A Compression-Inspired Framework for Macro Discovery

Francisco M. Garcia
University of Massachusetts
Amherst, Massachusetts, USA
fmgarcia@cs.umass.edu

Bruno C. da Silva
Federal University Rio Grande do Sul
Porto Alegre, Rio Grande, Brazil
bsilva@inf.ufrgs.br

Philip Thomas
University of Massachusetts
Amherst, Massachusetts, Brazil
pthomas@cs.umass.edu

ABSTRACT

In this paper we consider the problem of how a reinforcement learning agent tasked with solving a set of related Markov decision processes can use knowledge acquired early on in its lifetime to improve its ability to more rapidly solve novel tasks. We propose a three-step framework in which an agent 1) generates a set of candidate open-loop macros based on samples drawn from near-optimal policies; 2) evaluates the value of each macro; and 3) selects a maximally diverse subset of macros that spans the space of policies typically required for solving tasks in the distribution. Our experiments show that extending the original primitive action-set of the agent with the identified macros allows it to more rapidly learn an optimal policy in unseen, but similar MDPs.

KEYWORDS

Reinforcement Learning; Hierarchical RL; Exploration

1 INTRODUCTION

One of the key aspects of human learning is our ability to construct building blocks upon which we can learn new skills. An infant learning how to walk may struggle with coordinating basic low-level motor movements at first. Later on in their life, that person might decide to learn how to play soccer. They are no longer concerned with how to walk or even how to run, given that these are skills they already possess; instead, their focus is on learning new soccer skills. In other words, a person is typically not required to learn new behaviors by always directly experimenting with low-level behaviors like they used to do as infants; they do, by contrast, simply bootstrap the knowledge they acquired early on in their lives. This suggests that it may be beneficial to use particularly useful (e.g. recurring) previously-acquired higher-level skills to more efficiently explore the consequences of an agent's actions when facing novel tasks.

In the RL literature, higher-level actions are sometimes called *options* or *macros*. They introduce a bias in the behavior of the agent, which is key during exploration to efficiently learn how to solve new problems. Carefully constructed macros have been shown e.g. to improve learning by allowing an agent to quickly reach distant areas of the state space during training. However, if options or macros are not appropriate for the problem at hand, they may substantially degrade learning [8]. The question this paper focuses on is: *"How can an agent identify and leverage useful macros for a given class or distribution of problems?"*.

In this work we consider the scenario where an agent is required to solve large number of different but related tasks, which define a *problem class*. We propose a framework that, after the agent has learned an optimal policy for a few initial tasks, allows it to identify macros that would help in learning to solve the remaining tasks. In our approach, after an agent learns optimal (or near-optimal) policies for a set of training tasks, trajectories from these policies are sampled to generate, evaluate, and select effective macros for the specific class of problems at hand.

In this paper we make the following contributions:

- Present a general framework for identifying macro actions appropriate to the problem class. We posit that useful macros are pieces of reusable or maximally recurring behaviors.
- Introduce the notion of the *value of a macro*, which we call the W-value.
- Introduce a novel approach for evaluating distances between macros so that, combined with the value of a macro, allows us to select a subset of macros that is maximally diverse and spans the space of policies typically required for solving tasks in the distribution.

2 RELATED WORK

A critical component that determines the performance of an agent when learning to solve a new task is its ability to efficiently explore the state space. Typically, exploration is done through random walks, although it is known that this strategy scales poorly as the size of the state space increases [17].

A better approach to exploring the state space, which has become increasingly popular in the last years, is through *options* or *macros*, [11, 13], which define temporally extended actions. Options are sub-policies that the agent can invoke in any state $s \in \mathcal{I}$ and that can terminate in any state $s \in \mathcal{T}$, where $\mathcal{I}$ and $\mathcal{T}$ define the initiation and termination set, respectively. Macros, on the other hand, are their open-loop counterpart and are defined as a finite-length sequence of actions.[1] These techniques allow the agent to "commit" to some behavior for an extended period of time, as opposed to randomly execute actions, and their demonstrated potential has led to the development of methods for identifying useful skills or macros to become an active area of research under the name of *skill discovery*.

One approach for option discovery is to identify important states in the transition graph of an MDP and learn policies that would lead the agent from any region of the state-set to those specific states. The work by McGovern and Barto [7] proposes splitting trajectories into successful and unsuccessful trajectories based on whether they were able to reach a pre-determined goal state. These trajectories

Proc. of the 18th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2019), N. Agmon, M. E. Taylor, E. Elkind, M. Veloso (eds.), May 2019, Montreal, Canada

https://doi.org/doi

[1] Different works have slight different definitions for macros [8]; in this work we define them as open-loop finite-length sequences of actions.

are then analyzed to identify *bottleneck states*, and options can be obtained by learning policies that cause the agent to reach those bottlenecks. A more recent approach based on a similar principle is the one presented by Machado et al .[6]. The authors extend the idea of using proto-value functions [5] to identify states of interest based on the eigen-values of the transition graph. They are, then, able to obtain options by learning an optimal policy that allows the agent to reach each of those states.

There are many other commonly-used approaches to option discovery that do not rely on finding bottleneck states [1, 3, 9]; however, many of them share the same drawbacks which limit how reusable the discovered options are: they assume that the transition graph will be maintained in future tasks. This is in contrast with our proposed framework, which allows us to extract generally useful open-loop macros by making minimal assumptions about the structure of the problem.

In this paper, we aim to develop a framework that is not constrained by these limitations. We analyze sample trajectories drawn from optimal policies to related tasks and use them to obtain open-loop macro actions that improve learning when facing new tasks in a given problem class.

3 BACKGROUND AND NOTATION

3.1 Background on Markov Decision Processes

A *Markov decision process* (MDP) is a tuple, $M = (\mathcal{S}, \mathcal{A}, P, R, \gamma, d_0)$, where $\mathcal{S}$ is the set of possible states of the environment, $\mathcal{A}$ is the set of possible actions that the agent can take, $P(s, a, s')$ is the probability that the environment will transition to state $s' \in \mathcal{S}$ if the agent executes action $a \in \mathcal{A}$ in state $s \in \mathcal{S}$, $R(s, a)$ is the real-valued reward received after taking action a in state s, d_0 is the initial state distribution, and $\gamma \in [0, 1]$ is a discount factor for rewards received in the future.

