayer perceptron layer (MLP) or a convolutional layer (CONV). Multiple layers in DNNs can have a lot of parameters, allowing them to represent highly non-linear problems. Deep Q-Network (DQN) has been proposed recently by Google Deepmind [10], which opens a new era of deep reinforcement learning and has influenced most later studies in deep reinforcement learning. In term of the architecture, a Q-network parameterized by θ, e.g., $Q(s, a | \theta)$ 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 MLP layer, which has a number of outputs equal to 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:

$$Q(s, a) := Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s, a)). \tag{11}$$

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., $\hat{Q}(s, a|\theta')$, and is a copy of the main Q-network. The target Q-network is used to estimate the loss function as follows:

$$\mathcal{L} = \mathbb{E}\left[(y - Q(s, a|\theta))^2\right], \qquad (12)$$

where $y = r + \gamma \max_{a'} \hat{Q}(s', a'|\theta')$. 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.

Many other models based on DQNs have been developed, such as Double DQNs [46], Dueling DQN [47], and Priority Experiment Replay [48]. 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 [5], estimating the advantage function [49], or a mixed actor network and critic network [50].

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. Our proposed method uses RNNs not only for solving the POMDP domains but also for solving these domains in a hierarchical form of reinforcement learning.

III. HIERARCHICAL DEEP RECURRENT Q-LEARNING IN PARTIALLY OBSERVABLE SEMI-MARKOV DECISION PROCESS

In this section, we describe the hierarchical recurrent Q learning algorithm (hDRQN), our proposed framework. First, we describe the model of hDRQN and explain the method of learning in this model. Second, we describe some components of our algorithm 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. The setting of POSMDP [26], [43] 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 update given a taken option ξ, observation o and current belief b is defined as

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

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 [24] and Options [21] to describe our problem. We denote $Q^M(o_t, \xi|\theta^M)$ 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 or history at time t, b_t (we will use them interchangeably: $Q^M(b, \xi)$ or $Q^M(o, \xi|\theta^M)$).

The multi-time observation model of option ξ [43], which is initiated in belief b and observe o, has the following formula:

$$\mathcal{P}(o|b, \xi) = \sum_{k=1}^{\infty} \sum_{s'} \sum_{s} \gamma^k \mathcal{P}(s', o, k|s, \xi)b(s). \qquad (13)$$

Using the above formula, we can write the Bellman equation for the value function of the meta-controller policy π over options as follows:

$$\mathcal{V}^M(b) = \sum_{\xi} \pi(\xi|b)\left[\mathcal{R}_b^{\xi} + \sum_{o'} \mathcal{P}(o'|b, \xi)\mathcal{V}^M(b')\right], \qquad (14)$$

and the option-value function as

$$Q^M(b, \xi) = \mathcal{R}_b^{\xi} + \sum_{o'} \mathcal{P}(o'|b, \xi) \sum_{\xi' \in \Xi} \mu(\xi'|b')Q^M(b', \xi').$$
$$(15)$$

Similarly to the MAX-Q framework, the reward term R_b^{ξ} is the total reward collected by executing the option ξ and defined as $V^{\xi}(b)$. Its corresponding Q-value function is defined as $Q^S(b, a)$ (the value function for sub-controllers). In the use of RNN, we also denote $Q^S(b, a)$ as $Q^S(\{o_t, g_t\}, a|\theta^S)$, 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.

Our hierarchical frameworks are illustrated in Fig. 1. The framework in Fig. 1a is inspired by a related idea in [37]. The difference is that our framework is built on
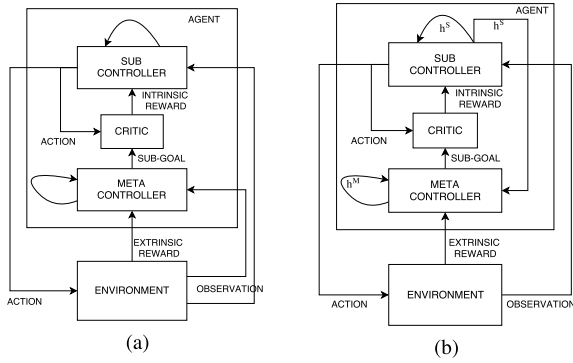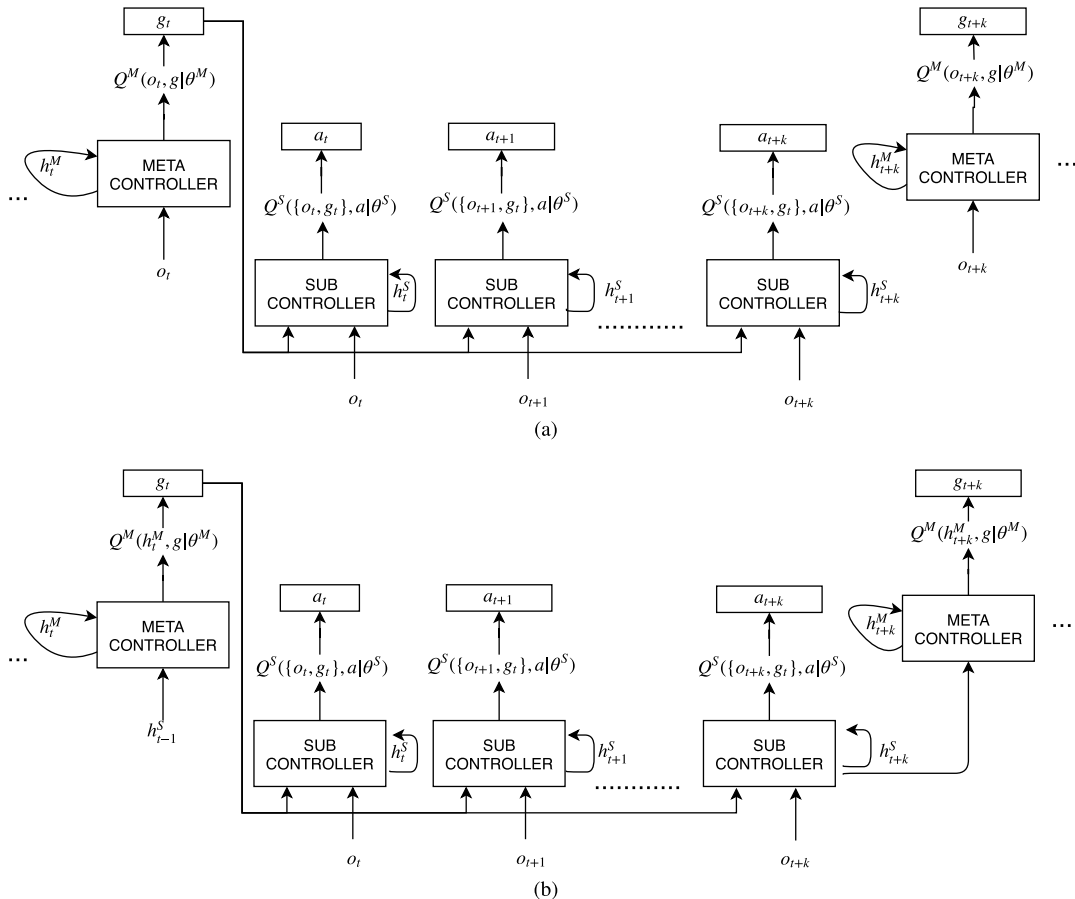
FIGURE 1. Hierarchical deep recurrent Q-network frameworks.
(a) Framework 1. (b) Framework 2.

