Deep Reinforcement Learning with Temporal Logic Specifications

by

Qitong Gao

Department of Mechanical Engineering and Materials Science
Duke University

Date: ____________________________
Approved:

____
Michael M. Zavlanos, Supervisor

____
Leila Bridgeman

____
Miroslav Pajic

Thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science in the Department of Mechanical Engineering and Materials
Science
in the Graduate School of Duke University
2018

Abstract

In this thesis, we propose a model-free reinforcement learning method to synthesize control policies for mobile robots modeled by Markov Decision Process (MDP) with unknown transition probabilities that satisfy Linear Temporal Logic (LTL) specifications. The key idea is to employ Deep Q-Learning techniques that rely on Neural Networks (NN) to approximate the state-action values of the MDP and design a reward function that depends on the accepting condition of the Deterministic Rabin Automaton (DRA) that captures the LTL specification. Unlike relevant works, our method does not require learning the transition probabilities in the MDP, constructing a product MDP, or computing Accepting Maximal End Components (AMECs). This significantly reduces the computational cost and also renders our method applicable to planning problems where AMECs do not exist. In this case, the resulting control policies minimize the frequency with which the system enters bad states in the DRA that violate the task specifications. To the best of our knowledge, this is the first model-free deep reinforcement learning algorithm that can synthesize policies that maximize the probability of satisfying an LTL specification even if AMECs do not exist. We validate our method through numerical experiments.

Contents

List of Tables

List of Figures

1 Introduction

Traditionally, the robot motion planning problem consists of generating robot trajectories that reach a desired goal region starting from an initial configuration while avoiding obstacles Karaman and Frazzoli (2011). More recently, a new class of planning approaches have been developed that can handle a richer class of tasks than the classical point-to-point navigation, and can capture temporal and boolean specifications. Such approaches typically rely on formal languages, such as Linear Temporal logic (LTL), to represent complex tasks and on discrete models, such as transition systems Guo and Dimarogonas (2015); Kantaros and Zavlanos (2017) or Markov Decision Processes (MDPs)Ding et al. (2014a, 2011a); Guo and Zavlanos (2008); Ulusoy et al. (2014), to capture the robot dynamics and the uncertainty in the workspace.

Control of MDPs under LTL specifications has been extensively studied recently Ding et al. (2014a, 2011a); Guo and Zavlanos (2008); Ulusoy et al. (2014). Often, the common assumption is that the transition probabilities in the MDPs are known. In this case, tools from probabilistic model checking Baier et al. (2008) can be used to design policies that maximize the probability of satisfying the assigned LTL task. For example, in Ding et al. (2014a, 2011a); Guo and Zavlanos (2008); Ulusoy et al.

(2014), first a product MDP is constructed by combining the MDP that captures robot mobility and the Deterministic Rabin Automaton (DRA) that represents the LTL specification and then, by computing the Accepting Maximum End Components (AMECs) of the product MDP, control policies that maximize the probability of satisfying the LTL formula are synthesized. However, construction of the product MDP and computation of AMECs is computationally expensive.

In this thesis, we consider robots modeled as MDPs with unknown transition probabilities that are responsible for accomplishing complex tasks captured by LTL formulas. Our goal is to design policies that maximize the probability of satisfying the LTL specification. For this, we employ the Double Deep Q-Network approach that is a model-free deep reinforcement learning method that employs Neural Networks (NN) to approximate the state-action values of the MDP. Unlike relevant approaches Fu and Topcu (2014); Sadigh et al. (2014); Wang et al. (2015), our proposed algorithm completely avoids the construction of a product MDP and AMECs. This signficantly decreases the computational cost and also renders the proposed method applicable to planning problems where AMECs do not exist. To achieve that, we first assign rewards to the transitions of the product MDP so that high rewards correspond to transitions that lead to accepting states that need to be visited infinitely often, as in Sadigh et al. (2014). Transitions in the product MDP are essentially determined by transitions in the MDP given the current action and the next MDP state. Since the DRA is a deterministic automaton, given the current DRA states and the next MDP states, DRA transitions can be uniquely determined and combined with MDP transitions to form the transitions in the product MDP. This allows us to design a policy for the product MDP that maximizes the collection of the rewards. By construction of the rewards and by the acceptance condition of the DRA, maximizing the collection of the rewards implies that the probability of satisfying the LTL formula is maximized. Exploration in the proposed off-policy learning algorithm can be achieved by adding noise to the current policy to obtain an exploration policy. We note that training efficiency can be further improved using recently developed asynchronous training methods Mnih et al. (2016); Wang et al. (2016); Wu et al. (2017).

To the best of our knowledge, the most relevant works are Ding et al. (2014b, 2011b); Fu and Topcu (2014); Li et al. (2017a,b); Sadigh et al. (2014); Wang et al. (2015). Specifically, Sadigh et al. (2014) converts the LTL specification into a DRA and takes the product between the MDP and the DRA to form a Product MDP. Then it employs a model-based learning method to learn the transition probabilities and optimize the current policy using Temporal Difference (TD) learning Sutton and Barto (2011). However, since this method stores all transition probabilities and state values, it can typically be used to solve problems with small and low dimensional state spaces. Furthermore, in this model-based approach, if the model of the environment and robot dynamics are not learned accurately, then the learned policy is not optimal. In our work, the state-action values of each state in the product MDP are approximated by Neural Networks which require much fewer resources to store and are also more expressive. To avoid the dependence of model-based methods on learning an accurate enough model of the environment, model-free learning algorithms can be used instead. Specifically, Fu and Topcu (2014); Wang et al. (2015) find AMECs in the learned product MDP to determine the accepting states and then learn the policy through value iteration Fu and Topcu (2014) or actor-critic type learning Wang et al. (2015). Nevertheless, the quadratic complexity of finding AMECs prohibits these algorithms from handling MDPs with large state spaces and complex LTL specifications, i.e., DRAs with large numbers of states. This is not the case with our method that does not construct a product MDP nor does it require the computation of AMECs. Moreover, note that Fu and Topcu (2014); Wang et al. (2015) return no policies if there do not exist AMECs in the product MDP. To the contrary, in such cases, our proposed control synthesis algorithm can compute a policy that maximizes the probability of satisfying the acceptance condition of the DRA. In practice, this means that the policy returned by our method minimizes the frequency with which the system enters bad states in the DRA that violate the task specifications. This is relevant to applications where the LTL specifications can be thought of as soft constraints whose violation is not destructive for the system, as in our recent work Guo and Zavlanos (2008). In Ding et al. (2014b, 2011b); Li et al. (2017a,b), control of MDPs with unknown transition probabilities under Signal Temporal Logic (STL) and Truncated Linear Temporal Logic (TLTL) specifications is considered. However, STL and TLTL specifications are satisfied by finite paths. To the contrary, here we consider LTL tasks that are satisfied by infinite paths.

2

Priliminaries

In this chapter, we briefly review preliminaries on LTL, MDPs, and Deep Q-Learning.

2.1 LTL Specifications

The basic ingredients of Linear Temporal Logic are a set of atomic propositions $\mathcal{AP}$, the boolean operators, i.e., conjunction $\wedge$, and negation $\neg$, and two temporal operators, next $\bigcirc$ and until $\mathcal{U}$. LTL formulas over a set $\mathcal{AP}$ can be constructed based on the following grammar: $\phi ::= \text{true} \mid \xi \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \bigcirc \phi \mid \phi_1 \; \mathcal{U} \; \phi_2$, where $\xi \in \mathcal{AP}$. For the sake of brevity we abstain from presenting the derivations of other Boolean and temporal operators, e.g., *always* $\square$, *eventually* $\lozenge$, *implication* $\Rightarrow$, which can be found in Baier et al. (2008). Any LTL formula can be translated into a Deterministic Rabin Automaton (DRA) defined as follows Baier et al. (2008).

