Using Reward Machines for High-Level Task Specification and Decomposition in Reinforcement Learning

Rodrigo Toro Icarte [1 2] Toryn Q. Klassen [1] Richard Valenzano [3] Sheila A. McIlraith [1 2]

Abstract

In this paper we propose Reward Machines – a type of finite state machine that supports the specification of reward functions while exposing reward function structure to the learner and supporting decomposition. We then present Q-Learning for Reward Machines (QRM), an algorithm which appropriately decomposes the reward machine and uses off-policy q-learning to simultaneously learn subpolicies for the different components. QRM is guaranteed to converge to an optimal policy in the tabular case, in contrast to Hierarchical Reinforcement Learning methods which might converge to suboptimal policies. We demonstrate this behavior experimentally in two discrete domains. We also show how function approximation methods like neural networks can be incorporated into QRM, and that doing so can find better policies more quickly than hierarchical methods in a domain with a continuous state space.

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, a priori, the transition probabilities or reward function manifest in the environment. To learn optimal behavior, 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 (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 it to decompose the problem and speed up learning. We consider a way to do so in this paper.

Previous work on giving an agent knowledge about the reward function focus on defining a task specification language, usually based on sub-goal sequences (Singh, 1992a;b) or linear temporal logic (Li et al., 2017; Camacho et al., 2017; Littman et al., 2017; Toro Icarte et al., 2018; Hasanbeig et al., 2018), 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 concatenations, 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 for decomposing it.

Our second contribution is to introduce an algorithm, called *Q-Learning for Reward Machines (QRM)*, that can exploit a reward machine's internal structure to decompose the problem and thereby improve sample efficiency. QRM's task decomposition does not prune optimal policies and uses *q*-

[1] Department of Computer Science, University of Toronto, Toronto, Ontario, Canada [2] Vector Institute, Toronto, Ontario, Canada [3] Element AI, Toronto, Ontario, Canada. Correspondence to: Rodrigo Toro Icarte <rntoro@cs.toronto.edu>.

Proceedings of the 35^{th} International Conference on Machine Learning, Stockholm, Sweden, PMLR 80, 2018. Copyright 2018 by the author(s).

learning to update each sub-task policy in parallel. We show that QRM is guaranteed to converge to an optimal policy in the tabular case, and also how to combine QRM with *deep reinforcement learning* methods. Experiments in three domains demonstrate that QRM can be effectively applied in both discrete and continuous environments and can find optimal policies in cases where hierarchical reinforcement learning methods can only find suboptimal policies.

2. Preliminaries

2.1. Reinforcement Learning

The RL problem consists of an agent interacting with an unknown environment (Sutton & Barto, 1998). As is standard, 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*.

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 a reward $r(s_t, a_t, s_{t+1})$ from the environment. The process then repeats from s_{t+1}. 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* 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 $q^*(s, a)$ in every state s. In the next subsection, we explain how to use the agent's experience to estimate q^*.

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 estimate, which is called the q-value, 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 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 $x \leftarrow x + \alpha \cdot (y - x)$.

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* 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.

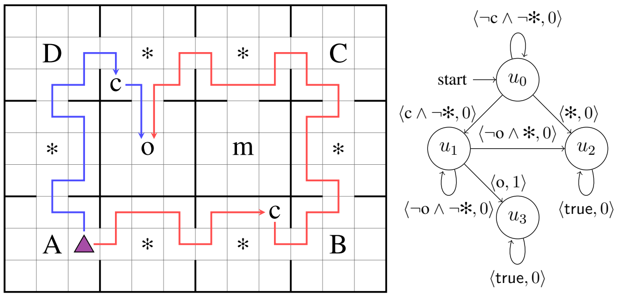
2.3. Deep Q-Networks (DQN)