two deep recurrent neural policies: a meta-controller and a sub-controller. The meta-controller is equivalent to a "policy over subgoals" that receives the current observation o_t and determines the new subgoal g_t. The sub-controller is equivalent to the option's policy, which directly interacts with the environment by performing action a_t. The sub-controller receives both g_t and o_t as inputs to the deep neural network. Each controller contains an arrow (Fig. 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. In contrast to the framework in Fig. 1a, the framework in Fig. 1b 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.

As mentioned in the previous section, RNNs are used in our framework to enable learning in POMDP. Particularly, CNNs are used to learn low-dimensional representations for image inputs, while a RNN is used to capture the temporal relations among the observation data. Without using a recurrent layer as in RNNs, CNNs cannot accurately approximate the state features from observations in the POMDP domain. The procedure of the agent is clearly illustrated in Fig. 2. The meta-controller and sub-controller use the Deep Recurrent Q Network (DRQN), as described in [41]. Particularly, at step t, the meta-controller takes an observation o_t from the environment (framework 1) or the last hidden state of the sub-controller generated by the previous sequence of observations (framework 2), extracts state features through some deep

FIGURE 2. Schematic of Hierarchical Deep Recurrent Q Learning (hDRQN); the framework is based on [37]. (a) Framework 1.
(b) Framework 2.

neural layers, internally constructs a hidden state h_t^M, and produces the Q subgoal values $Q^M(o_t, g_t|\theta^M)$. The Q subgoal values then are used to determine the 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 Q action values $Q^S(\{o_t, g_t\}, a_t|\theta^S)$, which are used to determine the next action a_{t+1}. Those explanations are formalized into the following equations:

$$\Phi^M = \begin{cases} f^{extract}(o_t) & \text{framework 1} \\ f^{extract}(h^S) & \text{framework 2} \end{cases} \quad (16)$$

$$h_t^M, Q^M(o_t, g_t|\theta^M) = f^M(\Phi^M, h_{t-1}^M) \quad (17)$$

$$\Phi^S = f^{extract}(o_t, g_t) \quad (18)$$

$$h_t^S, Q^S(\{o_t, g_t\}, a_t|\theta^S) = f^S(\Phi^S, h_{t-1}^S), \quad (19)$$

where Φ^M and Φ^S are the features after the extraction process of the meta-controller and sub-controller, respectively. f^M and f^S are the respective recurrent networks of the meta-controller and sub-controller, which receive the state features and a hidden state and then provide the next hidden state and Q value function.

The networks for the controllers are illustrated in Fig. 3. For framework 1, we use the networks demonstrated in Fig. 3a for both the meta-controller and sub-controller. A sequence of four CONV layers and ReLU layers interleaved together is used to extract information from the raw observations. A RNN layer, especially LSTM, is employed in front of the feature to memorize the information from previous observations. The output of the RNN layer is split

(a)

(b)

FIGURE 3. Network models. **(a) The network for framework 1. (b) Meta-controller network in framework 2.**

into Advantage stream A and Value stream V before being unified with the Q-value. This architecture is inherited from the Dueling architecture [47], which effectively learns states without having to learn the effect of an action on that state. For framework 2, we use the network in Fig. 3a for the sub-controller and use the network in Fig. 3b for the meta-controller. The meta-controller in framework 2 uses a state that is the internal hidden state of the sub-controller. By passing through four fully connected layers and ReLU layers, the features of the meta-controller are extracted. The rest of the network is the same as the network for framework 1.

A. LEARNING MODEL

We use a state-of-the-art Double DQN to learn the parameters of the network. Particularly, the controllers estimate the following Q value functions:

$$\begin{aligned} Q^M(\hat{o}_t, g_t) = Q^M(\hat{o}_t, g_t) \\ + \alpha(r^{ex} + \gamma^k \max_{g_{t+k}} Q^M(\hat{o}_{t+k}, g_{t+k}) \\ - Q^M(\hat{o}_t, g_t)) \end{aligned} \quad (20)$$

and

$$\begin{aligned} Q^S(\{o_t, g_t\}, a_t) = Q^S(\{o_t, g_t\}, a_t) \\ + \alpha(r^{in} + \gamma \max_{a_{t+1}} Q^S(\{o_{t+1}, g_t\}, a_{t+1}) \\ - Q^S(\{o_t, g_t\}, a_t)), \end{aligned} \quad (21)$$

where $\hat{o}$ can be a direct observation or an internal hidden state generated by the sub-controller. Let θ^M and θ^S be parameters (weights and bias) of the deep neural networks, which parameterize the networks $Q^M(\hat{o}_t, g_t)$ and $Q^S(\{o_t, g_t\}, a_t)$, correspondingly, e.g. $Q^M(\hat{o}_t, g_t) = Q^M(\hat{o}_t, g_t|\theta^M)$ and $Q^S(\{o_t, g_t\}, a_t) = Q^S(\{o_t, g_t\}, a_t|\theta^S)$. Then, Q^M and Q^S are trained by minimizing the loss function L^M and L^S, respectively. L^M can be formulated as:

$$L^M = \mathbb{E}_{(o,g,o',g',r^{ex})\sim\mathcal{M}^M}[y_i^M - Q^M(o, g|\theta^M)], \quad (22)$$

where $\mathbb{E}$ denotes the expectation over a batch of data which is uniformly sampled from an experience replay M^M, i is the iteration number in the batch of samples and

$$y_i^M = r^{ex} + \gamma Q^{M'}(o', \underset{g'}{\operatorname{argmax}} Q^M(o', g'|\theta^M)|\theta^{M'}). \quad (23)$$

Similarly, the formula of L^S is

$$L^S = \mathbb{E}_{(o,g,a,r^{in})\sim\mathcal{M}^S}[y_i^S - Q^S(\{o, g\}, a|\theta^S)], \quad (24)$$

where

$$y_i^S = r^{in} + \gamma Q^{S'}(\{o', g\}, \underset{a'}{\operatorname{argmax}} Q^S(\{o', g\}, a'|\theta^S)|\theta^{S'}). \quad (25)$$