Definition 2.1.1 (DRA)**.** *A DRA over* $2^{\mathcal{AP}}$ *is a tuple* $\mathcal{R}_\phi = (Q, q_0, \Sigma, \delta, F)$ *where* Q *is a finite set of states;* $q_0 \subseteq Q$ *is the set of initial states;* $\Sigma = 2^{\mathcal{AP}}$ *is the input alphabet;* $\delta : Q \times \Sigma \to Q$ *is the transition function and* $F = \{(G_1, B_1), \ldots, (G_n, B_n)\}$ *is a set of accepting pairs where* $G_i, B_i \subseteq Q, i \in \{1, \ldots, n\}$*.*

A run of $\mathcal{R}_\phi$ over a infinite word $w_\Sigma = w_\Sigma(1)w_\Sigma(2)w_\Sigma(3)\cdots \in \Sigma^\omega$ is a infinite sequence $w_Q = w_Q(1)w_Q(2)w_Q(3)\cdots$, where $w_Q(1) \in q_0$ and $w_Q(k+1) \in \delta(w_Q(k), w_\Sigma(k))$ for all $k \geqslant 1$. Let $\text{Inf}(w_Q)$ denote the set of states that appear infinitely often in w_Q. Then, a run w_Q is accepted by $\mathcal{R}_\phi$ if $\text{Inf}(w_Q) \cap G_i \neq \varnothing$ and $\text{Inf}(w_Q) \cap B_i = \varnothing$ for at least one pair $i \in \{1, \ldots, n\}$ of accepting states $i = 1, \ldots, n$ Belta et al. (2017).

2.2 MDP Robot Models

Robot mobility in the workspace can be represented by a Markov Decision Process (MDP) defined as follows.

Definition 2.2.1 (MDP)**.** *A Markov Decision Process (MDP) is a tuple $M = (S, s_0, A, P, R, \gamma)$, where S is a finite set of states; s_0 is the initial state; A is a finite set of actions; P is the transition probability function defined as $P : S \times A \times S \to [0,1]$; $R : S \times A \times S \to \mathbb{R}$ is the reward function; $\gamma \in [0,1]$ is the discount factor.*

Particularly, for any Markov state at time step t, which is denoted by $s_t \in S$, the future state s_{t+1} should be independent of the past states $s_1, \ldots, s_{t-1}$ given the present state s_t, where:

$$\mathbb{P}[s_{t+1}|s_t] = \mathbb{P}[s_{t+1}|s_1, \ldots, s_t].$$

In order to model uncertainty in both the robot motion and the workspace properties, we extend Definition 2.2.1 to include probabilistic labels giving rise to labeled MDP defined as follows Sadigh et al. (2014).

Definition 2.2.2 (Labeled MDP)**.** *A labeled MDP is a tuple $\mathcal{M} = (S, s^0, A, P, R, \gamma, \mathcal{AP}, L)$, where S is a finite set of states; s_0 is the initial state; A is a finite set of actions; P is the transition probability function defined as $P : S \times A \times S \to [0,1]$; $R : S \times A \times S \to \mathbb{R}$*

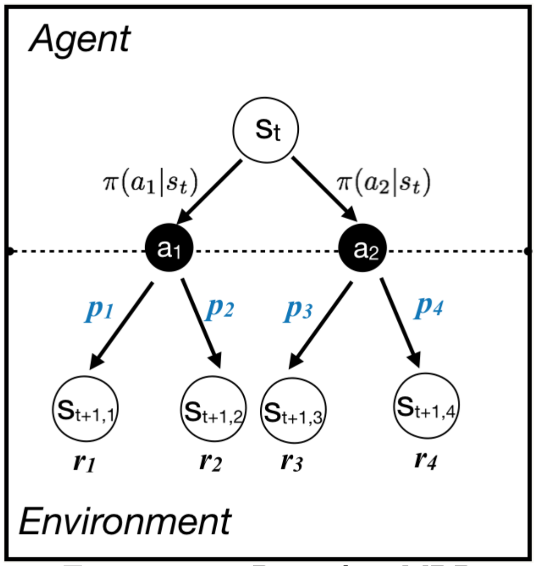

FIGURE 2.1: Part of an MDP

is the reward function; $\gamma \in [0, 1]$ *is the discount factor;* $\mathcal{AP}$ *is the set of atomic propositions;* $L : S \to 2^{AP}$ *is the labeling function that returns the atomic propositions that are satisfied at a state* $s \in S$*.*

Definition 2.2.3 (Deterministic Policy of MDP)**.** *A deterministic policy* π *of a labeled MDP* $\mathcal{M}$ *is a function,* $\pi : S \to A$*, that maps each state* $s \in S$ *to an action* $a \in A$*.*

Observe that given an initial state s^0 and a policy π, an *infinite path* r_π of the MDP is generated under the policy π, defined as an infinite sequence $r_\pi = s^0, s^1, \ldots, s^t, \ldots$, where $s^t \in S$ is the state of the MDP at the stage t. Note that, given an action a^t due to the policy π, a transition from $s^t \in S$ to $s^{t+1} \in S$ in the labeled MDP occurs with probability $P(s^t, a^t, s^{t+1})$, and a scalar reward r^t is generated.

Moreover, given a policy π we can define the accumulated reward over a finite horizon starting from stage t as follows.

Definition 2.2.4 (Accumulated Return)**.** *Given a labeled MDP* $\mathcal{M} = (S, s^0, A, P, R, \gamma, \mathcal{AP}, L)$

and a policy π, the accumulated return over an finite horizon starting from the stage t and ending at stage $t+T$, $T > 0$, is defined as $G_t = \sum_{k=0}^{T} \gamma^k r_{t+k}$, where r_{t+k} is the return at the stage $t+k$.

2.3 Reinforcement Learning

Reinforcement learning is a machine learning technique focusing on how agents ought to take actions in an environment so as to maximize some notion of cumulative reward. It's learning process emphasis on the agent's own experience obtained by directly interacting with the environment, without depending on supervision or complete environment model.

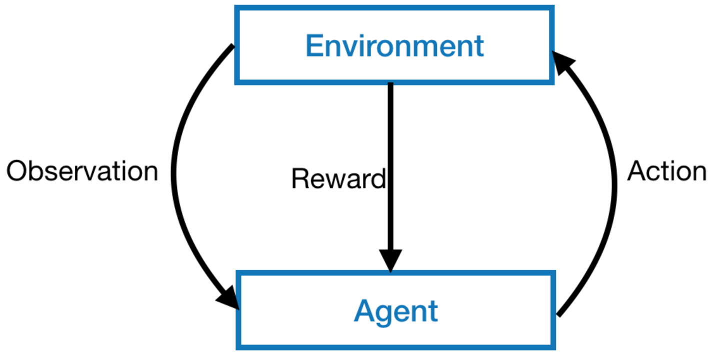

FIGURE 2.2: Reinforcement learning agent interacts with environment

To find the optimal policy π^* that can maximize the accumulated reward, several dynamic programming techniques can be applied which are value iteration and policy iteration. In this thesis, since we only consider the cases where the policy π is deterministic, the approaches introduced below have been slightly modified for better fitting deterministic policy. We recommend readers refering to Sutton and Barto (2011) for a complete review of these techniques.

