Teaching Multiple Tasks to an RL Agent using LTL

Rodrigo Toro Icarte
University of Toronto
Department of Computer Science & Vector Institute
rntoro@cs.toronto.edu

Toryn Q. Klassen
University of Toronto
Department of Computer Science
toryn@cs.toronto.edu

Richard Valenzano
Element AI
rick.valenzano@elementai.com

Sheila A. McIlraith
University of Toronto
Department of Computer Science
sheila@cs.toronto.edu

ABSTRACT

This paper examines the problem of how to teach multiple tasks to a *Reinforcement Learning (RL)* agent. To this end, we use Linear Temporal Logic (LTL) as a language for specifying multiple tasks in a manner that supports the composition of learned skills. We also propose a novel algorithm that exploits LTL progression and off-policy RL to speed up learning without compromising convergence guarantees, and show that our method outperforms the state-of-the-art approach on randomly generated Minecraft-like grids.

KEYWORDS

Deep RL; LTL; Task Specification; Task Decomposition

ACM Reference Format:
Rodrigo Toro Icarte, Toryn Q. Klassen, Richard Valenzano, and Sheila A. McIlraith. 2018. Teaching Multiple Tasks to an RL Agent using LTL. In *Proc. of the 17th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2018), Stockholm, Sweden, July 10–15, 2018,* IFAAMAS, 10 pages.

1 INTRODUCTION

Reinforcement Learning (RL) algorithms are capable of learning effective behaviours through trial and error interactions with their environment [40]. The recent combination of these algorithms with deep learning has enabled RL systems to perform well in complex environments such as Atari games [32, 37] and robotics [3, 4, 18].

Our concern in this paper is with teaching an RL agent to perform multiple tasks. For example, imagine that you have just purchased a robot for your home. Ideally, you would like it to learn how to perform several chores, such as picking up the dirty laundry, sorting it, and popping it in the washer. However, if your robot uses RL, it needs a reward signal to learn from, and therein lies a problem. Your dirty socks are not going to reward the robot for being picked up. Further the robot must have an adequate state representation (or it needs to know what to remember) in order for these temporally extended tasks – a selection of state properties that must occur over time in accordance with some temporal pattern – to be learned by RL algorithms that assume a Markovian reward function.

In this paper, we propose a means of teaching or instructing an RL agent to perform multiple tasks by specifying those tasks in Linear Temporal Logic (LTL) and then defining reward functions that provide positive reward for their successful completion. LTL is a propositional, modal temporal logic first developed for the verification of reactive systems [35]. It augments propositional logic with modalities such as $\Diamond$ (*eventually*), $\Box$ (*always*), and U (*until*) in support of expressing statements such as *"Always if clothes are on the floor, put them in the hamper"* or *"Eventually make dinner."* Such statements can be combined via logical connectives and nesting of modal operators to provide task specifications. The syntax is natural and compelling and, as a formal language, it has a well-defined semantics and thus is unambiguously interpretable. Moreover, it is possible to translate natural language into LTL [13].

To define a task, we use a high-level domain specific vocabulary comprised of a set of propositions that relate to properties of the environment or the occurrence of events that can be determined to be true or false in the environment (e.g., `clothes_on_floor`, `made_dinner`). This vocabulary can then be used by a human teacher to specify any number of tasks that they would like the RL agent to learn. By teaching the agent using a pre-defined vocabulary, we obviate the need for the teacher to know the state representation used by the agent. Instead, we require the existence of event- or property-detectors that determine the truth or falsity of the propositions in our domain-specific language, much in the way object-detectors are used in vision applications. Further, the burden of tracking the satisfaction of the temporally extended (and potentially non-Markovian) LTL reward function is left to the LTL, removing the need for the agent to know a priori what it needs to remember of the past in order to learn to complete any task it may be given in the future.

The use of LTL for task specification can also be beneficial for learning. For example, while many hierarchical RL techniques also decompose tasks into subtasks, these methods then solve the subtasks in a locally optimal way such that global optimality can be lost when they are aggregated back together. In contrast, our LTL specification enables an interleaving of subtasks that supports global optimization. In addition, given a collection of tasks specified in LTL, the agent can learn all of them simultaneously using off-policy RL, exploiting a technique called *LTL progression* to extract (possibly shared) subtasks. We then solve each task, one at the time, while getting better at all of them using off-policy RL. Our approach not only preserves global optimality guarantees, it also has good empirical performance, as we show in the experimental section.

The notion of teaching an RL agent is not new. Past work has largely been associated with *non-technical* humans teaching RL agents by providing positive/negative feedback (e.g. [17, 29, 30, 48]),

Proc. of the 17th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2018), M. Dastani, G. Sukthankar, E. André, S. Koenig (eds.), July 10–15, 2018, Stockholm, Sweden. © 2018 International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). All rights reserved.

demonstrations (e.g. [1, 4, 43]), or advice (e.g. [19, 20, 31, 49]). In contrast, we employ a formal language to describe tasks we wish the agent to learn, with no strong claim of how easy (or hard) it would be for a lay person to define LTL specifications. The focus of this research is to demonstrate the appeal of LTL in the context of multi-task RL and how to take advantage of its structure.

There is also a large body of work on multi-task or multiple goal RL (e.g. [3, 11, 26, 33, 36, 39, 44, 46, 53]). While recent work has used variants of LTL in RL [14, 27, 28, 51, 52], this was not in the context of teaching multiple tasks to an RL agent. Another particularly related work uses policy sketches to learn multiple modular tasks [2]. However, sketches are less expressive than LTL and might prune optimal policies from consideration. Further discussion of this and other related work can be found in Section 6.

There are two main contributions of this work. The first is that we propose the use of LTL as a language for teaching RL agents multiple tasks. Our task specifications are compositional and elaboration tolerant, supporting the learning and transfer of interleaved realizations of multiple tasks. Our second main contribution is a novel framework for multi-task learning with LTL task specifications, which exploits properties of LTL to speed up learning while preserving guarantees of convergence to an optimal policy.

2 PRELIMINARIES

2.1 Markov Decision Processes

A *Markov Decision Process (MDP)* is a tuple $\mathcal{M} = \langle S, A, T, R, \gamma \rangle$ where S is a finite set of *states*, A is a finite set of *actions*, $\gamma \in (0, 1]$ is the *discount factor*, $T : S \times A \times S \to [0, 1]$ is the *transition probability distribution*, and $R : S \times A \times S \to \Pr(\mathbb{R})$ is the *reward function*.

By performing actions in an MDP an agent moves between states according to the transition probability distribution, and accumulates reward according to the reward function. A *policy* π for an MDP is a probability distribution over actions for each state. An agent following policy π will take action a in state s with probability $\pi(a|s)$. The *state-action value* or *Q-value*, denoted $Q^{\pi}(s, a)$, is the *expected discounted return* of selecting action a in state s and then selecting actions according to π. A policy is *optimal* if the expected discounted reward received by following that policy is maximal for every state $s \in S$. The Q-value function of the optimal policy is denoted by Q^*. Given Q^*, the optimal policy is to select the action a in every state s with the highest value of $Q^*(s, a)$.

In RL [40], the agent does not know the transition probability distribution or reward function, and must find an effective policy through interaction with the environment (i.e., the MDP). At each time step, the agent selects an action $a \in A$, executes it in the current state s, and receives reward r and a new state s' (which are sampled from R and T, respectively). The process repeats from s'.

2.2 The Q-Learning Algorithm

Off-policy RL methods learn a *target* policy while using some other *behaviour* policy for action selection. In an off-policy method, the policy being learned is called the *target policy*, while the policy being used for action selection is the *behaviour policy*. In this work, we use the well-known off-policy *Q-learning* algorithm [50] to simultaneously learn policies for different tasks during the same interaction with the environment. Q-learning begins by initializing the Q-values of all state-action pairs (often to zero). At every step, the algorithm then uses some behaviour policy to pick an action a in the current state s, for which a new state s' and reward r are returned from the environment. The current estimation of $Q(s, a)$ is then updated as follows, where α is the *learning rate*:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)] \qquad (1)$$

The algorithm is guaranteed to converge to the optimal Q-values as long as the behaviour policy visits every state-action pair an infinite number of times. One way to fulfill this condition is to set the behaviour policy to be ϵ-greedy on the target policy. That is, on each step, the behaviour policy selects a random action with probability ϵ and the action with the highest Q-value with probability $1 - \epsilon$.