$\mathcal{M}^S$ is the experience replay, which stores the transition data from the sub-controller, and $Q^{S'}$ is the target network of Q^S. Intuitively, in contrast to DQN, which uses the maximum operator for both selecting and evaluating an action, Double DQN separately uses the main Q network to greedily estimate the next action and the target Q network to estimate the
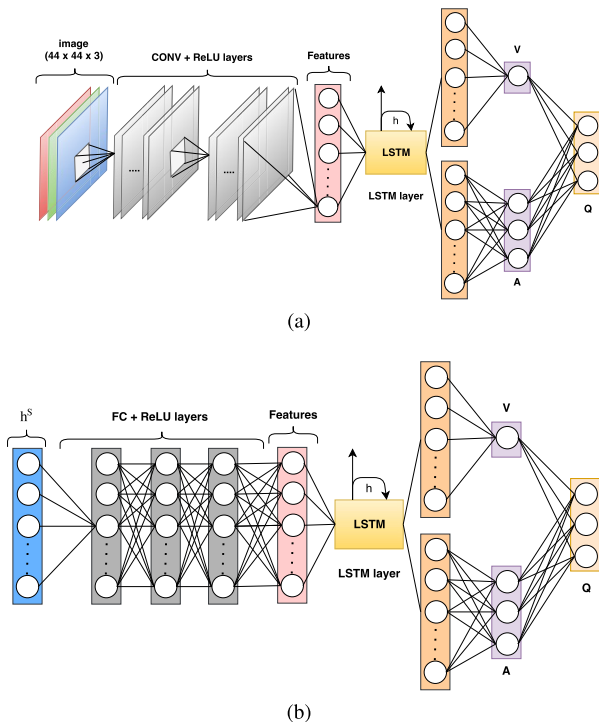
value function. This method has been shown to achieve better performance than DQN regarding Atari games [46].

B. MINIBATCH SAMPLING STRATEGY

For updating RNNs in our model, we need to analyze a sequence 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 [41], 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 [41], has been evaluated to achieve the same performance as Bootstrapped Sequential Updates while also reducing the computational complexity. The strategy is called Bootstrapped Random Updates. This strategy randomly selects a batch of episodes from the experience replay. Then, for each episode, we begin at a random transition and select a sequence of n transitions. The value of n, which affects to the performance of our algorithm, is analyzed in Section IV. We apply the same procedure of Bootstrapped Random Updates to our algorithm.

In addition, the mechanism explained in [51] is also applied. That study discusses a problem when updating DRQN: using the first observations in a sequence of transitions to update the 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.

C. 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. However, 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 [52], where entities or objects in the environment are decoded as subgoals.

D. INTRINSIC AND EXTRINSIC REWARD FUNCTIONS

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 motivation is based on psychology, cognitive science, and neuroscience [53] and has been applied to hierarchical RL [54]–[59]. Our framework follows the model of intrinsic motivation in [55]. Particularly, within an option (or skill), the agent needs to learn an option's policy (sub-controller in our framework) to obtain a subgoal (a salient event) under reinforcement of an intrinsic reward while for the overall task, a policy over options (meta-controller) is learned to generate a sequence of subgoals
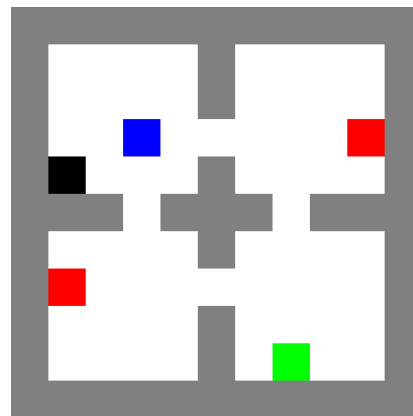

FIGURE 4. Example domain for illustrating the notions of intrinsic and extrinsic motivation.

under reinforcement of an extrinsic reward. Defining "good" intrinsic reward 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, Fig. 4 describes the domain of multiple goals in four-rooms, which is used to evaluate our algorithm in Section IV. 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 Section IV.

E. LEARNING ALGORITHM

In this section, our contributions are summarized through pseudo-code Algorithm 1. The algorithm learns four neural networks: two networks for meta-controller (Q^M and $Q^{M'}$) and two networks for sub-controller (Q^S and $Q^{S'}$). They are parameterized by θ^M, $\theta^{M'}$, θ^S and $\theta^{S'}$. The architectures of the networks are described in Section III. In addition, the algorithm separately maintains two experience replay

Algorithm 1 hDRQN in POMDP

Require:
1: POMDP $M = \{\mathcal{S}, \mathcal{A}, \mathcal{P}, r, \Omega, \mathcal{Z}\}$
2: Meta-controller with the network Q^M (main) and $Q^{M'}$ (target) parameterized by θ^M and $\theta^{M'}$, respectively.
3: Sub-controller with the network Q^S (main) and $Q^{S'}$ (target) parameterized by θ^S and $\theta^{S'}$, respectively.
4: Exploration rate ϵ^M for meta-controller and ϵ^S for sub-controller.
5: Experience replay memories M^M and M^S of meta-controller and sub-controller, respectively.
6: A pre-defined set of subgoals $\mathcal{G}$.
7: f^M and f^S are recurrent networks of meta-controller and sub-controller, respectively.

Ensure:
8: **Initialize:**
- Experiences replay memories M^M and 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

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

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

Require:
1: x: input of the Q network
2: h: internal hidden states
3: $\mathcal{B}$: a set of outputs
4: ϵ: exploration rate
5: Q network and recurrent function f

Ensure:
6: $h \leftarrow f(x, h)$
7: **if** $random() < \epsilon$ **then**
8:   $o \leftarrow$ An element from the set of output $\mathcal{B}$
9: **else**
10:   $o = \text{argmax}_{m \in \mathcal{B}} Q(x, m)$
11: **end if**
12: **Return** o, h

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

Require:
1: M^M: experience replay memory of meta-controller

Ensure:
2: **Sample** a mini-batch of $\{o, g, r^{ex}, o'\}$ from M^M as the strategy explained at III-B
3: **Update the network** by minimizing the loss function:

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

where

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

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

---

memories M^M and M^S to store transition data from the meta-controller and 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 loops 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; 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 M^S (Line 19) for updating the sub-controller. Meanwhile, the extrinsic reward is directly received from the environment and is stored in M^M for updating the meta-controller (Line 24). Updating the controllers at

Line 20 and 21 is described in Section III-A and is summarized in Algorithm 3 and Algorithm 4.

