Reward Machines: Exploiting Reward Function Structure in Reinforcement Learning

Rodrigo Toro Icarte RNTORO@CS.TORONTO.EDU
University of Toronto, Vector Institute

Toryn Q. Klassen TORYN@CS.TORONTO.EDU
University of Toronto, Vector Institute

Richard Valenzano RICK.VALENZANO@ELEMENTAI.COM
Element AI

Sheila A. McIlraith SHEILA@CS.TORONTO.EDU
University of Toronto, Vector Institute

Abstract

Reinforcement learning (RL) methods usually treat reward functions as black boxes. As such, these methods must extensively interact with the environment in order to discover rewards and optimal policies. In most RL applications, however, users have to program the reward function and, hence, there is the opportunity to treat reward functions as white boxes instead – to show the reward function's code to the RL agent so it can exploit its internal structures to learn optimal policies faster. In this paper, we show how to accomplish this idea in two steps. First, we propose reward machines (RMs), a type of finite state machine that supports the specification of reward functions while exposing reward function structure. We then describe different methodologies to exploit such structures, including automated reward shaping, task decomposition, and counterfactual reasoning for data augmentation. Experiments on tabular and continuous domains show the benefits of exploiting reward structure across different tasks and RL agents.

1. Introduction

A standard assumption in reinforcement learning (RL) is that the agent does not have access to the environment model (Sutton & Barto, 1998). This means that it does not know the environment's transition probabilities or reward function. To learn optimal behaviour, an RL agent must therefore interact with the environment and learn from its experience. While assuming that the transition probabilities are unknown seems reasonable, there is less reason to hide the reward function from the agent. Artificial agents cannot inherently perceive reward from the environment; someone must program those rewards functions. This is true even if the agent is interacting with the real world. Typically, though, a programmed reward function is given as a black box to the agent. The agent can query the function for the reward in the current situation, but does not have access to whatever structures or high-level ideas the programmer may have used in defining it. However, an agent that had access to the specification of the reward function might be able to use such information to learn optimal policies faster. We consider different ways to do so in this work.

Previous work on giving an agent knowledge about the reward function focused on defining a task specification language – usually based on sub-goal sequences (Singh, 1992a, 1992b) or linear temporal logic (Li et al., 2017; Littman et al., 2017; Toro Icarte et al., 2018b; Hasanbeig et al., 2018; Camacho et al., 2019; De Giacomo et al., 2019; Shah et al., 2020) – and then generate a reward function towards fulfilling that specification. In this work, we instead directly tackle the problem of defining reward functions that expose structure to the agent. As such, our approach is able to reward behaviors to varying degrees in manners that cannot be expressed by previous approaches.

There are two main contributions of this work. First, we introduce a type of finite state machine, called the *reward machine*, which we use in defining rewards. A reward machine allows for composing different reward functions in flexible ways, including concatenation, loops, and conditional rules. As an agent acts in the environment, moving from state to state, it also moves from state to state within a reward machine (as determined by high-level events detected within the environment). After every transition, the reward machine outputs the reward function the agent should use at that time. For example, we might construct a reward machine for "*delivering coffee to an office*" using two states. In the first state, the agent does not receive any rewards, but it moves to the second state whenever it gets the coffee. In the second state, the agent gets rewards after delivering the coffee. The advantage of defining rewards this way is that the agent knows that the problem consists of two stages and might use this information to speed up learning.

Our second contribution is a collection of RL methods that can exploit a reward machine's internal structure to improve sample efficiency. These methods include using the reward machine for decomposing the problem, shaping the reward functions, and using counterfactual reasoning to learn policies faster. We also discuss conditions under which these approaches are guaranteed to converge to optimal policies. Finally, we empirically demonstrate the value of exploiting reward structures in discrete and continuous domains.

This paper builds upon Toro Icarte et al. (2018c) – where we originally proposed reward machines and an approach, called *q-learning for reward machines (QRM)*, to exploit the structure exposed by a reward machine. This paper also covers an approach for automated reward shaping from a given reward machine that we first introduced in Camacho et al. (2019). Since then, we have gathered additional practical experience and theoretical understanding about reward machines that are reflected in this paper. Concretely, we provide a cleaner definition of a reward machine and propose two novel approaches to exploit its structure, called *counterfactual experiences for reward machines (CRM)* and *hierarchical RL for reward machines (HRM)*. We expanded the related work discussion to include recent trends in reward machine research and included new empirical results in single task, multitask, and continuous control learning problems. Finally, we have released a new implementation of our code that is fully compatible with the OpenAI gym API (Brockman et al., 2016). We hope that this paper and code will facilitate future research on reward machines.

2. Preliminaries

In this section, we define the relevant terminology and notation regarding reinforcement learning. We also describe a family of off-policy approaches to RL, including tabular q-learning (Watkins & Dayan, 1992), deep q-networks (Mnih et al., 2015), and deep deterministic policy gradient (Lillicrap et al., 2016) methods.

2.1 Reinforcement Learning

The RL problem consists of an agent interacting with an unknown environment. Usually, the environment is modeled as a *Markov decision process (MDP).* An MDP is a tuple $\mathcal{M} = \langle S, A, r, p, \gamma \rangle$ where S is a finite set of *states*, A is a finite set of *actions*, $r : S \times A \times S \to \mathbb{R}$ is the *reward function*, $p(s_{t+1}|s_t, a_t)$ is the *transition probability distribution*, and $\gamma \in (0, 1]$ is the *discount factor.* In some cases, a subset of the states are labelled as *terminal states.*

A *policy* $\pi(a|s)$ is a probability distribution over the actions $a \in A$ given a state $s \in S$. At each time step t, the agent is in a particular state $s_t \in S$, selects an action a_t according to $\pi(\cdot|s_t)$, and executes a_t. The agent then receives a new state $s_{t+1} \sim p(\cdot|s_t, a_t)$ and an immediate reward $r(s_t, a_t, s_{t+1})$ from the environment. The process then repeats from s_{t+1} until potentially reaching a terminal state. The agent's goal is to find a policy π^* that maximizes the expected discounted future reward from every state in S.

The *q-function* $q^\pi(s, a)$ under a policy π is defined as the expected discounted future reward of taking action a in state s and then following policy π. It is known that every *optimal policy* π^* satisfies the *Bellman optimality* equations (where $q^* = q^{\pi^*}$):

$$q^*(s,a) = \sum_{s' \in S} p(s'|s,a) \left(r(s,a,s') + \gamma \max_{a' \in A} q^*(s',a') \right) \ ,$$

for every state $s \in S$ and action $a \in A$. Note that, if q^* is known, then an optimal policy can be computed by always selecting the action a with the highest value of $q^*(s, a)$.

2.2 Tabular Q-Learning

Tabular q-learning (Watkins & Dayan, 1992) is a well-known approach for RL. This algorithm works by using the agent's experience to estimate the optimal q-function. It begins with an initialization (often just a random initialization) of the estimated value of every state-action pair (s, a). We denote this q-value estimate as $\tilde{q}(s, a)$. On every iteration, the agent observes the current state s and chooses an action a according to some exploratory policy. One common exploratory policy is the ϵ-greedy policy, which selects a random action with probability ϵ, and $\arg\max_a \tilde{q}(s, a)$ with probability $1 - \epsilon$. Given the resulting state s' and immediate reward $r(s, a, s')$, this experience is used to update $\tilde{q}(s, a)$ as follows:

$$\tilde{q}(s,a) \xleftarrow{\alpha} r(s,a,s') + \gamma \max_{a'} \tilde{q}(s',a') \ ,$$

where α is an hyperparameter called the *learning rate*, and we use $x \xleftarrow{\alpha} y$ as shorthand notation for $x \leftarrow x + \alpha \cdot (y - x)$. Note that $\tilde{q}(s, a) \xleftarrow{\alpha} r(s, a, s')$ when s' is a terminal state.

Tabular q-learning is guaranteed to converge to an optimal policy in the limit as long as each state-action pair is visited infinitely often. This algorithm is an *off-policy* learning method since it can learn from the experience generated by any policy. Unfortunately, tabular q-learning is impractical when solving problems with large state spaces. In such cases, *function approximation* methods like DQN are often used.

2.3 Deep Q-Networks (DQN)

Deep Q-Network (DQN), proposed by Mnih et al. (2015), is a method which approximates the q-function with an estimate $\tilde{q}_\theta(s, a)$ using a deep neural network with parameters θ.

To train the network, mini-batches of experiences (s, a, r, s') are randomly sampled from an *experience replay* buffer and used to minimize the square error between $\tilde{q}_\theta(s, a)$ and the Bellman estimate $r + \gamma \max_{a'} \tilde{q}_{\theta'}(s', a')$. The updates are made with respect to a *target network* with parameters θ'. The parameters θ' are held fixed when minimizing the square error, but updated to θ after a certain number of training updates. The role of the target network is to stabilize learning. DQN inherits the off-policy behavior from tabular q-learning, but is no longer guaranteed to converge to an optimal policy.

Since its original publication, several improvements have been proposed to DQN. We consider one of them in this paper: *Double DQN* (Van Hasselt et al., 2016). Double DQN uses two neural networks, parameterized by θ and θ', to decouple action selection from value estimation, and thereby decreases the overestimation bias that DQN is known to suffer from. Note that double DQN usually outperforms DQN while preserving its off-policy nature.

2.4 Deep Deterministic Policy Gradient (DDPG)

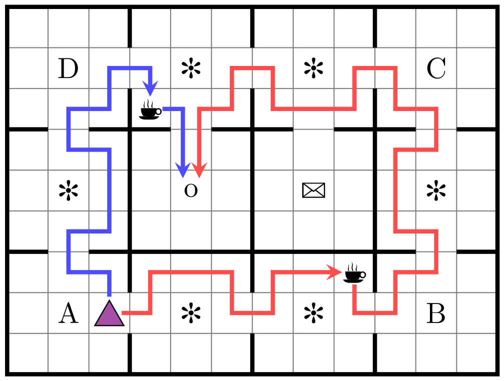
DQN cannot solve continuous control problems because DQN's network have one output unit per possible action and the space of possible actions is infinite in continuous control problems. For those cases, actor-critic approaches such as *Deep Deterministic Policy Gradient (DDPG)* (Lillicrap et al., 2016) are preferred. DDPG is an off-policy actor-critic approach that also uses neural networks for approximating the q-value $\tilde{q}_\theta(s, a)$. However, in this case the action can take continuous values. To decide which action to take in a given state s, an actor network $\pi_\mu(s)$ is learned. The actor network receives the current state and outputs a (possibly continuous) action to execute in the environment.

Training $\tilde{q}_\theta(s, a)$ is done by minimizing the Bellman's error and letting the actor policy select the next action. This is, given a set of experiences (s, a, r, s') sampled from the experience replay buffer, θ is updated towards minimizing the square error between $\tilde{q}_\theta(s, a)$ and $r + \gamma \tilde{q}_{\theta'}(s', \pi_{\mu'}(s'))$, where θ' and μ' are the parameters of target networks for the q-value estimate and the actor policy. Training the actor policy $\pi_\mu(s)$ is done by moving its output towards $\arg\max_a \tilde{q}_\theta(s, a)$. To do so, $\pi_\mu(s)$ is updated using the expected gradient of $\tilde{q}_\theta(s, a)$ with respect to $a = \pi_\mu(s)$: $\nabla_\mu[\tilde{q}_\theta(s, a)|s = s_t, a = \pi_\mu(s_t)]$. This gradient is approximated using sampled mini-batches from the experience replay buffer. Finally, the target networks' parameters are periodically updated as follows: $\theta' \xleftarrow{\tau} \theta$ and $\mu' \xleftarrow{\tau} \mu$, where $\tau \in (0, 1)$.

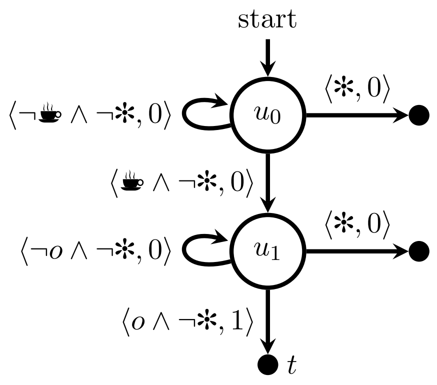
3. Reward Machines

In this section, we introduce a novel type of finite state machine, called the *reward machine (RM)*. An RM takes abstracted descriptions of the environment as input, and outputs reward functions. The intuition is that the agent will be rewarded by different reward functions at different times, depending on the state in the RM. Hence, an RM can be used to define temporally extended (and as such, non-Markovian) tasks and behaviors. We then show that an RM can be interpreted as specifying a single reward function over a larger state space, and consider types of reward functions that can be expressed using RMs.