Deep Q-Network (DQN) (Mnih et al., 2015) is a method which approximates $\tilde{q}(s, a) \approx \tilde{q}_\theta(s, a)$ 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's estimate $r(s, a, s') + \gamma \max_{a'} \tilde{q}_{\theta'}(s', a')$. 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 (a comprehensive summary was recently provided by Hessel et al. (2018)). We consider two of them: *Double DQN* (Van Hasselt et al., 2016) and *Prioritized Experience Replay* (Schaul et al., 2015). In short, double DQN decreases the overestimation bias of DQN by selecting the next action a' in the Bellman's estimate using $\tilde{q}_\theta$, i.e., the estimate is $r(s, a, s') + \gamma \tilde{q}_{\theta'}(s', \arg\max_{a'} \tilde{q}_\theta(s', a'))$. Prioritized experience replay biases the mini-batch sampling from the replay buffer towards experiences that the network is failing to predict. We focus on these two extensions because they improve performance while maintaining the off-policy nature of DQN.

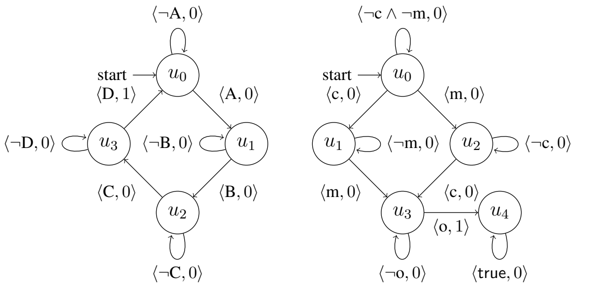
3. Reward Machines for Task Specification

In this section, we introduce a novel type of finite state machine, called a *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 transitions made in the RM. Hence,


D
✻
✻
C
c
✻
o
m
✻
c
A
✻
✻
B
$\langle \neg c \wedge \neg ✻, 0 \rangle$
start
u_0
$\langle c \wedge \neg ✻, 0 \rangle$
$\langle ✻, 0 \rangle$
$\langle \neg o \wedge ✻, 0 \rangle$
u_1
u_2
$\langle o, 1 \rangle$
$\langle \neg o \wedge \neg ✻, 0 \rangle$
u_3
$\langle \mathsf{true}, 0 \rangle$
$\langle \mathsf{true}, 0 \rangle$

(a) The office gridworld (b) A reward machine

Figure 1. An example environment and one task for it


$\langle \neg A, 0 \rangle$
start
u_0
$\langle D, 1 \rangle$
$\langle A, 0 \rangle$
$\langle \neg D, 0 \rangle$
u_3
$\langle \neg B, 0 \rangle$
u_1
$\langle C, 0 \rangle$
$\langle B, 0 \rangle$
u_2
$\langle \neg C, 0 \rangle$
$\langle \neg c \wedge \neg m, 0 \rangle$
start
u_0
$\langle c, 0 \rangle$
$\langle m, 0 \rangle$
u_1
$\langle \neg m, 0 \rangle$
u_2
$\langle \neg c, 0 \rangle$
$\langle m, 0 \rangle$
$\langle c, 0 \rangle$
u_3
u_4
$\langle o, 1 \rangle$
$\langle \neg o, 0 \rangle$
$\langle \mathsf{true}, 0 \rangle$

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

Figure 2. Two more reward machines for the office gridworld

an RM can be used to define temporally extended (and as such, non-Markovian relative to the environment) 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 what types of reward functions can be expressed using a reward machine in a given environment.

As a running example, consider the *office gridworld* presented in Figure 1a. In this environment, the agent can move in the 4 cardinal directions. It picks up coffee if at location c, picks up the mail if at location m, and delivers the coffee and mail to an office if at location o. The building contains decorations, marked with the symbol ✻, which the agent breaks if it steps on them. Finally, there are 4 marked locations: A, B, C, and D. In the rest of this section, we will show how to define multiple tasks for an RL agent in this environment using reward machines.

