Reinforcement Learning with Option Machines

Floris den Hengst\textsuperscript{1,2}\textsuperscript{*}, Vincent François-Lavet\textsuperscript{2}, Mark Hoogendoorn\textsuperscript{2}, Frank van Harmelen\textsuperscript{2}
\textsuperscript{1}ING Bank N.V.
\textsuperscript{2}Vrije Universiteit Amsterdam
Floris.den.Hengst@ing.com, \{Vincent.FrancoisLavet, M.Hoogendoorn, Frank.van.Harmelen\}@vu.nl

Abstract

Reinforcement learning (RL) is a powerful framework for learning complex behaviors, but lacks adoption in many settings due to sample size requirements. We introduce a framework for increasing sample efficiency of RL algorithms. Our approach focuses on optimizing environment rewards with high-level instructions. These are modeled as a high-level controller over temporally extended actions known as options. These options can be looped, interleaved and partially ordered with a rich language for high-level instructions. Crucially, the instructions may be underspecified in the sense that following them does not guarantee high reward in the environment. We present an algorithm for control with these so-called option machines (OMs), discuss option selection for the partially ordered case and describe an algorithm for learning with OMs. We compare our approach in zero-shot, single- and multi-task settings in an environment with fully specified and underspecified instructions. We find that OMs perform significantly better than or comparable to the state-of-art in all environments and learning settings.

1 Introduction

Reinforcement Learning (RL) is a powerful framework for learning complex behaviors. Sample efficiency, however, remains an open challenge in RL and prevents adoption in many real-world settings [Dulac-Arnold et al., 2019; den Hengst et al., 2020]. Sample efficiency is often improved with knowledge of a good solution, e.g. with demonstrations, increasingly complex tasks [Bengio et al., 2009], intermediate rewards [Ng et al., 1999] and by decomposing the task into subtasks that are easier to learn [Dietterich, 2000].

Recently, approaches have become popular for making RL more sample efficient with high-level symbolic knowledge. These methods combine the clear semantics, verifiability and well-understood compositional and computational characteristics of symbolic methods at a high level of abstraction with the power and flexibility of RL at large, low-level action and state spaces [Yang et al., 2018; Toro Icarte et al., 2018b; Toro Icarte et al., 2018a; Camacho et al., 2019; Lyu et al., 2019; Illanes et al., 2020; den Hengst et al., 2022]. These works demonstrate that symbolic instructions form a compelling complement to RL. A drawback of existing methods, however, is that they require the instructions to fully define the task at hand. Specifically, these assume that high rewards are always obtained if the instructions are followed. Such rich instructions, however, may be hard to attain in practice.

Firstly, knowledge of a good solution may be tacit. Secondly, the solution space may be so large that only partial instructions are feasible, e.g. chess opening and closing strategies. Finally, the quality of a solution may not be known a priori, e.g. when it depends on the agents’ capabilities or user preferences.

We therefore target a setting in which an agent is to optimize an environment reward with the help of underspecified instructions. These instructions define a solution at a high level of abstraction and, crucially, do not define the task at hand completely: following these instructions does not guarantee a high environment reward. Such instructions are abundant in a vast range of domains, including driving directions and clinical guidelines. In this paper, we propose and evaluate a framework for sample-efficient RL with underspecified instructions.

The framework consists of a high-level controller over a set of temporally extended actions known as options [Sutton et al., 1999] and uses a formalism that allows for looping, interleaving and partial ordering of such options. The policies for these options are trained to optimize an environment return and can be reused both within a single task and across tasks. We compare our approach with the state of the art on an environment with instructions that fully specify and underspecified instructions. In summary, the contributions of this paper are:

- the first approach to increase sample efficiency of an RL agent with high-level and underspecified instructions;
- methods for specification, control and learning for options with rich initiation and termination conditions;
- intuitive instruction semantics that allow reuse of options both within a single task and across multiple tasks;
- state of the art performance in a single-task setting and significant outperformance of the state of the art in zero-
related work and introducing preliminaries, we introduce our framework in Section 4. We detail how instructions are formalized and used for control, then present a learning algorithm in Section 5, an experimental evaluation in Section 6 and a discussion in Section 7.