As a running example, consider the *office gridworld* presented in Figure 1a. In this environment, the agent can move in the four cardinal directions. It picks up coffee if at location ☕, picks up the mail if at location ✉, and delivers the coffee and mail to an office if at location o. The building contains decorations ✻, which the agent breaks if it steps on



(a) The office gridworld


start
u_0
u_1
$\langle \neg☕ \wedge \neg ✻, 0\rangle$
$\langle ✻, 0\rangle$
$\langle ☕ \wedge \neg ✻, 0\rangle$
$\langle \neg o \wedge \neg ✻, 0\rangle$
$\langle ✻, 0\rangle$
$\langle o \wedge \neg ✻, 1\rangle$
t

(b) A simple reward machine

Figure 1: An example environment and one reward machine for it

them. Finally, there are four marked locations: A, B, C, and D. In the rest of this section, we will show how to define tasks for an RL agent in this environment using RMs.

A reward machine is defined over a set of propositional symbols $\mathcal{P}$. Intuitively, $\mathcal{P}$ is a set of relevant high-level events from the environment that the agent can detect. For example, in the office gridworld environment, we can define $\mathcal{P} = \{☕, ✉, o, ✻, A, B, C, D\}$, where event $e \in \mathcal{P}$ occurs when the agent is at location e. We can now formally define a reward machine as follows:

Definition 3.1 (reward machine)**.** *Given a set of propositional symbols $\mathcal{P}$, a set of (environment) states S, and a set of actions A, a reward machine (RM) is a tuple $\mathcal{R}_{\mathcal{P}SA} = \langle U, u_0, F, \delta_u, \delta_r\rangle$ where U is a finite set of states, $u_0 \in U$ is an initial state, F is a finite set of terminal states (where $U \cap F = \emptyset$), δ_u is the state-transition function, $\delta_u : U \times 2^{\mathcal{P}} \rightarrow U \cup F$, and δ_r is the state-reward function, $\delta_r : U \rightarrow [S \times A \times S \rightarrow \mathbb{R}]$.*

A reward machine $\mathcal{R}_{\mathcal{P}SA}$ starts in state u_0, and at each subsequent time is in some state $u_t \in U \cup F$. At every step t, the machine receives as input a *truth assignment* σ_t, which is a set that contains exactly those propositions in $\mathcal{P}$ that are currently true in the environment. For example, in the office gridworld, $\sigma_t = \{e\}$ if the agent is at a location marked as e. Then the machine moves to the next state $u_{t+1} = \delta_u(u_t, \sigma_t)$ according to the state-transition function, and outputs a reward function $r_t = \delta_r(u_t)$ according to the state-reward function. This process repeats until the machine reaches a terminal state. Note that reward machines can model never-ending tasks by defining $F = \emptyset$.

In our examples, we will be considering *simple* reward machines (which we later prove are a particular case of reward machines), defined as follows:

Definition 3.2 (simple reward machine)**.** *Given a set of propositional symbols $\mathcal{P}$, a simple reward machine is a tuple $\mathcal{R} = \langle U, u_0, F, \delta_u, \delta_r\rangle$ where U, u_0, F, and δ_u are defined as in a RM, but the state-reward function $\delta_r : U \times 2^{\mathcal{P}} \rightarrow \mathbb{R}$ depends on $2^{\mathcal{P}}$ and returns a number instead of a function.*

Figure 1b shows a graphical representation of a simple reward machine for the office gridworld. Every node in the graph is a state of the machine, u_0 being the initial state.

Terminal states are represented by a black circle. Each edge is labelled by a tuple $\langle \varphi, c\rangle$, where φ is a propositional logic formula over $\mathcal{P}$ and c is a real number. An edge between u_i and u_j labelled by $\langle \varphi, c\rangle$ means that $\delta_u(u_i, \sigma) = u_j$ whenever $\sigma \models \varphi$ (i.e., the truth assignment σ satisfies φ), and $\delta_r(u_i, \sigma)$ returns a reward of c. For instance, the edge between the state u_1 and the terminal state t labelled by $\langle o \wedge \neg \text{✻}, 1\rangle$ means that the machine will transition from u_1 to t if the proposition o becomes true and ✻ is false, and output a reward of one. Intuitively, this machine outputs a reward of one if and only if the agent delivers coffee to the office while not breaking any decoration. The blue path in Figure 1a shows an optimal way to complete this task, and the red path shows a sub-optimal way.

Now that we have defined a reward machine, we can use it to reward an agent. The overall idea is to replace the standard reward function in an MDP by a reward machine. To do so, we require a *labelling function* $L : S \times A \times S \to 2^{\mathcal{P}}$. L assigns truth values to symbols in $\mathcal{P}$ given an environment experience $e = (s, a, s')$, where s' is the resulting state after executing action a from state s. The labelling function plays the key role of producing the truth assignments that are input to the reward machine, as discussed below.

Definition 3.3. *A Markov decision process with a reward machine (MDPRM) is a tuple $\mathcal{T} = \langle S, A, p, \gamma, \mathcal{P}, L, U, u_0, F, \delta_u, \delta_r\rangle$, where S, A, p, and γ are defined as in an MDP, $\mathcal{P}$ is a set of propositional symbols, L is a labelling function $L : S \times A \times S \to 2^{\mathcal{P}}$, and U, u_0, F, δ_u, and δ_r are defined as in a reward machine.*

The RM in an MDPRM $\mathcal{T}$ is updated at every step of the agent in the environment. If the RM is in state u and the agent performs action a to move from state s to s' in the MDP, then the RM moves to state $u' = \delta_u(u, L(s, a, s'))$ and the agent receives a reward of $r(s, a, s')$, where $r = \delta_r(u)$. For a simple reward machine, the reward is $\delta_r(u, L(s, a, s'))$.

In the running example (Figure 1), for instance, the reward machine starts in u_0 and stays there until the agent reaches a location marked with ✻ or ☕. If ✻ is reached (i.e., a decoration is broken), the machine moves to a terminal state, ending the episode and providing no reward to the agent. In contrast, if ☕ is reached, the machine moves to u_1. While the machine is in u_1, two outcomes might occur. The agent might reach a ✻, moving the machine to a terminal state and returning no reward, or it might reach the office o, also moving the machine to a terminal state but giving the agent a reward of 1.

Note that the rewards the agent gets may be non-Markovian relative to the environment (the states of S), though they are Markovian relative to the elements in $S \times U$. As such, when making decisions on what action to take in an MDPRM, the agent should consider not just the current environment state $s_t \in S$ but also the current RM state $u_t \in U$.

A policy $\pi(a|\langle s, u\rangle)$ for an MDPRM is a probability distribution over actions $a \in A$ given a pair $\langle s, u\rangle \in S \times U$. We can think of an MDPRM as defining an MDP with state set $S \times U$, as described in the following observation.

Observation 1. *Given an MDPRM $\mathcal{T} = \langle S, A, p, \gamma, \mathcal{P}, L, U, u_0, F, \delta_u, \delta_r\rangle$, let $\mathcal{M}_{\mathcal{T}}$ be the MDP $\langle S', A', r', p', \gamma'\rangle$ defined such that $S' = S \times (U \cup F)$, $A' = A$, $\gamma' = \gamma$,*

$$p'(\langle s', u'\rangle | \langle s, u\rangle, a) = \begin{cases} p(s'|s,a) & \textit{if } u \in F \textit{ and } u' = u \\ p(s'|s,a) & \textit{if } u \in U \textit{ and } u' = \delta_u(u, L(s,a,s')) \\ 0 & \textit{otherwise} \end{cases} ,$$

and $r'(\langle s,u\rangle, a, \langle s',u'\rangle) = \delta_r(u)(s,a,s')$ if $u \notin F$ (zero otherwise). Then any policy for $\mathcal{M}_{\mathcal{T}}$ achieves the same expected reward in $\mathcal{T}$, and vice versa.

We can now see why simple reward machines are a particular case of reward machines. Basically, for any labelling function, we can set the Markovian reward functions in a reward machine to mimic the reward given by a simple reward machine, as shown below.

Proposition 3.4. *Given any labelling function $L : S \times A \times S \to 2^{\mathcal{P}}$, a simple reward machine $\mathcal{R} = \langle U, u_0, F, \delta_u, \delta_r\rangle$ is equivalent to a reward machine $\mathcal{R}_{\mathcal{P}SA} = \langle U, u_0, F, \delta_u, \delta'_r\rangle$ where $\delta'_r(u)(s,a,s') = \delta_r(u, L(s,a,s'))$ for all $u \in U$, $s \in S$, $a \in A$, and $s' \in S$. That is, both $\mathcal{R}$ and $\mathcal{R}_{\mathcal{P}SA}$ will be at the same RM state and output the same reward for every possible sequence of environment state-action pairs.*

Finally, we note that RMs can express any Markovian and some non-Markovian reward functions. In particular, given a set of states S and actions A, the following properties hold:

1. Any Markovian reward function $R : S \times A \times S \to \mathbb{R}$ can be expressed by a reward machine with one state.
2. A non-Markovian reward function $R : (S \times A)^* \to \mathbb{R}$ can be expressed using a reward machine if the reward depends on the state and action history $(S \times A)^*$ only to the extent of distinguishing among those histories that are described by different elements of a finite set of regular expressions over elements in $S \times A \times S$.
3. Non-Markovian reward functions $R : (S \times A)^* \to \mathbb{R}$ that distinguish between histories via properties not expressible as regular expressions over elements in $S \times A \times S$ (such as counting how many times a state has been reached) cannot be expressed using a reward machine.

In other words, reward machines can return different rewards for the same transition (s, a, s') in the environment, for different histories of states and actions seen by the agent, as long as the history can be represented by a regular language. This means that RMs can specify structure in the reward function that includes loops, conditional statements, and sequence interleaving, as well as behavioral constraints, such as safety constraints. To allow for structure beyond what is expressible by regular languages requires that the agent has access to an external memory, which we leave as future work.

Bibliographical Remarks A reader familiar with automata theory will recognize that, except for the terminal states, reward machines are Moore machines (with an output alphabet of reward functions) and simple reward machines are Mealy machines (with an output alphabet of numbers). As such, it seems reasonable to consider a more general form of Mealy reward machine where reward functions (and not just numbers) are output by each RM transition. That was actually our original definition of a reward machine (Toro Icarte et al., 2018c; Camacho et al., 2019). However, following the same argument from Proposition 3.4, we can see that any such Mealy reward machine can be encoded by a (Moore) reward machine using fewer reward functions (one per node instead of per edge). For reward machines that just output numbers, on the other hand, Mealy machines have the advantage of, in some cases, requiring fewer states to represent the same reward signal (also, their first output can depend on the input, unlike for a Moore machine).

Algorithm 1 The cross-product baseline using tabular q-learning.

```
1: Input: $S$, $A$, $\gamma \in (0,1]$, $\alpha \in (0,1]$, $\epsilon \in (0,1]$, $\mathcal{P}$, $L$, $U$, $u_0$, $F$, $\delta_u$, $\delta_r$.
2: Initialize $\tilde{q}(s,u,a)$, for all $s \in S$, $u \in U$, and $a \in A$ arbitrarily
3: for $l \leftarrow 0$ to num_episodes do
4:    Initialize $u \leftarrow u_0$ and $s \leftarrow$ EnvInitialState()
5:    while $s$ is not terminal and $u \notin F$ do
6:       Choose action $a$ from $(s,u)$ using policy derived from $\tilde{q}$ (e.g., $\epsilon$-greedy)
7:       Take action $a$ and observe the next state $s'$
8:       Compute the reward $r \leftarrow \delta_r(u)(s,a,s')$ and next RM state $u' \leftarrow \delta_u(u, L(s,a,s'))$
9:       if $s'$ is terminal or $u' \in F$ then
10:          $\tilde{q}(s,u,a) \xleftarrow{\alpha} r$
11:      else
12:          $\tilde{q}(s,u,a) \xleftarrow{\alpha} r + \gamma \max_{a' \in A} \tilde{q}(s',u',a')$
13:      Update $s \leftarrow s'$ and $u \leftarrow u'$
```

4. Exploiting the RM Structure in Reinforcement Learning

In this section, we describe a collection of RL approaches to learn policies for MDPRMs. We begin by describing a baseline that uses standard RL. We then discuss three approaches that exploit the information in the reward machine to facilitate learning. In all these cases, we include pseudo-code for their tabular implementation, describe how to extend them to work with deep RL, and discuss their convergence guarantees.

4.1 The Cross-Product Baseline

As discussed in Observation 1, MDPRMs are regular MDPs when considering the cross-product between the environment states S and the reward machine states U. As such, any RL algorithm can be used to learn a policy $\pi(a|s,u)$ – including tabular RL methods and deep RL methods. If the RL algorithm is guaranteed to converge to optimal policies, then it will also find optimal policies for the MDPRM.