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 $\mathcal{P} = \{c, m, o, ✻, A, B, C, D\}$, where event $e \in \mathcal{P}$ occurs when the agent is at location e. Now we can 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 $\mathcal{R}_{\mathcal{P}SA} = \langle U, u_0, \delta_u, \delta_r \rangle$ where U is a finite set of states, $u_0 \in U$ is an initial state, δ_u is the state-transition function, $\delta_u : U \times 2^{\mathcal{P}} \to U$, and δ_r is the reward-transition function, $\delta_r : U \times U \to [S \times A \times S \to \mathbb{R}]$.*

An RM $\mathcal{R}_{\mathcal{P}SA}$ starts in state u_0, and at each subsequent time is in some state $u \in U$. At every step t, the RM receives as input a *truth assignment* σ_t, which is a set that contains exactly those propositions in $\mathcal{P}$ that are true in s_t. For example, in the office gridworld, $\sigma_t = \{e\}$ if the agent is at a location marked as e, then the RM 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 $r_t = \delta_r(u_t, u_{t+1})$ according to the reward-transition function.

For our examples, we will be considering *simple* reward machines, defined as follows:

Definition 3.2 (simple reward machine)**.** *A reward machine $\mathcal{R}_{\mathcal{P}SA} = \langle U, u_0, \delta_u, \delta_r \rangle$ is simple if for each $\langle u, u' \rangle \in U \times U$, there is some $c \in \mathbb{R}$ such that $\delta_r(u, u')(s, a, s') = c$ for all $\langle s, a, s' \rangle \in S \times A \times S$ (i.e., the function $r = \delta_r(u, u')$ is constant).*

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. 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 $\delta_r(u_i, u_j)$ returns a constant reward function equal to c. For instance, the edge between u_1 and u_3 labelled by $\langle o, 1 \rangle$ means that the machine will transition from u_1 to u_3 if the proposition o becomes true (regardless of the truth assignment of the other symbols) and output a reward function equal to 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 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* $L : S \to 2^{\mathcal{P}}$ which assigns truth values to the symbols in $\mathcal{P}$ given an environment state. The labelling function produces the truth assignments that are input to the RM.

Definition 3.3. *An MDP with a Reward Machine (MDPRM) is a tuple $\mathcal{T} = \langle S, A, p, \gamma, \mathcal{P}, L, U, u_0, \delta_u, \delta_r \rangle$, 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 $L : S \to 2^{\mathcal{P}}$, and U, u_0, δ_u, and δ_r are defined as in an RM.*

The reward machine 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, the RM moves to state $u' = \delta_u(u, L(s'))$ and the agent receives reward $r(s, a, s')$, where $r = \delta_r(u, u')$.

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 c. If ✻ is reached (i.e., a decoration is broken), the machine moves to u_2, from where the agent cannot receive further rewards. In contrast, if c is reached, the machine moves to u_1. While the machine is in u_1, two outcomes might occur. The agent might reach a ✻, in which case the machine moves to u_2 and returns no reward, or it might reach the office o, moving the machine to u_3 and receiving 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 $S \times U$. 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 $\mathcal{T} = \langle S, A, p, \gamma, U, u_0, \mathcal{P}, \delta_u, \delta_r, L\rangle$, let $\mathcal{M}_{\mathcal{T}} = \langle S', A', r', p', \gamma'\rangle$ be the MDP defined such that $S' = S \times U$, $A' = A$, $\gamma' = \gamma$,*

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

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

Finally, we note that reward machines can express any Markovian and some non-Markovian reward functions. In particular, given a set of states S, actions A, and propositional symbols $\mathcal{P}$, 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 history S^* only to the extent of distinguishing among those histories that are described by different elements of a finite set of regular expressions over $\mathcal{P}$.
3. Non-Markovian reward functions $R : S^* \times A \to \mathbb{R}$ that distinguish between histories via properties not expressible as regular expressions over $\mathcal{P}$ (such as counting how many times a proposition has been true) cannot be expressed using a reward machine.

In other words, RMs can return different rewards for the same transition (s, a, s') in the environment, for different histories of states 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 the safety constraints that are typical in safety critical systems. 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.

4. Reward Machines for Task Decomposition

In this section, we introduce *Q-learning for Reward Machines (QRM)*, an approach for learning policies for tasks defined by reward machines. QRM learns one subpolicy per state in the RM and uses off-policy learning to train each subpolicy in parallel. Then, we present the convergence guarantees of QRM and discuss its scalability with respect to the size of the RM. We end the section by showing how to combine QRM with Deep Learning.

4.1. Q-Learning for Reward Machines (QRM)

In this section, we describe a version of QRM that supports both single and multi task learning of RMs.