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

Require:
1: M^S: experience replay memory of meta-controller
2: **Sample** a mini-batch of $\{\{o, g\}, a, r^{in}, \{o', g'\}\}$ from M^S as the strategy explained at III-B
3: **Update the main network** by minimizing the loss function:

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

where

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

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

IV. EXPERIMENTS

In this section, we evaluate two versions of the hierarchical deep recurrent network algorithm. 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 benchmark the proposed algorithm. Finally, one of the most challenging games in ATARI 2600 [9], called Montezuma's Revenge, is used to confirm the efficiency of our proposed framework.

A. IMPLEMENTATION DETAILS

We use Tensorflow [60] to implement our algorithms. 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 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 observed around the agent (either 3×3 unit or 5×5 unit). The image feature of 256 values extracted through four CNNs and ReLUs is put into a LSTM layer of 256 states to generate 256 output values, and an internal hidden state of 256 values is also constructed. For the hDRQNv2 algorithm, a hidden state of 256 values is put into the network of the meta-controller. The state is passed through four fully connected layers and ReLU layers instead of four CONV layers. The output is a feature of 256 values. The algorithm uses ADAM [61] for learning the neural network parameters with the learning rate 0.001 for both the meta-controller network and the sub-controller. To update the target network, a τ value of 0.001 is applied. The algorithm uses a discount factor of 0.99. The capacity of M^S and M^M is $1E5$ and $5E4$, respectively.
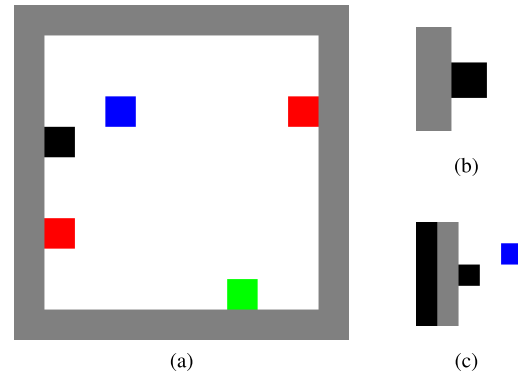
FIGURE 5. Multiple goals in gridworld. (a) Multiple goals in gridworld. (b) 3 × 3 unit. (c) 5 × 5 unit.

B. MULTIPLE GOALS IN A GRIDWORLD

The domain of multiple goals in a gridworld is a simple form of multiple goals in four-rooms, which is described in Section III-D. In this domain, a gridworld map of size 11×11 units is used instead of the four-rooms map. At each time step, the agent only observes a part of the surrounding environment, either a 3×3 units (Fig. 5b) or 5×5 units (Fig. 5c). The agent is allowed to choose one of four actions (top, left, right, bottom), which are deterministic. The agent can not 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:

$$r^{in} = \begin{cases} 1 & \text{obtain the goal} \\ -1 & \text{hit the obstacle} \end{cases} \tag{26}$$

and

$$r^{ex} = \begin{cases} 1 & \text{reach goals in order} \\ 0.01 & \text{otherwise} \end{cases} \tag{27}$$

The first evaluation reported in Fig. 6 is a comparison of the different lengths of selected transitions discussed in Section III-B. The agent in this evaluation can observe an area of 5×5. 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. In the figures on the left (hDRQNv1) and in the middle (hDRQNv2), we use a fixed length of meta-transitions ($n^M = 1$) and compare different lengths of sub-transitions. Otherwise, the figures on the right show the performance of the algorithm using a fixed length of sub-transitions ($n^S = 8$) and compare different lengths of meta-transitions. With a fixed length of meta-transition, 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. Intuitively, the RNN needs a sequence of transitions that is long enough
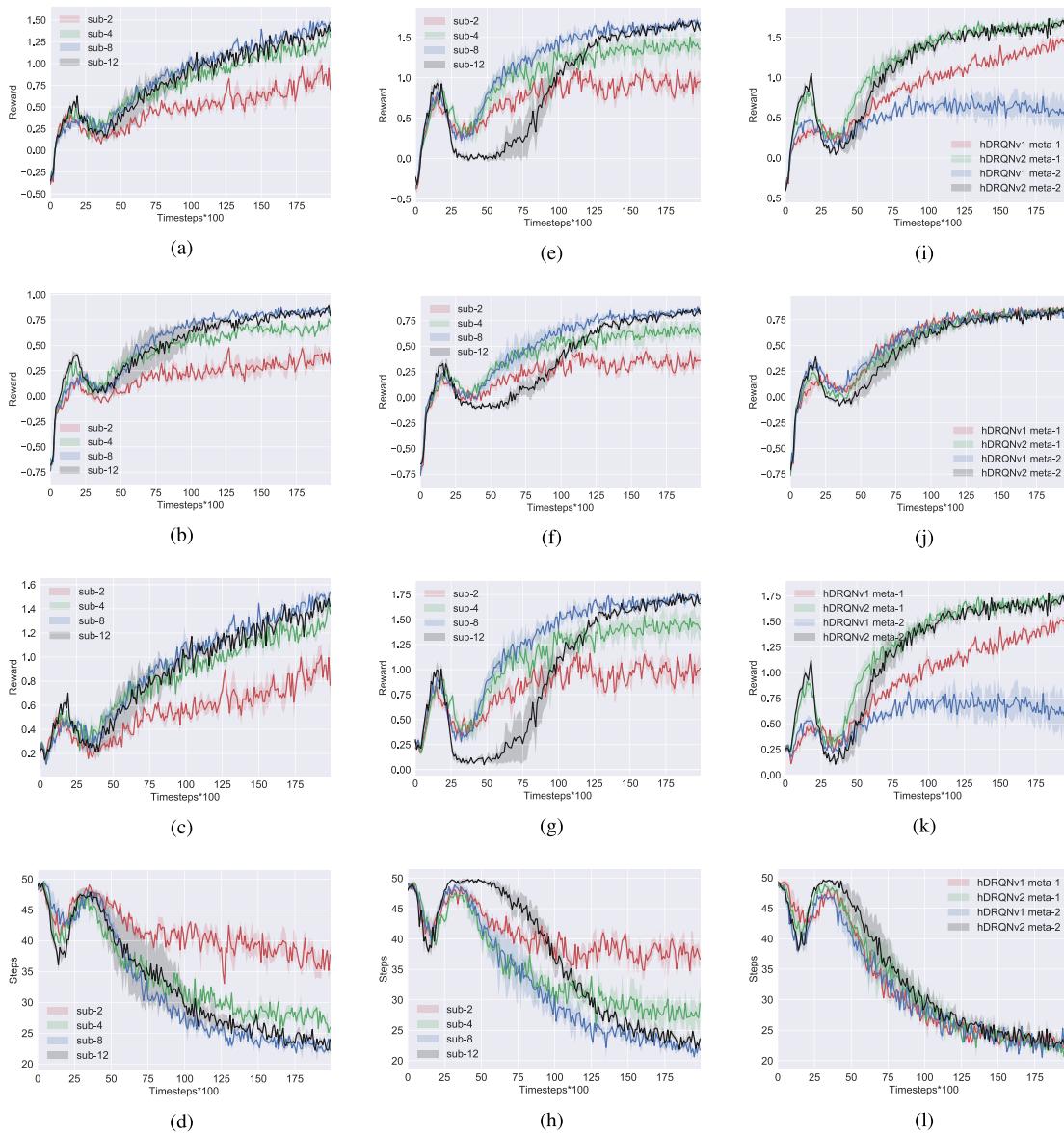
FIGURE 6. Evaluation of different lengths of transitions. In the left-side figures, we use a fixed length of meta-transitions ($n^M = 1$); the figures on the right use a fixed length of sub-transitions ($n^S = 8$). (a) hDRQNv1: reward. (b) hDRQNv1: intrinsic. (c) hDRQNv1: extrinsic. (d) hDRQNv1: steps. (e) hDRQNv2: reward. (f) hDRQNv2: intrinsic. (g) hDRQNv2: extrinsic. (h) hDRQNv2: steps. (i) Reward. (j) Intrinsic. (k) Extrinsic. (l) Steps.