As a concrete example, Algorithm 1 shows pseudo-code for solving MDPRMs using tabular q-learning. The only difference with standard q-learning is that it also keeps track of the current RM state u and learns q-values over the cross-product $\tilde{q}(s,u,a)$. This allows the agent to consider the current environment state s and RM state u when selecting the next action a. Given the current experience $\langle s,u,a,r,s',u' \rangle$, where $\langle s',u' \rangle$ is the cross-product state reached after executing action a in state $\langle s,u \rangle$ and receiving a reward r, the q-value $\tilde{q}(s,u,a)$ will be updated as follows: $\tilde{q}(s,u,a) \xleftarrow{\alpha} r + \gamma \max_{a'} \tilde{q}(s',u',a')$.

While this method has the advantage of allowing for the use of any RL method to solve MDPRMs, it does not exploit the information exposed by the RM. Below, we discuss different approaches that make use of such information to learn policies for MDPRMs faster.

4.2 Counterfactual Experiences for Reward Machines (CRM)

Our first method to exploit the information from the reward machine is called *counterfactual experience for reward machines (CRM)*. This approach also learns policies over the

Algorithm 2 Tabular q-learning with counterfactual experiences for RMs (CRM).

```
1: Input: $S$, $A$, $\gamma \in (0,1]$, $\alpha \in (0,1]$, $\epsilon \in (0,1]$, $\mathcal{P}$, $L$, $U$, $u_0$, $F$, $\delta_u$, $\delta_r$.
2: Initialize $\tilde{q}(s,u,a)$, for all $s \in S$, $u \in U$, and $a \in A$ arbitrarily
3: for $l \leftarrow 0$ to num_episodes do
4:    Initialize $u \leftarrow u_0$ and $s \leftarrow$ EnvInitialState()
5:    while $s$ is not terminal and $u \notin F$ do
6:       Choose action $a$ from $(s,u)$ using policy derived from $\tilde{q}$ (e.g., $\epsilon$-greedy)
7:       Take action $a$ and observe the next state $s'$
8:       Compute the reward $r \leftarrow \delta_r(u)(s,a,s')$ and next RM state $u' \leftarrow \delta_u(u, L(s,a,s'))$
9:       Set experience $\leftarrow \{\langle s, \bar{u}, a, \delta_r(\bar{u})(s,a,s'), s', \delta_u(\bar{u}, L(s,a,s'))\rangle \mid \forall \bar{u} \in U\}$
10:      for $\langle s, \bar{u}, a, \bar{r}, s', \bar{u}'\rangle \in$ experience do
11:         if $s'$ is terminal or $\bar{u}' \in F$ then
12:            $\tilde{q}(s,\bar{u},a) \xleftarrow{\alpha} \bar{r}$
13:         else
14:            $\tilde{q}(s,\bar{u},a) \xleftarrow{\alpha} \bar{r} + \gamma \max_{a' \in A} \tilde{q}(s', \bar{u}', a')$
15:      Update $s \leftarrow s'$ and $u \leftarrow u'$
```

cross-product $\pi(a|s,u)$, but uses counterfactual reasoning to generate *synthetic* experiences. These experiences can then be used by an off-policy learning method, such as tabular q-learning, DQN, or DDPG, to learn a policy $\pi(a|s,u)$ faster.

Suppose that the agent performed action a when in the cross-product state $\langle s, u\rangle$ and reached state $\langle s', u'\rangle$ and received a reward of r. For every RM state $\bar{u} \in U$, we know that if the agent had been at $\bar{u}$ when a caused the transition from s to s', then the next RM state would have been $\bar{u}' = \delta_u(\bar{u}, L(s,a,s'))$ and the agent would have received a reward of $\bar{r} = \delta_r(\bar{u})(s,a,s')$. This is the key idea behind CRM. What CRM does is that, after every action, instead of feeding only the actual experience $\langle s,u,a,r,s',u'\rangle$ to the RL agent, it feeds one experience per RM state, i.e., the following set of experiences:

$$\{\langle s, \bar{u}, a, \delta_r(\bar{u})(s,a,s'), s', \delta_u(\bar{u}, L(s,a,s'))\rangle \mid \forall \bar{u} \in U\}.$$

Note that incorporating CRM into an off-policy learning method is trivial. For instance, Algorithm 2 shows that CRM can be included in tabular q-learning by adding two lines of code (lines 9 and 10). CRM can also be easily adapted to other off-policy methods by adjusting how the generated experiences are then used for learning. For example, for both DQN and DDPG, the counterfactual experiences would simply be added to the experience replay buffer and then used for learning as is typically done with these algorithms.

In Section 5, we will show empirically that CRM can be very effective at learning policies for MDPRMs. The intuition behind its good performance is that CRM allows the agent to reuse experience to learn the right behaviour at different RM states. Consider, for instance, the RM from Figure 1b, which rewards the agent for delivering a coffee to the office. Suppose that the agent gets to the office before getting the coffee. The cross-product baseline would use that experience to learn that going to the office is not an effective way to get coffee. In contrast, CRM would also use that experience to learn how to get to the office. As such, CRM will already have made progress towards learning a policy that will finish the task as soon as it finds the coffee, since it will already have experience about how to get to the office. Importantly, CRM also converges to optimal policies when combined with q-learning:

Theorem 4.1. *Given an MDPRM $\mathcal{T} = \langle S, A, p, \gamma, \mathcal{P}, L, U, u_0, F, \delta_u, \delta_r \rangle$, CRM with tabular q-learning converges to an optimal policy for $\mathcal{T}$ in the limit (as long as every state-action pair is visited infinitely often).*

Proof. The convergence proof provided by Watkins and Dayan (1992) for tabular q-learning follows directly for CRM when we consider that each experience produced by CRM is still sampled according its transition probability $p(\langle s', u' \rangle | \langle s, u \rangle, a) = p(s'|s, a)$. □

4.2.1 Q-learning for reward machines (QRM)

In the original paper on reward machines (Toro Icarte et al., 2018c), we proposed *q-learning for reward machines (QRM)* as a way to exploit reward machine structure. CRM and QRM are both based on the same fundamental idea: To reuse experience to simultaneously learn optimal behaviours for the different RM states. The key difference is that CRM learns a single q-value function $\tilde{q}(s, u, a)$ that takes into account both the environment and RM state, while QRM learns a separate q-value function $\tilde{q}_u$ for each RM state $u \in U$. Formally, QRM uses any experience $\langle s, a, s' \rangle$ to update each $\tilde{q}_u$ as follows:

$$\tilde{q}_u(s, a) \xleftarrow{\alpha} \delta_r(u)(s, a, s') + \gamma \max_{a' \in A} \tilde{q}_{\delta_u(u, L(s,a,s'))}(s', a')$$

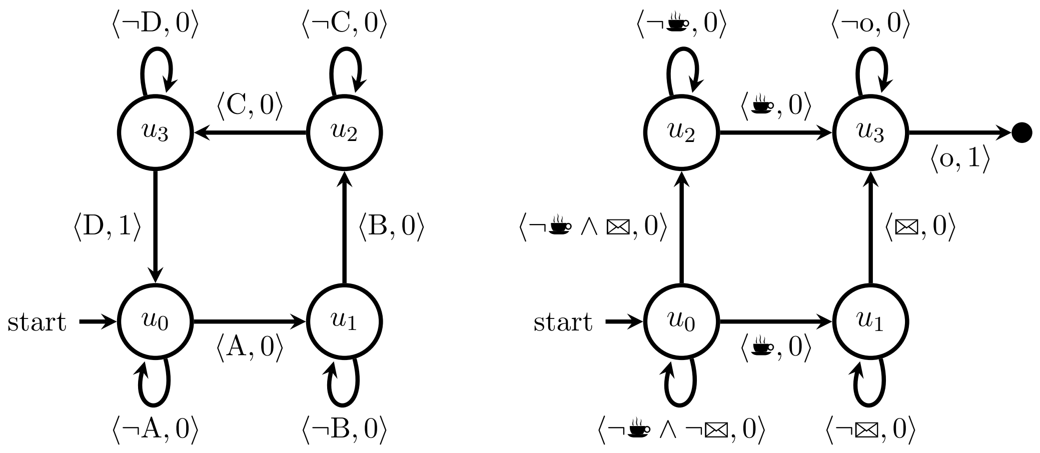
Note that QRM will behave identically to q-learning with CRM in the tabular case. Intuitively, this is because the q-value function $\tilde{q}(s, u, a)$ of CRM can be partitioned by reward machine state, to yield reward machine state specific q-value functions just as in QRM. The corresponding updates can then be seen to be identical.

However, QRM and CRM will be different when using function approximation. Consider, for example, the case of using deep q-networks for function approximation. While the definition of QRM suggests training separate q-networks for each RM state, CRM will learn a single q-network for all the RM states. We note that the need for separate q-networks make the implementation of deep QRM fairly complex. In contrast, combining CRM with DQN or DDPG is trivial: it only requires adding the reward machine state to the experiences when they are being added to the experience replay buffer. We will also see that CRM performs slightly better than QRM in our deep RL experiments.

4.3 Hierarchical RL for Reward Machines (HRM)

Our second approach to exploit the structure of a reward machine is based on hierarchical RL, in particular, the options framework (Sutton et al., 1999). The overall idea is to decompose the problem into subproblems, called *options*, that are potentially simpler to solve. Formally, an option is a triple $\langle \mathcal{I}, \pi, \beta \rangle$, where $\mathcal{I}$ is the initiation set (the subset of the state space in which the option can be started), π is the policy that chooses actions while the option is being followed, and β gives the probability that the option will terminate in each state. Note that an option can be thought of as a macro-action.

In our case, the agent will learn a set of options for the cross-product MDP, that focus on learning how to move from one RM state to another RM state. Then, a higher-level policy will learn how to select among those options in order to collect reward.



(a) Patrol A, B, C, and D (b) Deliver a coffee and the mail

Figure 2: Two more reward machines for the office gridworld

As an example, consider the reward machine from Figure 2b. That machine rewards the agent when it delivers a coffee and the mail to the office. To do so, the agent might first get the coffee, then the mail, and go to the office, or get the mail first, then the coffee, and then go to the office. For this reward machine, our hierarchical RL method will learn one option per edge for a total of five options[1]. That is, the method will learn one policy to get a coffee before getting the mail (moving from u_0 to u_1), one policy to get the mail before getting a coffee (moving from u_0 to u_2), one policy to get a coffee (moving from u_2 to u_3), one policy to get the mail (moving from u_1 to u_3), and one policy to go to the office (moving from u_3 to the terminal state). The role of the high-level policy is to decide which option to execute next among the available options. For instance, if the RM state is in u_0, the high-level policy will decide whether to get a coffee first (moving to u_1) or the mail (moving to u_2). To make this decision, it will consider the current environment state and, thus, it can learn to get the coffee or mail depending on which one is closer to the agent.

More generally, we learn one option for each pair of RM states $\langle u, u_t \rangle$ that are connected in the RM. We will name the options with the pairs of RM states $\langle u, u_t \rangle$ that they correspond to. This means that the set of options is $\mathcal{A} = \{\langle u, \delta_u(u, \sigma)\rangle \mid u \in U, \sigma \in 2^{\mathcal{P}}\}$. Notice that this includes "self-loop" edges where $u_t = u$. The option $\langle u, u_t \rangle$ will have its initiation set defined to contain all the states in the cross-product MDP where the RM state is u: $\mathcal{I}_{\langle u, u_t \rangle} = \{\langle s, u \rangle : s \in S\}$. The termination condition is then defined as follows:

$$\beta_{\langle u, u_t \rangle}(s', u') = \begin{cases} 1 & \text{if } u' \neq u \text{ or } s' \text{ is terminal} \\ 0 & \text{otherwise} \end{cases}$$

That is, the option $\langle u, u_t \rangle$ terminates (deterministically) when a new RM state is reached or a terminal environment state is reached. Since $\langle u, u_t \rangle$ can only be executed when the RM state is u, its policy can be described in terms of the environment state s only. As such, we refer to the option policy as $\pi_{u,u_t}(a|s)$.