2.3 Function Approximation and DQN

In the simplest form of Q-learning, a table is used to store the Q-value for each state-action pair. For problems with large (or even infinite) state spaces, this is impractical, and some form of *function approximation* is used on the Q-value function. This means that the Q-value function is defined as a function of state features and Q-value updates involve updating the function instead of just entries in a table. We note that doing so generally means that Q-learning is no longer guaranteed to converge to an optimal policy.

Deep Q-Networks (DQN) [32] use a deep neural network for Q-value function approximation. To stabilize learning, an experience replay buffer and a target network are used. The agent's experiences, of the form (s, a, r, s'), are stored in the buffer and randomly sampled to train the network over time. This helps to reduce the correlation between the experiences used to make consecutive updates to the neural net. The Q-learning updates are also computed with respect to a *target* network, which is only updated periodically, as a way to decrease the chance of the policy diverging.

3 LTL AND MULTI-TASK RL

In this section we review Linear Temporal Logic (LTL), illustrate how tasks and reward functions are specified using this language, and define the multi-task reinforcement learning problem.

Illustrative Example: We use a Minecraft-like grid world domain, proposed by Andreas et al. [2], to illustrate and constrast our work. Figure 1 shows an example grid. In this world, the agent can interact with different objects, extract raw material from its environment, and use this material to make new objects. The domain is comprised of the following properties and events that are detectable by the agent: {`got_wood`, `got_iron`, `got_grass`, `used_factory`, `used_toolshed`, `is_night`, `at_shelter`}.

3.1 Specifying Tasks in Linear Temporal Logic

Linear Temporal Logic (LTL) is a propositional modal logic with temporal modalities [35]. Here we use LTL formulae to specify tasks in terms of patterns of properties or events that characterize the successful execution of a task. These properties and events are drawn from a domain-specific set of propositions.

3.1.1 LTL Syntax. LTL formulae are constructed from a set $\mathcal{P}$ of propositional symbols, the standard Boolean operators, and a

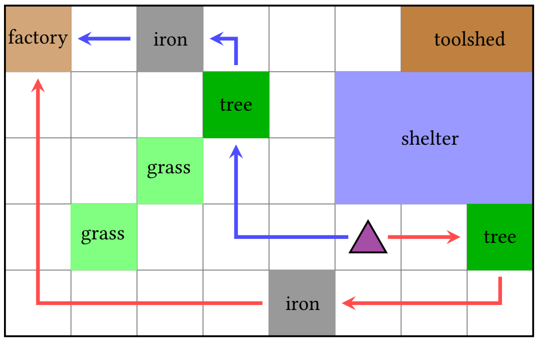

Figure 1: Minecraft-inspired example (adapted from [2]). The blue path is the optimal policy for task $\varphi_{\texttt{bridge}}$. The red path is a globally suboptimal policy that would be computed when decomposing the subtasks, learning an optimal policy for each, and then combining them together.

set of temporal operators. The Boolean operators are $\wedge$ (*and*), $\neg$ (*negation*), $\vee$ (*or*), and $\rightarrow$ (*implication*). The temporal operators are the unary operator $\bigcirc$ (*next*), and the binary operator U (*until*). From these we can also define $\square$ (*always*) and $\Diamond$ (*eventually*). E.g., $\Diamond\varphi \equiv \text{true}\,\mathsf{U}\,\varphi$. The syntax of an LTL formula is defined as follows:

$$\varphi ::= p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \bigcirc\varphi \mid \varphi_1\,\mathsf{U}\,\varphi_2 \text{ with } p \in \mathcal{P}$$

We have omitted $\vee$ and $\rightarrow$, since these can be defined according to the following equivalences: $\varphi_1 \vee \varphi_2 \equiv \neg(\neg\varphi_1 \wedge \neg\varphi_2)$ and $\varphi_1 \rightarrow \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2$. We also define the symbols true and false through the following equivalences: true $\equiv p \vee \neg p$ and false $\equiv \neg$true.

Example (cont.): Returning to our Minecraft-like example, the set of propositions $\mathcal{P}$ is the set {`got_wood`, `got_iron`, `got_grass`, `used_factory`, `used_toolshed`, `is_night`, `at_shelter`}. Using these propositions together with the logical and modal constructs of LTL, we can specify a variety of tasks such as:

- $\varphi_{\texttt{rope}} \triangleq \Diamond(\texttt{got\_grass} \wedge \Diamond\texttt{used\_toolshed})$
  Make a rope by eventually collecting grass and then eventually using the toolshed.
- $\varphi_{\texttt{bridge}} \triangleq \Diamond(\texttt{got\_wood} \wedge \Diamond\texttt{used\_factory}) \wedge \Diamond(\texttt{got\_iron} \wedge \Diamond\texttt{used\_factory})$
  Make a bridge by collecting wood and iron, and using the factory afterwards. Note that the wood and iron can be collected in either order, and that one factory use suffices to fulfill the task if done after collecting both.

3.1.2 LTL Semantics. The truth value of an LTL formula is determined relative to an infinite sequence of *truth assignments*, $\sigma = \langle\sigma_0, \sigma_1, \sigma_2, \ldots\rangle$, for the propositions in $\mathcal{P}$, where each σ_i is simply an assignment of true or false to each proposition in $\mathcal{P}$. We say $p \in \sigma_i$ for a proposition $p \in \mathcal{P}$ to indicate that p is true in σ_i.

Intuitively, the temporal operators allow us to describe how the propositions behave over time. For example, the formula $\bigcirc p$ will hold at some time step i in a sequence if the proposition p holds in the next (i.e. $(i+1)$-th) time step. Similarly, the formula $p\,\mathsf{U}\,q$ will hold at some time step i if p is true at that time and going forward until a time step is reached in which q is true.

Let us now formally define the notion that a sequence σ *models* or *satisfies* φ at time $i \geq 0$, denoted by $\langle\sigma, i\rangle \models \varphi$, as follows:

- $\langle\sigma, i\rangle \models p$ iff $p \in \sigma_i$, where $p \in \mathcal{P}$
- $\langle\sigma, i\rangle \models \neg\varphi$ iff $\langle\sigma, i\rangle \not\models \varphi$
- $\langle\sigma, i\rangle \models (\varphi_1 \wedge \varphi_2)$ iff $\langle\sigma, i\rangle \models \varphi_1$ and $\langle\sigma, i\rangle \models \varphi_2$
- $\langle\sigma, i\rangle \models \bigcirc\varphi$ iff $\langle\sigma, i+1\rangle \models \varphi$
- $\langle\sigma, i\rangle \models \varphi_1\,\mathsf{U}\,\varphi_2$ iff there exists j such that $i \leq j$ and $\langle\sigma, j\rangle \models \varphi_2$, and for all k such that $i \leq k < j$, $\langle\sigma, k\rangle \models \varphi_1$

A sequence σ is then said to *model* or *satisfy* φ iff $\langle\sigma, 0\rangle \models \varphi$.

Recall that while the semantics of the LTL formulae is defined with respect to the sequence of truths assignments of the propositions in $\mathcal{P}$, the state of the RL agent may be defined in a different way. As such, as noted in Section 1, the evaluation of each proposition in $\mathcal{P}$ is determined by a set of event- and property-detectors. These event- and property-detectors are mappings from the state of the RL agent to the truth or falsity of propositions in $\mathcal{P}$. We refer to this mapping as a *labeling function*, $L : S \rightarrow 2^{\mathcal{P}}$, which maps the agent's state into a truth evaluation of the propositions in $\mathcal{P}$.