2.3.1 Value Iteration

Value iteration is a method to find a optimal policy for a given MDP. To do value iteration, we start with defining state value functions and action-state value functions:

Definition 2.3.1 (State Value Function)**.** *Given a labeled MDP $\mathcal{M} = (S, s^0, A, P, R, \gamma, \mathcal{AP}, L)$ and a deterministic policy π, the value function of state $s \in S$ under deterministic policy π is denoted as $V^\pi(s)$ and defined as the expected return starting from state s, where:*

$$V^\pi(s) = \mathbb{E}_\pi[G_t \mid s^t = s] \tag{2.1}$$

Definition 2.3.2 (State-Action Value Function)**.** *Given and MDP $\mathcal{M} = (S, s^0, A, P, R, \gamma, \mathcal{AP}, L)$ and deterministic policy π, the state-action value function $Q^\pi(s, a)$ is defined as the expected return for taking action a when at state s following a deterministic policy π, i.e., $Q^\pi(s, a) = \mathbb{E}[G_t|s^t = s, a^t = a]$.*

Definition 2.3.3 (Optimal State-Action Value Function)**.** *The optimal state-action value function $Q^*(s, a) = \max_\pi Q^\pi(s, a)$ is the maximum state-action value for state s and action a for some deterministic policy π.*

The goal of value iteration is to find the optimal value function $V^*(s)$ that can maximize the $V^\pi(s)$ for all $s \in S$:

$$V^*(s) = \max_\pi \; V^\pi(s) \tag{2.2}$$

And then the optimal deterministic policy π^* can be determined by:

$$\pi^* = \text{argmax}_\pi \; V^\pi(s) \tag{2.3}$$

To find the optimal value function, we first derive the Bellman Equation:

$$\begin{aligned}
V_\pi(s) &= \mathbb{E}_\pi[G_t \mid s^t = s] && (2.4)\\
&= \mathbb{E}_\pi[\sum_{k=0}^{T} \gamma^k R_{t+k+1} \mid s^t = s] && (2.5)\\
&= \mathbb{E}_\pi[R_{t+1} + \sum_{k=1}^{T} \gamma^k R_{t+k+1} \mid s^t = s] && (2.6)\\
&= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid s^t = s] && (2.7)\\
&= \mathbb{E}_\pi[R_{t+1} + \gamma \mathbb{E}_\pi[G_{t+1} \mid s^{t+1} = s'] \mid s^t = s] && (2.8)\\
&= \mathbb{E}_\pi[R_{t+1} + \gamma V^\pi(s^{t+1}) | \; s^t = s] && (2.9)\\
&= \sum_{s',r} p(s', r | s, \pi(s))(r + \gamma V^\pi(s')). && (2.10)
\end{aligned}$$

Equation (2.10) is the bellman equation for V^π. It connects the value of a state and the values of its successor states. And since $Q^\pi(s, a)$ reflects the benefit of taking action a while at state s following a deterministic policy π, (2.9) can be rewritten in terms of $Q^\pi(s, a)$:

$$\begin{aligned}
Q^\pi(s,a) &= \mathbb{E}_\pi[R_{t+1} + \gamma Q^\pi(s^{t+1}, a^{t+1}) | \; s^t = s, a^t = a] && (2.11)\\
&= \sum_{r,s'} p(r, s' | s, a)(r + \gamma Q^\pi(s', \pi(s'))). && (2.12)
\end{aligned}$$

And the optimal value functions can be written as:

$$
\begin{aligned}
V^*(s) &= \max_{a \in A} Q^{\pi^*}(s, a) && (2.13) \\
&= \max_a \mathbb{E}_{\pi*}[G_t | s^t = s, a^t = a] && (2.14) \\
&= \max_a \mathbb{E}_{\pi*}[\sum_{k=0}^{T} \gamma^k R_{t+k+1} \mid s^t = s, a^t = a] && (2.15) \\
&= \max_a \mathbb{E}_{\pi*}[R_{t+1} + \sum_{k=1}^{T} \gamma^k R_{t+k+1} \mid s^t = s, a^t = a] && (2.16) \\
&= \max_a \mathbb{E}_{\pi*}[R_{t+1} + \gamma G_{t+1} \mid s^t = s, a^t = a] && (2.17) \\
&= \max_a \mathbb{E}_{\pi*}[R_{t+1} + \gamma \mathbb{E}_{\pi*}[G_{t+1} \mid s^{t+1} = s', a^{t+1} = a'] \mid s^t = s, a^t = a] && (2.18) \\
&= \max_a \mathbb{E}_{\pi*}[R_{t+1} + \gamma V^*(s^{t+1}) | \, s^t = s, a^t = a] && (2.19) \\
&= \max_a \sum_{s', r} p(s', r | s, \pi(s))(r + \gamma V^*(s')). && (2.20)
\end{aligned}
$$

Equation (2.20) is the Bellman optimality equation for V^*. And similarly, the Bellman optimality equation for Q^* is:

$$
\begin{aligned}
Q^*(s, a) &= \mathbb{E}_{\pi*}[R_{t+1} + \gamma \max_{a' \in A} Q^*(s^{t+1}, a') | \, s^t = s, a^t = a] && (2.21) \\
&= \sum_{r, s'} p(r, s' | s, a)(r + \gamma \max_{a' \in A} Q^*(s', a')). && (2.22)
\end{aligned}
$$

Given the Bellman equations and Bellman optimality equations introduced above, we can use dynamic programming to apply the Bellman optimality equation 2.20 iteratively to find the optimal deterministic policy π^*. In each iteration, we update the value functions for all states $s \in S$ by using the following update equation:

$$V_{k+1}(s) \leftarrow \max_{a \in A} \sum_{s', r} p(s', r | s, a)(r + \gamma V_k(s')), \tag{2.23}$$

where V_k and V_{k+1} denotes the value functions at the kth and the $k+1$th iteration, respectively. Basically, value iteration formally requires an infinite number of iterations to converge exactly to V^*, however, we can terminate the process after the value function changes by only a small amount. And the optimal deterministic policy π^* can be obtained after the termination requirement is satisfied:

$$\pi^*(s) = \text{argmax}_{a \in A} \sum_{s',r} p(s', r|s, a)(r + \gamma V(s')). \tag{2.24}$$

Note that in value iteration, unlike the policy iteration method that is introduced later, there is no explicit policy that can be shown during the entire learning process. The optimal policy can be only determined by using (2.24) after the termination requirement is satisfied and the intermediate value functions may not correspond to any policy.

2.3.2 Policy Evaluation

If the MDP is known and a deterministic policy π is given, we are able to evaluate $V^\pi(s)$ for all $s \in S$. This process can be achieved through policy evaluation.

According to the Bellman equation (2.10), when π is given, we can compute $V^\pi(s)$ for all $s \in S$ by using the following update equation:

$$V_{k+1}(s) = \sum_{s',r} p(s', r|s, \pi(s))(r + \gamma V_k(s')), \tag{2.25}$$

where V_k and V_{k+1} denotes the value functions at the kth and the $k + 1$th iteration, respectively. Similar to value iteration, $V_k(s)$ can converge to $V^\pi(s)$ when $k \to +\infty$ and we can terminate this process after $\max_{s \in S} |V_{k+1}(s) - V_k(s)|$ is small.