1. We will ignore edges that are "self-loop" edges for now as a way to simplify the explanation.

Algorithm 3 Tabular hierarchical RL for reward machines (HRM).

```
 1: Input: $S$, $A$, $\gamma \in (0,1]$, $\alpha \in (0,1]$, $\epsilon \in (0,1]$, $\mathcal{P}$, $L$, $U$, $u_0$, $F$, $\delta_u$, $\delta_r$.
 2: $\mathcal{A}(u) \leftarrow \{u_t \mid u_t = \delta_u(u,\sigma)$ for some $u_t \in U \cup F, \sigma \in 2^{\mathcal{P}}\}$ for all $u \in U$
 3: Initialize high-level $\tilde{q}(s,u,u_t)$, for all $s \in S$, $u \in U$, and $u_t \in \mathcal{A}(u)$ arbitrarily
 4: Initialize option $\tilde{q}_{u,u_t}(s,a)$, for all $s \in S$, $u \in U$, $u_t \in \mathcal{A}(u)$, and $a \in A$
 5: for $l \leftarrow 0$ to num_episodes do
 6:   Initialize $u \leftarrow u_0$, $s \leftarrow$ EnvInitialState(), and $u_t \leftarrow \emptyset$
 7:   while $s$ is not terminal and $u \notin F$ do
 8:     if $u_t = \emptyset$ then
 9:       Choose option $u_t \in \mathcal{A}(u)$ using policy derived from $\tilde{q}$ (e.g., $\epsilon$-greedy)
10:       Set $r_t \leftarrow 0$ and $t \leftarrow 0$
11:     Choose action $a$ from $s$ using policy derived from $\tilde{q}_{u,u_t}$ (e.g., $\epsilon$-greedy)
12:     Take action $a$ and observe the next state $s'$
13:     Compute the reward $r \leftarrow \delta_r(u)(s,a,s')$ and next RM state $u' \leftarrow \delta_u(u, L(s,a,s'))$
14:     for $\bar{u} \in U, \bar{u}_t \in \mathcal{A}(\bar{u})$ do
15:       if $\delta_u(\bar{u}, L(s,a,s')) \neq u$ or $s'$ is terminal then
16:         $\tilde{q}_{\bar{u},\bar{u}_t}(s,a) \overset{\alpha}{\leftarrow} r_{\bar{u},\bar{u}_t}(s,a,s')$
17:       else
18:         $\tilde{q}_{\bar{u},\bar{u}_t}(s,a) \overset{\alpha}{\leftarrow} r_{\bar{u},\bar{u}_t}(s,a,s') + \gamma \max_{a' \in A} \tilde{q}_{\bar{u},\bar{u}_t}(s',a')$
19:     if $s'$ is terminal or $\delta_u(u, L(s,a,s')) \neq u$ then
20:       if $s'$ is terminal or $u' \in F$ then
21:         $\tilde{q}(s,u,u_t) \overset{\alpha}{\leftarrow} r_t + \gamma^t r$
22:       else
23:         $\tilde{q}(s,u,u_t) \overset{\alpha}{\leftarrow} r_t + \gamma^t r + \gamma^{t+1} \max_{u'_t \in \mathcal{A}(u')} \tilde{q}(s',u',u'_t)$
24:       Set $u_t \leftarrow \emptyset$
25:     Update $s \leftarrow s'$, $u \leftarrow u'$, $r_t \leftarrow r_t + \gamma^t r$, and $t \leftarrow t+1$
```

Since the objective of option π_{u,u_t} is to induce the reward machine to transition to u_t as soon as possible, we train the option policy $\pi_{u,u_t}(a|s)$ using the following reward function:

$$r_{u,u_t}(s,a,s') = \begin{cases} \delta_r(u)(s,a,s') & \text{if } u = \delta_u(u, L(s,a,s')) \\ r^+ & \text{if } u_t = \delta_u(u, L(s,a,s')) \text{ and } u_t \neq u \\ r^- & \text{otherwise} \end{cases},$$

where r^+ and r^- are hyper-parameters. This reward function states that the policy π_{u,u_t} gets a reward of r^+ (a bonus) when the transition (s,a,s') causes the RM state to move from u to u_t (unless $u = u_t$), a reward of r^- (a penalization) when it causes the RM state to move from u to some other state $\bar{u} \notin \{u, u_t\}$, and the usual reward $\delta_r(u)(s,a,s')$ if (s,a,s') does not change the RM state. Crucially, the policies for all the options will be learned simultaneously, using off-policy RL and counterfactual experience generation.

The high-level policy decides which option to execute next from the set of available options. The policy $\pi(u_t|s,u)$ being learned will determine the probability of executing each option $\langle u, u_t \rangle \in \mathcal{A}$ given the current environment state s and RM state u. We note that this high-level policy can only choose among the options that start at the current RM state u. To train this policy, we use the reward coming from the reward machine.

Algorithm 3 shows pseudocode for this approach, which we call *hierarchical reinforcement learning for reward machines (HRM)*, when using tabular q-learning. Tabular q-learning could be replaced by any other off-policy method such as DQN or DDPG. The algorithm begins by initializing one q-value estimate $\tilde{q}(s, u, u_t)$ for the high-level policy and one q-value estimate $\tilde{q}_{u,u_t}(s, a)$ for each option $\langle u, u_t \rangle \in \mathcal{A}$. At every step, the agent first checks if a new option has to be selected and do so using $\tilde{q}(s, u, u_t)$ (lines 8-10). This option takes the control of the agent until it reaches a terminal transition. The current option selects the next action $a \in A$, executes it, and reaches the next state s' (lines 11-12). The experience (s, a, s') is used to compute the next RM state $u' = \delta_u(u, L(s, a, s'))$ and reward $r = \delta_r(u)(s, a, s')$ (line 13), and also to update the option policies by giving a reward of $r_{\bar{u},\bar{u}_t}(s, a, s')$ to each option $\langle \bar{u}, \bar{u}_t \rangle \in \mathcal{A}$ (lines 14-18). Finally, the high-level policy is updated when the option ends (lines 19-24) and the loop starts over from s' and u'.

HRM can be very effective at quickly learning good policies for MDPRMs. Its strength comes from its ability to learn policies for all of the options simultaneously through off-policy learning. However, it might converge to sub-optimal solutions, even in the tabular case. This is because the options-based approach is myopic: the learned option policies will always try to transition as quickly as possible without considering how that will affect performance after the transition occurs. An example of this behaviour is shown in Figure 1. The task consists of delivering a coffee to the office. As such, the optimal high-level policy will correctly learn to go for the coffee at state u_0 and then go to the office at state u_1. However, the optimal option policy for getting the coffee will move to the closest coffee station (following the sub-optimal red path in Figure 1a) because (i) that option gets a large reward when it reaches the coffee, and (ii) optimal policies will always prefer to collect such a reward *as soon as* possible. As a result, HRM will converge to a sub-optimal policy.

We note that HRM can use prior knowledge about the environment to prune useless options. For example, in our experiments we do not learn options for the self-loops, since no optimal high-level policy would need to self-loop in our domains. We also do not learn options that lead to "bad" terminal states, such as breaking decorations in Figure 1b.

4.4 Automated Reward Shaping (RS)

Our last method for exploiting reward machines builds on the idea of *potential-based reward shaping* (Ng et al., 1999). The intuition behind reward shaping is that some reward functions are easier to learn policies for than others, even if those functions have the same optimal policy. To that end, Ng et al. (1999) formally showed that given any MDP $\mathcal{M} = \langle S, A, r, p, \gamma \rangle$ and function $\Phi : S \to \mathbb{R}$, changing the reward function of $\mathcal{M}$ to

$$r'(s, a, s') = r(s, a, s') + \gamma \Phi(s') - \Phi(s) \tag{1}$$

will not change the set of optimal policies. Thus, if we find a function Φ – referred to as a *potential function* – that allows us to learn optimal policies more quickly, we are guaranteed that the found policies are still optimal with respect to the original reward function.

In this section, we consider the use of *value iteration* over the RM states as a way to compute a potential function. Intuitively, the idea is to approximate the cumulative

Algorithm 4 Value Iteration for Automatic Reward Shaping

```
Input: $U$, $F$, $\mathcal{P}$, $\delta_u$, $\delta_r$, $\gamma$
for $u \in U \cup F$ do
  $v(u) \leftarrow 0$ {initializing v-values}
$e \leftarrow 1$
while $e > 0$ do
  $e \leftarrow 0$
  for $u \in U$ do
    $v' \leftarrow \max\{\delta_r(u,\sigma) + \gamma v(\delta_u(u,\sigma)) \mid \forall \sigma \in 2^{\mathcal{P}}\}$
    $e = \max\{e, |v(u) - v'|\}$
    $v(u) \leftarrow v'$
return $v$
```

discounted reward of being in any RM state by treating the RM itself as an MDP. As a result, a potential will be assigned to each RM state over which equation (1) will be used to define a shaped reward function that will encourage the agent to make progress towards solving the task. This method works only for simple reward machines since it does not use information from the environment states S and actions A.

Formally, given a simple RM $\langle U, u_0, F, \delta_u, \delta_r \rangle$, we construct an MDP $\mathcal{M} = \langle S, A, r, p, \gamma \rangle$, where $S = U \cup F$, $A = 2^{\mathcal{P}}$, $r(u, \sigma, u') = \delta_r(u, \sigma)$ if $u \in U$ (zero otherwise), $\gamma < 1$, and

$$p(u'|u,\sigma) = \begin{cases} 1 & \text{if } u \in F \text{ and } u' = u \\ 1 & \text{if } u \in U \text{ and } u' = \delta_u(u,\sigma) \\ 0 & \text{otherwise} \end{cases} .$$

Intuitively, this is an MDP where every transition in the RM corresponds to a deterministic action. For every state u in $\mathcal{M}$, we compute $v^*(u) = \max_\sigma q^*(u, \sigma)$ using value iteration. The overall process of computing v^* given U, F, $\mathcal{P}$, δ_u, δ_r, and γ, is shown in Algorithm 4.

Once we have computed v^*, we then define the potential function as $\Phi(s, u) = -v^*(u)$ for every environment state s and RM state u. As we will see below, the use of negation encourages the agent to transition towards RM states that correspond to task completion.

To make this approach more clear, consider the example task of *delivering coffee to the office while avoiding decorations* from Figure 1b. What makes this task difficult for an RL agent is the sparsity of the reward. The agent only gets a +1 reward by completing the whole task. The goal of including reward shaping is to provide some intermediate rewards as the agent gets closer to complete the task. In this case, passing this simple reward machine through Algorithm 4 with $\gamma = 0.9$ results in the potential-based function shown in Figure 3. In the figure, nodes represent RM states, and each state has been labelled with the computed potential in red (the potential of terminal states is always zero). Each transition has also been labelled by a pair $\langle c, r + rs \rangle$, where c is a logical condition to transition between the states, r is the reward that the agent receives for the transition according to δ_r, and rs is the extra reward given by equation (1). Note that, with reward shaping, the agent is given a reward of 0.09 for self-looping before getting a coffee and a reward of 0.1 for self-looping after getting the coffee. This gives an incentive for collecting coffee and, as such, making

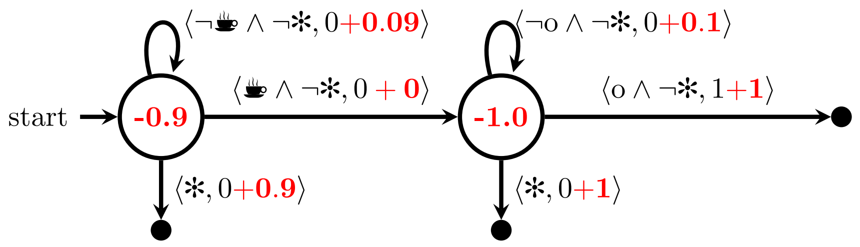

start
-0.9
-1.0
$\langle \neg ☕ \wedge \neg ✻, 0+0.09 \rangle$
$\langle ☕ \wedge \neg ✻, 0+0 \rangle$
$\langle ✻, 0+0.9 \rangle$
$\langle \neg o \wedge \neg ✻, 0+0.1 \rangle$
$\langle o \wedge \neg ✻, 1+1 \rangle$
$\langle ✻, 0+1 \rangle$

Figure 3: Reward shaping example with $\gamma = 0.9$.

progress on the overall task. In addition, we know that the optimal policy is preserved since we are using equation (1) to shape the rewards.

5. Experimental Evaluation

In this section, we provide an empirical evaluation of our methods in domains with a variety of characteristics: discrete states, continuous states, and continuous action spaces. Some domains include multitask learning and single task learning. As a brief summary, our results show the following:

1. CRM and HRM outperform the cross-product baselines in all our experiments.

2. CRM converges to the best policies in all but one experiment.

3. HRM tends to initially learn faster than CRM but converges to suboptimal policies.

4. The gap between CRM/HRM and the cross-product baseline increases when learning in a multitask setting.

5. Reward shaping helps CRM in discrete domains but it does not in continuous domains.

5.1 Results on Discrete Domains

We evaluated our methods on two gridworlds. Since these are tabular domains, we use tabular q-learning as the core off-policy learning method for all the approaches. Specifically, we tested q-learning alone over the cross-product (QL), q-learning with reward shaping (QL+RS), q-learning with counterfactual experiences (CRM), q-learning with CRM and reward shaping (CRM+RS), and our hierarchical RL method (HRM). We do not report results for QRM since it is equivalent to CRM in tabular domains. We use $\epsilon = 0.1$ for exploration, $\gamma = 0.9$, $\alpha = 0.5$, and an optimistic initialization of the q-values.

The first domain is the office world described in Section 3 and Figure 1a. This is a multitask domain, consisting of the four tasks described in Table 1. We begin by evaluating how long it takes the agents to learn a policy that can solve those four tasks. The agent will iterate through the tasks, changing from one to the next at the completion of each episode. Note that CRM and HRM are good fits for this problem setup because they can use experience from solving one task to update the policies for solving the other tasks.