3.1.3 LTL progression. An LTL formula can also be *progressed* along a given sequence of truth assignments [6]. In the context of an RL agent, the formula can be updated during an episode to reflect those aspects of the formula that have been satisfied by states seen so far. The progressed formula will only include those parts of the original formula that remain to be satisfied. For example, the formula $\Diamond(p \wedge \bigcirc\Diamond q)$ (eventually p and, then, eventually q) can be progressed to $\Diamond q$ after the agent reaches a state where p is true.

We now formally define the progression of a formula through a truth assignment (similarly to Bacchus and Kabanza [6, Table 2]):

Definition 3.1. Given an LTL formula φ and a truth assignment σ_i over $\mathcal{P}$, $\text{prog}(\sigma_i, \varphi)$ is defined as follows:

- $\text{prog}(\sigma_i, p) = \text{true}$ if $p \in \sigma_i$, where $p \in \mathcal{P}$
- $\text{prog}(\sigma_i, p) = \text{false}$ if $p \notin \sigma_i$, where $p \in \mathcal{P}$
- $\text{prog}(\sigma_i, \neg\varphi) = \neg\,\text{prog}(\sigma_i, \varphi)$
- $\text{prog}(\sigma_i, \varphi_1 \wedge \varphi_2) = \text{prog}(\sigma_i, \varphi_1) \wedge \text{prog}(\sigma_i, \varphi_2)$
- $\text{prog}(\sigma_i, \bigcirc\varphi) = \varphi$
- $\text{prog}(\sigma_i, \varphi_1\,\mathsf{U}\,\varphi_2) = \text{prog}(\sigma_i, \varphi_2) \vee (\text{prog}(\sigma_i, \varphi_1) \wedge \varphi_1\,\mathsf{U}\,\varphi_2)$

The prog operator can be applied after each step in an episode to update the task specification formula to reflect which parts of the original formula have become satisfied or unsatisfied. This is because a sequence satisfies a given formula at time i if the formula progressed through σ_i is satisfied at time $i+1$. This is stated formally by the next theorem which follows immediately from an analagous result for a first-order version of LTL [6, Theorem 4.3].

Theorem 3.2. *Given any LTL formula φ and an infinite sequence of truth assignments $\sigma = \langle\sigma_i, \sigma_{i+1}, \sigma_{i+2}, \ldots\rangle$ for the variables in $\mathcal{P}$, $\langle\sigma, i\rangle \models \varphi$ iff $\langle\sigma, i+1\rangle \models \text{prog}(\sigma_i, \varphi)$.*

3.1.4 Co-Safe LTL. LTL is interpreted over infinite traces. In this work, we focus on tasks that are completed in a finite episode. Therefore, we will describe tasks with *co-safe* LTL formulae. Co-safe LTL [22] is the subset of LTL for which the truth of formulae can be ensured after a finite number of steps. For example, $\Diamond p$ ("eventually, p must be true") is co-safe, because once p has been made true, what happens afterwards is irrelevant. Note that $\neg\Diamond q$, which means that "q must always be false", is not co-safe, because it can only be satisfied if q is never true over an infinite number of steps. However, we can define the co-safe task $\neg q \cup p$, which states that "q must always be false until p becomes true". As such, a co-safe LTL formula can still define properties that should be avoided or ensured (i.e., $\neg q$) at every step while some sub-task (i.e., p) is being completed. It is known that the syntactic restrictions of only using the temporal operators $\bigcirc$, $\cup$, $\Diamond$ and applying $\neg$ only to atomic propositions ensure that an LTL formula is co-safe [24].

3.2 Learning Tasks Specified in LTL

Our RL agent learns to perform multiple tasks specified as LTL formulae by garnering reward for their completion. In order to define this problem formally, we use the concept of a non-Markovian reward decision process (NMRDP) (e.g., [5, 8, 10, 47].)

Definition 3.3. A non-Markovian reward decision process (NMRDP) is a tuple $\mathcal{M} = \langle S, A, T, R, \gamma \rangle$ where S, A, T, and γ are defined as in an MDP, but where the reward function R is defined over state histories, $R : S^* \to \Pr(\mathbb{R})$.

Given an NMRDP $\mathcal{M} = \langle S, A, T, R, \gamma \rangle$, the Q-value function of a policy π can be defined over sequences of states:

$$Q^{\pi}(\langle s_0, \ldots, s_t \rangle, a) = \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k R(\langle s_0, \ldots, s_{t+k+1} \rangle) \,\middle|\, A_t = a \right].$$

We will be defining reward functions for NMRDPs in terms of completing tasks defined by co-safe LTL formulae.

Definition 3.4. A multi-task co-safe LTL specification (MTCLS) is a tuple $\mathcal{T} = \langle S, A, T, \mathcal{P}, L, \Phi, \gamma \rangle$, where S, A, T, and γ are defined as in an MDP or NMRDP, $\mathcal{P}$ is a set of propositional symbols, $L : S \to 2^{\mathcal{P}}$ is the *labelling function*, and Φ, the *set of tasks*, is a finite non-empty set of co-safe LTL formulae over $\mathcal{P}$.

Intuitively, the labelling function specifies what propositions are true in which states. A MTCLS $\mathcal{T} = \langle S, A, T, \mathcal{P}, L, \Phi, \gamma \rangle$ can be used to specify a set of NMRDPs, $\{\langle S, A, T, R_\varphi, \gamma \rangle : \varphi \in \Phi\}$, where each reward function R_φ is defined so that the agent receives a reward of 1 if and only if the formula φ becomes satisfied by the sequence of (labels of) states visited in the episode. Formally, that means that

$$R_\varphi(\langle s_0, \ldots, s_n \rangle) = \begin{cases} 1 & \sigma_{0:n-1} \not\models \varphi \text{ and } \sigma_{0:n} \models \varphi \\ 0 & \text{otherwise} \end{cases}$$

where $\sigma_{i:j} = \langle \sigma_i, \ldots, \sigma_j \rangle = \langle L(s_i), \ldots, L(s_j) \rangle$.

Given a MTCLS $\mathcal{T}$, an RL agent will not know the transition probability distribution of the domain, but they will have access to the labelling function and the set of tasks Φ. Such a specification is natural in the context of describing and teaching tasks to an agent. Nevertheless, it differs from the standard RL approach in which the agent only gets a reward signal upon executing actions during exploration. As we will see in Section 5, this approach will allow for significant performance improvements.

4 OFF-POLICY LEARNING WITH LTL

In this section, we describe a framework called *LTL Progression for Off-Policy Learning (LPOPL)* for teaching an RL agent how to accomplish multiple tasks that are specified using co-safe LTL. For clarity, we describe LPOPL in the tabular case and then describe how Q-value function approximation can be incorporated.

4.1 LPOPL Overview and Example

The overall idea behind LPOPL is to extract sub-tasks from the given set of tasks, and then use Q-learning to simultaneously learn sub-policies for each of them. An overview of LPOPL is given in Algorithm 1. To help describe this method, we use an example in which the set of tasks is $\Phi = \{\Diamond(b \wedge \Diamond c), \Diamond(d \wedge \Diamond c)\}$. The first of these formulae means "eventually reach a state where b is true and then eventually reach a state where c is true." The second formula specifies an analogous task with d in the role of b.

In the first step of LPOPL, subtasks are extracted from Φ by progressing every task in Φ into simpler tasks (Algorithm 2). In our example, only $\Diamond c$ will be extracted, as both $\Diamond(b \wedge \Diamond c)$ and $\Diamond(d \wedge \Diamond c)$ can be progressed to $\Diamond c$. LPOPL will then learn a separate policy for each task and extracted sub-task. That is, three different Q-value functions will be learned — $Q_{\Diamond(b\wedge\Diamond c)}$, $Q_{\Diamond(d\wedge\Diamond c)}$, and $Q_{\Diamond c}$ — one for each of $\Diamond(b \wedge \Diamond c)$ and $\Diamond(d \wedge \Diamond c)$, and $\Diamond c$. Each of these Q-value functions is then initialized. For example, we can set $Q_\varphi(s, a)$ to 0 for every state s, action a, and formula φ in the tabular case.

