A Framework for Macro Discovery for Efficient State-Set Exploration

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

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

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

Abstract

In this paper we consider the problem of how a reinforcement learning agent tasked with solving a set of Markov decision processes can use knowledge acquired early in its lifetime to improve its ability to learn how to solve novel, but related, tasks. Specifically, we propose a three step framework in which an agent **1)** generates a set of candidate open-loop macros, **2)** evaluates the value of each macro, and **3)** selects a diverse subset of macros. Our experiments show that extending the original action-set of the agent with the identified macros leads to significant improvement in learning an optimal policy in unseen MDPs.

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 that they already possess, and their focus would be on learning new soccer skills. In other words, the person no longer learns new behaviors by experimenting with low-level behavior like they used to do as an infant, and simply bootstrap the knowledge acquired early on in their lives. This person, in other words, is able to use these already-acquired higher level skills to more efficiently explore the consequences of his actions when facing new tasks.

In the RL literature, these 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 speed up learning. Carefully constructed macros have been shown to improve learning by allowing an agent to reach distant areas of the state space quickly during training. However, if they are not appropriate for the problem at hand, they may substantially degrade learning (McGovern and Sutton 1998). 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 make this definition more precise in the following section.

We propose a framework that, after the agent has learn an optimal policy for a few initial tasks, identifies macros that would help in learning to solve the remaining tasks. In our approach, after an agent learns optimal policies for a set of tasks, trajectories from these policies are sampled to generate, evaluate, and select effective macros for the specific type of problem.

In this paper we make the following contributions:

- Present a general framework for identifying macro actions appropriate to the problem class.
- Introduce the notion of the value of a macro, called the W-value.
- Introduce a new way of evaluating distances between macros.

Related Work

A critical component that determines the performance of an agent when learning to solve a new task is their ability to efficiently explore the state space. Typically, exploration is done through random walks, although it is known that his strategy scales poorly as the size of the state space increases, (Whitehead 1991).

A better approaches to exploring the state space, which has become increasingly popular in the last years, is through *options* or *macros*, (Sutton, Precup, and Singh 1999; **?**), which define temporally extended actions. Options are subpolicies that the agent can invoke in any state $s \in \mathcal{I}$ and 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 close-loop counterpart, and they 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 executing actions, and developing methods for identifying useful skills or macros became an active area of research commonly referred to as *option discovery*.

[1] Different work have slight different definitions for macros, in this work we defined them as open look finite length sequence of actions.

One approach for option discovery is to identify important states in the transition graph and learn policies that would take the area to that state. The work by McGovern and Barto (2001) proposes splitting trajectories into successful and unsuccessful trajectories based on whether they were able to reach a pre-determined goal state. These trajectories are then analyzed to identify *bottleneck states*, and options can be obtained by learning policies that cause the agent to reach those states. A more recent approach based on the transition graph is the one presented by Marlos C. Machado (2017). The authors propose using proto-value functions (Mahadevan 2005) to identify states of interest and extract options by learning an optimal policy that allows the agent to reach each of those states.

Many other commonly used approaches to option discovery do not rely on finding bottleneck states (Krishnamurthy et al. 2016; **?**; **?**) however, they all share the same limitation which limits how reusable they are: they all 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 on 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 the problem class

Background

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 takes 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 $\gamma \in [0, 1]$.

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^c. 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 $\mathcal{C}$, the *problem class*, to be the set of all tasks or problems c that an agent may face, where $c = (\mathcal{S}, \mathcal{A}, P_c, R_c, d_0^c)$. In particular, note that we define $\mathcal{C}$ such that all $c \in \mathcal{C}$ 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, if $\mathcal{C}$ correspond to problems related to navigating a maze to reach a goal location, each task $c \in \mathcal{C}$ could refer to navigating a specific maze (with, e.g., particular wall configurations) or a same maze but with different goal locations.