We use $t \in \{0, 1, 2, \ldots, T\}$ to index the time-step and write S_t, A_t, and R_t to denote the state, action, and reward at time t. We assume that T, the horizon, is finite, after which the environment resets to an initial state drawn from d_0. This process defines an episode and thus we restrict our discussion to *episodic* MDPs. A *policy*, $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$, provides a conditional distribution over actions given each possible state: $\pi(s, a) = \Pr(A_t = a|S_t = s)$.

In this paper, we define C, the *problem class*, as the set of all related tasks or problems c that an agent may face, where $c = (\mathcal{S}, \mathcal{A}, P_c, R_c, \gamma, d_0^c)$. In particular, note that we define C such that all $c \in C$ are MDPs sharing the same state-set $\mathcal{S}$ and action-set $\mathcal{A}$, but may have different transition functions P_c, reward functions R_c, and initial state distributions d_0^c. For example, Liu et al. [4] showed how RL can be used to efficiently allocate resources to servers in the cloud and reduce power usage from collected usage profiles. In this case C would correspond to the "resource allocation" problem for servers and each task $c \in C$ could refer a specific set of servers whose resources need to be managed.

A *trajectory* from a policy π is a sequence of state, action, and rewards $\tau = (s_0, a_0, r_0 \ldots, s_n, a_n, r_n)$, $n \leq T$, and is obtained by following the policy for T time-steps or until a terminal state is reached. We use τ_a to refer only to the sequences of actions in a trajectory τ; we refer to τ_a as an *action-trajectory*.

The value of an action a in state s under a policy π in a task c is referred to as the Q-value and is determined by the Q function Q_c^π: $Q_c^\pi(s, a) = \mathbf{E}\left[\sum_{t=0}^{T} \gamma^t R_t | S_t = s, A_t = a\right]$. A useful property of the Q-function is given by *Bellman equation*:

$$Q_c^\pi(s, a) = \mathbf{E}\left[R_t + \gamma\ Q(S_{t+1} = s', A_{t+1} = a')|S_t = s, A_t = a\right]$$

This implies that the Q value at S_t and A_t can be determined from knowing the expected Q value at S_{t+1}, A_{t+1} and the expected value of R_t.

Finally, we define a macro of length l to be a sequence of actions $m = (a_0, \cdots, a_l)$. We denote by $m_{(i)}$ the i^{th} action in macro m, and define $Q_c^\pi(s, m)$ to be the Q value of state-macro pair (s, m). Given a set of macros, $\mathcal{M}$, we define an *extended* action set of an MDP with action set $\mathcal{A}$ as $\mathcal{A}_\mathcal{M} = \mathcal{A} \cup \mathcal{M}$. That is, an extended action set is composed of both the primitive actions in $\mathcal{A}$ and the macros in $\mathcal{M}$. Our goal in this work (formalized in Section 4) is to find a set of macros that maximizes performance on the problem class.

3.2 Background on Compression Algorithms

The goal of compression is to represent messages or data in a compact manner by drastically reducing the number of bits needed to express the same information. Many compression algorithms share the same building blocks and their differences lie in how those elements are constructed and used.

Given an initial set of symbols Σ, called an alphabet, compression techniques seek to identify the most frequently used symbols in the alphabet and generate a *codebook* where each symbol is assigned a unique binary representation—a unique *codeword*. Once the codebook is built, new messages can be expressed in binary form by mapping each symbol (or sequence of symbols) in the message to a codeword in the codebook. For example, consider an alphabet $\Sigma = \{a, i, h\}$ and two different codebooks associating a codeword with each symbol: codebook $A = \{0, 1, 01\}$ and codebook $B = \{01, 0, 1\}$. Furthermore, consider encoding a message α = "hi" under each different codebook. The binary representation of α under codebook A would be 011 ($h = 01$, $i = 1$); however, under codebook B, it would be represented as 10 ($h = 1$, $i = 0$). Compression techniques seek to find a compact representation to express messages.

In this work we will consider the action-trajectories obtained from a trajectory analogous to messages, and primitive actions analogous to an initial alphabet. By taking this perspective, compressing a set of sampled action-trajectories will naturally result in generating a set of macros that are able to re-express sampled trajectories in a compact manner (using fewer symbols), thereby reducing the number of decision an agent must make.

4 PROBLEM STATEMENT

We consider the setting where an agent is required to solve a set of tasks $c \in C'$, $C' \subset C$, where C is a given problem class, and assume that when solving a particular task, it can interact with it for I episodes. After the agent has trained on a subset $C_{train} \subset C'$ of tasks, we are interested in identifying a set of macros to be used for improving learning in the set of remaining tasks $C_{test} \subset C'$. Notice that all tasks belong to the same problem class and so we wish to identify patterns in the optimal policies for problems already solved

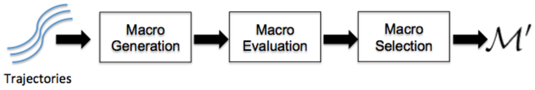


Figure 1: Diagram depicting proposed framework.

to improve learning in the remaining problems. We work under the assumption that macros identified in trajectories of optimal policies in a set of representative training tasks will be useful to solve other related tasks drawn from the same problem class. This assumption is supported by our experimental results.

As a concrete example, consider the case of an agent tasked to allocate cloud resources as described by Liu et al. [4]. In early stages of training, while the agent has not yet found a good performing policy, resources are allocated sub-optimally, which results in large latency and power consumption relative to an optimal policy. If the agent is able to leverage optimal policies for related problems in order to improve learning, it would be able quickly reach an efficient resource allocation policy in any new problem it might face. In this scenario, the agent could learn optimal policies for a small sub-set of resource allocation problems, C_{train}, analyze trajectories from these policies and identify macros to quickly obtain efficient policies for a set of novel problems, C_{test}.