2 Related Work

The literature on improving RL sample efficiency is vast and contains many task- or domain-specific approaches. We limit the discussion here to generic methods for expressing and supplying knowledge to the learner.

2.1 Hierarchical RL

Our work uses the expressive formalism of finite state transducers (FSTs) to specify initiation and termination conditions of temporally extended actions and can hence be seen as an extension of the options framework [Sutton et al., 1999], see Section 3.1. Our framework specifically proposes the use of a, to the best of our knowledge, novel kind of option with non-Markovian initiation and termination conditions, see Section 4.3. In the context of hierarchical RL, both sequential [Singh, 1992] and subroutine-based [Dietterich, 2000] formalisms have been used to define options. Unlike our proposed approach, these formalisms do not allow for interleaving, looping or partial ordering of options.

2.2 Classical Planning and RL

High-level control with classical planning and primitive control with RL goes back to Ryan [2002] who proposed to use plans obtained from high-level teleo-operators mapping states to suitable behaviors. Another early example used STRIPS planning and was extended with reward shaping [Grounds and Kudenko, 2005; Grzes and Kudenko, 2008]. More recently, Yang et al. [2018] and Lyu et al. [2019] proposed to use an action language from which subtasks are derived. Solutions to these are combined to solve new tasks and are optimized using intrinsic rewards. Illanes et al. [2020] introduced the problem of ‘taskable RL’ and propose a solution based on decomposition. Unfortunately, these works all require a planning goal that specifies the task completely and requires a planning model whereas our approach is robust against underspecified instructions and relies on instructions formalized as an FST which can be specified as e.g. LTL constraints.

2.3 Automata, Temporal Logics and RL

The first to recognize that automata can drastically improve RL sample efficiency were Parr and Russell [1998]. They proposed a ‘hierarchy of abstract machines’ to constrain the agent action space. This work was extended by iteratively refining the automata with data [Leonetti et al., 2012; Leonetti et al., 2016]. These automata operate on primitive actions and have no abstraction over actions.

Another line of work proposes to specify tasks in temporal logic formulas. These formulas are then converted into a reward function with the aim for the agent to learn how to satisfy the formula [Sadigh et al., 2014; Fu and Topcu, 2014; Li et al., 2017; Brafman et al., 2018; den Hengst et al., 2022]. These works require the full task to be specified whereas we target optimizing an unknown environment reward function using possibly underspecified instructions.

Some works consider decomposition of tasks specified in a temporal logic formula with the option framework. Andreas et al. [2017] introduced an approach for learning modular behaviors over sequences of subtasks. This approach optimizes an environment reward but does not support looping or interleaving subtasks and requires learning when to switch to a new subtask. Toro Icarte et al. [2018a] similarly learn a policy per subtask, but infer subtasks from an LTL formula using LTL progression. The same authors propose to learn a policy per state of an automaton representation of the formula [Toro Icarte et al., 2018b; Camacho et al., 2019]. These approaches specify temporally extended behaviors implicitly, i.e. there is no transparency at the meta-controller level, whereas we use explicitly named options. Use of options is therefore limited and their approach may not be applicable to certain zero-shot settings. On top of this, many policies may need to be learned, as the size of the automaton may grow exponentially in the size of the formula. Most importantly, these approaches also require that the entire task is specified upfront, whereas we target optimizing an unknown environment reward with possibly underspecified instructions.

3 Preliminaries

3.1 Reinforcement Learning

The RL framework can be used to maximize the amount of collected rewards in an environment by selecting an action at each time step [Sutton and Barto, 2018]. Such problems are formalized as a Markov Decision Problem (MDP) $M \equiv (S, A, T, R, \gamma, S_0)$ with a set of environment states $S = \{s^1, \ldots, s^n\}$, a set of agent actions $A = \{a^1, \ldots, a^n\}$, a probabilistic transition function $T : S \times A \to P(S)$ function $R : S \times A \times S \to R_{\min}, R_{\max}$ with $R_{\min}, R_{\max} \in \mathbb{R}$, a discount factor $\gamma \in [0, 1]$ to balance current and future rewards and $S_0$ a distribution of initial states at time step $t = 0$. At each time step $t$, the agent observes an environment state $s_t$ and performs some action $a_t \sim \pi \in \Pi : S \to P(A)$ and collects reward $r_t = R(s_t, a_t, s_{t+1})$. An optimal policy $\pi^\ast$ yields the highest obtainable discounted cumulative rewards. For complex tasks it may be difficult to discover any positive rewards. The agent can be given progressively more complex tasks known as curriculum learning [Bengio et al., 2009].