2.3.3 Policy Iteration

After evaluating the value functions given a policy π, we can improve that policy based on following steps:

1. Evaluate the policy $V_\pi(s) = [\sum_{k=0}^{T} \gamma^k R_{t+k+1} \mid s^t = s]$ for all $s \in S$.

2. Improve the policy by acting greedily with respect to V^π.

To ensure the new policy is always better than or at least as good as the old one, we first introduce policy improvement theorem:

Theorem 2.3.4 (Policy Improvement Theorem). *If π and π' are any pair of deterministic policies such that for all $s \in S$:*

$$Q^\pi(s, \pi'(s)) \geqslant V^\pi(s).$$

Then the policy π' is as good as or better than π, that is for all $s \in S$:

$$V^{\pi'}(s) \geqslant V^\pi(s).$$

So in the policy improvement step (step 2), the new greedy policy π' is constructed as:

$$\pi'(s) = \text{argmax}_a Q^\pi(s, a) \tag{2.26}$$

$$= \text{argmax}_a \mathbb{E}_\pi[R_{t+1} + \gamma V^\pi(s^{t+1}) | s^t = s, a^t = a] \tag{2.27}$$

$$= \text{argmax}_a \sum_{s', r} p(s', r | s, a)(r + \gamma V^\pi(s')). \tag{2.28}$$

In words, in each policy improvement step, the new greedy policy takes the action that looks best in the short term (after one-step overlook) according to V^π. And notice that ties will be broken arbitrarily while doing argmax_a. Now based on the new greedy policy π', we can evaluate the new value functions $V^{\pi'}$ for all $s \in S$ following from (2.27):

$$V^{\pi'}(s) = \max_{a \in A} \mathbb{E}_{\pi'}[R_{t+1} + \gamma V^{\pi'}(s^{t+1}) | s^t = s, a^t = a] \tag{2.29}$$

$$= \text{argmax}_a \sum_{s', r} p(s', r | s, a)(r + \gamma V^{\pi'}(s')). \tag{2.30}$$

Notice that this equation is the same as the bellman optimality equation (2.20), and thus $V^{\pi'}$ should be optimal value functions and both π and π' are optimal policies. As a result, the policy improvement step can give us a strictly better policy unless the current policy is already optimal.

Different from value iteration, policy iteration improves the current policy directly in the policy domain instead of in the value domain. As a result, there is always an explicit policy that corresponds to each learning epoch and it requires far more less iterations for the learned policy converging to the optimal policy. But the drawback is that in the policy evaluation step, the iterative evaluation of value functions requires multiple sweeps in the state domain and we cannot proceed to the policy improvement step until the evaluation result converges to the optima.

2.3.4 Deep Q-Learning with Experience Replay

To be able to learn a optimal policy, both value iteration and policy iteration require the knowledge of transition probabilities in the MDP, $p(s', r|s, a)$, as shown in (2.24) and (2.28). However, it is not always the case that transition probabilities are remain known, especially in an environment with uncertain dynamics. Here we introduce Q-Learning algorithm that is model-free and can learn a greedy policy that maximizes the state-action value functions.

Consider the labeled MDP defined in Definition 2.2.2 with unknown transition probabilities P. Our goal in this section is to compute a policy π^* that maximizes the expected accumulated return from the initial stage, i.e.,

$$\pi^* = \operatorname{argmax}_\pi \mathbb{E}_{s^{t\geqslant 1}\sim S, r^{t\geqslant 1}\sim R, a^{t\geqslant 1}\sim \pi}[G_1], \tag{2.31}$$

where $s^{t\geqslant 1} \sim S$ means that the states $s^1, s^2, \ldots, s^T$ are selected from the finite state set S, $r^{t\geqslant 1} \sim R$ means that the rewards are determined by the reward function R of the MDP $\mathcal{M}$, and $a^{t\geqslant 1} \sim \pi$ means that the actions are determined by the policy π.

To solve the optimization problem (2.31) we can use Q-Learning, which relies on the state-action value function defined previously.

Q-Learning learns a greedy deterministic policy $\pi(s^t) = \text{argmax}_a Q^\pi(s^t, a^t)$. In what follows, we present a Deep Q-Learning approach that relies on Neural Networks (NNs) to represent the state-action value function that depends on an unknown parameter θ. The parameter θ and the respective state-action value function, denoted by $Q(s, a; \theta)$, can be learned iteratively by minimizing the following loss function:

$$J(\theta) = \mathbb{E}_{s\sim\rho^\beta, r\sim\mathcal{M}, a\sim\beta}[(Q(s^t, a^t; \theta) - y_t)^2], \tag{2.32}$$

where y_t is the training target defined as

$$y_t = R(s^t, a^t, s^{t+1}) + \gamma Q(s^{t+1}, \mu(s^{t+1}); \theta). \tag{2.33}$$

In (2.32), β is an exploration policy, such as ϵ-greedy, ρ^β is the state visitation distribution over policy β, and θ are the weights of the NN. Note that the training target y_t depends on the current network weights θ, that change with every iteration causing overestimation problems Van Hasselt et al. (2016). In supervised learning, the training target should be independent of the training weight. In Mnih et al. (2015), this issue is addressed using a separate network, with weights θ', to generate the Q values needed to form the new target $y_t^{DoubleQ}$:

$$y_t^{DoubleQ} = R(s^t, a^t, s^{t+1}) + \gamma Q(s^{t+1}, \mu(s^{t+1}); \theta')$$

The authors in Mnih et al. (2015) also use an experience replay buffer to reduce the correlation between state-action pairs that appear in the same trajectory. This replay buffer is defined as follows:

Definition 2.3.5 (Replay Buffer)**.** *Let $e^t = (s^t, a^t, s^{t+1}, r^t)$, denote an agent's experience at time step t. Then, the replay buffer is defined as the set $\mathcal{B}^t = \{e^1, \ldots, e^t\}$ that collects all prior experiences up to time t.*

Using the replay buffer, minibatches of experiences $\{(s^t, a^t, s^{t+1}, r^t)\}_\mathcal{B} \sim \mathcal{B}$, drawn uniformly from the buffer can be used for learning. In this case, the loss function becomes:

$$J^{ER}(\theta) = \mathbb{E}_{\{(s^t, a^t, s^{t+1}, r^t)\}_{\mathcal{B}} \sim \mathcal{B}}[(Q(s^t, a^t; \theta) - y_t)^2], \tag{2.34}$$

where the superscript ER stands for 'Experience Replay'.

3

Problem Formulation

Consider a robot operating in a workspace $\mathcal{W} \subseteq \mathbb{R}^d$, $d = 2, 3$ that is responsible for accomplishing a high level complex task captured by an LTL formula ϕ. Robot mobility in $\mathcal{W}$ is captured by a labeled MDP $\mathcal{M} = (S, s^0, A, P, R, \gamma, \mathcal{AP}, L)$, as defined in Definition 2.2.2. Our goal is to design a policy that maximizes the probability of satisfying the assigned LTL specification ϕ. To achieve that, first we compose the MDP $\mathcal{M}$ and the DRA defined in Definition 2.1.1 to obtain a product MDP.

Definition 3.0.1. *A product MDP between the MDP $\mathcal{M} = (S, s_0, A, P, \gamma, \mathcal{AP}, L)$ and the DRA $\mathcal{R}_\phi = (Q, q_0, \Sigma, \delta, F)$ is a tuple $\mathcal{P} = (S_\mathcal{P}, s^0_\mathcal{P}, A_\mathcal{P}, P_\mathcal{P}, F_\mathcal{P}, R_\mathcal{P}, \gamma_\mathcal{P})$, where $S_\mathcal{P} = S \times Q$ is the set of states; $s_{0\mathcal{P}} = (s_0, q_0)$ is the initial state; $A_\mathcal{P}$ is the set of actions inherited from MDP, so that $A_\mathcal{P}((s,q)) = A(s)$; $P_\mathcal{P}$ is the set of transition probabilities, so that for $s_\mathcal{P} \in S_\mathcal{P}, a_\mathcal{P} \in A_\mathcal{P}$ and $s'_\mathcal{P} \in S_\mathcal{P}$, if $q' = \delta(q, L(s))$, then $P_\mathcal{P}(s_\mathcal{P}, a_\mathcal{P}, s'_\mathcal{P}) = P(s, a, s')$; $F_\mathcal{P} = \{(\mathcal{G}_1, \mathcal{B}_1), \ldots, (\mathcal{G}_n, \mathcal{B}_n)\}$ is the set of accepting states, where $\mathcal{G}_i = S \times G_i$ and $\mathcal{B}_i = S \times B_i$; $R_\mathcal{P} : S_\mathcal{P} \times A_\mathcal{P} \times S_\mathcal{P} \to \mathbb{R}$ is the reward function; and $\gamma_\mathcal{P}$ is the discounting rate inherited from the MDP.*