We ran 30 independent trials and report the average reward per step across the four tasks in Figure 4-left. We normalized the average reward per step to be 1 for an optimal policy

Table 1: Tasks for the office world.

#	Description
1	deliver coffee to the office without breaking any decoration
2	deliver mail to the office without breaking any decoration
3	patrol locations A, B, C, and D, without breaking any decoration
4	deliver a coffee and the mail to the office without breaking any decoration

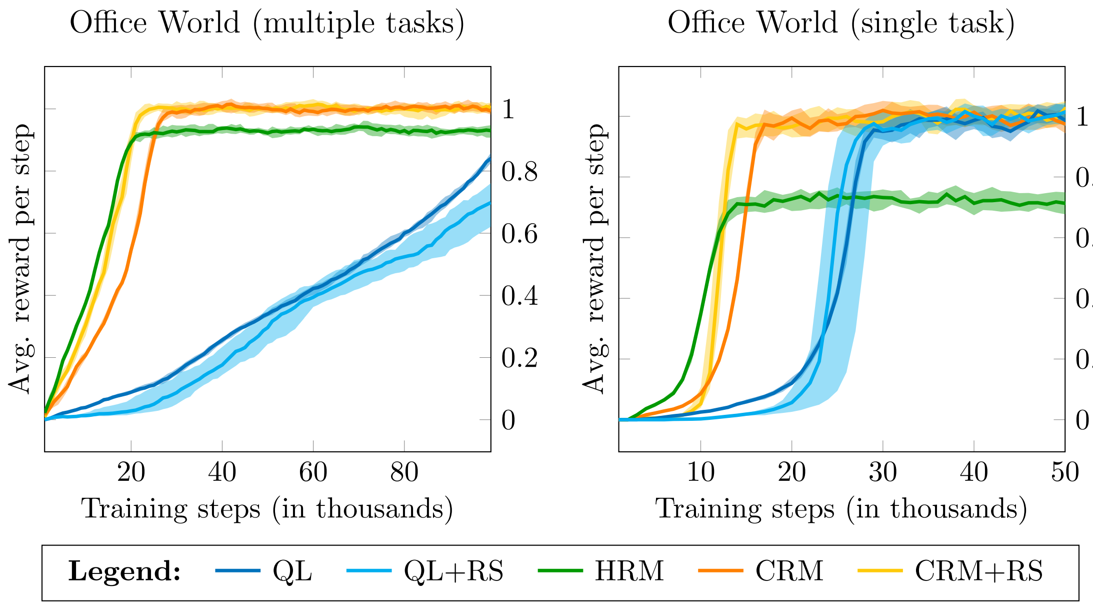


Figure 4: Results on the office gridworld.

(which we pre-computed using value iteration) and show the median performance over those 30 runs, as well as the 25th and 75th percentiles in the shadowed area. The results show that CRM and CRM+RS quickly learn to optimally solve all the tasks – largely outperforming the cross-product baseline (QL). HRM also outperforms QL and initially learns faster than CRM, but converges to suboptimal policies. Finally, we note that adding reward shaping improved the performance of CRM but decreased the performance of QL.

We also run a single-task experiment in the office world. We took the hardest task available (task 4), and ran 30 new independent runs where the agent had to solve only that task. The results are shown in Figure 4-right. We note that CRM still outperforms the other methods, though the gap between CRM and QL decreased. Again, HRM learns faster than CRM, but is overtaken since it converges to a suboptimal policy.

Our second tabular domain is the Minecraft-like gridworld introduced by Andreas et al. (2017). In this world, the grid contains raw materials that the agent can extract and use to make new objects. Andreas et al. defined 10 tasks to solve in this world (shown in Table 2), that consist of making an object by following a sequence of sub-goals (called a *sketch*). For instance, the task *make a bridge* consists of *get iron*, *get wood*, and *use factory*. Note that Andreas et al.'s approach forces an unnecessary ordering when describing the

Table 2: Tasks for the Minecraft domain. Each task is described as a sequence of events.

#	Task name	Description
1	make plank	get wood, use toolshed
2	make stick	get wood, use workbench
3	make cloth	get grass, use factory
4	make rope	get grass, use toolshed
5	make bridge	get iron, get wood, use factory (the iron and wood can be gotten in any order)
6	make bed	get wood, use toolshed, get grass, use workbench (the grass can be gotten at any time before using the workbench)
7	make axe	get wood, use workbench, get iron, use toolshed (the iron can be gotten at any time before using the toolshed)
8	make shears	get wood, use workbench, get iron, use workbench (the iron can be gotten at any time before using the workbench)
9	get gold	get iron, get wood, use factory, use bridge (the iron and wood can be gotten in any order)
10	get gem	get wood, use workbench, get iron, use toolshed, use axe (the iron can be gotten at any time before using the toolshed)

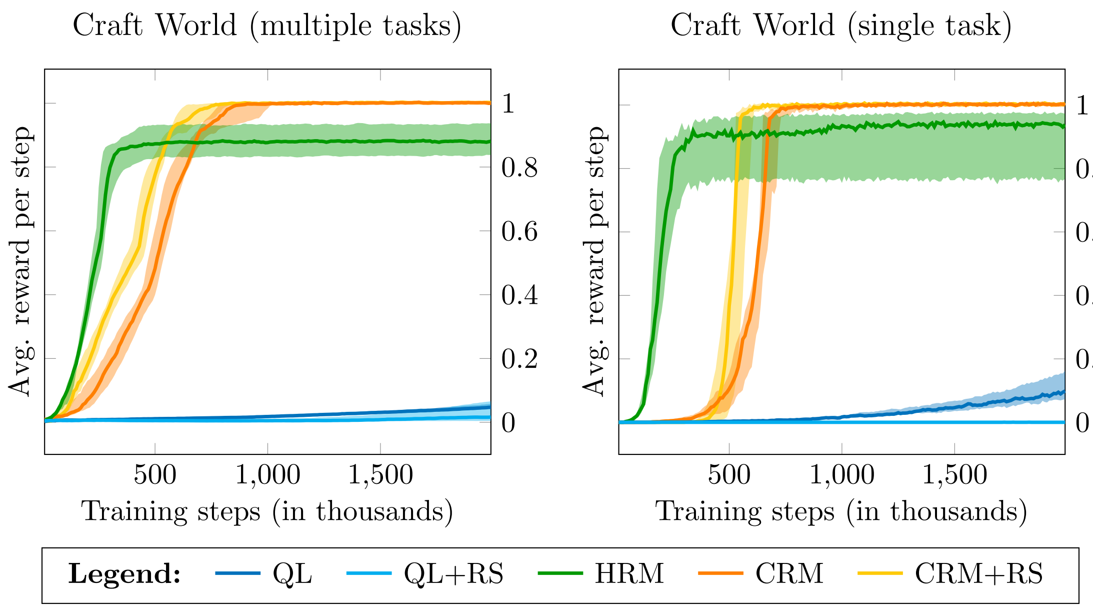


Figure 5: Results on the Minecraft-like gridworld.

extraction of raw materials (in the example, the agent can actually collect *wood* and *iron* in any order before using the *factory*). As reward machines are more expressive than sketches, we encoded the same 10 tasks, removing any unnecessary order constraints.

Figure 5 shows performance over 10 randomly generated maps, with 3 trials per map. We report results for multitask and single task learning. From these results, we can draw

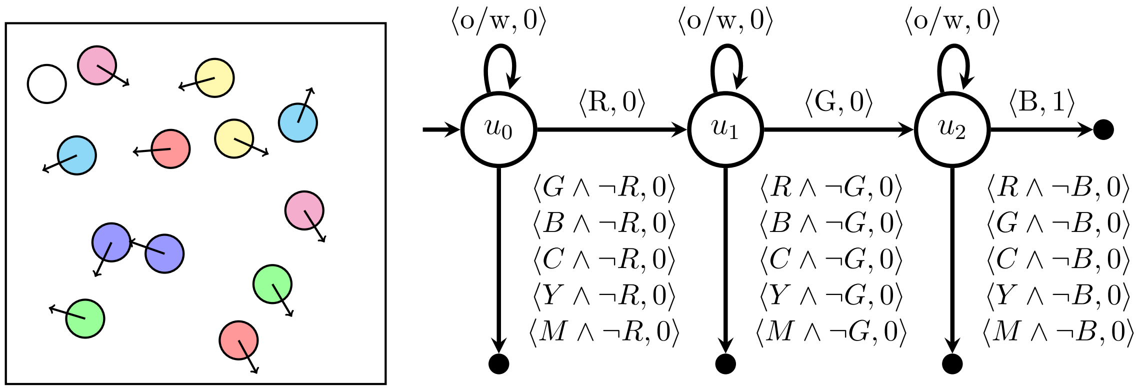

⟨o/w, 0⟩
⟨o/w, 0⟩
⟨o/w, 0⟩
⟨R, 0⟩
⟨G, 0⟩
⟨B, 1⟩
u_0
u_1
u_2
$\langle G \wedge \neg R, 0\rangle$
$\langle B \wedge \neg R, 0\rangle$
$\langle C \wedge \neg R, 0\rangle$
$\langle Y \wedge \neg R, 0\rangle$
$\langle M \wedge \neg R, 0\rangle$
$\langle R \wedge \neg G, 0\rangle$
$\langle B \wedge \neg G, 0\rangle$
$\langle C \wedge \neg G, 0\rangle$
$\langle Y \wedge \neg G, 0\rangle$
$\langle M \wedge \neg G, 0\rangle$
$\langle R \wedge \neg B, 0\rangle$
$\langle G \wedge \neg B, 0\rangle$
$\langle C \wedge \neg B, 0\rangle$
$\langle Y \wedge \neg B, 0\rangle$
$\langle M \wedge \neg B, 0\rangle$

Figure 6: Water world domain and an examplary task. The events R, G, B, C, Y, and M represent touching a red ball, green ball, blue ball, cyan ball, yellow ball, and magenta ball, respectively. The label o/w stands for *otherwise*. The RM encodes task 10 from Table 3.

similar conclusions as for the office world. However, notice that the gap between methods that exploit the structure of the reward machine (CRM and HRM) and methods that do not (QL) is much larger in the Minecraft domain. The reason is that the Minecraft domain is more complex and the reward more sparse – making the use of q-learning alone hopeless.

Overall, the results on these two domains show that exploiting the RM structure can greatly increase the performance in tabular domains for single task and multitask learning. From our methods, CRM seems to be the best compromise between performance and convergence guarantees. HRM can find good policies quickly, but it often converges to suboptimal solutions. Finally, note that reward shaping helped CRM but did not help QL.

5.2 Results on Continuous State Domains

We tested our approaches in a continuous state space problem called the *water world* (Karpathy, 2015). This environment consists of a two dimensional box with balls of different colors in it (see Figure 6 for an example). Each ball moves in one direction at a constant speed and bounces when it collides with the box's edges. The agent, represented by a white ball, can increase its velocity in any of the four cardinal directions. As the ball positions and velocities are real numbers, this domain cannot be tackled using tabular RL.

We defined a set of 10 tasks for the water world over the events of touching a ball of a certain color. For instance, one simple task consists of touching a *cyan ball* after a *blue ball*. Other more complicated tasks include touching a sequence of balls, such as *red*, *green*, and *blue*, in a strict order, such that the agent fails if it touches a ball of a different color than the next in the sequence. The complete list of tasks can be found in Table 3.

In these experiments, we replaced q-learning with Double DQN. Concretely, we evaluated double DQN over the cross-product (DDQN), DDQN with reward shaping (DDQN+RS), DDQN with counterfactual experiences (CRM), DDQN with CRM and reward shaping (CRM+RS), and our hierarchical RL method (HRM). For all approaches other than HRM, we used a feed-forward network with 3 hidden layers and 1024 relu units per layer. We trained the networks using the Adam optimizer (Kingma & Ba, 2015) with a learning rate

Table 3: Tasks for the water world. By "(color1 then color2)" we mean the task of touching a ball of color1 and then touching a ball of color2. A task like "(color1 then color2 then color3)" is similar but with three types of balls to touch. By "(subtask1) and (subtask2)" we mean that the agent must complete the tasks described by subtask1 and subtask2, but in any order. By "(color1 strict-then color2)" we mean the task which is like "(color1 then color2)" but where the agent is not allowed to touch balls of other colors during execution.

#	Description
1	(red then green)
2	(blue then cyan)
3	(magenta then yellow)
4	(red then green) and (blue then cyan)
5	(blue then cyan) and (magenta then yellow)
6	(red then green) and (magenta then yellow)
7	(red then green) and (blue then cyan) and (magenta then yellow)
8	(red then green then blue) and (cyan then magenta then yellow)
9	(cyan strict-then magenta strict-then yellow)
10	(red strict-then green strict-then blue)

of $1e-5$. On every step, we updated the q-functions using $32n$ sampled experiences from a replay buffer of size $50000n$, where $n = 1$ for DDQN and $n = |U|$ for CRM. The target networks were updated every 100 training steps and the discount factor γ was 0.9. For HRM, we use the same feed-forward network and hyperparameters to train the option's policies (although $n = |\mathcal{A}|$ in this case). The high-level policy was learned using DDQN but, since the high-level decision problem is simpler, we used a smaller network (2 layers with 256 relu units) and a larger learning rate ($1e-3$). Our DDQN implementation was based on the code from OpenAI Baselines (Hesse et al., 2017).