Algorithm 1 shows pseudo-code for QRM. It receives as input the set of propositional symbols $\mathcal{P}$, the labelling function L, the discount factor γ, and a list of reward machines $\Sigma = [\ldots \langle U^i, u_0^i, \delta_u^i, \delta_r^i\rangle \ldots]$ over $\mathcal{P}$. The goal is to learn an optimal policy for each of the tasks.

As a running example, consider the office gridworld and the three RM tasks defined in Figures 1 and 2. The first task T_0 rewards the agent when it brings coffee to the office without breaking any decoration (Figure 1b). Task T_1 rewards the agent after patrolling locations A, B, C, and D, in that order (Figure 2a). The final task T_2 rewards the agent when it delivers a coffee and the mail to the office (Figure 2b).

QRM decomposes the tasks by learning one q-value function per state in a reward machine. These q-functions are stored in $\widetilde{Q}$, and $\tilde{q}_j^i \in \widetilde{Q}$ corresponds to the q-value function for state j from task i. In the running example, $\widetilde{Q}$ would have 13 q-value functions in total, as the RMs have 4, 4, and 5 states, respectively.

After setting $\widetilde{Q}$, the algorithm has 3 nested loops. The first loop is over the number of episodes that we are running (line 3). Before running an episode, a task to run is selected (line 4). While different curriculum learning techniques can be used for task selection, we currently just repeatedly run through the tasks in the same order that they appear in Σ. In the running example, this means that, in the first episode, the

Algorithm 1 Q-learning for Reward Machines (QRM).

```
1:  Input: $\mathcal{P}$, $L$, $\gamma$, $\Sigma = [\ldots \langle U^i, u^i_0, \delta^i_u, \delta^i_r \rangle \ldots]$.
2:  $\widetilde{Q} \leftarrow$ InitializeQValueFunctions($\Sigma$)
3:  for $l = 0$ to num_episodes do
4:      $i \leftarrow$ GetTask($\Sigma$, $l$)
5:      $u^i_p \leftarrow u^i_0$; $s \leftarrow$ EnvInitialState()
6:      for $t = 0$ to length_episode do
7:          if EnvDeadEnd($s$) then
8:              break
9:          end if
10:         $a \leftarrow$ GetActionEpsilonGreedy($\tilde{q}^i_p$, $s$)
11:         $s' \leftarrow$ EnvExecuteAction($s$, $a$)
12:         for $\tilde{q}^o_j \in \widetilde{Q}$ {Learning loop} do
13:             $u^o_k \leftarrow \delta^o_u(u^o_j, L(s'))$
14:             $r \leftarrow \delta^o_r(u^o_j, u^o_k)$
15:             if EnvDeadEnd($s'$) then
16:                 $\tilde{q}^o_j(s,a) \xleftarrow{\alpha} r(s,a,s')$
17:             else
18:                 $\tilde{q}^o_j(s,a) \xleftarrow{\alpha} r(s,a,s') + \gamma \max_{a'} \tilde{q}^o_k(s',a')$
19:             end if
20:         end for
21:         $u^i_p \leftarrow \delta^i_u(u^i_p, L(s'))$; $s \leftarrow s'$
22:     end for
23: end for
```

agent will try to deliver coffee to the office. In the second episode, it will patrol. In the third, it will deliver coffee and the mail. Then, it will go back to the first task and so on.

The second loop runs an episode on the current task for a maximum number of steps (line 6). It contains the standard RL steps whereby the agent starts in a state s, selects an action a (line 10), executes a in the environment (line 11), and learns from this experience (lines 12-20). The loop also includes steps for keeping track of the current RM state u^i_p, where i is the task the agent is solving and p is the state id inside that RM. Note that the agent selects actions using the q-function corresponding to the current RM state q^i_p. This simply means that, while solving task i, the agent drives exploration using the policies associated to task i.