LPOPL then iteratively performs a series of episodes in the environment. On each iteration, some task φ is selected from Φ using some *curriculum learning* algorithm (*i.e.* the call to `GetEpisodeTask`). A new episode is then started with φ as its objective.

On each step of any episode, the given task is progressed so that it takes into account the sequence of states seen thus far. The progressed formula is then used to identify the Q-value function that will be used to inform action selection in the next step. For example, if the given task of $\varphi = \Diamond(b \wedge \Diamond c)$ is progressed to $\Diamond c$, actions will be selected ϵ-greedy on $Q_{\Diamond c}$. All of the Q-value functions will also be updated in each step using off-policy Q-learning updates. This will allow the agent to gain experience regarding how to solve tasks and sub-tasks that the episode is not currently focusing on.

These off-policy updates continue until the episode terminates. This will happen if the task is satisfied (*i.e.* completed), falsified (*i.e.* cannot be completed given the states seen), or an environment dead-end is encountered. The curriculum learning method is then used to select a new task for the next episode.

4.2 Algorithm Components

4.2.1 Sub-task extraction. The process for using progression on a set of tasks to extract sub-tasks is shown in Algorithm 2. The new set of tasks and sub-tasks is denoted as Φ^+. The basic idea is to iteratively generate new formulae by progressing every formula in Φ^+ — which is initialized as Φ — over every possible truth assignment of the propositions in $\mathcal{P}$. New formulae are then added to Φ^+, and the process continues until a stable point is reached. The set Φ^+ will then necessarily contain any formula that is reachable by progressing the formulae in Φ over any possible sequence of states in the environment. When performed on the example in Section 4.1, Φ^+ will contain $\Diamond(b \wedge \Diamond c)$, $\Diamond(d \wedge \Diamond c)$, and $\Diamond c$.

4.2.2 The LPOPL behaviour policy. Let φ be the task to solve on the current episode, during which states $s_0, \ldots, s_k$ have been encountered, and let φ' be the result of progressing φ through states s_0, ..., s_k. Then the LPOPL behaviour policy will be ϵ-greedy on $Q_{\varphi'}(s_k, a)$ (Section 2.2). Notice that this behaviour policy uses the Q-value function most relevant to the situation at hand. For example, suppose $\varphi = \Diamond(b \wedge \Diamond c)$. At any point during the episode prior to

```
Function LPOPL($\gamma, \Phi, L, \mathcal{P}, N$)
    $t \leftarrow 0, i \leftarrow 0$;
    $\Phi^+$ = ExtractSubtasks($\Phi, \mathcal{P}$);
    $Q \leftarrow$ InitializeQValueFunctions($\Phi^+$);
    while $t < N$ do
        $\varphi \leftarrow$ GetEpisodeTask($\Phi$);
        $t \leftarrow t +$ RunEpisode($Q, \varphi, L, \gamma, N - t$);
    return $Q^+$;
```

Algorithm 1: The main LPOPL episodic learning loop.

b being encountered, φ' will be equal to φ, and so actions will be selected greedily on $Q_{\Diamond(b \wedge \Diamond c)}$. However, once b is encountered, φ' will be $\Diamond c$, and actions will be selected greedily on $Q_{\Diamond c}$.

4.2.3 Q-value function updates in LPOPL. Algorithm 3 shows the process that LPOPL uses on each episode. During the episode, the task formula φ is progressed and used for action selection as described above. When an action a is executed in state s with the result being state s', an update is performed for each Q-value function in Q. To update a particular Q_ψ, this method first computes the reward that would be received if the agent was currently trying to solve ψ. By definition, the reward is 1 if ψ becomes satisfied by the transition, and 0 otherwise. This is evaluated by progressing ψ through s' and checking if the resulting formula, denoted ψ', is true. Note that it is possible that $\psi = \psi'$. There are now two possible cases. In the first, ψ' is neither true nor false, and s' is not a dead-end in the environment. The update then made in the tabular case is the following modification of the standard Q-learning rule:

$$Q_\psi(s,a) \leftarrow Q_\psi(s,a) + \alpha\left(r + \gamma \max_{a'} Q_{\psi'}(s',a') - Q_\psi(s,a)\right)$$

Notice that $Q_\psi(s,a)$ is updated using the maximum over every action a' of $Q_{\psi'}(s',a')$. In doing so, the algorithm will also propagate Q-value estimates backwards from its sub-tasks.

If ψ' is either true or false, then s' is a terminal state in the context of trying to solve ψ. Thus, the future reward will be 0. The Q-learning update is therefore simplified to take that into account. The same holds if s' is a dead-end in the environment.

To illustrate, consider the example in Section 4.1 and suppose the current task is $\Diamond(b \wedge \Diamond c)$. After every state transition, each of the three Q-value functions is updated as if their corresponding formula was the current objective. For example, if a state is encountered where b and d are false and c is true, then $Q_{\Diamond c}$ is updated with a reward of 1, and the other two functions are updated with a reward of 0. If a state is encountered in which b is true and c is false, then $Q_{\Diamond(b \wedge \Diamond c)}$ is updated with a reward of 0, but $Q_{\Diamond c}$ is used to estimate how much reward the agent expects for completing the task.

4.2.4 Curriculum learning. On every iteration of LPOPL, a curriculum learning method is used to select a task as the objective of the next episode. While any curriculum learning approach can be used, we use the following simple method which assumes the tasks in Φ are in some order $\varphi_0, \ldots, \varphi_{k-1}$ where $|\Phi| = k$. For each i, we keep track of the percentage p_i of episodes for which an episode was run on task φ_i and that task was solved in some maximum number of steps $n_{\max}$. This follows Andreas et al. [2] as a way to evaluate the effectiveness of the policy on each task.

```
Function ExtractSubtasks($\Phi, \mathcal{P}$)
    $\Phi^+ \leftarrow \Phi$;
    repeat
        $\Phi_{\text{last}} \leftarrow \Phi^+$;
        for $\langle \tau, \varphi \rangle \in (2^{\mathcal{P}} \times \Phi_{\text{last}})$ do
            $\varphi' = \text{prog}(\tau, \varphi)$;
            if $\varphi' \notin \{\text{true}, \text{false}\}$ then
                $\Phi^+ \leftarrow \Phi^+ \cup \{\varphi'\}$;
    until $\Phi^+ = \Phi_{\text{last}}$;
    return $\Phi^+$;
```

Algorithm 2: Using progression to extract sub-tasks.

```
Function RunEpisode($Q, \varphi, L, \gamma, N$)
    $t \leftarrow 0$; $s \leftarrow$ GetInitialState();
    while $t < N$ do
        $\varphi \leftarrow \text{prog}(L(s), \varphi)$ ;
        if $\varphi \in \{\text{true}, \text{false}\}$ or EnvDeadEnd($s$) then
            break;
        $a \leftarrow$ GetActionEpsilonGreedy($Q_\varphi, s$);
        $s' \leftarrow$ EnvExecuteAction($s, a$);
        for $Q_\psi \in Q$ do
            $r \leftarrow 0$;
            $\psi' \leftarrow \text{prog}(L(s'), \psi)$;
            if $\psi' = \text{true}$ then
                $r \leftarrow 1$;
            if $\psi' \in \{\text{true}, \text{false}\}$ or EnvDeadEnd($s'$) then
                $Q_\psi(s,a) \leftarrow Q_\psi(s,a) + \alpha\left(r - Q_\psi(s,a)\right)$;
            else
                $Q_\psi(s,a) \leftarrow Q_\psi(s,a) +$
                    $\alpha\left(r + \gamma \max_{a'} Q_{\psi'}(s',a') - Q_\psi(s,a)\right)$;
        $s \leftarrow s'$; $t \leftarrow t + 1$;
    return $t$;
```

Algorithm 3: Performing an episode in the environment.

The selection process is then as follows. If φ_i is the task selected in the j-th episode, the $j+1$-th task selected is $\varphi_{i'}$, where i' is equal to $i+1$ modulo k if p_i is at least as large as some success probability p_{succ}, and $i' = i$ otherwise. For the first episode, φ_0 is selected.