We define the performance of a set of macros $\mathcal{M}$ in a particular task c to be $\rho(\mathcal{M}, c) = \mathbf{E}\left[\frac{1}{I}\sum_{i=0}^{I}\sum_{t=0}^{T} R_t^i \middle| \mathcal{A}_{\mathcal{M}}, c\right]$, where R_t^i is the reward at time step t during the i^{th} episode. This quantity expresses the expected average return an agent gets over I episodes on a task c using an extended action set $\mathcal{A}_{\mathcal{M}}$. This implies that the agent uses some learning algorithm to update its policy and the performance of a set of macros is defined by how quickly those macros allow an agent to improve its return during training.

Our goal is to find one (of possible many) optimal set of macros $\mathcal{M}^*$ for C' according to the following criterion:

$$\mathcal{M}^* \in \underset{\mathcal{M}}{\arg\max} \quad \frac{1}{|C'|}\sum_{c \in C'} \rho(\mathcal{M}, c). \tag{1}$$

Unfortunately, the domain of the objective in Eq. 1 is discrete, making the objective non-differentiable, and thus difficult to optimize. In this paper we posit that compression techniques provide a means to identify highly reusable macros which represent recurring behaviors in the problem class. These allow an agent to more effectively learn to solve new tasks, since they enable the agent to reproduce previously-observed recurring optimal behaviors, thereby allowing it to acquire optimal policies for novel tasks while making fewer decisions. In the next section, we propose using compression as a method for generating a set of candidate macros and approximating the set $\mathcal{M}^*$ by incorporating the top performing and diverse macros, $\mathcal{M}'$, to the agent action-set.

5 A HEURISTIC APPROACH FOR APPROXIMATING $\mathcal{M}^*$

The proposed framework can be summarized by the diagram shown in Figure 1. After the agent has already trained in a set of tasks $C_{train} \subset C'$, the agent obtains an optimal policy π_c^* for each task and samples n trajectories from each policy π_c^* for task c. Once these samples have been obtained, our framework generates a set of macros $\mathcal{M}'$ as an approximation to $\mathcal{M}^*$ by a 3-step process: **1)** macro generation, **2)** macro evaluation, and **3)** macro selection.

5.1 Macro Generation - A Compression Perspective to Identify Recurrent Action Sequences

There are many possible ways to generate macros from sampled trajectories. One approach would be to simply analyze all possible sequences of actions that can be obtained from these samples. However this would generate an extremely large number of macros; combinatorial in the length of the sampled trajectories, to be precise. As a practical strategy to deal with this issue, we propose using compression techniques to generate candidate macros.

Consider the problem of finding a compressed representation for a action-trajectory $\tau_a = (a, b, c, d)$ where $\{a, b, c, d\} \in \mathcal{A}$. From the perspective of compression, we can consider t_a akin to a message we wish to compress and $\{a, b, c, d\}$ to the symbols in the initial alphabet. Compressing τ_a, thus, would result in building larger repeating sequences of symbols that are incorporated to the alphabet. That implies that initially the alphabet is composed of only primitive actions and after compression, it will also contain macros.[2]

Following this intuition, the sampled action-trajectories are compressed and the symbols defined in the final codebook represent a set of candidate macros, $\mathcal{M}$, to be evaluated. Because these symbols are the ones that allow optimal trajectories to be compressed, they are (by construction) highly recurring in those trajectories. This implies that they are pieces of behaviors that often appear as part of optimal policies for tasks in the a specific class of problems and are, for this reason, good candidates for reusable and recurring macros. In this work, we selected LZW [16] as a compression algorithm because of its simplicity and efficiency in populating the codebook. Algorithm 1 shows our adaptation to encode action-trajectories as macros.

5.2 Macro Evaluation - The Value of a Macro

At this stage we have generated a possibly large set of candidate macros $\mathcal{M}$, but we do not have a sense of how useful they are in general in relation to each other when solving tasks from the problem class. One way of evaluating them would be to re-train the agent on the training tasks, adding each macro in turn to the action-set and assessing the resulting improvement in learning achieved with respect to only using primitive actions. However, this would

[2] It is worth noting that not all compression algorithms build their alphabet incrementally, but many popular ones (such as LZW) do.

Algorithm 1 LZW - macro codebook generation

```
$\Sigma = \mathcal{A}$
macro $m = ()$
for each action-trajectory $\tau_a$ do
    for each action $a$ in $\tau_a$ do
        $m = m + a$
        if $m \notin \Sigma$ then
            $\Sigma = \Sigma \cup \{m\}$
            $m = ()$
```

quickly become very expensive for large action spaces where there could be thousands of macros. We propose a score for evaluating a macro in a problem class that can be efficiently computed offline in closed-form based on the Q-values of primitives. We propose determining the value of a macro m over a problem class C by the W-function defined as:

$$W_C^{\pi}(m) = \mathbf{E}\ \left[Q_C^{\pi}(S, m)\right] \tag{2}$$

where the expectation is defined over both tasks C and states S sampled from the *on-policy distribution* [12]. In other words, we defined the value of a macro m to be the expected Q-value of m over all states in the problem class.

Assuming that compressing action-trajectories generates a large number of candidate macros, learning the true value of macros as defined in 2 for each candidate becomes computationally expensive, particularly if the size of $\mathcal{S}$ is large. However, this formulation allows us to compute the W-values for all macros in closed-form, provided we have access to the true Q-values for all $a \in \mathcal{A}$, to the transition function P_c, and π is greedy with respect to Q. This property is shown in Theorem 5.1 (for clarity we write $P_c(s^{(t)}, a, s^{(k)})$ in place of $\Pr(S_k = s^{(k)} | A_t = a, S_t = s^{(t)})$, $s^{(0)}$ refers to the state where a macro or action is executed and $s^{(k)}$ refers to the state visited after executing $k-1$ actions from state $s^{(0)}$).