Figure 7 shows the results on 10 randomly generated water world maps. We normalized the average reward per step using the run that got the highest average reward across all the approaches. In the multitask experiments, CRM performs the best and adding reward shaping did not improve the performance of CRM or DDQN. As in the discrete domains, HRM initially learns faster than CRM but converged to suboptimal policies. In the single task experiment, we evaluated the performance of all the approaches when trying to solve task 10 only (Table 3). In this case, CRM also converged to better policies.

Finally, we compare the performance of CRM and QRM. As discussed in Section 4.2, CRM and QRM are not equivalent when using function approximation. The most notable difference is that CRM uses a large network to learn a single policy $\pi(a|s,u)$ for all RM states whereas QRM uses a set of small networks, one to learn a policy $\pi_u(a|s)$ for each RM state $u \in U$. The results in Figure 8 show that CRM can perform better than QRM on the water world. However, to do so CRM requires using a larger network. In this experiment, QRM is learning networks of 6 layers with 64 relu units (6L/64N) – which are the same size as the networks used in Toro Icarte et al. (2018c). To get to the same performance as QRM, CRM has to use a network of 3 layers with 1024 relu units per layer. We leave it to future work to further investigate the trade-offs of using multiple small networks to solve a task (as in QRM) versus one network per task (as in CRM). That said, CRM has the

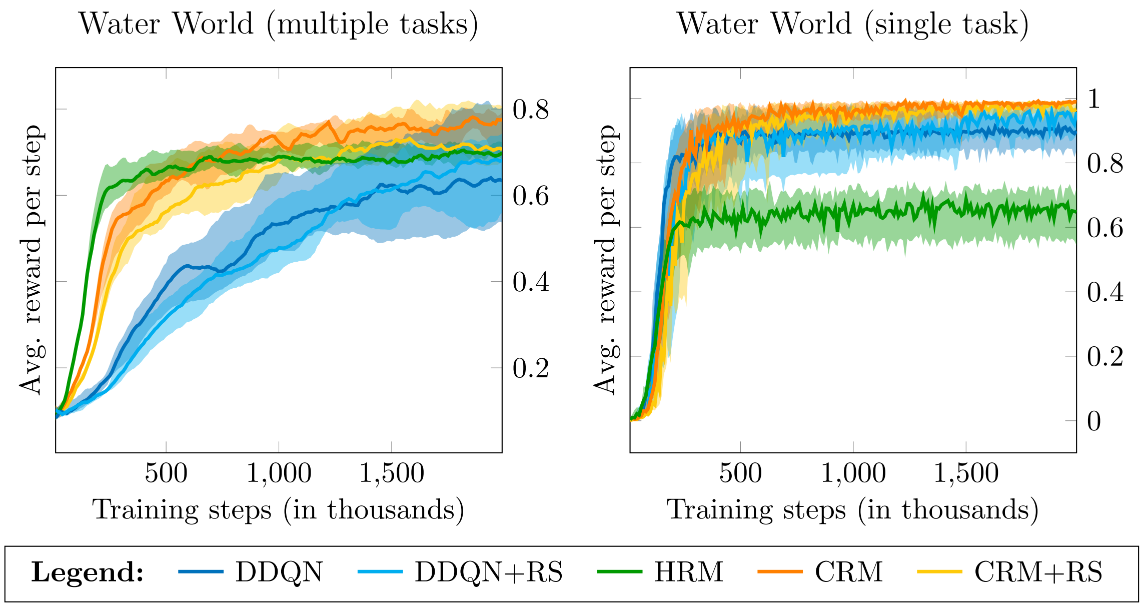


Figure 7: Results in the water world domain.

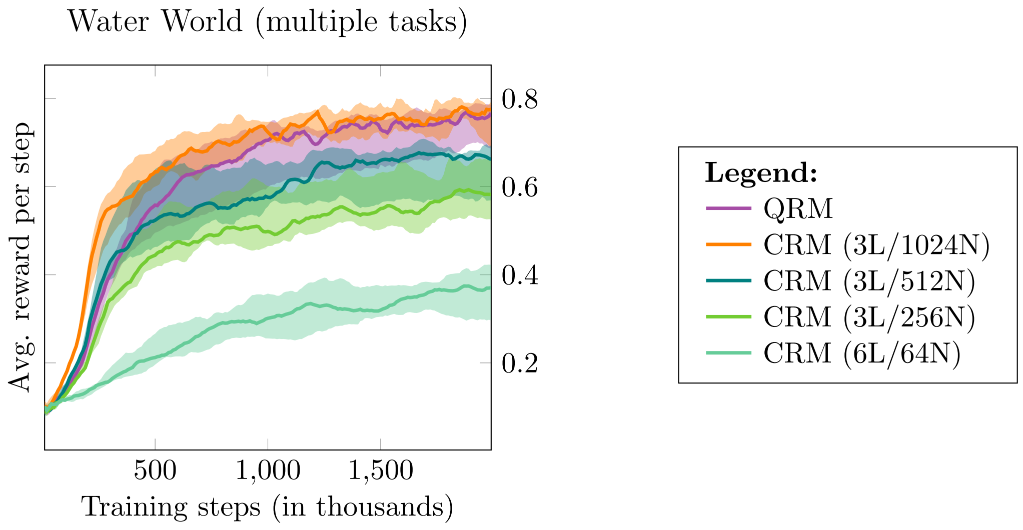


Figure 8: Results in the water world domain.

added advantage of being trivial to implement. In fact, we do not have results for QRM in the next section because it is unclear how to integrate QRM with DDPG.

5.3 Results on Continuous Control Tasks

Our final set of experiments consider the case where the action space is continuous. We ran experiments on the Half-Cheetah environment (Brockman et al., 2016), shown in Figure 9. In this environment, the agent is a cheetah-like robot. This agent has 6 joints that must

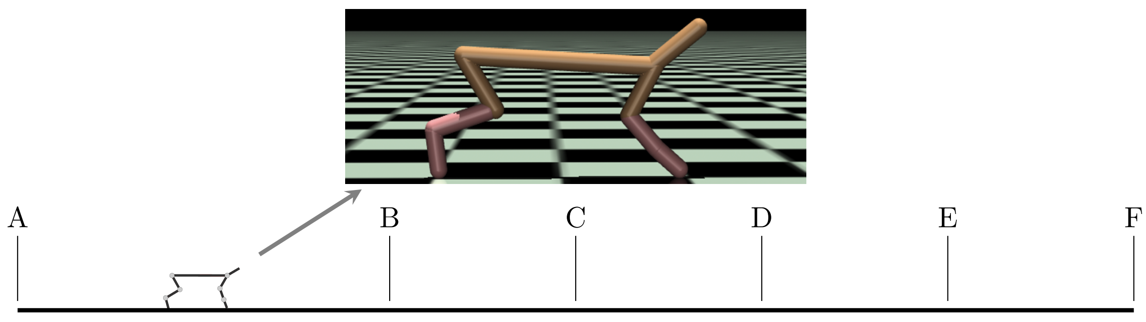


Figure 9: Half-cheetah domain. The first task consists of going from A to B and back as many times as possible. The second task consists of reaching F as soon as possible.

$$\begin{tikzcd}
\text{start} \arrow[r] & u_0 \arrow[loop above, "{\langle \neg \mathrm{B}, \texttt{CP} \rangle}"] \arrow[r, "{\langle \mathrm{B}, \texttt{CP} \rangle}"'] & u_1 \arrow[loop above, "{\langle \neg \mathrm{C}, \texttt{CP} \rangle}"] \arrow[r, "{\langle \mathrm{C}, \texttt{CP} \rangle}"'] & u_2 \arrow[loop above, "{\langle \neg \mathrm{D}, \texttt{CP} \rangle}"] \arrow[r, "{\langle \mathrm{D}, \texttt{CP} \rangle}"'] & u_3 \arrow[loop above, "{\langle \neg \mathrm{E}, \texttt{CP} \rangle}"] \arrow[r, "{\langle \mathrm{E}, \texttt{CP} \rangle}"'] & u_4 \arrow[loop above, "{\langle \neg \mathrm{F}, \texttt{CP} \rangle}"] \arrow[r, "{\langle \mathrm{F}, 1000 \rangle}"'] & \bullet
\end{tikzcd}$$

Figure 10: Reward machine for the second task on the half-cheetah domain. `CP` represents the control penalty usually used in this domain.

learn to control in order to stand up, move forward, or backwards. At each time step, the agent chooses how much force to apply to each joint, making the action space infinite. The state space is also continuous, including the location and velocities of each joint.

We evaluated the performance of our approaches in two tasks (independently). All the approaches use DDPG as the underlying off-policy learning approach. In the case of HRL, the option policies are learned using DDPG but the high-level policy uses DQN. All the approaches use a feed-forward network with 2 layers with 256 Relu units per layer. The batch size was $100n$ (where $n = 1$ in DDPG, $n = |U|$ in CRM, and $n = |\mathcal{A}|$ in HRM) and the rest of the hyperparameters were set to their default values (Hesse et al., 2017).

Figure 11-left shows average results over 30 runs for the first task. This task consists of moving back and forth from point A to point B (shown in Figure 9) as many times as possible given a time limit of 1000 steps. The corresponding reward machine gives a reward of 1000 every time the agent completes a lap. The agent also receives a small *control penalization* on every step – which is a standard penalization that discourages the agent from falling. The results show that CRM largely outperforms the other approaches, completing 9 laps on average by the end of learning. HRM also performs well, completing around 6 laps per episode by the end of learning. As before, the problem with HRM is that it optimizes for reaching the next subgoal (A or B) without considering what has to be done next. While CRM learns to slow down before reaching A or B so it can quickly jump back afterwards, HRM learns to run full speed until reaching A or B and, as a result, keeps advancing a few steps before being able to slow down and start moving in the opposite direction.

Note that the strength of CRM relies on being able to effectively share experience among different RM states. Being able to achieve one task while trying to achieve a different one

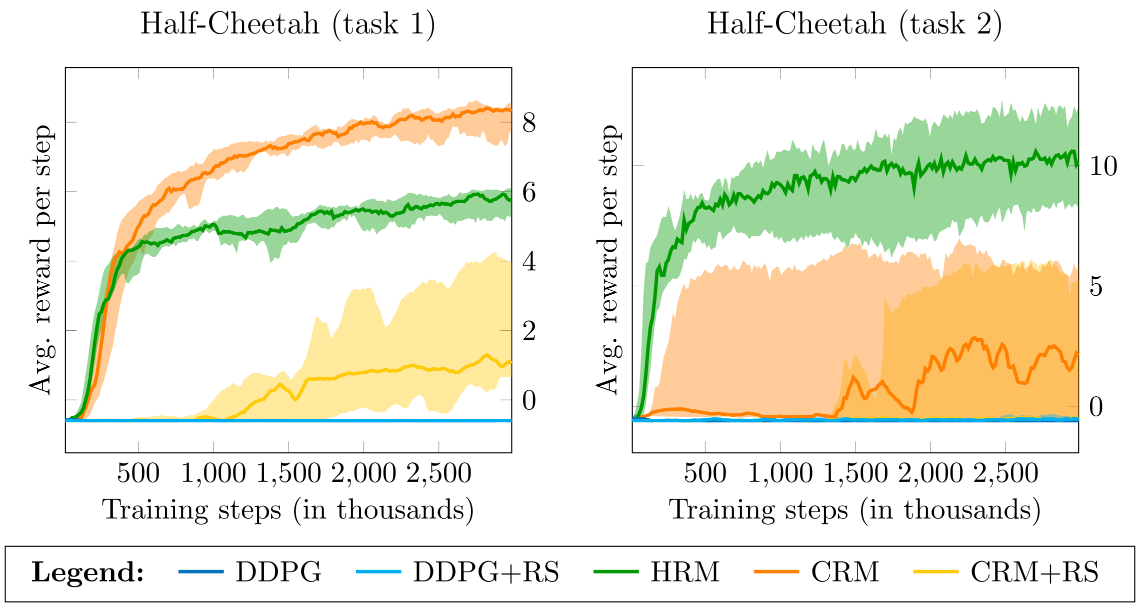


Figure 11: Results in the half-cheetah domain.

(for example, reaching point A while trying to get to point B) allows CRM to learn efficiently. If that sort of behaviour cannot happen, then CRM would not help much since the shared experience would not provide new insights into how to solve the problem. In contrast, HRM performs the best in tasks where it's problem decomposition preserves optimal policies. To provide a complete view of the strengths and weaknesses of these approaches, we designed a second task for the half-cheetah that is ideal for HRM and difficult for CRM.