4.3 Theoretical Properties of LPOPL

We now identify several theoretical properties of LPOPL. First, in the tabular case, LPOPL always converges to the optimal policy for all tasks of the given MCTLS. This statement is formally stated and proven in Appendix A. Secondly, in the worst case, Algorithm 2 may produce an exponential number of subtasks on the length of the formula, resulting in an exponential number of updates on each learning step (Algorithm 3). In our experience, this exponential behaviour does not occur in practice. In any case, its impact can also be alleviated by performing the Q-updates in parallel or by sampling a tractable subset of sub-policies to update at each iteration.

4.4 Function Approximation in LPOPL

To extend LPOPL to use function approximation merely requires an appropriate adjustment to the way that each Q-value function is represented, initialized, and updated. For example, when using a neural network for function approximation, initialization involves the construction of $|\Phi^+|$ neural networks, and Q-value function updates will adjust the network parameters instead of table entries.

Using function approximation allows LPOPL to solve harder problems at the cost of losing convergence guarantees. However, LPOPL does not prune optimal policies from being consider when decomposing a task. This is not necessarily true of other approaches and we show the impact of this in the experimental section.

5 EXPERIMENTAL EVALUATION

In this section, we empirically compare LPOPL against three strong baselines — one based on DQN and two based on Hierarchical RL — on Minecraft-like grid maps. The results show that the flexibility of LTL for expressing tasks can improve solution quality and that LPOPL can be effective for multi-task learning.

5.1 Experimental Setup

In this section, we describe the baseline algorithms tested, the test problems, and the hyperparameters used. We note that the source code is publicly available at https://bitbucket.org/RToroIcarte/lpopl.

5.1.1 Baseline algorithms. The first baseline is *Deep Q-Networks with LTL specifications (DQN-L)*, which follows the methodology proposed by Littman et al. [28] to learn LTL-based rewards. The input to a Q-value function is both the state and the progressed LTL task (i.e., using standard RL the baseline solves a cross-product MDP, the one which is described in the appendix in Definition A.1). This is the state-of-the-art approach for learning with a reward function specified in LTL. Our implementation uses one DQN network for the Q-value function of each task in Φ.

The second baseline, *Hierarchical RL with Event options (HRL-E)*, is based on the options framework [41] and its extension to use deep learning [21]. This method learns a meta controller over *options*, each defined by a triple $\langle \mathcal{I}_o, \pi_o, \mathcal{T}_o \rangle$, where $\mathcal{I}_o$ is the set of states where the option is applicable, π_o is the option policy, and $\mathcal{T}_o$ is the set of states where the option terminates. An option policy π_o is learned by rewarding the agent when it reaches states in $\mathcal{T}_o$. Following [21], we created one option per proposition $p \in \mathcal{P}$, where the terminal states are those in which p is true. The meta-controller also receives as input both the state and the progressed LTL task, and gets reward according to the task specification. Every option policy is learned in parallel using off-policy RL.

The third baseline is *Hierarchical RL with LTL pruning (HRL-L)*. This method augments HRL-E by constraining when each option can be applied. For instance, if the progressed task is $\Diamond a \wedge \Diamond b$, then only achieving either a or b will lead to further progression of the task. Intuitively, this means the only options that should be considered are π_a and π_b. The LTL specification is thus being used to prune the set of applicable options for the meta-controller.

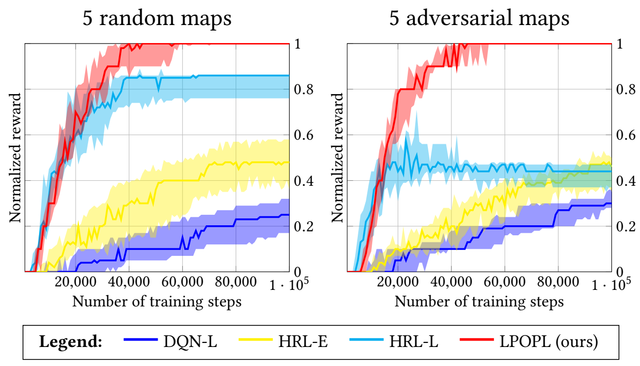

Figure 2: Experiments on sequence-based subtasks.

Intuitively, HRL-E and HRL-L are both decomposing a given task like $\Diamond(p \wedge \Diamond q)$ into two parts: $\Diamond p$ and $\Diamond q$. Ideally, the meta-controller will then learn to apply option π_p followed by option π_q. However, this would not preserve optimality even in the tabular case since the end goal of reaching q can change the best way to first achieve p, as shown in Figure 1. We will see this issue will also impact performance in the experiments below.

5.1.2 Test Problems and Evaluation. Our experiments compare LPOPL using DQN for Q-value function approximation to DQN-L, HRL-E, and HRL-L on two sets of five Minecraft-like grid maps. The first five were generated randomly. The second five, called adversarial maps, were generated specifically to understand how suboptimal the hierarchical methods can be. To do so, we randomly generated a set of $1,000$ maps and chose the five maps with the highest ratio between the locally and globally optimal solutions. Each algorithm was run three times independently per map. After every 100 learning steps on each map, the target policy was tested on all of the given tasks. The reported values are the average normalized discounted reward across all the tasks.

5.1.3 Hyperparameters and Features. All four algorithms use the same features, network architecture, optimizer, and learning parameters. The feature vector is given by the distance of every object from the agent, as used by Andreas et al. [2] and Andrychowicz et al. [3]. The DQN implementation was based on the code from OpenAI Baselines[15]. We use a feedforward network with 2 hidden layers and 64 ReLu units in each layer. The networks were trained using the Adam optimizer [16] with a learning rate of 0.0001. The DQN network learns on every step by randomly sampling 32 transitions from an experience replay buffer of size $25,000$; the target network was updated every 100 steps; the discount factor was 0.9. For curriculum learning, $n_{\max}$ was 100 and p_{succ} was 0.9.

5.2 Experiment 1: Sequences of Sub-Tasks

In the first set of experiments, the task set is given by the 10 Minecraft tasks in Andreas et al. [2], translated directly to LTL. All 10 tasks consist of a single sequence of properties to achieve. For instance, one of the tasks — to *make a bed* — is defined by the following sequence: `got_wood`, `used_toolshed`, `got_grass`, and `used_workbench`. The translation to LTL is $\Diamond(\texttt{got\_wood} \wedge \Diamond(\texttt{used\_toolshed} \wedge \Diamond(\texttt{got\_grass} \wedge \Diamond\texttt{used\_workbench})))$.

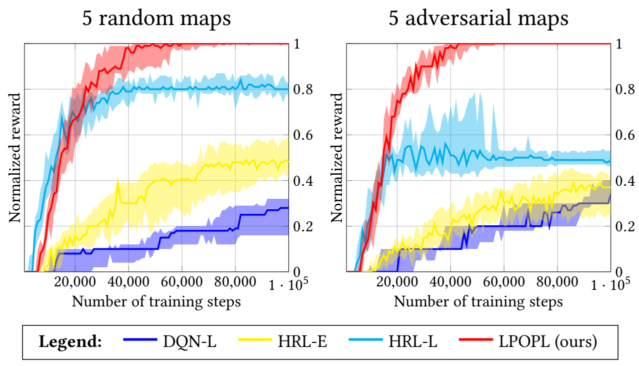

Figure 3: Experiments on tasks with unordered subtasks.

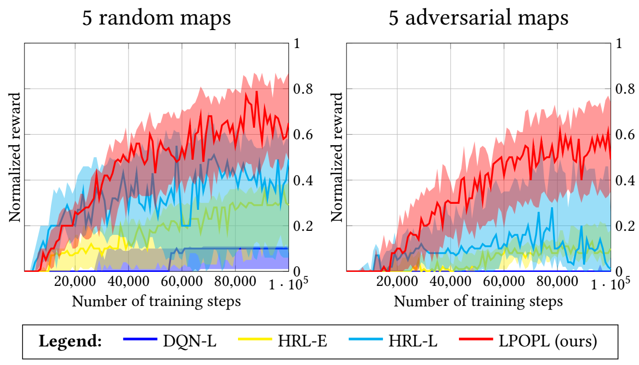

Figure 4: Experiments on tasks with safety constraints.