Theorem 5.1. *Let π be a policy, $c \in C$ a task in problem class C and $Q_C^{\pi}(s,a)$ be the Q-value of executing action $a \in \mathcal{A}$ in state $s \in \mathcal{S}$. The value of $Q_C^{\pi}(s,m)$ (and consequently $W_C^{\pi}(a)$) can be computed in closed-form by:*

$$\begin{aligned}
Q_c^{\pi}(s, m) = &\sum_{k=0}^{l_m - 1}(\gamma^k[\sum_{s^{(1)} \in \mathcal{S}} \cdots \sum_{s^{(l_m)} \in \mathcal{S}} \frac{\Pi_{i=0}^{l_m-1} P_c(s^i, m_{(i)}, s^{i+1})}{P_c(s^{(k)}, m_{(k)}, s^{(k+1)})}] \\
&\times [Q(s^k, m_{(k)}) - \sum_{s^{k+1} \in \mathcal{S}} P_c(s^{(k)}, m_{(k)}, s^{(k+1)}) \\
&\times \gamma \sum_{a' \in \mathcal{A}} \pi(a', s^{(k+1)}) Q(s^{(k+1)}, a')]) \\
&+ \sum_{s^{(1)} \in \mathcal{S}} \cdots \sum_{s^{(l_m)} \in \mathcal{S}} \Pi_{i=0}^{l_m-1} P_c(s^{(i)}, m_{(i)}, s^{(i+1)}) \\
&\times \gamma \sum_{a' \in \mathcal{A}} \pi(a', s^{(l_m)}) Q(s^{(l_m)}, a')
\end{aligned}$$

Proof. Assume we know the true Q function for a policy π in task c over all primitive actions. We can calculate the Q-value of a macro $m = (a, b, c)$, at state s, and obtain a general expression for a macro of arbitrary length l_m as follows:

$$\begin{aligned}
Q_c^{\pi}(s, m) = &\sum_{s^{(3)} \in \mathcal{S}} P_c(s^{(0)}, m, s^{(3)})(R_c(s^{(0)}, m, s^{(3)}) \\
&+ \gamma \sum_{a' \in \mathcal{A}} \pi(a', s^{(3)}) Q_c^{\pi}(s^{(3)}, a')) \\
= &\sum_{s^{(3)}, s^{(2)}, s^{(1)} \in \mathcal{S}} P_c(s^{(2)}, c, s^{(3)})\ P_c(s^{(1)}, b, s^{(2)})\ P_c(s^{(0)}, a, s^{(1)}) \\
&\times (R_c(s^{(0)}, a, s^{(1)}) + \gamma R_c(s^{(1)}, b, s^{(2)}) + \gamma^2 R_c(s^{(2)}, c, s^{(3)}) \\
&+ \gamma \sum_{a' \in \mathcal{A}} \pi(a', s^{(3)}) Q_c^{\pi}(s^{(3)}, a')) \\
= &\sum_{s^{(3)}, s^{(2)}, s^{(1)} \in \mathcal{S}} [P_c(s^{(2)}, c, s^{(3)}) \\
&\times P_c(s^{(1)}, b, s^{(2)})\ P_c(s^{(0)}, a, s^{(1)})\ R_c(s^{(0)}, a, s^{(1)})] \\
&+ [P_c(s^{(2)}, c, s^{(3)})\ P_c(s^{(1)}, b, s^{(2)})\ P_c(s^{(0)}, a, s^{(1)}) \\
&\times \gamma R_c(s^{(1)}, b, s^{(2)})] \\
&+ [P_c(s^{(2)}, c, s^{(3)})\ P_c(s^{(1)}, b, s^{(2)})\ P_c(s^{(0)}, a, s^{(1)}) \\
&\times \gamma^2 R_c(s^{(2)}, c, s^{(3)})] \\
&+ [P_c(s^{(2)}, c, s^{(3)})\ P_c(s^{(1)}, b, s^{(2)})\ P_c(s^{(0)}, a, s^{(1)}) \\
&\times \gamma \sum_{a' \in \mathcal{A}} \pi(a', s^{(3)}) Q_c^{\pi}(s^{(3)}, a')] \\
= &[(\sum_{s^{(3)}, s^{(2)}, s^{(1)} \in \mathcal{S}} P_c(s^{(2)}, c, s^{(3)})\ P_c(s^{(1)}, b, s^{(2)}) \\
&\times (Q(s^{(0)}, a) - \sum_{s^{(1)} \in \mathcal{S}} P(s^{(0)}, a, s^{(1)}) \\
&\times \gamma \sum_{a' \in \mathcal{A}} \pi(a', s^{(1)}) Q(s^{(1)}, a'))] \\
&+ [\gamma \sum_{s^{(3)}, s^{(2)}, s^{(1)} \in \mathcal{S}} P_c(s^{(2)}, c, s^{(3)})\ P_c(s^{(0)}, a, s^{(1)}) \\
&\times (Q(s^{(1)}, b) - \sum_{s^{(2)} \in \mathcal{S}} P(s^{(1)}, b, s^{(2)}) \\
&\times \gamma \sum_{a' \in \mathcal{A}} \pi(a', s^{(2)}) Q(s^{(2)}, a')] \\
&+ [\gamma^2 \sum_{s^{(3)}, s^{(2)}, s^{(1)} \in \mathcal{S}} P_c(s^{(1)}, b, s^{(2)})\ P_c(s^{(0)}, a, s^{(1)}) \\
&\times (Q(s^{(2)}, c) - \sum_{s^{(3)} \in \mathcal{S}} P_c(s^{(2)}, c, s^{(3)}) \\
&\times \gamma \sum_{a' \in \mathcal{A}} \pi(a', s^{(3)}) Q(s^{(3)}, a')] \\
&+ \sum_{s^{(3)}, s^{(2)}, s^{(1)} \in \mathcal{S}} P_c(s^{(2)}, c, s^{(3)})\ P_c(s^{(1)}, b, s^{(2)})\ P_c(s^{(0)}, a, s^{(1)}) \\
&\times \gamma \sum_{a' \in \mathcal{A}} \pi(a', s^{(3)}) Q(s^{(3)}, a') \\
= &\sum_{k=0}^{l_m - 1}(\gamma^k[\sum_{s^{(1)} \in \mathcal{S}} \cdots \sum_{s^{(l_m)} \in \mathcal{S}} \frac{\Pi_{i=0}^{l_m-1} P_c(s^i, m_{(i)}, s^{i+1})}{P_c(s^{(k)}, m_{(k)}, s^{(k+1)})}] \\
&\times [Q(s^k, m_{(k)}) - \sum_{s^{k+1} \in \mathcal{S}} P_c(s^{(k)}, m_{(k)}, s^{(k+1)}) \\
&\times \gamma \sum_{a' \in \mathcal{A}} \pi(a', s^{(k+1)}) Q(s^{(k+1)}, a')]) \\
&+ \sum_{s^{(1)} \in \mathcal{S}} \cdots \sum_{s^{(l_m)} \in \mathcal{S}} \Pi_{i=0}^{l_m-1} P_c(s^{(i)}, m_{(i)}, s^{(i+1)}) \\
&\times \gamma \sum_{a' \in \mathcal{A}} \pi(a', s^{(l_m)}) Q(s^{(l_m)}, a')
\end{aligned}$$

□

Notice that this expression is given in terms of the Q-values of primitives, and consequently, the W-values can be calculated in closed form. Having access to the Q-values of primitives is a reasonable assumption considering that algorithms like Q-learning, [15] and DQN [10] approximate the true Q-value. If the agent uses these techniques to learn an optimal policy for the tasks in C_{train} it will have a reasonable approximation to Q readily available.