to increase the probability that the agent will reach the sub-goal within that sequence. Another observation is that with $n^S = 8$ or $n^S = 12$, there is little difference in performance. This is reasonable because only eight transitions are needed for the agent to reach the subgoals. 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 $n^M = 1$. The reason is that while the sub-controller for the two settings has the same performance (Fig. 6f), the meta-controller with $n^M = 1$ performs better than the meta-controller with $n^M = 2$. Meanwhile, with the hDRQNv2 algorithm, the performance is the same at both settings $n^M = 1$ and $n^M = 2$. This means

that the hidden state from the sub-controller is a better input to determine the subgoal rather than a raw observation, as it causes the algorithm to not depend on the length of the meta-transition. The average number of steps to obtain two goals in order is around 22.

The next evaluation is a comparison at different levels of observation. Fig. 7 shows the performance of hDRQN algorithms with a 3 × 3 observable agent compared with a 5 × 5 observable agent and a fully observable agent. 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 quickly explore and localize the environment completely. As a result,
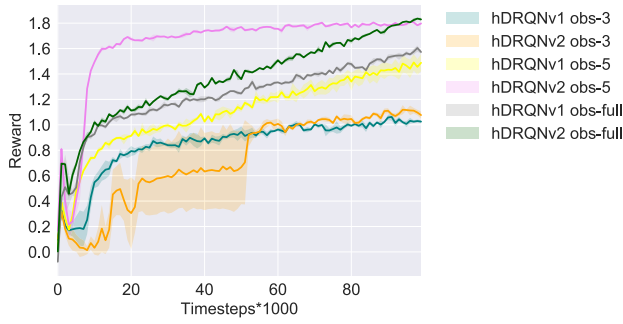
FIGURE 7. Evaluation on different levels of observation.

FIGURE 9. Multiple goals in the four-rooms domain. (a) The map of multiple goals in the four-rooms domain. (b) 3 × 3 unit. (c) 5 × 5 unit.

the performance of the agent with a larger observation area is better than the agents with smaller observing abilities. 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.

In the last evaluation of this domain, we compare the performance of the proposed algorithms with the well-known algorithms DQN, DRQN, and hDQN [37]. All algorithms assume that the agent only observes an area of 5 × 5 units around it. The results are shown in Fig. 8. For both domains with two goals and three goals, the hDRQN algorithms out-
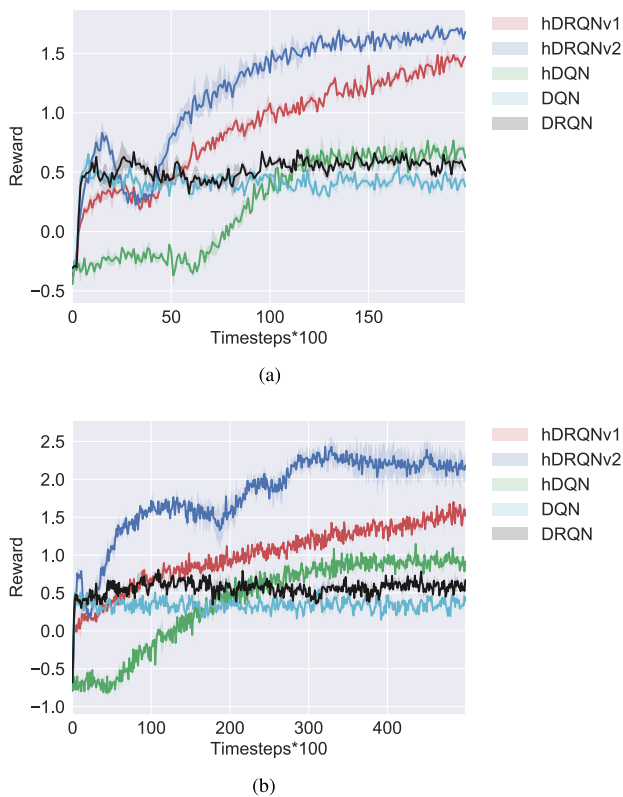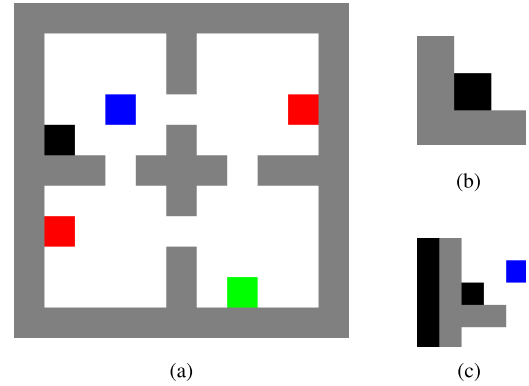
perform the other algorithms and hDRQNv2 has the best performance. The hDQN algorithm, which can operate in a hierarchical domain, is better than the flat algorithms but not better than the hDRQN algorithms. This may be because the hDQN algorithm is only for fully observable domains and has poor performance in partially observable domains.