Figure 2 shows the average normalized discounted reward across the 10 tasks for each algorithm. The shadowed areas show the 25th and 75th percentiles. As shown, LPOPL is able to converge to the optimal solution on all tasks, whereas HRL-L converges to a suboptimal one. The difference is especially clear in the adversarial maps. In addition, we note that the initial rate of learning is comparable between HRL-L and LPOPL despite the fact that LPOPL is learning 27 Q-value functions while HRL-L is learning only 8 option policies.

We also tested the policy sketches method [2] on these task sets, but it was unable to solve even the simplest task. We believe this is because it uses an actor-critic method with only one actor. As such, experiences across consecutive episodes on the same map are highly correlated. This can be catastrophic for such an agent. The actor-critic method is better suited for the experimental setup in the original paper in which a new random map is generated for each episode. However, even if it could be adapted to our setting, it would at best converge to a suboptimal policy, as HRL-L and HRL-E do, since it performs a similar kind of task decomposition.

5.3 Experiment 2: Interleaving Sub-Tasks

The policy sketches method [2] was designed for tasks consisting of sequences of sub-tasks since sketches impose a total order over sub-tasks. However, many of the tasks considered in that paper included subtasks that could have been ordered differently. For example, to make the bed, the agent needs *grass* and a *plank* which is made from wood in the toolshed. However, the order in which the grass is collected and the plank is made can be arbitrary. As such, we rewrote the sequence-based tasks by removing unnecessary orderings over parts of the task that are independent. For example, the bed-making task can be rewritten as follows:

$$\begin{aligned}&\Diamond(\texttt{got\_wood} \wedge \Diamond(\texttt{used\_toolshed} \wedge \Diamond\texttt{used\_workbench}))\\&\wedge \Diamond(\texttt{got\_grass} \wedge \Diamond\texttt{used\_workbench})\end{aligned} \tag{2}$$

Figure 3 shows the results after removing the unnecessary orders on all 10 tasks. In this case, LPOPL learns 34 policies, while HRL-L learns 8 option policies and 10 meta-controllers (in the previous task, meta-controllers were not needed). The results follow a similar trend to the previous experiment. Even though LPOPL is learning more policies than before, it still initially improves at a comparable rate to HRL-L and always converges to the optimal policy.

We note that the optimal reward on these tasks is 8% higher than on the totally ordered tasks. This demonstrates the potential of LTL's ability to allow partial orderings. It also motivates the need for methods like LPOPL, since approaches based on goal sequences like DYNA-H [38] and Policy Sketches [2] cannot handle such tasks.

5.4 Experiment 3: Safety Constraints

The last experiment explores another benefit of LTL: the ability to specify safety constraints. Here, we augment the 10 tasks with interleaved sub-tasks by requiring the agent to enter the shelter at night (to avoid zombies). To do so, we added a clock to the environment such that each step advances the clock by one hour. Episodes start at noon, sunrise is at 5:00 am and sunset at 9:00 pm.

For example, to incorporate this constraint in the bed-making task, we would modify Equation 2 by replacing each subformula of the form $\Diamond\varphi$ with $(\texttt{is\_night} \rightarrow \texttt{at\_shelter})\,\mathsf{U}\,(\varphi \wedge (\texttt{is\_night} \rightarrow \texttt{at\_shelter}))$. In so doing, the safety constraint of being at the shelter at night is enforced up to and including the time that φ is achieved. The result is that if the agent is outside the shelter at night, the formula becomes falsified and so the agent receives a reward of 0.

Figure 4 shows the results when adding this safety constraint. For these tasks, the feature set was augmented with the number of hours until the next sunset. The resulting set of 10 tasks were considerably more difficult to solve and so we set p_{succ} to 0.5. Still, LPOPL outperforms the rest of the approaches.

5.5 Discussion

Our experiments illustrate the merits of LPOPL with respect to alternative decomposition methods based on Hierarchical RL. LPOPL outperforms our three baselines across different sets of challenging tasks. However, our experiments do not provide a definitive evaluation of the effectiveness and scalability of this technique. Whether LPOPL would outperform Hierarchical RL in more complex environments remains an open question. We expect, though, that LPOPL would still outperform our baselines as long as off-policy RL can learn reasonably accurate policies for each subtask.

6 RELATED WORK

Andreas et al. [2] proposed a method that provides a task specification, called a *sketch*, in the form of a sequence of tokens that describes a solution to the task. The agent then learns one policy per token. When acting in the environment, the agent follows the policy corresponding to the current token and decides whether or not to move to the next token. This approach might not converge to an optimal policy because it reuses the same policies across different tasks. While the reward function is *not* defined by the sketch, the extension to that case is trivial. Hence, we consider policy sketches as the closest alternative to LTL specifications for multi-task RL. Our approach exploits similar ideas, but uses a more expressive language and does not prune optimal policies from consideration.

Our approach builds on the *Hierarchical Reinforcement Learning (HRL)* literature [7]. HRL algorithms look for ways to decompose complex tasks into simpler sub-tasks that can be reused over time. Some well-known HRL approaches are H-DYNA [38], MAXQ [12], HAMs [34], and Options [41]. All assume the existence of an external Markovian reward function. The hierarchy constrains the space of possible policies, which simplifies learning but may prune optimal policies. Our work exploits a formal specification of the reward function to automatically generate a decomposition that does not risk eliminating optimal policies, reuses sub-task policies, and also optimizes for each sub-task simultaneously using off-policy RL.

At least three previous works have used variants of LTL for task specification in RL [14, 27, 28]. While [14, 28] transform the specification into a reward function that maximizes the probability of satisfying an LTL formula, [27] guides the search for a policy using a measure of distance to satisfaction of the task. None of these approaches exploits the formula structure to decompose the problem. This is the main reason why LPOPL was able to outperform [28] in our multi-task RL experiments. Similarly, we expect LPOPL to outperform [14, 27]. Other works have used LTL to prune policies while maximizing an external reward function in an RL setting [51, 52]. Outside of RL, there has been work on using temporal logics – including co-safe LTL [23, 24], LTL$_f$ [10], and LDL$_f$ [8] – to specify non-Markovian rewards for MDP-like formalisms.

Previous work on multi-task RL has focused on creating agents that solve new tasks by transferring learning from previous tasks (e.g., [9, 25, 39, 42, 44–46, 53]). Unlike our approach, these methods do not receive information about the new task to solve (i.e., they cannot transfer experience or policies without handling the *transfer trade-off* [26]) and they cannot handle non-Markovian rewards.

7 CONCLUDING REMARKS

We presented an approach to specifying multiple tasks for an RL agent using co-safe LTL formulae, and an algorithmic framework, LPOPL, that exploits the structure of those LTL specifications, outperforming several baselines for learning tasks. Trivial extensions include handling the case where new tasks arrive online and handling arbitrary LTL formulae, or finite LTL. Future directions include running experiments in other environments and incorporating other off-policy RL algorithms into this framework.

A PROOF OF OPTIMALITY

We first construct a MDP $\mathcal{M}'$ for a given NMRDP $\mathcal{M}$ and show that an optimal policy for $\mathcal{M}'$ yields an optimal policy for $\mathcal{M}$.

Definition A.1. Let $\mathcal{T} = \langle S, A, T, \mathcal{P}, L, \Phi, \gamma \rangle$ be an MCTLS, $\mathcal{M} = \langle S, A, T, R_\varphi, \gamma \rangle$ be an NMRDP specified from $\mathcal{T}$ where $\varphi \in \Phi$, and Ψ be the set of formulae that can be progressed from φ. Then we define the MDP $\mathcal{M}' = \langle S', A', T', R', \gamma' \rangle$, where $S' = S \times \Psi$, $A' = A$, $\gamma' = \gamma$, $T'(\langle s, \psi \rangle, a, \langle s', \psi' \rangle) = T(s, a, s')$ iff $\psi' = \text{prog}(L(s'), \psi)$ (zero otherwise), and $R'(\langle s, \psi \rangle, a, \langle s', \psi' \rangle) = 1$ iff $\psi' = \text{prog}(L(s'), \psi) =$ true and $\psi \neq \psi'$ (zero otherwise).