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, which we call the *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: $Q_c^\pi(s, a) = \mathbf{E}\left[\sum_t^T \gamma^t R_t | S_t = s, A_t = a\right]$. A useful property of the Q function is the *Bellman equation* which states that:

$$Q_c^\pi(s, a) = \mathbf{E}\ [R_t + \gamma Q(S_{t+1}, A_{t+1}) | S_t = s, A_t = a]$$

In other words, 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 at 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 to be $\mathcal{A}_\mathcal{M} = \mathcal{A} \cup \mathcal{M}$. 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) is to find the set of macros with maximum expected performance on the problem class.

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 $\alpha =$ "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 occur frequently in the trajectories from where they originated.

Problem Statement

We consider the setting where an agent is required to solve a set of tasks $c \in \mathcal{C}$. We assume that when solving a particular task, an agent can interact with it for I episodes, and assume that there is a distribution $d_{\mathcal{C}}$ from which tasks are sampled and presented to the agent. After the agent has trained on a subset $\mathcal{C}_{train} \subset \mathcal{C}$, it is allowed to identify a set of macros to be used for improving learning in any new task. We will consider the new tasks to be part of a test set $\mathcal{C}_{test} \subset \mathcal{C}$, where $\mathcal{C}_{train} \cap \mathcal{C}_{test} = \emptyset$.

We define the performance of a set of macros $\mathcal{M}$ in a particular task c to be $\rho(\mathcal{M}, c) = \mathbf{E}\left[\sum_{i=0}^{I}\sum_{t=0}^{T}\gamma^t R_t^i \middle| \mathcal{A}_{\mathcal{M}}, c\right]$, 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 much they improve learning for an agent during training.

Let C be a random variable denoting a sample task from $d_{\mathcal{C}}$. Our goal is to find one (of possible many) optimal set of macros $\mathcal{M}^*$ according to the following criterion:

$$\mathcal{M}^* \in \arg\max_{\mathcal{M}} \quad \mathbf{E}\left[\rho(\mathcal{M}, C)\right]. \tag{1}$$

Unfortunately, the domain of the objective in Eq.1 is discrete, making the objective non-differentiable, and thus, difficult to optimize.

We argue that macros identified via compression techniques are highly reusable and represent recurring behaviors in the task distribution that will allow an agent to speed up learning. 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.

A Heuristic Approach for Approximating $\mathcal{M}^*$

The proposed framework can be summarized by the diagram shown in Figure 1. Given a set of training tasks $\mathcal{C}_{train} \subset \mathcal{C}$, the agent learns an optimal policy π_c^* for each $c \in \mathcal{C}_{train}$ and samples n trajectories from each policy π_c^* in 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.

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 look at 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 $t_a = (a, b, c, d)$ 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 t_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 also contains macros. [2]

Following this intuition, the sampled action-trajectories are compressed and the symbols in the final codebook define a set of candidate macros, $\mathcal{M}$, to be evaluated. In this work, we selected LZW (Welch 1984) as a compression algorithms because of its simplicty and efficiency in populating the codebook.

Macro Evaluation - The Value of a Macro

At this stage we have generated set of candidate macros $\mathcal{M}$, but do not have a sense of how useful they are in general in relation to each other. Knowing this could help us determine which macros are preferable for the problem class. One way of evaluating them, would be to re-train the agent with each macro individually and assessing the improvement in learning over a set of tasks. However, this would 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 $\mathcal{C}$ by the W-function defined as:

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

where the expectation is defined over both tasks C and states S.

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 the process previously described generates a large number of candidate macros, learning the true value of equation 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 $a \in \mathcal{A}$ and π is greedy with respect to Q. It can be shown that the Q-value of a macro m at state s for a task c under policy π is given by:

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

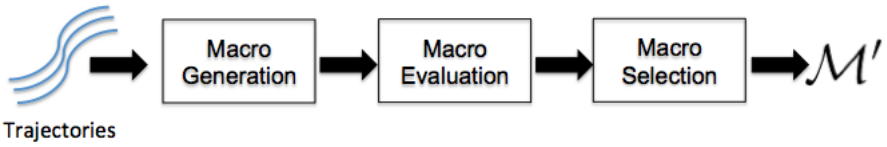


Figure 1: Diagram depicting proposed framework.