This second task consists of reaching F (shown in Figure 9) and it is represented by the RM from Figure 10. This RM is a chain of 6 states that advances as the agent reaches B, C, D, E, and F. The agent gets a reward of 1000 when it reaches F and the control penalization (`CP`) otherwise. This is a complex task for DDPG because the reward is too sparse. Adding CRM could make the reward less sparse if the probability that some RM state gets a positive reward increased by adding counterfactual experiences. This was the case in the first task, but it is not in this second task. On the other hand, HRM is a good fit for this problem for two reasons. First, the reward is less sparse for HRM since the agent gets some reward signal every time B, C, D, E, or F are reached. And second, the policy that results from composing optimal option policies is globally optimal.

The average results over 30 runs for the second task are shown in Figure 11-right. As expected, HRM is the approach that performs the best, being able of reaching F in less than 100 steps. Interestingly, adding CRM to DDPG did help a bit, allowing the agent to reach F in 200 steps in some runs, but its performance was unreliable.

5.4 Code

Our code is available at `github.com/RodrigoToroIcarte/reward_machines`, including our environments, raw results, and implementations of the cross-product baseline, automated reward shaping, CRM, and HRM using tabular q-learning, DDQN, and DDPG. For the experiments with QRM, we use the following implementation: `bitbucket.org/RToroIcarte/qrm`. Our methods are fully integrated with the OpenAI gym API (Brockman et al., 2016).

6. Related Work

In this section, we discuss existing works about reward machines and how this paper fits within that body of literature. We then discuss how reward machines relate more generally with approaches for reward specifications and knowledge exploitation in RL.

6.1 Reward Machine Research

We originally proposed reward machine in an ICML publication (Toro Icarte et al., 2018c). Back then, a few works were proposing to use Linear Temporal Logic (LTL) or related languages to reward agents in MDPs (e.g., Bacchus et al., 1996; Lacerda et al., 2014, 2015; Camacho et al., 2017; Brafman et al., 2018) and RL (e.g., Li et al., 2017; Aksaray et al., 2016; Littman et al., 2017; Hasanbeig et al., 2018; Li et al., 2018). The common principle was to translate the LTL specification into a finite state machine, reward the agent when the machine hits an accepting state, and learn policies using the cross-product baseline. In this context, we proposed a novel approach, called LPOPL, that exploited the structure of LTL to learn policies faster than the cross-product baseline and hierarchical RL (Toro Icarte et al., 2018b). LPOPL is QRM's predecessor as it relies on the same learning principle: It decomposes the LTL tasks into many subtasks and learns policies for them in parallel via off-policy learning.

Reward machines generalize LTL specifications as they can compose sets of Markovian reward functions in ways that cannot be represented in LTL. QRM also generalizes LPOPL and outperforms the cross-product baseline and Hierarchical RL. The main contributions of our ICML paper were to introduce reward machines and QRM (Toro Icarte et al., 2018c). Our automated reward shaping approach was introduced later (Camacho et al., 2019). Since then, reward machines have been used for solving problems in robotics (Shah et al., 2020; Shah & Shah, 2020), planning (Illanes et al., 2019, 2020), multi-agent systems (Neary et al., 2020), and partial observability (Toro Icarte et al., 2019a, 2019b). There has also been prominent work on how to learn RMs from experience (Toro Icarte et al., 2019a, 2019b; Xu et al., 2020a, 2020b; Furelos-Blanco et al., 2020a, 2020b; Rens & Raskin, 2020).

As previously mentioned, the original formulation of reward machines (Toro Icarte et al., 2018c; Camacho et al., 2019) was as Mealy machines, i.e. the outputs were associated with transitions rather than states. Recently, De Giacomo et al. (2020) considered both Mealy and Moore versions of reward machines, though theirs only output numbers (like our simple reward machines) instead of reward functions.

Since our previous work, we have gained practical experience and new theoretical insights about reward machines – which were reflected in this paper. In particular, we provided a cleaner definition of reward machines and QRM. On the RM side, we changed δ_r from returning a reward function on each transition to returning a reward function on each state (i.e., changed from a Mealy to Moore formulation). This is as expressive as before but simpler. We also added terminal states to reward machines because terminal states naturally arise in most practical applications. On the QRM side, we proposed CRM as a novel view of QRM that is simpler to understand and implement. Another improvement was the addition of HRM. In our ICML paper, hierarchical RL was a baseline. We hand-picked the set of options and learned policies following recommendations from Sutton et al. (1999) and Kulkarni et al. (2016). We now know that HRL performs too well to be just a baseline and, as such, we proposed HRM as a general HRL-based method for solving MDPRMs. The key difference between HRM and our previous HRL baseline is that HRM automatically extracts the set of options from the reward machine.

The experimental evaluation in this work is stronger too. First, we used a more standard performance metric (the average reward per step instead of the normalized discounted reward). Second, we added experiments on a continuous control environment (these are the first known results on continuous control for reward machines). And third, we include single-task experiments. Our ICML paper only had multitask experiments, which created the misconception that QRM only worked for multitask learning. We addressed those concerns in this paper. Finally, we reimplemented our code and made it fully compatible with OpenAI gym. We hope this will facilitate future research on reward machines.

6.2 Reward Specification

Beside standard reward functions, the most popular approaches for reward specification are *demonstrations* (e.g., Abbeel & Ng, 2004; Argall et al., 2009; Taylor & Chernova, 2010) and *feedback* (e.g., Thomaz et al., 2006; Knox & Stone, 2008; MacGlashan et al., 2017). When using demonstrations, tasks are specified using a set of expert traces. Then, inverse RL is used to transform traces into reward functions. In the case of feedback, an expert observes the agent behaving and rewards it by providing positive or negative feedback. Demonstrations and feedback are a useful proxy for task specifications, but they do not specify the task itself. As such, the agent can end up optimizing the wrong signal. In contrast, reward machines are a direct specification of the task to be accomplished.

Different reward specification languages have been proposed for RL. The approach of Williams et al. (2017) learns a natural language parser from single goal instructions to a sparse reward function. In contrast, Fasel et al. (2009) define an elaborate programming language for specifying rewards, actions, macro actions, advice, and task decomposition, among other things. In both cases, it is unclear how to automatically exploit their reward specification language to learn policies faster.

More recently, there has been significant interest in using formal languages to specify tasks, constraints, and advice in reinforcement learning (e.g., Li et al., 2017; Littman et al., 2017; Toro Icarte et al., 2017, 2018a, 2018b; Hasanbeig et al., 2018; Li & Belta, 2019; Li et al., 2019; Ringstrom & Schrater, 2019; Quint et al., 2019; Jothimurugan et al., 2019; Bozkurt et al., 2019; Koroglu & Sen, 2019; Shah et al., 2020; Shah & Shah, 2020; Gaon & Brafman, 2019; Ghasemi et al., 2020; De Giacomo et al., 2020; Li, 2020; Leon et al., 2020; Jiang et al., 2020). We hope to see more research in this direction in the next years. The use of formal languages can facilitate the problem of specifying correct reward functions for complex systems in RL. In addition, they also expose problem structure that RL agents could exploit to learn policies faster. That said, creating learning methods tailored for each language – LTL, LDL (Linear Dynamic Logic), and so on – could require a lot of work. We think that this problem could be ameliorated if we adopt reward machines as a *lingua franca* – a normal form for representing reward functions – that different formal languages could be mapped to. Then, we could focus our efforts in two subproblems: (i) understanding how to map particular formal languages into equivalent RMs and (ii) understanding how to exploit the RM structure to learn policies faster. We have made progress in both ends. While this paper shows methods for exploiting the RM structure, Camacho et al. (2019) shows a way of mapping regular languages into RMs. Note that any LTL or LDL formula (interpreted over finite histories) describes a regular language.

6.3 Exploiting Prior Knowledge

Our approaches for exploiting the RM structure are inspired by methods for exploiting prior knowledge in RL. In particular, prior knowledge has been used for problem decomposition (Parr & Russell, 1998; Sutton et al., 1999; Dietterich, 2000), data augmentation (Andrychowicz et al., 2017; Pitis et al., 2020), and reward shaping (Ng et al., 1999).

Hierarchical reinforcement learning is the most successful methodology to exploit decompositions in RL. Some foundational HRL works include *H-DYNA* (Singh, 1992a), *MAXQ* (Dietterich, 2000), *HAMs* (Parr & Russell, 1998), and *Options* (Sutton et al., 1999). The role of the hierarchy is to decompose the task into a set of sub-tasks that are reusable and easier to learn. However, these methods cannot guarantee convergence to optimal policies. The reason is that hierarchies constrain the policy space and, hence, might prune optimal policies. An RM can be viewed as a form of hierarchy that also defines the reward function. This allows us to define methods that can speed up learning and still guarantee convergence to optimal policies (e.g., QRM, CRM, or RS). We also proposed HRM, which automatically extracts an option-based hierarchy from an RM to solve MDPRMs faster (although it inherits the possibility of convergence to suboptimal policies from the option framework).

Singh (1992a, 1992b) proposed an alternative to HRL which defines tasks as sequences of sub-goals. Independent policies are trained to achieve each sub-goal, and then a gating function learns to switch from one policy to the next. The same idea was exploited by *policy sketches* (Andreas et al., 2017) but without the need for an external signal when a sub-goal is reached. In contrast, reward machines are considerably more expressive than sub-goal sequences and sketches, as they allow for interleaving, loops, and compositions of entire reward functions. Indeed, regular expressions can be captured in finite state machines.

CRM exploits a similar learning principle as Hindsight Experience Replay (HER) and Counterfactual Data Augmentation (CoDA). HER was proposed by Andrychowicz et al. (2017) and, like CRM, relies on relabelling experiences to learn policies faster. However, CRM relabels experiences using the RM and HER uses goal states. In this sense, RMs and CRM are more general as they work over combinations of reward functions. This allows CRM to learn policies that cannot be learned by HER, such as reaching a goal state while avoiding some objects (tasks 9 and 10 in Section 5.2) or loopy behaviours (task 1 in Section 5.3). CoDA was recently proposed by Pitis et al. (2020) as a new technique for generating counterfactual experiences in reinforcement learning. This approach consists of combining two experiences to generate new counterfactual experiences by exploiting local independent causal factors. In contrasts, CRM exploits the RM to generate multiple counterfactual experiences from one single environment experience. Further exploring synergies between RMs, HER, and CoDA is a promising direction for future work.

Our method for automated reward shaping was inspired by Camacho et al. (2018). In that work, Camacho et al. proposed an approach for automated reward shaping over automata in service of finding a policy for a fully specified MDP with an LTL-specified reward function. To do so, they defined a potential-based function that considered the distance between each automata state and its closest accepting state. In this paper, we extended Camacho et al.'s approach to work over simple reward machines.

7. Concluding Remarks

In this paper we introduced the notion of reward machines – a form of finite state machine that can be used to specify the reward function of an RL agent. Reward machines support the specification of arbitrary rewards, including sparse rewards and rewards for temporally extended behaviors. Reward machines expose structure in the reward function and, in so doing, can speed up learning as demonstrated in our experiments. We proposed different methodologies to exploit the structure in the reward machine to learn policies faster. They included automated reward shaping, counterfactual reasoning, and decomposition methods. We discussed the convergence guarantees of these approaches in the tabular case and empirically evaluated their effectiveness in discrete and continuous domains.

We believe there is significant potential in reward machines beyond what has been described in this paper. For one, reward machines can decrease the overhead of defining new tasks in a given environment since, having defined a set of relevant events, creating a new reward machine is straightforward. In fact, it would be possible to automatically create random tasks and their corresponding natural language descriptions using reward machines. This would allow for generating training data for a deep network that could learn to map natural language commands into policies in the same way that functional programs are available as training data for learning how to interpret questions in CLEVR (Johnson et al., 2017). Another unexplored research direction is to go up in the Chomsky hierarchy and study combinations of reward machines with context-free and context-sensitive grammars.

Finally, we see many opportunities for using formal languages to define correct reward specification via reward machines and defining novel RL methodologies to exploit the knowledge within reward machines – resulting in agents that can understand humans' instructions and use them to solve problems faster.

Acknowledgments

We gratefully acknowledge funding from the Natural Sciences and Engineering Research Council of Canada (NSERC), the Canada CIFAR AI Chairs Program, and Microsoft Research. The first author also gratefully acknowledges funding from ANID (Becas Chile). Resources used in preparing this research were provided, in part, by the Province of Ontario, the Government of Canada through CIFAR, and companies sponsoring the Vector Institute for Artificial Intelligence `www.vectorinstitute.ai/partners`.

References

Abbeel, P., & Ng, A. Y. (2004). Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the 21st International Conference on Machine Learning (ICML)*.