In the case of a greedy policy notice that: $\sum_{a'\in\mathcal{A}} \pi(a', s) \times Q_c^\pi(s, a') = \max_{a'} Q_c^\pi(s, a')$. Furthermore, by definition, there is always a state-primitive pair whose Q-value is no smaller than the largest state-macro pair Q-value. Consequently, we have that $\max_{a'\in\mathcal{A}} Q_c^\pi(s, a') \geq \max_{a'\in\mathcal{A}_\mathcal{M}} Q_c^\pi(s, a')$ for any set of macros $\mathcal{M}$. This implies that the addition of new macros to the action-set does not affect the value of the existing elements in the set.

In the case of stochastic policies, introducing a new element in the original action-set of the agent affects the summation in the last term over all actions since the probability distribution defined by π changes as well, so the actual value of a macro can no longer be calculated in closed form. However, it can still be calculated efficiently by applying the *Bellman equation* using the Q-values of primitives as a starting point.

5.3 Macro Selection - Encouraging Macro Diversity

Once the value of a macro has been estimated, it can be used to predict which macros we generally believe will lead to higher rewards. However, there is a trade-off we must account for when extending the agent's action set. If too few macros are included in the action set, the agent might miss on the ability to better explore the state space; on the other hand, including too many will result in the agent having too large of an action-set, which will hinder learning. This trade-off has also been observed in the context of options by Machado et al. [6]. We tackle this problem by establishing a distance metric between macros and only including those that are dissimilar enough to the rest of the action-set.

Let S_t be a random variable denoting the state where m is executed and S_{t+l_m} the state where m finishes execution. Furthermore, let $S' = d(S_{t+l_m}, S_t)$ denote a random variable describing the change in state caused by the execution of a macro, where d is a distance measure for the state space, and let p_m be the distribution for S' for macro m. We refer to p_m as the *end-state distribution*. We define the distance between two macros m_1 and m_2 to be the KL divergence between p_{m_1} and p_{m_2}, that is:

$$D_{KL}(p_{m_1}||p_{m_2}) = -\sum_{S'} p_{m_1}(S') \ \log\left(\frac{p_{m_2}(S')}{p_{m_1}(S')}\right).$$

In the case of continuous state spaces, we discretize the distribution into appropriately sized bins or buckets.

Figure 2 shows the empirical end-state distribution calculated for four macros in the maze navigation problem class (introduced in the next section). The macros m_1, m_2, m_3, m_4 are defined by repeating the same primitive action 5 times. The possible primitive actions are given by r, l, u, d and they allow the agent to move in the environment right, left, up or down, respectively. The figure intends to show that macros reflect their similarity (or differences) in the effect that they have in the distribution of state transitions, and we can measure the similarity between two macros by measuring the distance between their distributions.

The set $\mathcal{M}'$ is then incrementally built by only including those macros that have a minimum distance δ to all other macros that have already been included in the set. By selecting macros in descending order according to their W-value, their W-function defines a preference criterion by which macros can be selected.

Pseudocode describing our macro discovery framework is given in Algorithm 2.

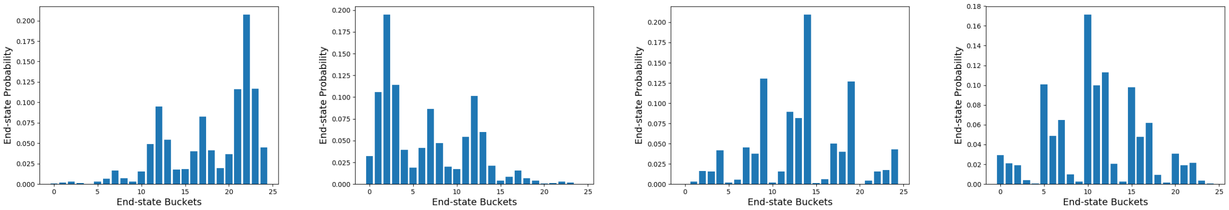
Algorithm 2 Macro discovery framework

```
1. Macro Generation
Learn optimal policy $\pi_c^*$ for all $c \in C_{train}$.
Collect action-trajectories $\tau_a$ from each $\pi_c^*$ in task $c$.
Generate macros $\mathcal{M}$ from all $\tau_a$ by Algorithm 1

2. Macro Evaluation
Sort all $m \in \mathcal{M}$ by $W_C^{\pi_c^*}(m)$ in descending order.

3. Macro Selection
$\mathcal{A}_{\mathcal{M}'} = \mathcal{A}$
for $m \in \mathcal{M}$ do
    if $\min D_{KL}(p_m||p_{m'}) > \delta, \forall m' \in \mathcal{A}_{\mathcal{M}'}$ then
        $\mathcal{A}_{\mathcal{M}'} = \mathcal{A}_{\mathcal{M}'} \cup \{m'\}$
```

6 EXPERIMENTAL RESULTS

In this section we present experimental results providing empirical evidence that the identified macros lead to improved learning. We first analyze two simple problem classes: chain and maze navigation, whose transition models can be defined apriori and the true Q-values for any policy can be accurately estimated in tabular form. These problems allow us to study the properties of our method in detail and visualize how the identified macros affect the behavior of the agent during learning. We then further extend our experiments to more complex problem classes by relaxing the assumption of access to the true Q-values of primitive actions, using function approximation to estimate Q and learning a model from data to estimate the transition function.