$$Q_c^\pi(s,m) = \sum_{k=1}^{l} \Bigg[\sum_{s^{(1)}\in\mathcal{S}} \cdots \sum_{s^{(l_m)}\in\mathcal{S}} \Bigg(Q(s^{(k-1)}, m_{(k)}) - \gamma^{k-1} \sum_{a'\in\mathcal{A}} \pi(a', s^{(k)}) Q_c^\pi(s^{(k)}, a') \Bigg) \times \frac{\prod_{i=1}^{l} P_c(s^{(i-1)}, m_{(i)}, s^{(i)})}{P_c(s^{(k-1)}, m_{(k)}, s^{(k)})} \Bigg] + \gamma \sum_{a'\in\mathcal{A}} \pi(a', s^{(l)}) Q_c^\pi(s^{(l)}, a') \Big) \tag{3}$$

Notice that this expression is given in terms of the Q-values of primitives, and consequently, the W-values can be calculated in close form [3].

This is a reasonable assumption when we consider that algorithms like Q-learning, (Watkins and Dayan 1992), and DQN, (Mnih et al. 2015), approximate the true Q-value and behave greedily with respect to Q. If the agent uses these techniques to learn an optimal policy for the tasks in $\mathcal{C}_{train}$ it will have a reasonable approximation to Q readily available.

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, 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.

Macro Selection - Encouraging Macro Diversity

Having defined a value for a macro allow us to estimate 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 to 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 Marlos C. Machado (2017).

[3]The proof for this derivation will be available at `http://www-all.cs.umass.edu`.

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 l_m denote the length of macro m, S_0 a random variable denoting a state where m is executed and S_l the state where m finishes execution. Furthermore, let $S'_m = S_l - S_0$ denote a random variable describing the change in state caused by the execution of m and p_m the corresponding distribution for S'_m. We refer to p_m as the *end-state distribution*.

In this paper, we define the distance between two macros $m1$ and $m2$ to be the KL divergence between p_{m1} and p_{m2}, that is:

$$D_{KL}(p_{m1}||p_{m2}) = -\sum_{s\in\mathcal{S}} p_{m1}(s) \ \log\left(\frac{p_{m2}(s)}{p_{m1}(s)}\right)$$

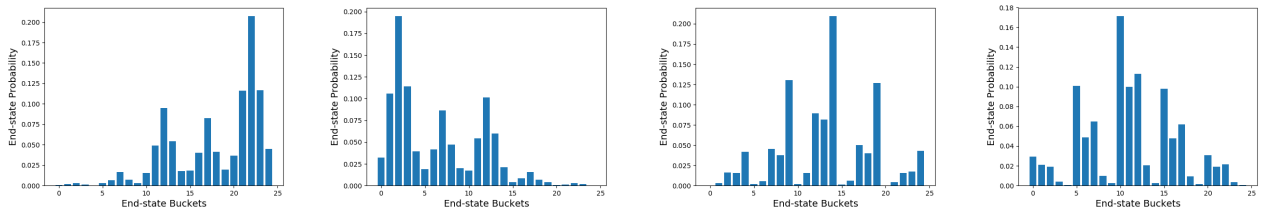
In the case of continuous state spaces, we discretize the distribution into appropriately sized bins.

Figure 2 shows the empirical distribution for the change of state 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 very macros reflect their similarity (or differences) in the effect that they have in the distribution of state transitions. 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-value value defines a preference criterion by which macros can be selected.

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.



(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.

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 first two experiments, the agent was trained using Q-learning with tabular representation and in the remaining experiments the policy was trained using DQN Mnih et al. (2015). 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. All figures depict mean performance and standard error over a set of testing tasks.

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.

(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.