Theorem A.2. *Let $\mathcal{T} = \langle S, A, T, \mathcal{P}, L, \Phi, \gamma \rangle$ be an MCTLS, $\mathcal{M} = \langle S, A, T, R_\varphi, \gamma \rangle$ be an NMRDP specified from $\mathcal{T}$ where $\varphi \in \Phi$, and $\mathcal{M}' = \langle S', A', T', R', \gamma' \rangle$ be the MDP constructed from $\mathcal{M}$ as in Definition A.1. If π' is an optimal policy for $\mathcal{M}'$, then the policy π, defined as $\pi(\langle s_0, \ldots, s_n \rangle) = \pi'(\langle s_n, \psi \rangle)$ where ψ is the progression of φ using $\langle L(s_0), \ldots, L(s_n) \rangle$, is an optimal policy for $\mathcal{M}$.*

Proof. Consider any policy π_o, state sequence $s_{0:t} = \langle s_0, \ldots, s_t \rangle$, and action a for $\mathcal{M}$. By definition:

$$Q^{\pi_o}_{\mathcal{M}}(s_{0:t}, a) = \mathbb{E}_{\pi_o, T}\left[\sum_{k=0}^{\infty} \gamma^k R_\varphi(\sigma_{0:t+k+1}) \,\middle|\, A_t = a \right]$$

where $\sigma_{i:j} = \langle \sigma_i, \ldots, \sigma_j \rangle = \langle L(s_i), \ldots, L(s_j) \rangle$. Using Theorem 3.2, we can progress φ to ψ over the sequence $\sigma_{0:t}$:

$$Q^{\pi_o}_{\mathcal{M}}(s_{0:t}, a) = \mathbb{E}_{\pi_o, T}\left[\sum_{k=0}^{\infty} \gamma^k R_\psi(\sigma_{t:t+k+1}) \,\middle|\, A_t = a \right]$$

By definition, $R_\psi(\sigma_{t:t+k+1})$ is *one* iff $\sigma_{t:t+k} \not\models \psi$ and $\sigma_{t:t+k+1} \models \psi$, and *zero* otherwise. As $\sigma_{t:t+k} \not\models \psi$ iff $\psi_k = \text{prog}(\sigma_{t:t+k}, \psi)$ is not true and $\sigma_{t:t+k+1} \models \psi$ iff $\psi_{k+1} = \text{prog}(\sigma_{t:t+k+1}, \psi)$ is true, then $R_\psi(\sigma_{t:t+k+1})$ is equivalent to $R'(\langle s_{t+k}, \psi_k \rangle, \cdot, \langle s_{t+k+1}, \psi_{k+1} \rangle)$:

$$Q^{\pi_o}_{\mathcal{M}}(s_{0:t}, a) = \mathbb{E}_{\pi_o, T}\left[\sum_{k=0}^{\infty} \gamma^k R'(\cdots) \,\middle|\, A_t = a \right]$$

The previous expected value is over π_o and T that are defined in terms of $\mathcal{M}$. However, we can construct an equivalent policy π'_o for $\mathcal{M}'$, where $\pi_o(s_{t:t+k+1}) = \pi'_o(\langle s_{t+k+1}, \psi_{k+1} \rangle)$. Moreover, $T(s_{t+k}, a', s_{t+k+1})$ is equivalent to $T'(\langle s_{t+k}, \psi_k \rangle, a', \langle s_{t+k+1}, \psi_{k+1} \rangle)$ for every $a' \in A$. Replacing π_0 by π'_0 and T by T' in the expectation, we get the following equivalence: $Q^{\pi_o}_{\mathcal{M}}(s_{0:t}, a) = Q^{\pi'_o}_{\mathcal{M}'}(\langle s_t, \psi \rangle, a)$ (where ψ is the progression of φ using $\sigma_{0:t}$) for any policy π_o, action a, and state sequence $s_{0:t}$. In particular, if π' is optimal in $\mathcal{M}'$, then $Q^{\pi}_{\mathcal{M}}(s_{0:t}, a) = Q^{\pi'}_{\mathcal{M}'}(\langle s_t, \psi \rangle, a) \geq Q^{\pi'_o}_{\mathcal{M}'}(\langle s_t, \psi \rangle, a) = Q^{\pi_o}_{\mathcal{M}}(s_{0:t}, a)$ for every policy π'_o and action a. Therefore, π is optimal in $\mathcal{M}$. □

We can now show our desired result:

Theorem A.3. *Let $\mathcal{T} = \langle S, A, T, \mathcal{P}, L, \Phi, \gamma \rangle$ be an MCTLS. Then LPOPL with Q-learning converges to an optimal policy for every NMRDP $\{\langle S, A, T, R_\varphi, \gamma \rangle : \varphi \in \Phi\}$ specified by $\mathcal{T}$.*

Proof. The Q-updates applied to every $Q_\varphi \in Q$ in Algorithm 3 are equivalent to solving the cross-product MDP $\mathcal{M}'$ between $\mathcal{M} = \langle S, A, T, R_\varphi, \gamma \rangle$ and φ (Definition A.1). The use of an ϵ-greedy behaviour policy means every state-action pair is updated an infinite number of times in the limit. Hence, every $Q_\varphi(s, a) \in Q$ converges to $Q^*_\varphi(s, \varphi, a)$ in $\mathcal{M}'$ [50]. Since the optimal policy for $\mathcal{M}'$ is also optimal in $\mathcal{M}$ by Theorem A.2, LPOPL converges to an optimal policy for $\mathcal{M}$ for every $\varphi \in \Phi$, in the limit. □

ACKNOWLEDGMENTS

We gratefully acknowledge funding from NSERC and CONICYT.

REFERENCES

[1] Pieter Abbeel and Andrew Y. Ng. 2004. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the 21st International Conference on Machine Learning (ICML)*. 1.

[2] Jacob Andreas, Dan Klein, and Sergey Levine. 2017. Modular Multitask Reinforcement Learning with Policy Sketches. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*. 166–175.

[3] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. 2017. Hindsight experience replay. In *Proceedings of the 30th Conference on Advances in Neural Information Processing Systems (NIPS)*. 5048–5058.

[4] Brenna D. Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. 2009. A survey of robot learning from demonstration. *Robotics and autonomous systems* 57, 5 (2009), 469–483.

[5] Fahiem Bacchus, Craig Boutilier, and Adam J. Grove. 1996. Rewarding Behaviors. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI)*. 1160–1167.

[6] Fahiem Bacchus and Froduald Kabanza. 2000. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence* 116, 1-2 (2000), 123–191.

[7] Andrew G. Barto and Sridhar Mahadevan. 2003. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems* 13, 4 (2003), 341–379.

[8] Ronen I Brafman, Giuseppe De Giacomo, and Fabio Patrizi. 2018. LTLf/LDLf Non-Markovian Rewards. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI)*. to appear.

[9] Emma Brunskill and Lihong Li. 2013. Sample Complexity of Multi-task Reinforcement Learning. In *Proceedings of the 29th Conference on Uncertainty in Artificial Intelligence (UAI)*. 122.

[10] Alberto Camacho, Oscar Chen, Scott Sanner, and Sheila A. McIlraith. 2017. Non-Markovian Rewards Expressed in LTL: Guiding Search Via Reward Shaping. In *Proceedings of the 10th Symposium on Combinatorial Search (SOCS)*. 159–160.

[11] Misha Denil, Sergio Gómez Colmenarejo, Serkan Cabi, David Saxton, and Nando de Freitas. 2017. Programmable agents. *arXiv preprint arXiv:1706.06383* (2017).

[12] Thomas G. Dietterich. 2000. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research* 13 (2000), 227–303.