Aksaray, D., Jones, A., Kong, Z., Schwager, M., & Belta, C. (2016). Q-learning for robust satisfaction of signal temporal logic specifications. In *Proceedings of the 55th IEEE Conference on on Decision and Control (CDC)*, pp. 6565–6570.

Andreas, J., Klein, D., & Levine, S. (2017). Modular multitask reinforcement learning with policy sketches. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*, pp. 166–175.

Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong, R., Welinder, P., McGrew, B., Tobin, J., Abbeel, O. P., & Zaremba, W. (2017). Hindsight experience replay. In *Proceedings of the 30th Conference on Advances in Neural Information Processing Systems (NIPS)*, pp. 5048–5058.

Argall, B. D., Chernova, S., Veloso, M., & Browning, B. (2009). A survey of robot learning from demonstration. *Robotics and autonomous systems*, *57*(5), 469–483.

Bacchus, F., Boutilier, C., & Grove, A. J. (1996). Rewarding behaviors. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI)*, pp. 1160–1167.

Bozkurt, A. K., Wang, Y., Zavlanos, M. M., & Pajic, M. (2019). Control synthesis from linear temporal logic specifications using model-free reinforcement learning. *CoRR*, *abs/1909.07299*.

Brafman, R. I., De Giacomo, G., & Patrizi, F. (2018). LTLf/LDLf non-Markovian rewards. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI)*, pp. 1771–1778.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). OpenAI gym..

Camacho, A., Chen, O., Sanner, S., & McIlraith, S. A. (2017). Non-Markovian rewards expressed in LTL: Guiding search via reward shaping. In *Proceedings of the 10th Symposium on Combinatorial Search (SOCS)*, pp. 159–160.

Camacho, A., Chen, O., Sanner, S., & McIlraith, S. A. (2018). Non-markovian rewards expressed in LTL: Guiding search via reward shaping (extended version). In *GoalsRL, a workshop collocated with ICML/IJCAI/AAMAS*.

Camacho, A., Toro Icarte, R., Klassen, T. Q., Valenzano, R., & McIlraith, S. A. (2019). LTL and beyond: Formal languages for reward function specification in reinforcement learning. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 6065–6073.

De Giacomo, G., Favorito, M., Iocchi, L., Patrizi, F., & Ronca, A. (2020). Temporal logic monitoring rewards via transducers. In *Proceedings of the 17th International Conference on Knowledge Representation and Reasoning (KR)*, pp. 860–870.

De Giacomo, G., Iocchi, L., Favorito, M., & Patrizi, F. (2019). Foundations for restraining bolts: Reinforcement learning with LTLf/LDLf restraining specifications. In *Pro-*

ceedings of the 29th International Conference on Automated Planning and Sched. (ICAPS), pp. 128–136.

De Giacomo, G., Iocchi, L., Favorito, M., & Patrizi, F. (2020). Restraining bolts for reinforcement learning agents.. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI)*, pp. 13659–13662.

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

Fasel, I., Quinlan, M., & Stone, P. (2009). A task specification language for bootstrap learning. In *Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 1169–1170.

Furelos-Blanco, D., Law, M., Jonsson, A., Broda, K., & Russo, A. (2020a). Induction and exploitation of subgoal automata for reinforcement learning. *CoRR*, *abs/2009.03855*.

Furelos-Blanco, D., Law, M., Russo, A., Broda, K., & Jonsson, A. (2020b). Induction of subgoal automata for reinforcement learning.. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI)*, pp. 3890–3897.

Gaon, M., & Brafman, R. I. (2019). Reinforcement learning with non-Markovian rewards. *CoRR*, *abs/1912.02552*.

Ghasemi, M., Bulgur, E. A., & Topcu, U. (2020). Task-oriented active perception and planning in environments with partially known semantics. In *Proceedings of the 37th International Conference on Machine Learning (ICML)*.

Hasanbeig, M., Abate, A., & Kroening, D. (2018). Logically-constrained reinforcement learning. *CoRR*, *abs/1801.08099*.

Hesse, C., Plappert, M., Radford, A., Schulman, J., Sidor, S., & Wu, Y. (2017). OpenAI baselines. `https://github.com/openai/baselines`.

Illanes, L., Yan, X., Toro Icarte, R., & McIlraith, S. A. (2019). Symbolic planning and model-free reinforcement learning: Training taskable agents. In *Proceedings of the 4th Multi-disciplinary Conference on Reinforcement Learning and Decision (RLDM)*, pp. 191–195.

Illanes, L., Yan, X., Toro Icarte, R., & McIlraith, S. A. (2020). Symbolic plans as high-level instructions for reinforcement learning. In *Proceedings of the 30th International Conference on Automated Planning and Sched. (ICAPS)*, pp. 540–550.

Jiang, Y., Bharadwaj, S., Wu, B., Shah, R., Topcu, U., & Stone, P. (2020). Temporal-logic-based reward shaping for continuing learning tasks. *CoRR*, *abs/2007.01498*.

Johnson, J., Hariharan, B., van der Maaten, L., Fei-Fei, L., Zitnick, C. L., & Girshick, R. B. (2017). CLEVR: A diagnostic dataset for compositional language and elementary visual reasoning. In *Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1988–1997.

Jothimurugan, K., Alur, R., & Bastani, O. (2019). A composable specification language for reinforcement learning tasks. In *Proceedings of the 32nd Conference on Advances in Neural Information Processing Systems (NeurIPS)*, pp. 13041–13051.

Karpathy, A. (2015). REINFORCEjs: WaterWorld demo. `http://cs.stanford.edu/people/karpathy/reinforcejs/waterworld.html`.

Kingma, D. P., & Ba, J. (2015). Adam: A method for stochastic optimization. In Bengio, Y., & LeCun, Y. (Eds.), *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*.

Knox, W. B., & Stone, P. (2008). Tamer: Training an agent manually via evaluative reinforcement. In *Proceedings of the 7th IEEE International Conference on Development and Learning (ICDL)*, pp. 292–297.

Koroglu, Y., & Sen, A. (2019). Reinforcement learning-driven test generation for Android GUI applications using formal specifications. *CoRR*, *abs/1911.05403*.

Kulkarni, T. D., Narasimhan, K., Saeedi, A., & Tenenbaum, J. (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)*, pp. 3675–3683.

Lacerda, B., Parker, D., & Hawes, N. (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)*, pp. 1511–1516.

Lacerda, B., Parker, D., & Hawes, N. (2015). Optimal policy generation for partially satisfiable co-safe LTL specifications. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1587–1593.

Leon, B. G., Shanahan, M., & Belardinelli, F. (2020). Systematic generalisation through task temporal logic and deep reinforcement learning. *CoRR*, *abs/2006.08767*.

Li, X. (2020). *A formal methods approach to interpretability, safety and composability for reinforcement learning*. Ph.D. thesis, Boston University.

Li, X., & Belta, C. (2019). Temporal logic guided safe reinforcement learning using control barrier functions. *CoRR*, *abs/1903.09885*.

Li, X., Ma, Y., & Belta, C. (2018). A policy search method for temporal logic specified reinforcement learning tasks. In *Proceedings of the 2018 Annual American Control Conference (ACC)*, pp. 240–245.

Li, X., Serlin, Z., Yang, G., & Belta, C. (2019). A formal methods approach to interpretable reinforcement learning for robotic planning. *Science Robotics*, *4*(37).

Li, X., Vasile, C. I., & Belta, C. (2017). Reinforcement learning with temporal logic rewards. In *Proceedings of the 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 3834–3839.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., & Wierstra, D. (2016). Continuous control with deep reinforcement learning. In Bengio, Y., & LeCun, Y. (Eds.), *Proceedings of the 4th International Conference on Learning Representations (ICLR)*.

Littman, M. L., Topcu, U., Fu, J., Jr., C. L. I., Wen, M., & MacGlashan, J. (2017). Environment-independent task specifications via GLTL. *CoRR*, *abs/1704.04341*.

MacGlashan, J., Ho, M. K., Loftin, R. T., Peng, B., Wang, G., Roberts, D. L., Taylor, M. E., & Littman, M. L. (2017). Interactive learning from policy-dependent human feedback. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*, pp. 2285–2294.

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*(7540), 529–533.

Neary, C., Xu, Z., Wu, B., & Topcu, U. (2020). Reward machines for cooperative multi-agent reinforcement learning. *CoRR*, *abs/2007.01962*.

Ng, A. Y., Harada, D., & Russell, S. J. (1999). Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the 16th International Conference on Machine Learning (ICML)*, pp. 278–287.

Parr, R., & Russell, S. J. (1998). Reinforcement learning with hierarchies of machines. In *Proceedings of the 11th Conference on Advances in Neural Information Processing Systems (NIPS)*, pp. 1043–1049.

Pitis, S., Creager, E., & Garg, A. (2020). Counterfactual data augmentation using locally factored dynamics. *CoRR*, *abs/2007.02863*.

Quint, E., Xu, D., Dogan, H., Hakguder, Z., Scott, S., & Dwyer, M. (2019). Formal language constraints for Markov decision processes. *CoRR*, *abs/1910.01074*.

Rens, G., & Raskin, J.-F. (2020). Learning non-Markovian reward models in MDPs. *CoRR*, *abs/2001.09293*.

Ringstrom, T. J., & Schrater, P. R. (2019). Constraint satisfaction propagation: non-stationary policy synthesis for temporal logic planning. *CoRR*, *abs/1901.10405*.

Shah, A., Li, S., & Shah, J. (2020). Planning with uncertain specifications (PUnS). *IEEE Robotics and Automation Letters*, *5*(2), 3414–3421.

Shah, A., & Shah, J. (2020). Interactive robot training for non-Markov tasks. *CoRR*, *abs/2003.02232*.

Singh, S. (1992a). Reinforcement learning with a hierarchy of abstract models. In *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI)*, pp. 202–207.

Singh, S. (1992b). Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning*, *8*(3-4), 323–339.

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning - an introduction*. Adaptive computation and machine learning. MIT Press.

Sutton, R. S., Precup, D., & Singh, S. (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, *112*(1-2), 181–211.

Taylor, M. E., & Chernova, S. (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*, p. 23.

Thomaz, A. L., Hoffman, G., & Breazeal, C. (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)*, pp. 352–357.

Toro Icarte, R., Klassen, T. Q., Valenzano, R., & McIlraith, S. A. (2017). Using advice in model-based reinforcement learning. In *Proceedings of the 3rd Multi-disciplinary Conference on Reinforcement Learning and Decision (RLDM)*, pp. 199–203.

Toro Icarte, R., Klassen, T. Q., Valenzano, R., & McIlraith, S. A. (2018a). Advice-based exploration in model-based reinforcement learning. In *Proceedings of the 31st Canadian Conference on Artificial Intelligence (Canadian AI)*, pp. 72–83.

Toro Icarte, R., Klassen, T. Q., Valenzano, R., & McIlraith, S. A. (2018b). Teaching multiple tasks to an RL agent using LTL. In *Proceedings of the 17th International Conference on Autonomous Agents and Multiagent Systems (AAMAS).* 452–461.

Toro Icarte, R., Klassen, T. Q., Valenzano, R., & McIlraith, S. A. (2018c). Using reward machines for high-level task specification and decomposition in reinforcement learning. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*, pp. 2112–2121.

Toro Icarte, R., Waldie, E., Klassen, T. Q., Valenzano, R., Castro, M. P., & McIlraith, S. A. (2019a). Learning reward machines for partially observable reinforcement learning. In *Proceedings of the 32nd Conference on Advances in Neural Information Processing Systems (NeurIPS)*, pp. 15497–15508.

Toro Icarte, R., Waldie, E., Klassen, T. Q., Valenzano, R., Castro, M. P., & McIlraith, S. A. (2019b). Searching for Markovian subproblems to address partially observable reinforcement learning. In *Proceedings of the 4th Multi-disciplinary Conference on Reinforcement Learning and Decision (RLDM)*, pp. 22–26.

Van Hasselt, H., Guez, A., & Silver, D. (2016). Deep reinforcement learning with double q-learning.. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI)*, pp. 2094–2100.

Watkins, C. J. C. H., & Dayan, P. (1992). Q-learning. *Machine learning*, *8*(3-4), 279–292.

Williams, E. C., Rhee, M., Gopalan, N., & Tellex, S. (2017). Learning to parse natural language to grounded reward functions with weak supervision. In *Proceedings of the 2017 AAAI Fall Symposium on Natural Communication for Human-Robot Collaboration.*

Xu, Z., Gavran, I., Ahmad, Y., Majumdar, R., Neider, D., Topcu, U., & Wu, B. (2020a). Joint inference of reward machines and policies for reinforcement learning. In *Proceedings of the 30th International Conference on Automated Planning and Sched. (ICAPS)*, Vol. 30, pp. 590–598.

Xu, Z., Wu, B., Neider, D., & Topcu, U. (2020b). Active finite reward automaton inference and reinforcement learning using queries and counterexamples. *CoRR*, *abs/2006.15714*.