The probability $\mathbb{P}^\mathcal{P}_\pi(\phi)$ that a policy π of the product MDP $\mathcal{P}$ satisfies an LTL formula ϕ is defined as

$$\mathbb{P}_{\pi}^{\mathcal{P}}(\phi) = \mathbb{P}(\{r_{\pi}^{\mathcal{P}} \in \mathcal{P}_{\pi} \mid L(\Pi|_{\mathcal{M}} r_{\pi}^{\mathcal{P}}) \models \phi\}), \tag{3.1}$$

where $\mathcal{P}_{\pi}$ is a set that collects all infinite paths $r_{\pi}^{\mathcal{P}}$ of the product MDP $\mathcal{P}$ (see also Definition 2.2.3) that are induced under the policy π. Also in (3.1), $\Pi|_{\mathcal{M}} r_{\pi}^{\mathcal{P}}$ stands for the projection of the infinite run $r_{\pi}^{\mathcal{P}}$ onto the state-space of $\mathcal{M}$, i.e., $\Pi|_{\mathcal{M}} r_{\pi}^{\mathcal{P}}$ is an infinite sequence of states in $\mathcal{M}$, and $L(\Pi|_{\mathcal{M}} r_{\pi}^{\mathcal{P}})$ denotes the word that corresponds to $\Pi|_{\mathcal{M}} r_{\pi}^{\mathcal{P}}$.

Then the problem that we address in this paper can be summarized as follows.

Problem 1. *Given a labeled MDP $\mathcal{M} = (S, s_0, A, P, \gamma, \mathcal{AP}, L)$ with unknown transition probabilities and an LTL specification ϕ, find a policy π such that the probability $\mathbb{P}_{\pi}^{\mathcal{P}}(\phi)$, defined in (3.1), of satisfying the ϕ is maximized.*

4

Proposed Algorithm

In this chapter, we propose a model-free reinforcement learning algorithm to solve Problem 1. Specifically, we employ Double Deep Q-Network learning to design optimal control policies that maximize the probability of satisfying an LTL specification ϕ without constructing the product MDP and AMECs. Also, our proposed method does not require to learn the transition probabilities in the MDP due to its model-free nature.

We first construct the reward function $R_{\mathcal{P}} : S_{\mathcal{P}} \times A_{\mathcal{P}} \times S_{\mathcal{P}} \to \mathbb{R}$ in Definition 3.0.1 as follows

$$R_{\mathcal{P}}(s_{\mathcal{P}}, a_{\mathcal{P}}, s'_{\mathcal{P}}) = \begin{cases} -1 & \text{if } s'_{\mathcal{P}} \in \mathcal{G}_i, \\ -10 & \text{if } s'_{\mathcal{P}} \in \mathcal{B}_i, \\ -100 & \text{if } q' \text{ is deadlock state}, \\ -1 & \text{otherwise} \end{cases} \tag{4.1}$$

for all $i \in \{1, \ldots, n\}$. In words, we assign high rewards to transitions that lead to states in the sets $\mathcal{G}_i$ and assign low rewards to transitions that end in states in the sets $\mathcal{B}_i$. This selection of rewards can guide the robot to maximize the probability of satisfying a given LTL formula ϕ with minimum number of movements. A negative reward with large absolute value is assigned to transitions that lead to deadlock states in DRA. This prevents violation of the safety constraints in the LTL formula (e.g., $\Box\neg\xi$). A negative reward with small absolute value is assigned to all other transitions so that the robot reaches the final states with the minimum number of movements. Then, to solve Problem 1, we design a policy π^* for the product MDP $\mathcal{P}$ that maximizes the accumulated rewards determined in (4.1), i.e.,

$$\pi^* = \text{argmax}_{\pi}\mathbb{E}_{s^{t\geqslant 1}, r^{t\geqslant 1}\sim R_{\mathcal{P}}, a^{t\geqslant 1}\sim\pi}[G_1], \tag{4.2}$$

where $s^t \in S \in \mathcal{P}$, $r^{t\geqslant 1} \sim R_{\mathcal{P}}$ means that the reward r^t at time step t is determined by the product MDP reward function $\mathcal{R}_{\mathcal{P}}$, and $a^{t\geqslant 1} \sim \pi$ means that the action a^t taken at time step t is determined by the policy π.

To obtain the reward $R_{\mathcal{P}}$ of the product MDP $\mathcal{P}$ directly, we introduce a reward $R_{\mathcal{R}} : Q \times A \times Q \to \mathbb{R}$ on the DRA $\mathcal{R}_{\phi}$ as

$$R_{\mathcal{R}}(q, a, q') = \begin{cases} -1 & \text{if } q' \in G_i, \\ -10 & \text{if } q' \in B_i, \\ -100 & \text{if } q' \text{ is deadlock state}, \\ -1 & \text{otherwise}, \end{cases} \tag{4.3}$$

for all $i \in \{1, \ldots, n\}$. Then, (4.2) becomes:

$$\pi^* = \text{argmax}_{\pi}\mathbb{E}_{s^{t\geqslant 1}\sim\mathcal{P}, r^{t\geqslant 1}\sim\mathcal{R}_{\phi}, a^{t\geqslant 1}\sim\pi}[G_1]. \tag{4.4}$$

The only difference between (4.2) and (4.4) is that in (4.4) the reward r^t, is determined by the DRA reward function (4.3) while in (4.2), r^t is determined by the DRA reward function (4.1).

Remark 4.0.1. *Note that different from the reward function defined in Sadigh et al. (2014), which only depends on the current state $s_{\mathcal{P}}$, the reward function we employ here is based on the transition $(s_{\mathcal{P}}, a_{\mathcal{P}}, s'_{\mathcal{P}})$ which is more suitable for Q-Learning.*

To solve the optimization problem (4.4), we propose a Double Deep Q-Network approach that is summarized in Algorithm 1. Specifically, two neural networks, the main network θ and the target network θ', are used to approximate the current and target Q-values and they both have the same structure. The input to the input layer is $((s, q), a)$, where s, q, a represent the current MDP state, the current rabin state, and the current action that was taken, respectively. Both networks have 2 hidden layers with 400 and 300 nodes, respectively, and a biased layer is associated with each hidden layer. All the weights of the hidden layers are initilized uniformly between 0 and 0.1 and the weights of the output layer are initialized uniformly between 0 and 0.01. All the weights of the bias layer are initialized at 0. The activation functions of the hidden layers are selected to be rectified linear functions. A batch normalization layer Ioffe and Szegedy (2015) is applied after each of the two hidden layers to adjust and scale the activation output.

In Algorithm 1, at the beginning of each episode, the current MDP state s is set to the initial MDP state s_0 and the current rabin state q is set to the initial rabin state q_0 [line 2, Algorithm 1]. Then, $x = (s, q)$ represents the current product state in the product MDP. The current action a is determined by the exploration policy β [lines 4-8, Algorithm 1]:

$$\beta(x) = \begin{cases} \text{argmax}_a Q(x, a; \theta) & \text{with probability } 1 - \epsilon \\ \text{random action} & \text{with probability } \epsilon \end{cases}$$

Then, action a is performed and the next MDP state s' is observed [line 9, Algorithm 1]. The next rabin state q' and the reward r, given the current Rabin state q, are determined by the function `CheckRabin()` in Algorithm 2, which is discussed in detail later in the text [line 10, Algorithm 1]. Then, the next product state $x' = (s', q')$ is constructed and the tuple (x, a, r, x') is stored in the experience replay buffer $\mathcal{B}$ [lines 11-12, Algorithm 1]. If the current number of elements in $\mathcal{B}$ is greater than the training minibatch size m, then the neural network is updated as follows [lines 13-25, Algorithm 1]. First, m tuples $\{(x_0, a_0, r_0, x'_0), \ldots, (x_{m-1}, a_{m-1}, r_{m-1}, x'_{m-1})\}_{\mathcal{B}}$, are sampled uniformly from $\mathcal{B}$ [line 14, Algorithm 1], and the training target y_i of