[13] Juraj Dzifcak, Matthias Scheutz, Chitta Baral, and Paul Schermerhorn. 2009. What to do and how to do it: Translating natural language directives into temporal and dynamic logic representation for goal management and action execution. In *Proceedings of the 2009 IEEE International Conference on Robotics and Automation (ICRA)*. 4163–4168.

[14] Mohammadhosein Hasanbeig, Alessandro Abate, and Daniel Kroening. 2018. Logically-Correct Reinforcement Learning. *arXiv preprint arXiv:1801.08099* (2018).

[15] Christopher Hesse, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. 2017. OpenAI Baselines. https://github.com/openai/baselines. (2017).

[16] Diederik Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[17] W. Bradley Knox and Peter Stone. 2008. Tamer: Training an agent manually via evaluative reinforcement. In *Proceedings of the 7th IEEE International Conference on Development and Learning (ICDL)*. 292–297.

[18] Jens Kober, J. Andrew Bagnell, and Jan Peters. 2013. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research* 32, 11 (2013), 1238–1274.

[19] Samantha Krening, Brent Harrison, Karen Feigh, Charles Isbell, Mark Riedl, and Andrea Thomaz. 2016. Learning from Explanations using Sentiment and Advice in RL. *IEEE Transactions on Cognitive and Developmental Systems* (2016).

[20] Gregory Kuhlmann, Peter Stone, Raymond Mooney, and Jude Shavlik. 2004. Guiding a reinforcement learner with natural language advice: Initial results in RoboCup soccer. In *Proceedings of the AAAI Workshop on Supervisory Control of Learning and Adaptive Systems*.

[21] Tejas D. Kulkarni, Karthik Narasimhan, Ardavan Saeedi, and Josh Tenenbaum. 2016. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. In *Proceedings of the 29th Conference on Advances in Neural Information Processing Systems (NIPS)*. 3675–3683.

[22] Orna Kupferman and Moshe Y. Vardi. 2001. Model checking of safety properties. *Formal Methods in System Design* 19, 3 (2001), 291–314.

[23] Bruno Lacerda, David Parker, and Nick Hawes. 2014. Optimal and dynamic planning for Markov decision processes with co-safe LTL specifications. In *Proceedings of the 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 1511–1516.

[24] Bruno Lacerda, David Parker, and Nick Hawes. 2015. Optimal Policy Generation for Partially Satisfiable Co-Safe LTL Specifications. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI)*. 1587–1593.

[25] Alessandro Lazaric and Mohammad Ghavamzadeh. 2010. Bayesian multi-task reinforcement learning. In *Proceedings of the 27th International Conference on Machine Learning (ICML)*. 599–606.

[26] Alessandro Lazaric and Marcello Restelli. 2011. Transfer from multiple MDPs. In *Proceedings of the 25th Conference on Advances in Neural Information Processing Systems (NIPS)*. 1746–1754.

[27] Xiao Li, Cristian Ioan Vasile, and Calin Belta. 2017. Reinforcement Learning With Temporal Logic Rewards. In *Proceedings of the 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 3834–3839.

[28] Michael L. Littman, Ufuk Topcu, Jie Fu, Charles Isbell, Min Wen, and James MacGlashan. 2017. Environment-Independent Task Specifications via GLTL. *arXiv preprint arXiv:1704.04341* (2017).

[29] James MacGlashan, Mark K. Ho, Robert Tyler Loftin, Bei Peng, Guan Wang, David L. Roberts, Matthew E. Taylor, and Michael L. Littman. 2017. Interactive Learning from Policy-Dependent Human Feedback. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*. 2285–2294.

[30] James MacGlashan, Michael Littman, Robert Loftin, Bei Peng, David Roberts, and Matthew E. Taylor. 2014. Training an agent to ground commands with reward and punishment. In *Proceedings of the AAAI Workshop on Machine Learning for Interactive Systems*.

[31] Richard Maclin and Jude W. Shavlik. 1996. Creating advice-taking reinforcement learners. *Machine Learning* 22, 1-3 (1996), 251–281.

[32] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529–533.

[33] Emilio Parisotto, Jimmy Lei Ba, and Ruslan Salakhutdinov. 2015. Actor-mimic: Deep multitask and transfer reinforcement learning. *arXiv preprint arXiv:1511.06342* (2015).

[34] Ronald Parr and Stuart J. Russell. 1998. Reinforcement learning with hierarchies of machines. In *Proceedings of the 11th Conference on Advances in Neural Information Processing Systems (NIPS)*. 1043–1049.

[35] Amir Pnueli. 1977. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science (FOCS)*. 46–57.

[36] Tom Schaul, Daniel Horgan, Karol Gregor, and David Silver. 2015. Universal value function approximators. In *Proceedings of the 32nd International Conference on Machine Learning (ICML)*. 1312–1320.

[37] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[38] Satinder P. Singh. 1992. Reinforcement learning with a hierarchy of abstract models. In *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI)*. 202–207.

[39] Matthijs Snel and Shimon Whiteson. 2011. Multi-task reinforcement learning: Shaping and feature selection. In *Proceedings of the European Workshop on Reinforcement Learning*. 237–248.

[40] Richard S. Sutton and Andrew G. Barto. 1998. *Reinforcement learning: An introduction*. Vol. 1. MIT press Cambridge.

[41] Richard S. Sutton, Doina Precup, and Satinder Singh. 1999. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence* 112, 1-2 (1999), 181–211.

[42] Fumihide Tanaka and Masayuki Yamamura. 2003. Multitask reinforcement learning on the distribution of MDPs. In *Proceedings of the 2nd IEEE International Symposium on Computational Intelligence in Robotics and Automation (CIRA)*. 1108–1113.

[43] Matthew E. Taylor and Sonia Chernova. 2010. Integrating human demonstration and reinforcement learning: Initial results in human-agent transfer. In *Proceedings of the AAMAS Workshop on Agents Learning Interactively with Human Teachers*. 23.

[44] Matthew E. Taylor and Peter Stone. 2007. Cross-domain transfer for reinforcement learning. In *Proceedings of the 24th International Conference on Machine Learning (ICML)*. 879–886.

[45] Matthew E. Taylor and Peter Stone. 2009. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research* 10, Jul (2009), 1633–1685.

[46] Yee Teh, Victor Bapst, Wojciech M Czarnecki, John Quan, James Kirkpatrick, Raia Hadsell, Nicolas Heess, and Razvan Pascanu. 2017. Distral: Robust multitask reinforcement learning. In *Proceedings of the 30th Conference on Advances in Neural Information Processing Systems (NIPS)*. 4499–4509.

[47] Sylvie Thiébaux, Charles Gretton, John K Slaney, David Price, Froduald Kabanza, et al. 2006. Decision-Theoretic Planning with non-Markovian Rewards. *Journal of Artificial Intelligence Research* 25 (2006), 17–74.

[48] Andrea L. Thomaz, Guy Hoffman, and Cynthia Breazeal. 2006. Reinforcement learning with human teachers: Understanding how people want to teach robots. In *Proceedings of the 15th IEEE International Symposium on Robot and Human Interactive Communication (ROMAN)*. 352–357.

[49] Rodrigo Toro Icarte, Toryn Q. Klassen, Richard Valenzano, and Sheila A. McIlraith. 2018. Advice-Based Exploration in Model-Based Reinforcement Learning. In *Proceedings of the 31sh Canadian Conference on Artificial Intelligence (CCAI)*. to appear.

[50] Christopher J. C. H. Watkins and Peter Dayan. 1992. Q-learning. *Machine learning* 8, 3-4 (1992), 279–292.
[51] Min Wen, Ivan Papusha, and Ufuk Topcu. 2017. Learning from Demonstrations with High-Level Side Information. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI)*. 3055–3061.
[52] Min Wen and Ufuk Topcu. 2016. Probably Approximately Correct Learning in Stochastic Games with Temporal Logic Specifications. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI)*. 3630–3636.
[53] Aaron Wilson, Alan Fern, Soumya Ray, and Prasad Tadepalli. 2007. Multi-task reinforcement learning: a hierarchical Bayesian approach. In *Proceedings of the 24th International Conference on Machine Learning (ICML)*. 1015–1022.