Actor-Critic Methods

Actor-critic (AC) methods optimize a set of weights $\theta$ on which the policy is conditioned: $a \sim \pi(s_t; \theta)$ [Williams, 1992; Konda and Tsitsiklis, 2000]. This actor is itself optimized with an estimated state-value $v_{\pi}(s; w)$, conditioned on a second set of weights $w$ referred to as the critic. Both sets of weights can then be optimized with the following update rules for given step sizes $\alpha^\theta, \alpha^w > 0$ and a given interaction with the environment $(s_t, a_t, r_t, s_{t+1})$ and resulting return $g = \sum_{j=t}^{\infty} \gamma^{j-t} R(s_j, a_j, s_{j+1})$ at time $t$:

$$w \leftarrow w + \alpha^w \left(\nabla v(s_t; w) \right) \left(g - \hat{v}(s_t; w)\right)$$  (1)
Options

The option framework introduces an abstraction over the space of actions [Sutton et al., 1999]. The agent selects a ‘primitive’ action $a \in A$ or ‘multi-step’ action at each time step. These options are formalized as a tuple $(O, \pi, \beta)$ where $O : S \rightarrow \{0, 1\}$ a function indicating in which states the option can be initiated, $\pi$ a policy that controls the agent when the option is active and $\beta : S \rightarrow \{0, 1\}$ a termination function that determines when the option becomes inactive. If the options are trained with an actor-critic method then each option $o$ can have its own actor $\theta_o$ and critic $w_o$. We denote the sets of all actors and critics for all options as $\Theta$ and $W$.

3.2 Finite State Transducers

Transducers are a generalization of finite state machines for control and define a mapping between two different types of information. We focus on deterministic FSTs whose output is determined by its current state and input, known in literature as a Mealy machine. We define a FST as a tuple $\langle \Sigma, \Omega, \varphi, F, \delta \rangle$ where $\Sigma$ is a finite input alphabet, $\Omega$ a finite output alphabet, $Q$ a finite set of states, $I \subseteq Q$ the set of initial states, $F \subseteq Q$ the set of terminal or final states, $\delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow Q \times (\Omega \cup \{\epsilon\})$ a transition functions where $\epsilon$ the empty string [Mealy, 1955]. A FST can be specified in a temporal logic such as LTL and then converted to a FST with out-of-the-box tools [Michaud and Colange, 2018].

4 The Option Machine Framework

In this section we introduce a framework for using underspecified instructions in RL. Specifications for Option Machines (OMs) can be underspecified in two ways. Firstly, the instructions specify what to do at a high level of abstraction rather than at the level of primitive actions. Secondly, a policy following the instructions in OMs is not assumed to always get high environment rewards. This contrasts with most related works, in which following the instructions is equated to high environment rewards. OMs, in contrast, use the environment reward as the canonical definition of the task and leverage instructions for reuse of obtained knowledge, improved exploration and better reward attribution.

Example 1. A recipe gives instructions for a particular type of pie. While each type of pie is a separate task, recipes refer to common steps such as mixing ingredients, pouring, baking etc. Solutions for these steps can be reused across recipes. A recipe may be underspecified and not guarantee a tasty result as baking requires more knowledge than just the recipes.

We now introduce OMs formally from the perspective of a curriculum of tasks. An OM curriculum is defined as a tuple $C : (S, A, T, \gamma, R, P, \Phi, L)$ where $S, A, T, \gamma$ are formalized as in RL, see Section 3.1. Tasks $R$ are formalized as a set of environment reward functions, $P$ a probability distribution over tasks $\mathcal{R}$ and instructions $\Phi$ as a set of FSTs. Each $\varphi_i \in \Phi$ corresponds to a particular task $R_i$ and has some $\Sigma_i$ of environment events as its input alphabet. We assume that a function for detecting these events $L : S \rightarrow \bigcup \Phi \Sigma_i$ is available. The main loop can be found in Algorithm 1 and contains components for control and learning.