In the case of the chain problem, we limit our experiments to compare the performance between our framework and using only primitive actions. For all other experiments, we also contrast our approach to using Eigen-Options [6] and the Option-Critic architecture [1]. These methods work in a similar setting to ours, where the agent first interacts with some specific environments and those experiences can then be leveraged to facilitate learning in novel, but related problems. Our experiments show that despite macros being a simple open-loop alternative to options, they are sufficient to generalize to novel problems and result in improved performance relative to the competing methods. These tests also highlight the fact that Eigen-Options and the Option-Critic are not well suited to adapt to different transition graphs, while the macros identified by our framework capture recurring patterns across the problem class.



(a) End-state distribution for macro m_1

(b) End-state distribution for macro m_2

(c) End-state distribution for macro m_3

(d) End-state distribution for macro m_4

Figure 2: End-state distribution for macros m_1, m_2, m_3 and m_4 in the maze navigation problem class (described in experiments section). The primitive action-set is composed of for actions r, l, u, d, and macros defined as follows: $m_1 = (r, r, r, r, r)$, $m_2 = (l, l, l, l, l)$, $m_3 = (u, u, u, u, u)$, $m_4 = (d, d, d, d, d)$, where primitive actions r, l, u, d move the agent one state right, left, up or down, respectively.

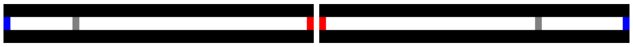
In the first two experiments, the agent was trained using Q-learning with tabular representation and in the remaining experiments the policy was trained using DQN [10]. Exploration was implemented with an ϵ-greedy strategy with an initial value of 0.9 and decreasing by a factor of 0.99 after each episode.

6.1 Chain Problem Class

In this problem class, the agent originally has at its disposal two primitive actions, $\mathcal{A} = \{a_1, a_2\}$. The states and transitions between states in each task form a chain, meaning that each state has two possible transitions, move to the state to the right or move to the state to the left. Given a state s_k at position k in the chain, action a_1 moves the agent to state s_{k+1} but with a small probability the agent moves to state s_{k-1}. Similarly, after taking action a_2, the agent moves to state s_{k-1} but with a small probability it moves to state s_{k+1}. The agent receives a large reward at either end of the chain, so if there are a total of n states in the chain, the agent receives a large reward R_0 or R_n upon reaching states s_0 or s_n, respectively. We ensure by construction that from the initial state in the chain the number of states at one end is much larger than the number of states at the other end of the chain, and that the reward obtained at the farther end is much larger than the reward obtained at the nearest end. In our implementation, when constructing a new task, an integer a between 0 and 100 is sampled uniformly to define the length of the chain to the right of the agent's initial position. The left side of the chain is assigned a length of $100 - a$. The end state at the end of the longest side of the chain results in a reward of +1000 and the one at the shortest end results in a reward of +10. In this experiment we set $\delta = 2.0$ to filter macros.

Two different different tasks within the chain problem class are shown in Figure 3. The agent's initial position is shown as a gray square within the chain, the state which results in the largest reward is shown in red at the farther end of the chain (relative to the initial position), and the state resulting in the smallest reward is shown in blue at the closer end of the chain.

We present this problem class as an intuitive example of the type of problems were simple open-loop macros can lead to a substantial improvement in the agent's performance. In this problem class, oftentimes the policy of the agent converges to the nearest end if exploration is done randomly using primitives, since it is unlikely that random exploration will reach the further end of the chain. However, if an agent has access to macros well suited for this type of problem, it is able to reach both ends of the chain early in its lifetime and learn the correct optimal policy for a specific task.



(a) Chain task example 1

(b) Chain task example 2

Figure 3: Example tasks for chain problem class. The agent starts in the location shown as a gray square within the chain. If it reaches the state at the farther end (shown in red) it receives a reward of +100, if it reaches the state at the closer end (shown in blue) it receives a reward of +10.

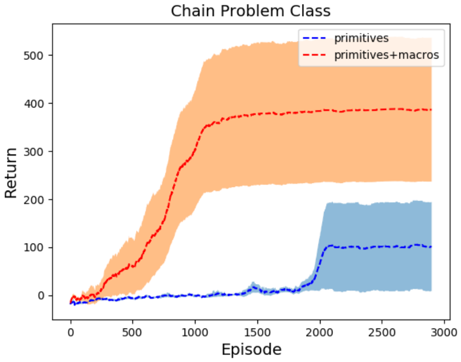


Figure 4: Comparison of mean learning curve over 20 randomly generated chains. The error bars indicate standard error.

Figure 4 depicts the mean reward accumulated by the agent during training over 20 different randomly generated chain tasks, after using only 4 tasks for training to generate candidate macros. The results show that, on average, the policy of an agent equipped only with primitive actions (shown in blue) converges to a sub-optimal behavior, since it hardly ever discovers the farthest end with the largest reward. As the action-set of the agent is augmented with the identified macros, the agent no longer only executes actions randomly, but rather they are guided by the macros identified for this type of problems.

Note that this does not mean that the agent is not able to represent an optimal policy using only primitive actions; in fact any policy that can be represented with macros can be represented with primitives. What these result show is that macros provide guidance to the agent which allow it to better explore the state space.

6.2 Maze Navigation Problem Class

The previous class of problems allowed us to asses the ability of the agent to reach an optimal policy with the identified macros, when having access to only primitive actions would fail. In this set of experiments, we extend our results to class of problem with a much larger state space and an action-set composed of four primitive actions. This experiment compares the performance of our method, when we know the true transition function and the true Q-values, to the performance of Eigen-Options and the Option-Critic. In this case, all methods were implemented in tabular form. In this experiment, to make a fair comparison in the setting for which the competing methods were designed, we allowed them to learn options defined in terms of the same transition graph where they were tested.