Algorithm 1 Policy Generation through Deep Q-Learning

```
Input: $\theta, \theta', \mathcal{B}, \alpha, \epsilon, \tau, \gamma, max\_epi, max\_step, m, s_0, q_0, \mathcal{M}$
Begin:
 1: for $epi = 0$ to $max\_epi - 1$ do
 2:     $s \leftarrow s_0$, $q \leftarrow q_0$, $x \leftarrow (s, q)$
 3:     for $step = 0$ to $max\_step - 1$ do
 4:         if explore then
 5:             $a \leftarrow$ take random action
 6:         else
 7:             $a \leftarrow \text{argmax}_a Q(x, a; \theta)$        ▷ $Q(\cdot, \cdot; \theta)$ is the output from main network with input $(\cdot, \cdot)$
 8:         end if
 9:         Get $s'$ following the dynamics in $\mathcal{M}$ after taking action $a$
10:         $q', r \leftarrow$ CheckRabin$(s, q, s')$
11:         $x' \leftarrow (s', q')$
12:         Append tuple $(x, a, r, x')$ to $\mathcal{B}$
13:         if size$(\mathcal{B})$ $>= m$ then
14:             Sample minibatch $\{(x_i, a_i, r_i, x'_i)\}_{\mathcal{B}}$ from $\mathcal{B}$
15:             for $(x_i, a_i, r_i, x'_i)$ in $\{(x_i, a_i, r_i, x'_i)\}_{\mathcal{B}}$ do
16:                 if CheckTeminal$(x'_i)$ then
17:                     $y_i \leftarrow r_i$
18:                 else
19:                     $y_i \leftarrow r_i + \gamma \max_{a'} Q(x'_i, a'; \theta')$        ▷ $Q(\cdot, \cdot; \theta')$ is the output from target network with input $(\cdot, \cdot)$
20:                 end if
21:             end for
22:             $J^{ER}(\theta) \leftarrow \frac{1}{2m} \sum_{k=0}^{m-1} (y_k - Q(x_k, a_k; \theta))^2 + C||\theta||_2^2$
23:             $\theta \leftarrow \theta - \alpha \nabla_\theta J^{ER}(\theta)$
24:             $\theta' \leftarrow (1 - \tau)\theta' + \tau\theta$
25:         end if
26:         $s \leftarrow s'$, $q \leftarrow q'$, $x \leftarrow x'$
27:         if CheckTerminal$(x)$ then
28:             break
29:         end if
30:     end for
31:     $\pi(\cdot) \leftarrow \text{argmax}_{a'} Q(\cdot, a'; \theta)$
32:     return $\pi$
33: end for
```

the i-th tuple for $i = 1, \ldots, m$ is determined as [lines 15-19, Algorithm 1]:

$$y_i = \begin{cases} r_i & \text{if } x'_i \text{ is terminal state} \\ r_i + \gamma \max_{a'} Q(s'_i, a'; \theta') & \text{otherwise,} \end{cases}$$

where a product state $x = (s, q)$ is a terminal state if $q \in \mathcal{G}_i$, for some $i \in \{1, \ldots, n\}$. Note that we terminate every training episode when $q \in \mathcal{G}_i$, since the goal of the robot is to learn a policy so that the states $q \in \mathcal{G}_i$ are visited infinitely often, as this satisfies the acceptance condition of the DRA.

Then, the weights of main network θ are updated by minimizing the loss function $J^{ER}(\theta)$ using a gradient descent method, $\theta \leftarrow \theta - \alpha \nabla_\theta J^{ER}(\theta)$, where α is the learning rate [line 23, Algorithm 1]. To stabilize the main neural network parameters θ and to prevent over-fitting, ℓ_2 regularization is employed. Specifically, we introduce the regularization term $C||\theta||_2^2$ to the cost function (2.34) [line 22, Algorithm 1]:

$$J^{ER}(\theta) = \frac{1}{2m} \sum_{k=0}^{m-1} (y_k - Q(x_k, a_k; \theta))^2 + C||\theta||_2^2,$$

where m is the size of minibatch and C is the regularization parameter. Different from the target network updating approach in Mnih et al. (2015); Van Hasselt et al. (2016), to enable the algorithm to fit in environments with different sizes, here we update the target network θ' in each step as per (4.5), using an update parameter τ selected to be always equal to 0.001 without the need of being tuned [line 24, Algorithm 1]:

$$\theta' \leftarrow (1 - \tau)\theta' + \tau\theta. \tag{4.5}$$

After this training process, s, q, x are set to s', q', x', respectively [line 26, Al.g 1]. Then, if x is a terminal state, the current training episode is completed [lines 27-28, Algorithm 1]. When all training episodes are completed, the learned policy is constructed as [line 31, Algorithm 1]:

$$\pi(x) = \text{argmax}_{a'} Q(x, a'; \theta),$$

where $x = (s, q), s \in S, q \in Q$.

The function `CheckRabin`(s, q, s') used in [line 10, Alg. 1] to obtain the next state q' and the reward r given an action a is described in Algorithm 2. This algorithm requires as an input the current MDP state s, the current rabin state q, the next MDP state s' after taking action a, the labeling function L from the MDP, and the

DRA reward function $\mathcal{R}_\phi$. First, the next rabin state q' is determined based on the DRA transition rule δ (see Definition 2.1.1) given the atomic propositions that are true at the next state s' [line 2, Alg. 1]. If q' is a deadlock state, then this means that the LTL specification is violated. In this case, following the same logic as in Sadigh et al. (2014), we reset the Rabin state to the Rabin state q of the previous step. In [line 10, Alg. 2], the reward is determined by the reward function $R_\mathcal{R}$ embedded in DRA $\mathcal{R}_\phi$.

Algorithm 2 `CheckRabin()`

```
Input:
1: Current MDP state $s$; Current Rabin state $q$; Next MDP state $s'$; MDP $\mathcal{M}$;
   DRA $\mathcal{R}_\phi$;
Begin:
2: $q' \leftarrow \delta(q, L(s'))$
3: if $q'$ is a deadlock state then
4:     $q' \leftarrow q$
5: end if
6: $r \leftarrow R_\mathcal{R}(q, q')$          ▷ $R_\mathcal{R}$ is the reward function in DRA $\mathcal{R}_\phi$
7: return $q', r$
```

5

Numerical Experiments

In this chapter, we illustrate the proposed algorithm on a planning task for a single robot. Algorithm 1 was implemented in Python 2.7, and the neural networks were defined using Tensorflow Abadi et al. (2015) and trained on the Amazon Web Services server using an Nvidia Tesla K80 graphics card and a quad-core Intel Xeon CPU E5-2686 2.30GHz. In what follows, we examine the performance of Algorithm 1 in a static and a dynamic environment. In both cases, the environment is represented as a 10×10 discrete grid world, where each discrete point is associated with a state of a labeled MDP that models the robot; see Figure 5.1. The set $\mathcal{AP}$ of atomic propositions is defined as $\mathcal{AP} = \{A, B, C, T, \varnothing\}$, where the atomic propositions A, B, T are observed in the respective regions shown in Figure 5.1, while $\varnothing$ denotes that nothing is observed. In the static environment the atomic propositions A, B, C and T are observed with probability 1 in the corresponding states, while in the dynamic environment the atomic propositions A, B, and T are observed with probability 0.8 and C is observed with probability of 0.05.