4.1 Instructions as an Option Machine

Our approach uses high-level instructions for a given task. In particular, instructions define traces of high-level behaviors based on high-level descriptions of environment states. This allows for the intuitive formalization of e.g. a recipe.

Example 1. (cont.) A recipe ‘mix ingredients until smooth, fill pie plate and bake in oven at 180°C until golden. Apply a home-made topping or use a topping from the pantry to finalize the pie.’ See Figure 1 for an example OM.

High-level descriptions of states consist of events that the agent can detect in the environment. These are formalized as a set of atomic propositions $AP^*$, to which some truth value in $\Sigma : 2^{AP^*}$ can be assigned. $\Sigma$ corresponds to the input alphabet for the FST associated with the current task. We assume that some function $L : S \rightarrow \Sigma$ for detecting these events in states is available, e.g. as a handcrafted or pretrained component. We return to our running example before we look at how events are used for high-level control.

Example 1. (cont.) Events {smooth, pie-plate-filled, golden} can be identified from pixel-level states.

High-level Behaviors are actions that take multiple time steps and can be reused across tasks. These are formalized as options and denoted with a set of atomic propositions $AP^O$, to which some truth values in $\Omega : 2^{AP^O}$ can be assigned. At each time step, the permissible options in an OM are determined by this FST output. The current FST state $q_t$ and detected events $L(s_t)$ trigger some FST transition $\delta(q_t, L(s_t))$ which produces a new FST state $q_{t+1}$ and an output $\omega_t \in \Omega \cup \epsilon$. The ‘true’ propositions in $\omega_t$ are interpreted as the set of permissible options at that particular time step and are denoted $O_t \subseteq AP^O$. An OM consists of policies associated with options, a FST that specifies which options are
permissible and a mechanism to select from these. We discuss selection mechanisms in the next section. If no options are explicitly defined, then this is represented by the empty string $O_t = \{\}$.

Example 1. \textit{(cont.)} Figure 1 shows that mix is the only permissible option until the event smooth is detected. From this point onward, the option fill is permissible until the event pie-plate-filled becomes true etc. When the event golden has been detected, the two options make-topping and pantry-topping become permissible simultaneously.

### 4.2 Control with Option Machines

Control in the OM framework assumes a given task $R_i$ with corresponding FST $\varphi_i$, and has a two-level structure, see Algorithm 2. At the upper, meta-controller level, a suitable option is selected using $\varphi_i$. The policy for this option is then executed at the lower level and generates a primitive action $a_t \in A$ to be executed by the agent. In particular, an option is selected based on the FST output. This output defines one or multiple permissible options $O_t$. For now, we simply assume these policies to exist and leave the details on how these are optimized from interactions with the environment to Section 5.

Example 1. \textit{(cont.)} It may not be clear to a recipe author whether their audience has the right actuators to create a topping. Further, it may not be known whether e.g. pantry has been detected, the two options make-topping and pantry-topping become permissible simultaneously.

### 4.3 Reusable Policies and Non-Markovian Options

Policies in the OM framework have names $AP^O$ and can therefore easily be reused within a task or across tasks. For example, the policy for mixing ingredients can be used for mixing both the dough and the filling in a single cake recipe.

---

### Algorithm 1 Main loop

**Input:** curriculum $C : \{S, A, T, \gamma, R, P, \Phi, L\}$, parameterizations $\pi(\cdot; s, \theta)$ and $\hat{\psi}(s; w)$

**Parameters:** learning steps $N$, batch size $D$

**Output:** set of actors $\Theta$ and set of critics $W$

1: $i \leftarrow 0$, $D \leftarrow \emptyset$, $\Theta \leftarrow \emptyset$, $W \leftarrow \emptyset$
2: $\forall o \in AP^O \cup \varepsilon$, add random weights $\theta_o$ to $\Theta, w_o$ to $W$.
3: while $i < N$
4: \hspace{1em} while $|D| < D$ do
5: \hspace{2em} sample $(R \in R, \varphi \in \Phi) \sim P$.
6: \hspace{2em} $d \leftarrow$ rollout for task $R$ and instructions $\varphi$. \{Alg. 2\}
7: \hspace{1em} $D \leftarrow D \cup d$.
8: \hspace{1em} end while
9: \hspace{1em} update parameters $\Theta, W$ with $D$. \{Alg. 3\}
10: $i \leftarrow i + 1$.
11: end while
12: return $\Theta, W$.