C. MULTIPLE GOALS IN A FOUR-ROOMS DOMAIN
In this domain, we apply the multiple goals domain to a complex map called four-rooms (Fig. 9). The dynamics of
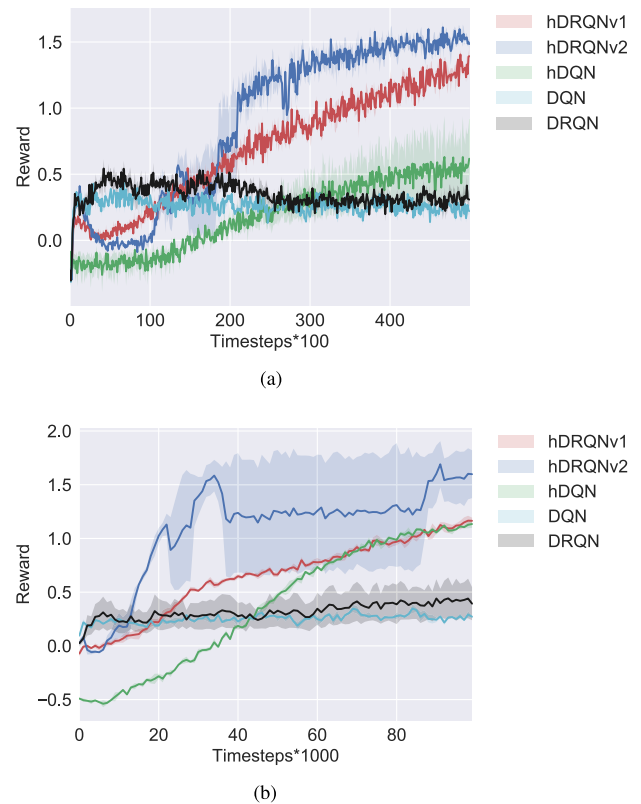

FIGURE 8. Comparing the hDRQN algorithms with some baseline algorithms. (a) Two goals in gridworld. (b) Three goals in gridworld.

FIGURE 10. Comparing the hDRQN algorithms with some baseline algorithms. (a) Two goals in four-rooms. (b) Three goals in four-rooms.

the environment in this domain is similar to that of the task in IV-B. The agent in this domain must usually pass through hallways to obtain goals that are randomly located in four rooms. Originally, the four-rooms domain was an environment for testing a hierarchical reinforcement learning algorithm [21].

The performance shown in Fig. 10a is averaged through three runs of 50000 episodes and each episode has 50 time steps. Meanwhile, the performance shown in Fig. 10b is averaged through three runs of 100000 episodes, and each episode has 100 time steps. As before, the proposed algorithms outperform other algorithms, especially, the hDRQNv2 algorithm.

D. MONTEZUMA's REVENGE GAME IN ATARI 2600

Montezuma's Revenge is one of the hardest games in ATARI 2600, and the DQN algorithm [10] can only achieve a score of zero. We use OpenAI Gym to simulate this domain [62]. 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 paper evaluates our proposed algorithms on the first screen of the game (Fig. 11). Particularly, the agent, which only observes a part of the environment (Fig. 11b), 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 transition to the next screen. The agent will earn 100 points after it obtains the key and 300 after it reaches any door. In total, the agent can receive 400 points for this screen.

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 Fig. 11a (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. 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 sub-controllers altogether. The architectures of the meta-controller and the sub-controllers are described in section IV-A. The length of sub-transition and meta-transition is 8 and 16, respectively. In this domain, the agent can observe an area of 70×70 pixels. Then, the observation area is resized to 44×44 to fit the input of a controller network.

The performance of the proposed algorithms compared to the baseline algorithms is shown in Fig. 12. DQN reported a score of zero, which is similar to the result from [10].
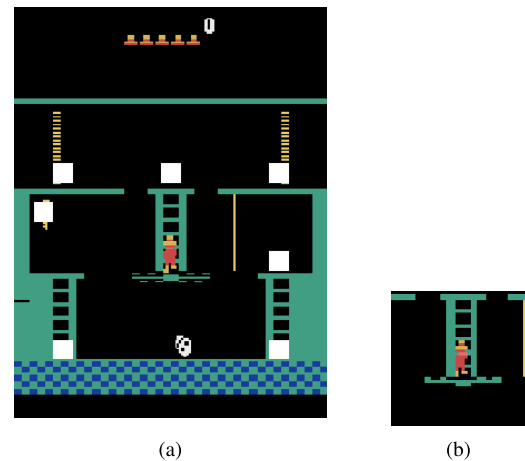

FIGURE 11. Montezuma's Revenge game in ATARI 2600. (a) The first screen of Montezuma's Revenge game. (b) Observable area.
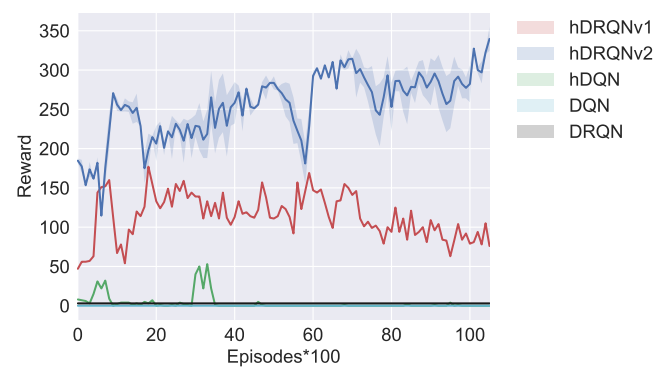

FIGURE 12. Comparing the hDRQN algorithms with some baseline algorithms for the Montezuma's Revenge game.
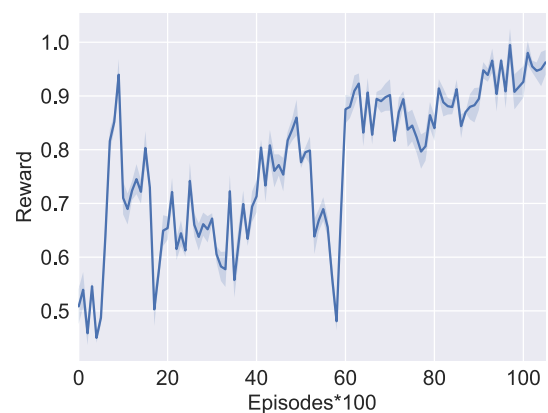

FIGURE 13. Success ratio for reaching the goal "key".

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. Meanwhile, hDQN can achieve a high score on this domain. However, it cannot perform well in a partial observability setting. The performance of hDQN in a full observability setting can
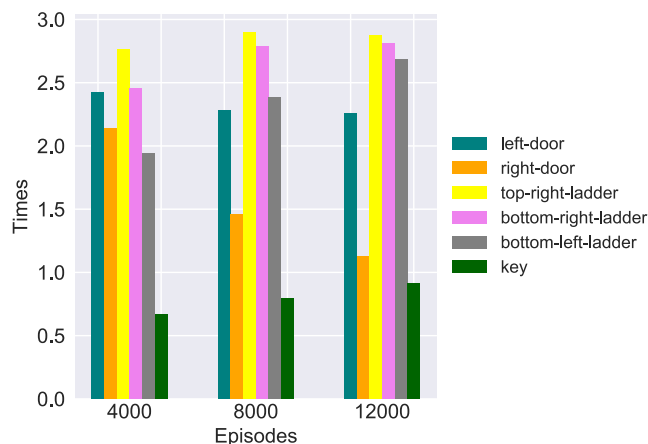
FIGURE 14. The number of times the agent visits subgoals.

be found in the paper of Kulkarni *et al.* [37]. 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 two frameworks (described in Section III) 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 the two algorithms, we report the success ratio for reaching the goal "key" in Fig. 13 and the number of time steps the agent explores each subgoal in Fig. 14. In Fig. 13, the agent using the hDRQNv2 algorithm almost picks up the "key" at the end of the learning process. Moreover, Fig. 14 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 right door.