This problem class demonstrates that oftentimes the policy of the agent converges to the nearest end if exploration is done randomly, since it is unlikely that random exploration will reach the further end of the chain. However, if the agent were to have macros well suited for this type of problem, it would be to reach both ends of the chain and learn the correct optimal policy for any particular task.

Figure 4 depicts the mean reward accumulated by the agent during training over 20 different randomly generated chain tasks, after using 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 never 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.

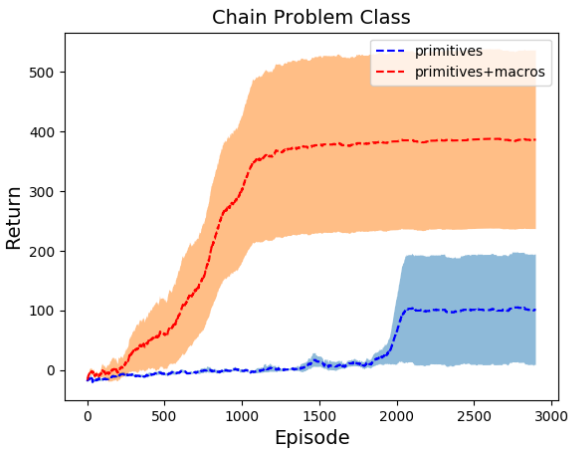


Figure 4: Comparison of mean learning curve over 20 randomly generated chains.

Maze Navigation Problem Class

The previous class of problem 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.

At the beginning of an episode, the agent is randomly place in an initial state, in a randomly generate maze, 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. In this experiment we set $\delta = 2.0$ and contrast our method with eigen-options Marlos C. Machado (2017).

The benefits of the identified macros can be seen empirically in Figure 5. The figures shows as a gray line the path taken by the agent (shown in red) selecting action 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.

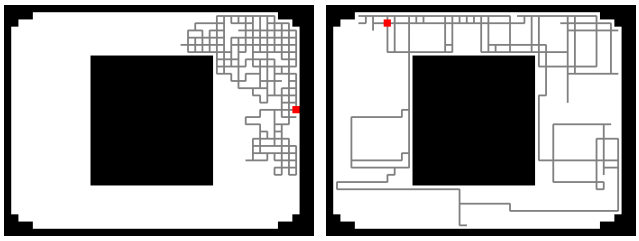


Figure 5: Trajectories obtained from pure exploration after 1000 steps using action-sets $\mathcal{A}$ (left) and $\mathcal{A}_{\mathcal{M}'}$ (right).

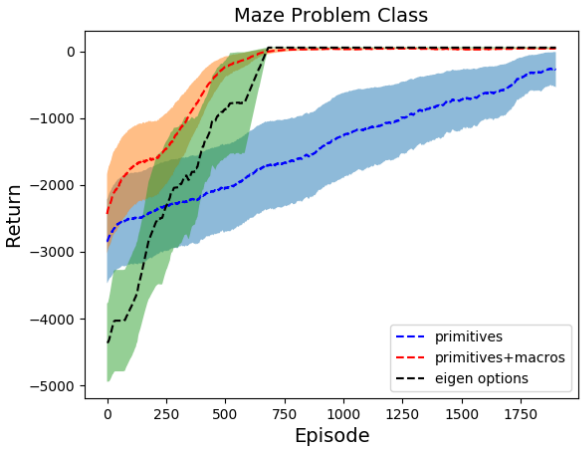


Figure 6: Mean performance on 20 testing tasks on maze navigation problem class. Macros evaluated using true Q function and transition function.

Figure **??** shows the mean performance of the agent in the 20 testing tasks from this problem class, which is in accordance with the intuition on exploration described above.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 similarly to eigen options, however it does not incur in the expensive process of solving several MDPs for each eigen vector.

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 demonstrates empirically, that these results hold when we use function approximation to estimate Q and approximate the transition model from data.

For all these experiments, the agent collected (s, a, s') transitions during training, and they were used to fit a model to estimate the transition probabilities $P(s, a, s')$. The results of Equation 1 for each method are reported in Table 1 [4].