At the beginning of an episode, the agent is randomly placed in an initial state, in a randomly generated maze of size 60 × 60, and the objective is to reach a specific goal state. The agent receives a reward of -1 after executing an action and receives a reward of +100 upon reaching the goal state. The state is represented by the xy-position in the environment and the agent can execute four possible actions: move right, move left, move up or move down. We test robustness to stochastic environment by introducing noise to each executed action: after selecting an action, with probability 0.8 the agent executes the selected action and with probability 0.2 the agent executes any action at random. If the agent executes an action that would move it to a state that is blocked (an obstacle), the agent remains in the same state. The agent trained on 6 different tasks to generate candidate macros, and tested on 20 different tasks. To be able to reuse options for Eigen-Option and Option-Critic, the test tasks were defined by changing the goal location in one of the environments previously used for training. In this experiment we set $\delta = 2.0$.

The benefits of the identified macros can be seen empirically in Figure 5. The figures show as gray lines the paths taken by the agent (shown in red) when selecting actions from a uniform distribution during a period of 1000 steps in one sample environment. We consider this a period of pure exploration. The figure on the left shows that when the agent explores using only primitive actions, it densely visits a small region of the state space but is unlikely to reach states that are far away. The figure on the right, on the other hand, shows that with the identified macros the agent is able to explore a much larger area of the state space. This latter approach allows the agent to learn at a global scale during early stages of training.

Figure 6 shows the mean performance and standard error of the agent in the 20 testing tasks from this problem class, which is in accordance with the intuition on exploration described above.

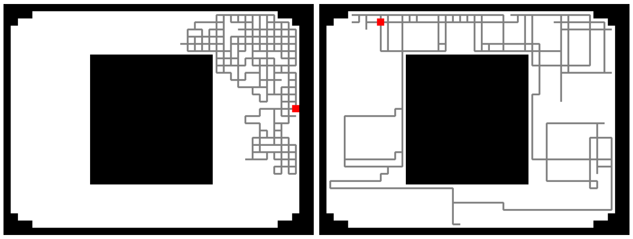


Figure 5: Trajectories obtained from pure exploration after 1000 steps using action-sets $\mathcal{A}$ (left) and $\mathcal{A}_{\mathcal{M}'}$ (right).

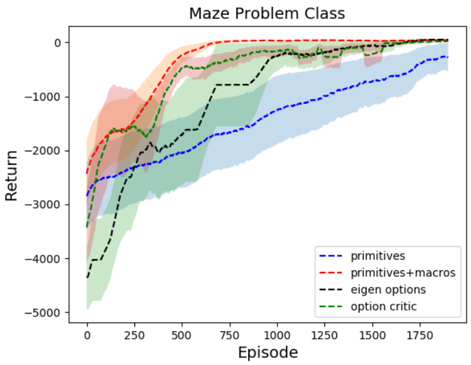


Figure 6: Mean performance on 20 testing tasks on maze navigation problem class. Macros evaluated using true Q function and transition function.

Just as it was the case in the previous experiment, extending the action-set with the identified macros led to a large performance improvement over using only primitives. In this scenario, our framework performs slightly better to Eigen-Option and Option-Critic, however, it is worth noting that the competing methods required to learn environment specific options, making them highly sample inefficient.

In the next sections we compare our method to the competitors in the more interesting case where the agent needs to identify options that are reusable across many tasks that might differ not only in terms of their reward functions, but also in terms of their transition graph; in that case, our method’s advantages are made even clearer.

6.3 Scaling up Results to Large State Spaces

In the previous experiments, we were able to precisely calculate the W-values of each macros since the transition probabilities and true Q-values for primitives were known. This section presents an empirical demonstration that these results hold when we use function approximation to estimate Q and approximate the transition model from data. In all of these experiments, the agent collected (s, a, s') transitions during training, and they were used to fit a model to estimate the transition probabilities $P(s, a, s')$. It is worth noticing that in the following three problems the transition graphs are not maintained across tasks and the competing methods are not well suited/capable of dealing with this more general learning setting:

Problem Class	Primitives	Primitives+Macro	Eigen-Options	Option-Critic
Maze Navigation (approximate)	-2355.44 ± 640.54	$\mathbf{-2016.50 \pm 643.71}$	-3444.06 ± 459.68	-2788.52 ± 696.03
Animat	-909.77 ± 199.53	$\mathbf{-752.89 \pm 188.59}$	-1432.46 ± 64.72	-1955.47 ± 41.22
Lunar Lander	-314.03 ± 44.09	$\mathbf{-246.89 \pm 28.99}$	-266.43 ± 5.22	-265.51 ± 7.42

Table 1: Average performance on test tasks with large state spaces.

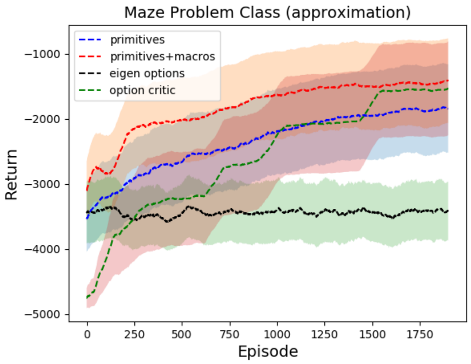


Figure 7: Mean performance on 20 testing tasks on maze navigation. Macros evaluated using approximate Q function and transition function.

(1) Maze Navigation Problem Class: We revisited the maze navigation problem class, this time approximating the true Q-values for primitives for the training tasks using DQN and randomly generating training and testing environments. Since in this problem the state-space is discrete, the transition function can be easily modeled by collecting samples of (s, a, s') tuples and estimating $P(s'|s, a)$ by looking at the frequency count.

Figure 7 shows the mean learning curve over 20 randomly sampled environments contrasting the reward accumulated by an agent using only primitives (blue), using the the identified extended action-set (red), using Eigen-Options (black) and Option-Critic (green). This case highlights how general the identified macros are compared to the competing methods. The Eigen-Option approach fails to generalize to new domains and actually prevents the agent from learning an optimal policy. The option critic approach shows an interesting behavior, improving performance in a step-wise fashion. This can be explained by noticing that, at the beginning of an episode, the options are poorly suited for the new problem. However, as the policy improves and options are updated, they cause large improvements in the reward obtained. Our method, in this case, obtains generally useful macros which are agnostic to the transition graph, leading to a significant overall improvement.