V. CONCLUSION

We introduced a new hierarchical deep reinforcement learning algorithm that is a learning framework for both full observability (MDP) and partial observability (POMDP). The algorithm takes advantage of deep neural networks (DNN, CNN, LSTM) to produce hierarchical policies that can solve domains with a highly hierarchical nonlinearity. We showed that the framework performs well when learning in hierarchical POMDP environments. Nevertheless, our approach contains some limitations as follows. First, our framework is built on two levels of a hierarchy, which does not work for domains with multiple levels of hierarchy. Second, 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. In the future, we plan to improve our framework by tackling those problems. In addition, we can

apply DRQN to multi-agent problems where the environment is partially observable and the task is hierarchical.

REFERENCES

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

[2] M. P. Deisenroth, G. Neumann, and J. Peters, "A survey on policy search for robotics," *Found. Trends Robot.*, vol. 2, nos. 1–2, pp. 1–142, Aug. 2013.

[3] J. Peters and S. Schaal, "Policy gradient methods for robotics," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, Oct. 2006, pp. 2219–2225.

[4] J. Peters and S. Schaal, "Natural actor-critic," *Neurocomputing*, vol. 71, nos. 7–9, pp. 1180–1190, Mar. 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0925231208000532

[5] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *Proc. 32nd Int. Conf. Mach. Learn. (ICML)*, 2015, pp. 1889–1897.

[6] J. W. Lee, "Stock price prediction using reinforcement learning," in *Proc. IEEE Int. Symp. Ind. Electron. (ISIE)*, vol. 1, Jun. 2001, pp. 690–695.

[7] J. Moody and M. Saffell, "Learning to trade via direct reinforcement," *IEEE Trans. Neural. Netw.*, vol. 12, no. 4, pp. 875–889, Jul. 2001.

[8] Y. Deng, F. Bao, Y. Kong, Z. Ren, and Q. Dai, "Deep direct reinforcement learning for financial signal representation and trading," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 28, no. 3, pp. 653–664, Mar. 2017.

[9] V. Mnih *et al.* (2013). "Playing Atari with deep reinforcement learning." [Online]. Available: https://arxiv.org/abs/1312.5602

[10] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, 2015.

[11] D. Silver *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[12] G. Tesauro, D. C. Gondek, J. Lenchner, J. Fan, and J. M. Prager, "Simulation, learning, and optimization techniques in Watson's game strategies," *IBM J. Res. Develop.*, vol. 56, no. 3, pp. 423–433, 2012.

[13] G. Tesauro, "TD-gammon: A self-teaching backgammon program," in *Applications of Neural Networks*. Boston, MA, USA: Springer, 1995, pp. 267–285, doi: 10.1007/978-1-4757-2379-3_11.

[14] D. Ernst, M. Glavic, and L. Wehenkel, "Power systems stability control: Reinforcement learning framework," *IEEE Trans. Power Syst.*, vol. 19, no. 1, pp. 427–435, Feb. 2004.

[15] J. Kober and J. Peters, "Reinforcement learning in robotics: A survey," in *Reinforcement Learning*. Berlin, Germany: Springer, 2012, pp. 579–610, doi: 10.1007/978-3-642-27645-3_18.

[16] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Proc. Adv. Neural Inf. Process. Syst.*, 2000, pp. 1057–1063.

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

[18] D. E. Moriarty, A. C. Schultz, and J. J. Grefenstette, "Evolutionary algorithms for reinforcement learning," *J. Artif. Intell. Res.*, vol. 11, pp. 241–276, Sep. 1999.

[19] V. R. Konda and J. N. Tsitsiklis, "Actor-critic algorithms," in *Proc. Adv. Neural Inf. Process. Syst.*, 2000, pp. 1008–1014.

[20] S. Sukhbaatar, A. Szlam, G. Synnaeve, S. Chintala, and R. Fergus. (2015). "MazeBase: A sandbox for learning from games." [Online]. Available: https://arxiv.org/abs/1511.07401

[21] R. S. Sutton, D. Precup, and S. Singh, "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning," *Artif. Intell.*, vol. 112, nos. 1–2, pp. 181–211, 1999.

[22] R. E. Parr, "Hierarchical control and learning for Markov decision processes," Ph.D. dissertation, Dept. Comput. Sci., Univ. California, Berkeley, Berkeley, CA, USA, 1998.

[23] R. E. Parr and S. J. Russell, "Reinforcement learning with hierarchies of machines," in *Proc. Adv. Neural Inf. Process. Syst.*, 1998, pp. 1043–1049.

[24] T. G. Dietterich, "Hierarchical reinforcement learning with the MAXQ value function decomposition," *J. Artif. Intell. Res.*, vol. 13, pp. 227–303, Nov. 2000.

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

[26] N. A. Vien and M. Toussaint, "Hierarchical Monte-Carlo planning," in *Proc. 29th AAAI Conf. Artif. Intell.*, Austin, TX, USA, Jan. 2015, pp. 3613–3619.

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

[28] A. G. Barto and S. Mahadevan, "Recent advances in hierarchical reinforcement learning," *Discrete Event Dyn. Syst.*, vol. 13, nos. 1–2, pp. 41–47, 2003.

[29] M. Bellemare, S. Srinivasan, G. Ostrovski, T. Schaul, D. Saxton, and R. Munos, "Unifying count-based exploration and intrinsic motivation," in *Proc. Adv. Neural Inf. Process. Syst.*, 2016, pp. 1471–1479.

[30] P.-L. Bacon, J. Harb, and D. Precup, "The option-critic architecture," in *Proc. AAAI*, 2017, pp. 1726–1734.