### Algorithm 2 Control with an Option Machine

**Input:** finite-state transducer $\varphi$, actors $\Theta$, critics $W$, labelling $L : S \rightarrow \Sigma$

**Parameters:** shaping reward $\rho \geq 0$

**Output:** episode $d$

1: initialize $o, d \leftarrow \emptyset, q \leftarrow q_0 \in \varphi, s \leftarrow 0$.
2: while $q$ and $s$ are not terminal do
3: \hspace{1em} $(q', O) \leftarrow \delta(q, L(s))$. \{Equation 3 or 4\}
4: \hspace{1em} $o \leftarrow$ select from $O$.
5: \hspace{1em} perform action $a \sim \pi(\cdot; \theta_o)$.
6: \hspace{1em} observe $r$ and $s'$.
7: \hspace{1em} append $(s, o, q, a, r, s')$ to $d$.
8: $s \leftarrow s', q \leftarrow q'$.
9: end while
10: return $d$.

Additionally, multiple recipes may require mixing dough. Named options enable reuse of policies in e.g. a zero-shot setting where an unseen task can be solved by combining previously encountered options.

The initiation and termination condition of options in our framework are defined by the FST and based on the history of observed events $L(s_0), L(s_1), \ldots, L(s_l)$. These conditions are therefore non-Markovian. This enables powerful yet intuitive control, including looping and interleaving of options.

### 5 Learning with Option Machines

In this section we look at the problem of learning optimal policies for options from environment interactions generated by a sequence of these options. A key challenge here is to attribute rewards to the appropriate option. If an option was in control at a particular point in time, should future rewards be attributed to this option or not? First, however, we detail how instructions in OMs can be used to guide the agent with shaping rewards.

Shaping rewards are small positive (or negative) intermediate rewards for actions or states that are promising (or to be
Algorithm 3 Learning with Option Machines

<table>
<thead>
<tr>
<th>Env.</th>
<th>Setting</th>
<th>Sketch</th>
<th>Fixed</th>
<th>Greedy</th>
<th>Sticky</th>
</tr>
</thead>
<tbody>
<tr>
<td>maze</td>
<td>isolation</td>
<td>0.09</td>
<td><strong>0.60</strong></td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>holdout</td>
<td>0.49</td>
<td><strong>0.54</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>craft</td>
<td>isolation</td>
<td>0.03</td>
<td><strong>0.90</strong></td>
<td>0.74</td>
<td>0.73</td>
</tr>
<tr>
<td>holdout</td>
<td>0.05</td>
<td><strong>0.86</strong></td>
<td>0.13</td>
<td>0.22</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Zero-shot total environment reward on 1K test episodes. **Bold** denotes significant best (Mann-Whitney U, $p < 0.01$).

Algorithm 3 lists a learning algorithm that implements these ideas on reward shaping and reward attribution. First, the final automaton state and active option are extracted and both the discounted cumulative environment return $g_e$ and shaping return $g_s$ are initialized (lines 1-6). In lines 8-12, the last executing option $o$ in a particular FST state $q$ is set as the target option $o'$ to optimize and the shaping rewards are calculated. These are added to the total reward (lines 14-15) and used to update the actor and critic parameters (lines 16-17). The learning algorithm thus leverages FST state information in two ways: firstly, shaping rewards can be supplied to promote exploration and reinforce subtask completion and secondly, interactions are mapped to a single option to ensure that the parameters of the appropriate option are updated.

6 Experiments

In this section, we provide an empirical evaluation of OMs in an environment with both fully specified and underspecified instructions. We evaluate OMs in single-task, multi-task and two zero-shot settings to answer the research questions:

1. Do the instructions improve sample efficiency?
2. What are effects of named options and reward shaping?
3. Which option selection method to use?