(2) Animat Problem Class: This type of problem was first introduced by Thomas and Barto [14] and presents the challenge of having a much larger action space than the previous problems. In this problem class, the agent is a circular creature that lives in a continuous state space. It has 8 independent actuators, angled around it in increments of 45 degrees. Each actuator can be either on or off at each time step, so the action set is $\{0, 1\}^8$, for a total of *256* actions. When an actuator is on, it produces a small force in the direction that it is pointing. The agent is tasked with moving to a goal location; it receives a reward of -1 at each time-step and a reward of $+100$ at the goal state. The different variations of the tasks correspond to randomized start and goal positions in different environments. The agent moves according to the following mechanics: let (x_t, y_t) define the state of the agent at time t and d be the total displacement given by actuator β with angle θ_β. The displacement of the agent for a set of active actuators, $\mathcal{B}$, is given by, $(\Delta_x, \Delta_y) = \sum_{\beta \in \mathcal{B}} (d \cos(\theta_\beta), d \sin(\theta_\beta))$. After taking an action, the new state is perturbed by 0-mean unit variance Gaussian noise. Notice that certain actuator combinations will not help the agent reach a goal; for example, if only actuators at angles 0 and 180 are activated, that action would leave the agent in the sample position where it previously was (ignoring noise effects). We used 4 training tasks and tested 10 task variations corresponding to different environments with distinct transition graphs.

(3) Lunar Lander Problem Class: The implementation for this problem class was obtained from OpenAI Gym [2]. The agent is tasked with landing a rocket in a specific platform and it has 4 actions at its disposal. Thrust left, right, up or do nothing. We modified the original code to obtain variations of the problem class by changing the landing location, terrain and the thrust force of the rocket. We use 4 tasks for training and 8 variations for testing.

Performance results (according to Equation 1) for each method, in the three domains described above, are reported in Table 1. Our method, as before, outperforms all competitors.

7 CONCLUSION

In the paper, we introduced a general framework for identifying reusable macros. By analyzing the trajectories of optimal or near-optimal policies for tasks drawn from a common class of related problems, we can identify behaviors that recur, that are associated high rewards, and that are diverse. These allow the agent to more efficiently explore the state space and acquire optimal policies for novel tasks more rapidly—even when the transition dynamics of the problem might change. We introduced a new approach to determine the value of a macro in closed-form and introduced a novel way of determining the similarity between macros, so as to ensure diversity and to control the size of the augmented action space. Our empirical results show that our approach outperforms state-of-the-art methods for option discovery when the transition graph is maintained across tasks, and that it performs those competitors significantly when this assumption relaxed and tasks are allowed to differ not only in their reward structure but also in terms of their transition dynamics.

ACKNOWLEDGMENTS

This work was partially supported by FAPERGS under grant no. 17/2551-000

REFERENCES

[1] Pierre-Luc Bacon, Jean Harb, and Doina Precup. 2017. The Option-Critic Architecture. In *AAAI*.

[2] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. OpenAI Gym. (2016). http://arxiv.org/abs/1606.01540 cite arxiv:1606.01540.

[3] Ramnandan Krishnamurthy, Aravind S. Lakshminarayanan, Peeyush Kumar, and Balaraman Ravindran. 2016. Hierarchical Reinforcement Learning using Spatio-Temporal Abstractions and Deep Neural Networks. *CoRR* (2016).

[4] Ning Liu, Zhe Li, Jielong Xu, Zhiyuan Xu, Sheng Lin, Qinru Qiu, Jian Tang, and Yanzhi Wang. 2017. A Hierarchical Framework of Cloud Resource Allocation and Power Management Using Deep Reinforcement Learning. *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)* (2017), 372–382.

[5] Sridhar Mahadevan. 2005. Proto-value Functions: Developmental Reinforcement Learning. In *Proceedings of the 22nd International Conference on Machine Learning (ICML-2005)*. ACM, 553–560.

[6] Michael Bowling Marlos C. Machado, Marc G. Bellemare. 2017. A Laplacian Framework for Option Discovery in Reinforcement Learning. *CoRR* (2017).

[7] Amy McGovern and Andrew G. Barto. 2001. Automatic Discovery of Subgoals in Reinforcement Learning Using Diverse Density. In *Proceedings of the Eighteenth International Conference on Machine Learning (ICML '01)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 361–368. http://dl.acm.org/citation.cfm?id=645530.655681

[8] A. McGovern and R. Sutton. 1998. *Macro Actions in Reinforcement Learning: An Empirical Analysis*. Technical Report. University of Massachusetts - Amherst, Massachusetts, USA.

[9] Ishai Menache, Shie Mannor, and Nahum Shimkin. 2002. Q-Cut - Dynamic Discovery of Sub-goals in Reinforcement Learning. In *Proceedings of the 13th European Conference on Machine Learning (ECML '02)*. Springer-Verlag, London, UK, UK, 295–306. http://dl.acm.org/citation.cfm?id=645329.650060

[10] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (Feb. 2015), 529–533. http://dx.doi.org/10.1038/nature14236

[11] Jette Randl. 1998. Learning Macro-Actions in Reinforcement Learning. In *NIPS*.

[12] Richard S. Sutton and Andrew G. Barto. 2018. *Introduction to Reinforcement Learning* (2nd ed.). MIT Press, Cambridge, MA, USA.

[13] Richard S. Sutton, Doina Precup, and Satinder P. Singh. 1999. Between MDPs and Semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning. *Artificial Intelligence* 112, 1-2 (1999), 181–211.

[14] Philip S. Thomas and Andrew G. Barto. 2011. Conjugate Markov Decision Processes. In *Proceedings of the 28th International Conference on International Conference on Machine Learning (ICML'11)*. Omnipress, USA, 137–144. http://dl.acm.org/citation.cfm?id=3104482.3104500

[15] Christopher J. C. H. Watkins and Peter Dayan. 1992. Q-learning. In *Machine Learning*. 279–292.

[16] T. A. Welch. 1984. A Technique for High-Performance Data Compression. *Computer* 17, 6 (June 1984), 8–19. https://doi.org/10.1109/MC.1984.1659158

[17] Steven D. Whitehead. 1991. Complexity and Cooperation in Q-Learning. In *Proceedings of the Eighth International Workshop (ML91), Northwestern University, Evanston, Illinois, USA*. 363–367.