The robot can take 5 actions: *UP, RIGHT, DOWN, LEFT, NONE*, where action *NONE* means the robot chooses to remain idle. We assume the robot has a noisy

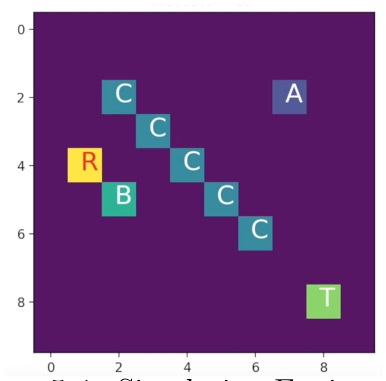

Figure 5.1: Simulation Environment.

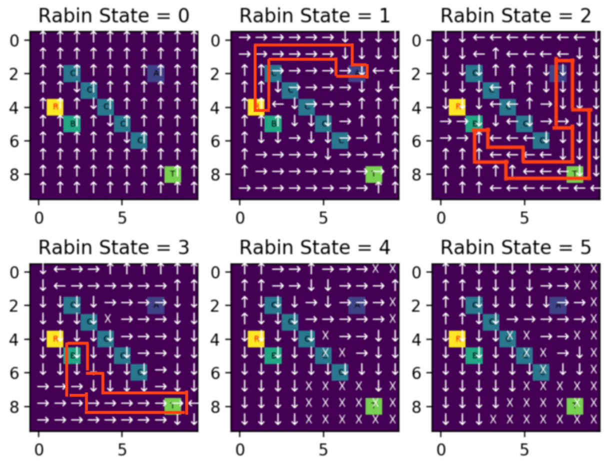

Figure 5.2: Case Study I: Graphical depiction of the policy designed for the static environment by Algorithm 1.

controller, which can only execute the desired action with probability of 0.6, and a random action among the other available ones is taken with probability of 0.4. The initial location of the robot is at $(4, 1)$. In what follows, we consider two case studies. The first one pertains to an LTL specification corresponding to a DRA with 6 states and the second one pertains to a DRA with 16 states. Note that in both case studies, there do not exist AMECs and, therefore, AMEC-based methods, such as Fu and Topcu (2014), cannot be applied to design policies that maximize the probability of satisfying the considered LTL formulas.

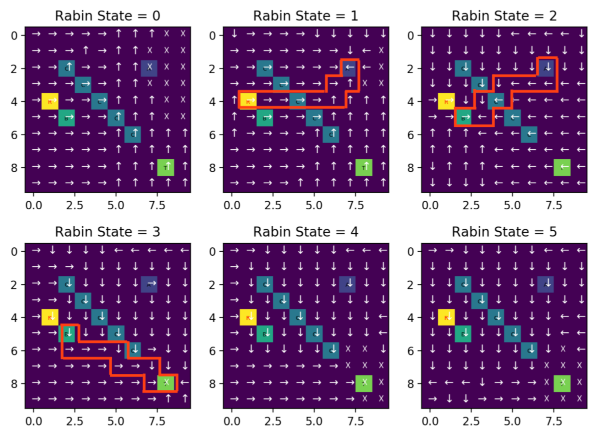

FIGURE 5.3: Case Study I: Graphical depiction of the policy designed for the dynamic environment by Algorithm 1.

5.1 Case Study I

In this case study, we assume that the robot must learn a policy to satisfy the following LTL task:

$$\phi_1 = \Diamond(A \wedge \Diamond B) \wedge \Box\neg C \wedge \Diamond\Box T. \tag{5.1}$$

In words, the LTL task ϕ_1 requires the robot to visit A first, then visit B, and eventually visit T and remain there forever while always avoiding C. The DRA $\mathcal{R}_{\phi_1}$ of LTL ϕ_1 has 6 states, i.e., $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$ and 81 edges, where state q_0 is the deadlock state and state q_5 is the terminal/accepting state that needs to be visited infinitely often. The robot needs to satisfy $A \wedge \neg C$, $B \wedge \neg C$, $T \wedge \neg C$ and $T \wedge \neg C$, to transition from q_1 to q_2, q_2 to q_3, q_3 to q_4 and q_4 to q_5, respectively.

The policy computed for the static environment was computed in 3 hours and is illustrated in Figure 5.2. Specifically, in Figure 5.2, we visualize the policy seperately for each rabin state q, since the robot is operating in the product MDP state-space that collects states of the form $x = (s, q)$; see Definition 3.0.1. Observe in Figure 5.2, that the policy learned by the robot in the rabin states q_1, q_2, q_3 and q_4 eventually leads to the accepting state q_5 with the minimum number of movements due to the design of the reward function (4.3). Observe also that in the rabin state q_2, according to the optimal learned policy, the robot learns to keep some distance from region C to reduce the chance of violating the LTL specification. The policy designed by Algorithm 1 for the dynamic environment was computed in 0.5 hours and is illustrated in Figure 5.3. Note that the designed policy guides the robot through C, as C can be observed with probability of 0.05.

5.2 Case Study II

In this case study, we consider the following LTL specification.

$$\phi_2 = \Diamond(A \wedge \Diamond(B \wedge \Diamond T)) \wedge \Box\Diamond A \wedge \Box\Diamond B \wedge \Box\neg C \tag{5.2}$$

In words, the LTL formula in (5.2) requires the robot to first visit A, B, and T in this order and visit infinitely often A and B while always avoiding C. The DRA that corresponds to ϕ_2 has 16 states and 241 edges.

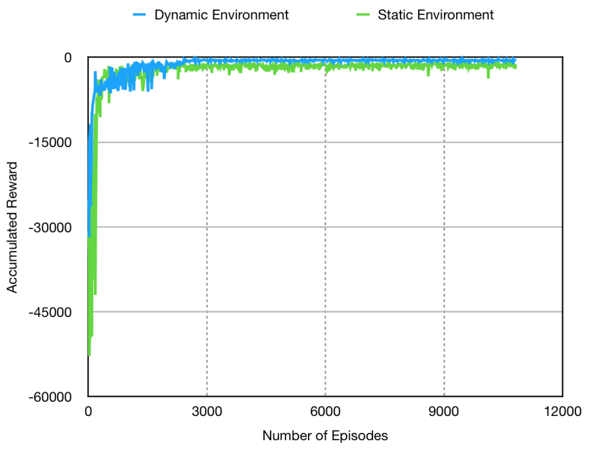

FIGURE 5.4: Case Study II: Graphical depiction of the reward-episode curve.

The policy for the static and dynamic environment was computed in 14 and 17 hours, respectively. Figure 5.4 compares the accumulated reward with respect to episodes for the dynamic and the static environment. In both environments, the learned policy starts converging to the optimal policy after 3000 episodes approximately. Also, observe that the robot operating in the dynamic environment has higher accumulated reward than in the static environment after convergence. This is because C can be observed with low probability and, as a result, the optimal policy often drives the robot from A to B through a path that goes through C.

5.3 Runtime Comparison

In this section, we conducted runtime comparison between our proposed algorithm and the AMECs-based learning algorithm proposed by Fu and Topcu (2014). Since AMECs are not exist with the environment and LTL specifications we experimented above, new testing environments and LTL specifications are developed for this section to ensure the existence of AMECs.

We tested both algorithm on a 5×5 and a 10×10 environment as shown in Figure 5.5 and 5.6. Both of these environments are static and the robot can take 5 actions: *UP, RIGHT, DOWN, LEFT, NONE*. And we considering the following LTL specifications:

$$\phi_3 = \Diamond(A \wedge \Diamond(B \wedge \Diamond T)) \tag{5.3}$$

$$\phi_4 = \Diamond(A \wedge \Diamond(B \wedge \Diamond T)) \wedge \Box\Diamond A \wedge \Box\Diamond B. \tag{5.4}$$

In words, the LTL formula in (5.3) requires the robot to first visit A, B, and T in this order. And (5.4) requires the robot to first visit A, B, and T in this order and visit infinitely often A and B. Table 5.1 compares our proposed algorithm to Fu and Topcu (2014) in terms of runtime and Table 5.2 specifies how large is the state space with each combination of environments and LTL specifications. In particular, observe in Table 5.1 that our proposed algorithm can generally find the optimal policy faster than Fu and Topcu (2014) except for the case which operates in 5×5 environment with LTL specification ϕ_3. Considering only 100 states appeared in the product MDP in this case, as shown in Table 5.2, it makes sense that directly decomposing AMECs can be slightly efficient than training a neural network when the underlying product MDP has a relatively small state space.