We include versions of OMs for each of the option selection mechanisms described in Section 4.2: OM-fixed selects based on an arbitrarily fixed order, OM-greedy selects according to Equation 3 and OM-sticky according to Equation 4.

6.1 Baselines

We compare option machines to three state-of-the-art approaches. Firstly, we include the ‘sketch’-based approach proposed by Andreas et al.[2017]. This approach targets the multi-task setting, uses a sequence of subtasks rather than the richer representation proposed here and learns option termination conditions. Secondly, we compare to reward machines (RM) by Icarte et al. [2018b] which assume that the instructions specify the task fully and require that the training and evaluation subtasks use the same events. This is not the case for the tasks included here and we therefore do not include RM in the zero-shot setting. For all algorithms, we use AC as the base learner and we include a vanilla AC baseline per task in the single-task setting, denoted ‘RL’.

6.2 Experimental Setup

Two benchmark environments by [Andreas et al., 2017] are used to evaluate the approach. In the ‘craft’ environment,
items can be obtained by collecting resources such as wood and iron and combining them at workshop locations. Instructions may specify multiple permissible options simultaneously or may fully specify tasks. In the ‘maze’ environment, the agent must navigate a series of rooms with doors. An event detector describes whether the agent is in a door or not. Critically, it does not differentiate doors leading to the desired room from other doors. As a result, instructions are underspecified. Furthermore, instructions only permit one option at a time. We therefore do not include OM-greedy and OM-sticky in this environment.

An existing curriculum learning setup was used for multi-task learning [Andreas et al., 2017]. Initially, only tasks associated with two options are presented. Once the mean reward on these reaches a threshold of 0.8, this limit is increased. Tasks within this limit are sampled inversely proportional to the obtained reward. Results were selected with a grid search over hyperparameters. Shaping reward hyperparameters $\rho = 0$ and $\rho = 0.1$ were selected for the maze and craft environment respectively. We report averages over five random seeds. A detailed description of the environments, tasks, hyperparameters etc. can be found in the Appendix.

6.3 Results

Single-task Results

The two leftmost graphs in Figure 2 show the single-task results on all tasks consisting of more than two options. The maze environment proves too challenging. The reason is its inherent exploration problem which cannot be mitigated by the instructions. Following these does not guarantee solving the task and hence shaping rewards do not help. In the craft environment, shaping is useful: the OM-fixed and RM indicate that the usage of named options increases sample efficiency significantly. Again, we see that OM-fixed outperforms the other OM variants and that using sticky option selection provides a slight benefit.

Zero-shot Results

We evaluate applicable approaches in two zero-shot settings. In the first setting, policies for all options are trained in isolation and then evaluated on tasks composed of these options. We include all tasks here. In the second setting, policies are trained on a set of training tasks and then evaluated on two unseen, held out, tasks. For OM-based approaches, we execute Algorithm 2 in both settings. Table 1 shows that all of the OM versions significantly outperform the baseline in both environments. OM-fixed outperforms all OM versions. The difference here is striking in the holdout case.

The holdout setting is challenging since policies are optimized in the context of tasks other than the evaluation task. As a result, a policy associated with some option $o$ is positively reinforced if it completes a subtask associated with a later option $o'$. If this subtask is not part of the evaluation task, completing it may harm performance. It could take time and affect later subtasks if these are not commutative. OM-fixed is less susceptible to this failure mode than the other variants, as it uses the same delineation across all episodes. This does not show in the ‘isolation’ training setting where the greedy and sticky variants perform significantly better than their counterparts trained in the holdout setting.

7 Discussion

We proposed a framework for sample efficient RL with underspecified instructions. These are represented with powerful and intuitive FSTs as a natural way to define shaping rewards and use named options for the reuse of learned behaviors. Experimental evaluations show state of the art performance in a single-task setting and significant outperformance of the state of the art in zero-shot and multi-task settings across environments with fully specified and with underspecified instructions. We have found indications that shaping rewards should not be used when instructions do not cover the task at hand completely but that named options provide a significant benefit. Finally, results indicate that named options significantly increase performance in the multi-task and zero-shot settings.

Future work includes the development of a calculus of instructions for RL with FST operations and the study of ways to derive OMs from interactions to communicate learned strategies with other agents and humans.
References