The third loop is the learning loop (lines 12-20). This is where, having just executed the action, the agent learns from the experience (s, a, s'). Given that the labelling function can detect which propositions, or events, from $\mathcal{P}$ are true in s', we can use the reward machines to compute how much reward the agent would have received if the reward machine had been in any RM state. In particular, for any RM state u^o_j, the reward machine is used to determine the RM state u^o_k that the RM would transition to due to experience (s, a, s'). The corresponding reward function is given by $r = \delta^o_r(u^o_j, u^o_k)$. Therefore, the update to $\tilde{q}^o_j$ is as follows:

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

Notice that the maximization step is over $\tilde{q}^o_k$, since those q-values would be used as part of action selection in s' since the RM state would then be in u^o_k.

As a concrete example, suppose that the agent is trying to solve the patrol task T_1. While exploring the environment, it reaches the office o. This experience is not particularly relevant for solving the patrol task. However, the agent can still use it to learn how to reach the office and, thus, improve its performance for the delivery tasks. The learning loop is an implementation of this idea using q-learning.

4.2. Convergence Guarantees and Scalability

QRM decomposes each task into a set of sub-tasks. An advantage of QRM with respect to alternative hierarchical methods (such as, options (Sutton et al., 1999), MAXQ (Dietterich, 2000), and policy sketches (Andreas et al., 2017)) is that QRM does not prune optimal policies when decomposing the task. As such, it is guaranteed to converge to optimal policies in the limit, as stated in the following theorem and proven in the supplementary materials.

Theorem 4.1. *Given a list of MDPRMs over the same environment, $\Sigma = [\ldots \langle S, A, p, \gamma, \mathcal{P}, L, U^i, u^i_0, \delta^i_u, \delta^i_r \rangle \ldots]$, QRM converges to an optimal policy in the limit (when the episode number and length go to infinity) for every MDPRM in Σ.*

QRM also learns every sub-task in parallel using off-policy RL. This significantly reduces the amount of experience needed to learn optimal policies, as we show in Section 5. While updating each sub-task policy on each step of the algorithm introduces a computational overhead, we can alleviate this by parallelizing the learning loop (line 12, Algorithm 1), as we did in our implementation. In cases where the number of sub-tasks is intractably large, a simple variant of QRM which only updates a random subset of policies during the learning loop could be used.

4.3. Deep Q-Learning for Reward Machines (DQRM)

Extending QRM to use deep learning is straight-forward. We just have to replace q-learning by Double DQN to learn the sub-task policies. The rest of the approach, including the task decomposition and methodology for computing the sub-task rewards, remains the same. We include prioritized experience replay by assigning priorities according to the networks average error across all sub-tasks. Our source code is publicly available at `https://bitbucket.org/RToroIcarte/qrm`.

This deep version of QRM is able to solve tasks in environments with continuous state spaces and still take advantage of task decomposition and off-policy learning. The price is, however, to lose convergence guarantees (as DQN might learn suboptimal policies for some tasks).

5. Experimental Evaluation

In this section, we provide an empirical evaluation of QRM and DQRM over domains with discrete and continuous state spaces. In each domain, we ask the agent to solve multiple tasks and report average performance across them. In our experiments we aim to investigate the following:

- Does using off-policy RL to learn sub-tasks in parallel improve the sample efficiency?
- Does our task decomposition help (D)QRM find better policies than alternative hierarchical RL methods?

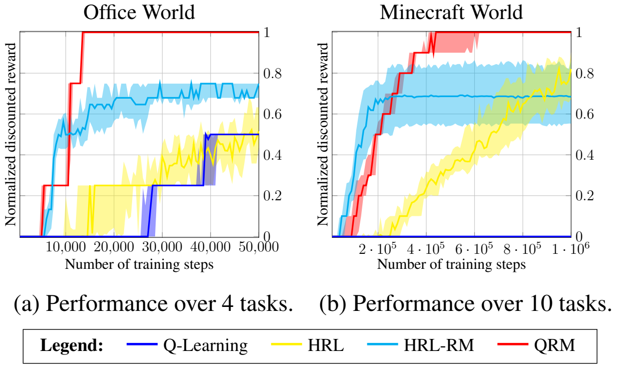
5.1. Baselines

We selected three baselines to compare against. The first baseline, *q-learning*, learns each task individually using standard q-learning (Watkins & Dayan, 1992) (for the continuous case, we use Double DQN (Van Hasselt et al., 2016) with prioritized experience replay (Schaul et al., 2015)). This baseline is intended to provide insights about whether (D)QRM's learning of sub-tasks in parallel (unlike in this baseline) improves the sample efficiency.