[31] R. Fox, S. Krishnan, I. Stoica, and K. Goldberg. (2017). "Multi-level discovery of deep options." [Online]. Available: https://arxiv.org/abs/1703.08294

[32] S. Lee, S.-W. Lee, J. Choi, D.-H. Kwak, and B.-T. Zhang. (2017). "Micro-objective learning: Accelerating deep reinforcement learning through the discovery of continuous subgoals." [Online]. Available: https://arxiv.org/abs/1703.03933

[33] A. S. Vezhnevets *et al.* (2017). "Feudal networks for hierarchical reinforcement learning." [Online]. Available: https://arxiv.org/abs/1703.01161

[34] I. P. Durugkar, C. Rosenbaum, S. Dernbach, and S. Mahadevan. (2016). "Deep reinforcement learning with macro-actions." [Online]. Available: https://arxiv.org/abs/1606.04615

[35] K. Arulkumaran, N. Dilokthanakul, M. Shanahan, and A. A. Bharath. (2016). "Classifying options for deep reinforcement learning." [Online]. Available: https://arxiv.org/abs/1604.08153

[36] P. Dayan and G. E. Hinton, "Feudal reinforcement learning," in *Proc. Adv. Neural Inf. Process. Syst.*, 1993, pp. 271–278.

[37] T. D. Kulkarni, K. R. Narasimhan, A. Saeedi, and J. B. Tenenbaum, "Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation," in *Proc. Adv. Neural Inf. Process. Syst.*, 2016, pp. 3675–3683.

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

[39] M. Stolle, "Automated discovery of options in reinforcement learning," Ph.D. dissertation, School Comput. Sci., McGill Univ., Montreal, QC, Canada, 2004.

[40] K. P. Murphy, "A survey of POMDP solution techniques," Univ. California, Berkeley, Berkeley, CA, USA, Tech. Rep., 2000, vol. 2, p. X3.

[41] M. Hausknecht and P. Stone, "Deep recurrent Q-learning for partially observable MDPs," in *Proc. AAAI Fall Symp. Ser.*, 2015.

[42] M. Egorov, "Deep reinforcement learning with POMDPs," Stanford Unv., Stanford, CA, USA, Tech. Rep., 2015.

[43] C. C. White, "Procedures for the solution of a finite-horizon, partially observed, semi-Markov optimization problem," *Oper. Res.*, vol. 24, no. 2, pp. 348–358, 1976.

[44] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra, "Planning and acting in partially observable stochastic domains," *Artif. Intell.*, vol. 101, nos. 1–2, pp. 99–134, 1998.

[45] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016.

[46] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double Q-learning," in *Proc. AAAI*, vol. 2, 2016, p. 5.

[47] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas. (2015). "Dueling network architectures for deep reinforcement learning." [Online]. Available: https://arxiv.org/abs/1511.06581

[48] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. (2015). "Prioritized experience replay." [Online]. Available: https://arxiv.org/abs/1511.05952

[49] S. Gu, T. Lillicrap, I. Sutskever, and S. Levine, "Continuous deep Q-learning with model-based acceleration," in *Proc. Int. Conf. Mach. Learn.*, 2016, pp. 2829–2838.

[50] T. P. Lillicrap *et al.* (2015). "Continuous control with deep reinforcement learning." [Online]. Available: https://arxiv.org/abs/1509.02971

[51] G. Lample and D. S. Chaplot, "Playing FPS games with deep reinforcement learning," in *Proc. AAAI*, 2017, pp. 2140–2146.

[52] C. Diuk, A. Cohen, and M. L. Littman, "An object-oriented representation for efficient reinforcement learning," in *Proc. 25th Int. Conf. Mach. Learn.*, 2008, pp. 240–247.

[53] R. M. Ryan and E. L. Deci, "Intrinsic and extrinsic motivations: Classic definitions and new directions," *Contemp. Educ. Psychol.*, vol. 25, no. 1, pp. 54–67, 2000.

[54] A. Stout, G. D. Konidaris, and A. G. Barto, "Intrinsically motivated reinforcement learning: A promising framework for developmental robot learning," Comput. Sci. Dept. Fac. Pub. Ser., no. 41, 2005. [Online]. Available: https://scholarworks.umass.edu/cs_faculty_pubs/41

[55] A. G. Barto, S. Singh, and N. Chentanez, "Intrinsically motivated learning of hierarchical collections of skills," in *Proc. 3rd Int. Conf. Develop. Learn.*, 2004, pp. 112–119.

[56] S. Singh, R. L. Lewis, A. G. Barto, and J. Sorg, "Intrinsically motivated reinforcement learning: An evolutionary perspective," *IEEE Trans. Auton. Mental Develop.*, vol. 2, no. 2, pp. 70–82, Jun. 2010.

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

[58] S. Mohamed and D. J. Rezende, "Variational information maximisation for intrinsically motivated reinforcement learning," in *Proc. Adv. Neural Inf. Process. Syst.*, 2015, pp. 2125–2133.

[59] J. Schmidhuber, "Formal theory of creativity, fun, and intrinsic motivation (1990–2010)," *IEEE Trans. Auton. Mental Develop.*, vol. 2, no. 3, pp. 230–247, Sep. 2010.

[60] M. Abadi *et al.* (2015). "TensorFlow: Large-scale machine learning on heterogeneous distributed systems." [Online]. Available: https://arxiv.org/abs/1603.04467

[61] D. Kingma and J. Ba. (2014). "Adam: A method for stochastic optimization." [Online]. Available: https://arxiv.org/abs/1412.6980

[62] G. Brockman *et al.* (2016). "OpenAI Gym." [Online]. Available: https://arxiv.org/abs/1606.01540

TUYEN P. LE received the B.S. degree in computer science from the HCMC University of Technology, Ho Chi Minh City, Vietnam, in 2013. He is currently pursuing the joint M.S.-Ph.D. degree with the Artificial Intelligence Laboratory, Department of Computer Science and Engineering, Kyung Hee University, South Korea. His current research interests include machine learning, reinforcement learning, and robotics.

NGO ANH VIEN received the B.S. degree in computer engineering from the Hanoi University of Science and Technology, Vietnam, in 2005, and the Ph.D. degree in computer engineering from Kyung Hee University, South Korea, in 2009. He was a Post-Doctoral Researcher with the National University of Singapore from 2009 to 2011, also with the Ravensburg-Weingarten University of Applied Sciences, Germany, from 2011 to 2013, and also with the Machine Learning and Robotics lab, University of Stuttgart, from 2013 to 2017. He has been a Lecturer (a.k.a Assistant Professor) at Queen's University Belfast, U.K., since 2017. His research interests include machine learning and robotics.

TAECHOONG CHUNG received the B.S. degree in electronic engineering from Seoul National University, South Korea, in 1980, and the M.S. and Ph.D. degrees in computer science from Korea Advanced Institute of Science & Technology, South Korea, in 1982 and 1987, respectively. Since 1988, he has been with the Department of Computer Engineering, Kyung Hee University, South Korea, where he is currently a Professor. His research interests include machine learning, metasearch, and robotics.

● ● ●