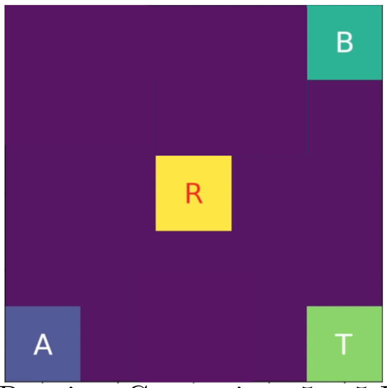

FIGURE 5.5: Runtime Comparison 5×5 Environment.

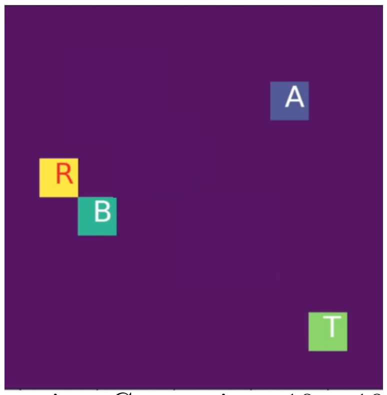

FIGURE 5.6: Runtime Comparison 10×10 Environment.

Taking advantage of the expressiveness of deep neural networks, the policy learned through our approach can handle the uncertainty well. In addition, the support of GPU acceleration helps making the training process of neural networks more efficient. As shown in Table 5.1, our algorithm requires much less time to find the optimal policy than Fu and Topcu (2014). Both algorithms have been implemented in Python and trained on Amazon Web Services server with quad-core Intel Xeon CPU E5-2697 2.60GHz and a Nvidia Tesla K80 graphics card. Neural networks is defined using Tensorflow Abadi et al. (2015).

Table 5.1: Runtime Comparison

Algorithm	5×5 Env w/ ϕ_3	10×10 Env w/ ϕ_3
Deep Q-Network	0.33 hrs	3.5 hrs
AMEC-based Fu and Topcu (2014)	0.22 hrs	>25 hrs
Algorithm	5×5 Env w/ ϕ_4	10×10 Env w/ ϕ_4
Deep Q-Network	3 hrs	9 hrs
AMEC-based Fu and Topcu (2014)	8 hrs	>25 hrs

Table 5.2: Number of states in the product MDP

Env	5×5 Env w/ ϕ_3	10×10 Env w/ ϕ_3
# of states	100	400
Env	5×5 Env w/ ϕ_4	10×10 Env w/ ϕ_4
# of states	375	1500

6

Conclusion

In this thesis, we proposed a model-free reinforcement learning method to synthesize control policies for mobile robots modeled by Markov Decision Process (MDP) with unknown transition probabilities that satisfy Linear Temporal Logic (LTL) specifications. Unlike relevant works, our method does not require learning the transition probabilities in the MDP, constructing a product MDP, or computing Accepting Maximal End Components (AMECs). This significantly reduces the computational cost and also renders our method applicable to planning problems where AMECs do not exist. Simulation studies verified the efficiency of the proposed algorithm.

Bibliography

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

Baier, C., Katoen, J.-P., and Larsen, K. G. (2008), *Principles of model checking*, MIT press.

Belta, C., Yordanov, B., and Gol, E. A. (2017), *Formal Methods for Discrete-Time Dynamical Systems*, vol. 89, Springer.

Ding, X., Smith, S. L., Belta, C., and Rus, D. (2014a), "Optimal control of Markov decision processes with linear temporal logic constraints," *IEEE Transactions on Automatic Control*, 59, 1244–1257.

Ding, X., Smith, S. L., Belta, C., and Rus, D. (2014b), "Optimal control of Markov decision processes with linear temporal logic constraints," *IEEE Transactions on Automatic Control*, 59, 1244–1257.

Ding, X. C., Smith, S. L., Belta, C., and Rus, D. (2011a), "MDP optimal control under temporal logic constraints," in *Decision and Control and European Control Conference (CDC-ECC), 2011 50th IEEE Conference on*, pp. 532–538, IEEE.

Ding, X. C., Smith, S. L., Belta, C., and Rus, D. (2011b), "MDP optimal control under temporal logic constraints," in *Decision and Control and European Control Conference (CDC-ECC), 2011 50th IEEE Conference on*, pp. 532–538, IEEE.

Fu, J. and Topcu, U. (2014), "Probably approximately correct mdp learning and control with temporal logic constraints," *arXiv preprint arXiv:1404.7073*.

Guo, M. and Dimarogonas, D. V. (2015), "Multi-agent plan reconfiguration under local LTL specifications," *The International Journal of Robotics Research*, 34, 218–235.

Guo, M. and Zavlanos, M. M. (2008), "Probabilistic Motion Planning under Temporal Tasks and Soft Constraints," *IEEE Transactions on Automatic Control*, PP, 11.

Ioffe, S. and Szegedy, C. (2015), "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on machine learning*, pp. 448–456.

Kantaros, Y. and Zavlanos, M. M. (2017), "Sampling-Based Optimal Control Synthesis for Multi-Robot Systems under Global Temporal Tasks," *IEEE Transactions on Automatic Control*, (accepted).

Karaman, S. and Frazzoli, E. (2011), "Sampling-based algorithms for optimal motion planning," *The international journal of robotics research*, 30, 846–894.

Li, X., Ma, Y., and Belta, C. (2017a), "A Policy Search Method For Temporal Logic Specified Reinforcement Learning Tasks," *arXiv preprint arXiv:1709.09611.*

Li, X., Vasile, C.-I., and Belta, C. (2017b), "Reinforcement learning with temporal logic rewards," in *Intelligent Robots and Systems (IROS), 2017 IEEE/RSJ International Conference on*, pp. 3834–3839, IEEE.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015), "Human-level control through deep reinforcement learning," *Nature*, 518, 529.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. (2016), "Asynchronous methods for deep reinforcement learning," in *International Conference on Machine Learning*, pp. 1928–1937.

Sadigh, D., Kim, E. S., Coogan, S., Sastry, S. S., and Seshia, S. A. (2014), "A learning based approach to control synthesis of markov decision processes for linear temporal logic specifications," in *Decision and Control (CDC), 2014 IEEE 53rd Annual Conference on*, pp. 1091–1096, IEEE.

Sutton, R. S. and Barto, A. G. (2011), "Reinforcement learning: An introduction," .

Ulusoy, A., Wongpiromsarn, T., and Belta, C. (2014), "Incremental controller synthesis in probabilistic environments with temporal logic constraints," *The International Journal of Robotics Research*, 33, 1130–1144.

Van Hasselt, H., Guez, A., and Silver, D. (2016), "Deep Reinforcement Learning with Double Q-Learning." in *AAAI*, vol. 16, pp. 2094–2100.

Wang, J., Ding, X., Lahijanian, M., Paschalidis, I. C., and Belta, C. A. (2015), "Temporal logic motion control using actor–critic methods," *The International Journal of Robotics Research*, 34, 1329–1344.

Wang, Z., Bapst, V., Heess, N., Mnih, V., Munos, R., Kavukcuoglu, K., and de Freitas, N. (2016), “Sample efficient actor-critic with experience replay,” *arXiv preprint arXiv:1611.01224*.

Wu, Y., Mansimov, E., Grosse, R. B., Liao, S., and Ba, J. (2017), “Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation,” in *Advances in neural information processing systems*, pp. 5285–5294.