The second baseline is a Hierarchical RL approach (*HRL*) based on the option framework (Sutton et al., 1999). This approach learns policies over macro-actions called *options*. Each option is a tuple $\langle \mathcal{I}_o, \pi_o, \mathcal{T}_o \rangle$, where $\mathcal{I}_o$ is a set of states where the option can be started, π_o is a policy to follow when the option is being executed, and $\mathcal{T}_o$ is a set of states where the option ends. Following Kulkarni et al. (2016), we defined one option per event $p \in \mathcal{P}$, that terminates whenever p becomes true. Each policy π_o is optimized in parallel (using off-policy RL) to reach any state in $\mathcal{T}_o$ as soon as possible. When solving tasks on continuous domains, we use the Deep version *(DHRL)* of this approach proposed by Kulkarni et al. with Double DQN and prioritized experience replay. We note that optimizing locally for an option's policy may not allow HRL to find a globally optimal policy even in the tabular case (as the red path in Figure 1a shows). This behavior will be seen in the experiments below.

The third baseline augments HRL by pruning options that do not lead to reward according to the RM *(HRM-RM)*. For example, when solving the patrol task (Figure 2 left) the only option that makes sense to try at the beginning of the episode is to go to A (the first position to patrol). As such, HRL-RM is intended to be a fair comparison between QRM and a hierarchy based approach that also exploits the reward machine structure. However, this technique will still not necessarily find a globally optimal policy.

The three baselines work over the cross-product MDP defined in Observation 1. Thus, they learn from a Markovian reward function, as usual.



(a) Performance over 4 tasks. (b) Performance over 10 tasks.

Figure 3. Results on discrete domains.

5.2. Experimental setup

We tested on discrete and continuous domains across multiple tasks. Each task is encoded by a reward machine which gives reward of one if and only if the task is completed (see supplementary materials for more details). Some tasks might require reaching an object while avoiding others (e.g., reach the mail box while avoiding decorations). For both hierarchical RL baselines, we constructed the options such that they consider those extra constraints.

We report performance every n steps on the environment (where n is 100 steps in discrete domains and 1000 in continuous domains). Every n steps, we stop learning and test the agent's average performance across all tasks. For each task, we normalize by using the maximal achievable discounted reward from the initial state. Each algorithm is independently tested over several runs. We report the median performance and percentile 25 to 75 over the runs.

5.3. Results on Discrete Domains

We evaluated QRM and the three baselines on two gridworlds. In both domains, we use $\epsilon = 0.1$ for exploration, $\gamma = 0.9$, and $\alpha = 1$. The first domain is the office world described in section 3 (Figure 1a). We ask the agent to solve 4 tasks in this domain, including patrolling A, B, C, and D (the patrol ends when D is reached, which is slightly different from the task in Figure 2a, which is never completed), and bringing coffee and mail to the office without breaking the decorations. We test these for 30 independent trials on the map in Figure 1a. The results in Figure 3a show that QRM learns to optimally solve each tasks quite quickly, whereas hierarchical methods converge to suboptimal policies. The number of training steps was not enough for the q-learning baseline to find an optimal policy.

We also tested in the Minecraft-like gridworld introduced by Andreas et al. (2017). In this world, the grid contains raw materials (such as grass, wood, and iron) that the agent can extract and use to make new objects. Andreas et al. defined

10 tasks to solve in this world, 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 must force an unnecessary ordering when extracting 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 while solving the task. Figure 3b shows performance over 10 randomly generated maps, with 3 trials per map. In this domain, QRM also outperforms the three baselines while hierarchical RL converges to a suboptimal policy.

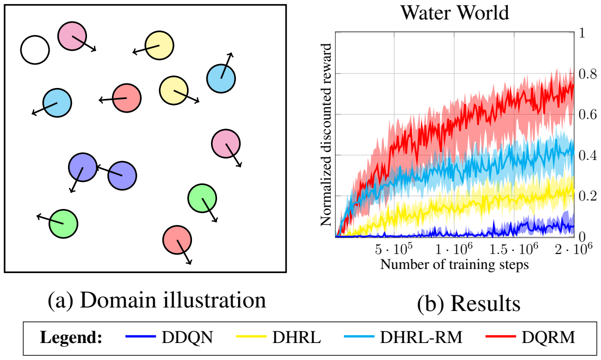
The results on these domains suggest affirmative answers to our experimental questions when using tabular representations. QRM seems to have the best of these two worlds: it learns optimal policies (like q-learning) and it learns at a comparable rate to Hierarchical RL methods.

5.4. Results on Continuous Domains

We tested the four approaches in the *water world* domain. This domain has a continuous state space and is based on Karpathy's "WaterWorld" (2015). Our version differs in that it is fully observable, and the reward depends on the given task. The environment consists of a two dimensional box with balls of different colors in it (see Figure 4a 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 by a constant factor. 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 the supplementary material.

In these experiments, we use versions of our four approaches that replace the tabular q-learning algorithm by Double DQN with prioritized experience replay. All the approaches uses the same feed-forward network with 6 hidden layers and 64 ReLu units in each layer. The network's input is a vector consisting of the relative position and velocity of each ball with respect to the agent and the absolute position and velocity of the agent in the box. We trained the networks using the Adam optimizer (Kingma & Ba, 2014) with a learning rate of $1e-5$. On every step, we updated the q-functions using 32 sampled experiences from a replay buffer of size 50000. The target networks were updated every 100 training steps. We set ϵ as 0.1 and the discount factor γ as 0.9. Our Double DQN implementation was based on the code from OpenAI Baselines (Hesse et al., 2017).



(a) Domain illustration

(b) Results

Figure 4. Results in the water world domain.

Figure 4b shows the results on 10 randomly generated water world maps. DQRM significantly outperforms our three baselines. This is particularly notable for two reasons. First, DQN is known to be unstable and, as such, training a subpolicy using experience from a different task might cause task interference, thereby destabilizing learning. This seems to not be the case in this domain. Second, while tabular QRM is expected to eventually outperform the hierarchical baselines since it is guaranteed to converge to an optimal policy, DQRM provides no such guarantee. Nevertheless, DQRM is outperforming hierarchical RL alternatives.

The results in the water world are encouraging. They show potential for DQRM to become a standard methodology for decomposing and learning multiple tasks in complex environments. They also suggest affirmative answers to our experimental questions. Still, further study is needed.

6. Related Work

Hierarchical reinforcement learning is the standard, and most successful, methodology to exploit task decomposition 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 subtasks that are reusable and easier to learn. However, none of these approaches can guarantee convergence to optimal policies. The reason is that hierarchies constrain the policy space and thus, since the hierarchy and reward functions are independent, HRL methods can potentially prune optimal policies. This is the main difference with respect to our approach. In our case, the reward machine is actually defining the reward function. As such, we can decompose the task, guarantee convergence, and also exploit off-policy learning to improve subpolicies in parallel.

That said, reward machines and QRM incorporate three main ideas from the Hierarchical RL literature. The first is to represent the hierarchy using some form of finite state machine, which can be tracked back to HAMs. In fact, our q-learning baseline is equivalent to transforming the RM into a HAM and solve it using *HAM-Q*. The second idea is to learn subpolicies in parallel using an off-policy method, which has been used when learning option policies. Finally, the idea of including rewards into the hierarchy is also used by options and MAXQ, though these rewards are decoupled from the actual reward function in these methods. It will be interesting to study which other insights from HRL can be exploited by reward machines.

Singh (1992a;b) proposed an alternative to HRL which defines tasks as sequence 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.

Another popular approach to task decomposition is to transform a single-agent problem into a multi-agent setting. The overall idea is to train separate agents using different reward functions and then have an *arbiter* combine the agents' opinions about which action to perform next (Karlsson, 1994; Russell & Zimdars, 2003; van Seijen et al., 2016). From these approaches, only (Russell & Zimdars, 2003) is guaranteed to converge to an optimal policy, and only if the sum of the agents' rewards is equal to the global reward. In contrast, our approach to task decomposition does not require an arbiter and always converges to an optimal policy.

Different task 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 task specifications. This language allows for specifying rewards, actions, macro actions, advice, and task decomposition, among other things. In both cases, it is unclear if tasks can be decomposed while still ensuring convergence to optimality, or if sub-task policies can be optimized in parallel.

Recently, there has been significant interest in using *Linear Temporal Logic (LTL)* to specify tasks in RL (Li et al., 2017; Littman et al., 2017; Toro Icarte et al., 2018; Hasanbeig et al., 2018). An LTL formula describes a pattern (e.g., "Eventually, the agent is at the key, and then eventually the agent is at the door") that might be satisfied (or violated) by a sequence of states. These approaches then use RL to find a policy that tries to satisfy the LTL formula. They are able to learn from non-Markovian task specifications and their language is quite expressive. However, reward machines can reward various behaviors to varying degrees as opposed to just rewarding the agent for completing tasks.

Reward machines are also related to reward shaping (Ng et al., 1999) as both allow for improving sample efficiency without compromising convergence guarantees. However, they exploit different principles for doing so. While reward shaping relies on guiding the exploration by providing artificial rewards to the agent, RMs rely on decomposing the task by exposing the reward structure to the agent. Combining these approaches is a promising direction for future work.

7. Discussion and 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 can expose structure in the reward function and, in so doing, can speed up learning as demonstrated in our experiments. We proposed a means of q-learning for reward machines (QRM) which decomposes the reward and uses off-policy learning to simultaneously learn subpolicies for the different components. QRM is guaranteed to converge to an optimal policy in the tabular case, in contrast to Hierarchical Reinforcement Learning.

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).

Reward machines also act as a small amount of memory for the agent. This feature would simplify learning when solving tasks in domains with partial observability.

Finally, as reward machines are a form of finite state machine, they can be created to correspond to sentences in many different formal languages, including regular expressions and various procedural programming languages. In future work, we plan to exploit these relationships to investigate novel behavioral specification languages for RL that can be mapped to reward machines and solved using QRM.

Acknowledgements

We gratefully acknowledge funding from the Natural Sciences and Engineering Research Council of Canada (NSERC). The first author also gratefully acknowledges funding from CONICYT (Becas Chile).

References

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

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

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

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

Hasanbeig, M., Abate, A., and Kroening, D. Logically-constrained reinforcement learning. *arXiv preprint arXiv:1801.08099*, 2018.

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

Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., and Silver, D. Rainbow: Combining improvements in deep reinforcement learning. *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI)*, pp. 3215–3222, 2018.

Johnson, J., Hariharan, B., van der Maaten, L., Fei-Fei, L., Zitnick, C. L., and Girshick, R. B. 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, 2017.

Karlsson, J. Task decomposition in reinforcement learning. In *Proceedings of the 1994 AAAI Spring Symposium on Goal-Driven Learning*, pp. 46–53, 1994.

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

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

Kulkarni, T. D., Narasimhan, K., Saeedi, A., and Tenenbaum, J. 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, 2016.

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

Littman, M. L., Topcu, U., Fu, J., Isbell, C., Wen, M., and MacGlashan, J. Environment-independent task specifications via GLTL. *arXiv preprint arXiv:1704.04341*, 2017.

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. Human-level control through deep reinforcement learning. *Nature*, 518(7540): 529–533, 2015.

Ng, A. Y., Harada, D., and Russell, S. J. 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, 1999.

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

Russell, S. J. and Zimdars, A. Q-decomposition for reinforcement learning agents. In *Proceedings of the 20th International Conference on Machine Learning (ICML)*, pp. 656–663, 2003.

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

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

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

Sutton, R. S. and Barto, A. G. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

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

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

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

van Seijen, H., Fatemi, M., Romoff, J., and Laroche, R. Separation of concerns in reinforcement learning. *arXiv preprint arXiv:1612.05159*, 2016.

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

Williams, E. C., Rhee, M., Gopalan, N., and Tellex, S. 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*, 2017.