[4]For all problems, we attempted to contrast our results with

Problem Class	Primitives	Primitives+Macro	Eigen-Options
Maze Navigation (approximate)	-2355.44 ± 640.54	$\mathbf{-2016.50 \pm 643.71}$	-3444.06 ± 459.68
Animat	-909.77 ± 199.53	$\mathbf{-752.89 \pm 188.59}$	—
Lunar Lander	-442.31 ± 23.38	$\mathbf{-354.85 \pm 23.02}$	—

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

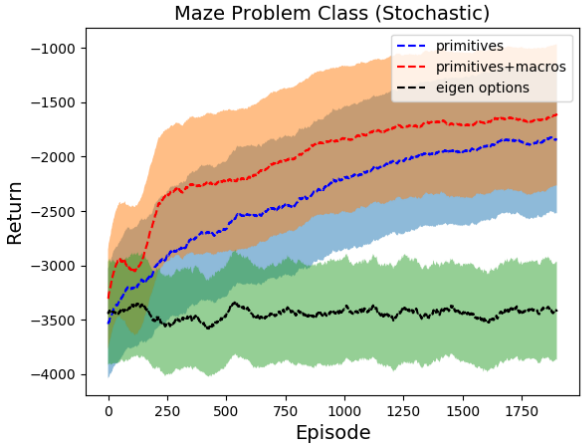


Figure 7: Mean performance on 20 testing tasks on maze navigation. Macros evaluated using approximate Q function and transition function.

Maze Navigation Problem Class: We revisited the maze navigation problem class, this time estimating the true Q-values for primitives for the training tasks using DQN. Since in this problem the state-space is discrete, the transition function can be easilly 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 agent using only primitives (blue) and agent equipped with the identified macros (red). In this case, we also highlight one of the drawbacks of eigen options which is its dependency on the transition graph. Just as we do with our approach, we optioned eigen options from a set of tasks with a different transition graph from the set of test tasks. As expected, the options actually hinder learning since they do not correspond to the test environment. Our method, on the other hand, obtains generally useful macros which are agnostic to the transition graph.

Animat Problem Class: This type of problem was first introduced by Thomas and Barto (2011) 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). The variations in task correspond to different environments with distinct transition graphs.

Lunar Lander Problem Class: The implementation for this problem class was obtained from OpenAI Gym (Brockman et al. 2016). 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. Variations of the problem class were obtained by changing the landing location and the thrust force.

Discussion and Future Work

In the paper, we presented a general framework for identifying reusable macros. By analyzing the trajectories of well performing policies, we can identify recurrent behavior that is associated with high reward and allows the agent to better explore the state space. We presented a new approach to determine the value of an action or a macro and introduce a novel way for determining the similarity between macros. As future work, we intend to study how to extend our approach to continuous action spaces.

Acknowledgement

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

References

Brockman, G.; Cheung, V.; Pettersson, L.; Schneider, J.; Schulman, J.; Tang, J.; and Zaremba, W. 2016. Openai gym. cite arxiv:1606.01540.

eigen options, however in cases of large transition graphs, where tabular form is not an option, we could not find a clear indication of how to identify the correct eigen options in the original paper. For that reason, there is no comparison in the case of animat and lunar lander.

Krishnamurthy, R.; Lakshminarayanan, A. S.; Kumar, P.; and Ravindran, B. 2016. Hierarchical reinforcement learning using spatio-temporal abstractions and deep neural networks. *CoRR*.

Mahadevan, S. 2005. Proto-value functions: Developmental reinforcement learning. In *Proceedings of the 22nd International Conference on Machine Learning (ICML-2005)*, 553–560. ACM.

Marlos C. Machado, Marc G. Bellemare, M. B. 2017. A Laplacian Framework for Option Discovery in Reinforcement Learning. *CoRR*.

McGovern, A., and Barto, A. G. 2001. Automatic discovery of subgoals in reinforcement learning using diverse density. In *Proceedings of the Eighteenth International Conference on Machine Learning*, ICML '01, 361–368. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

McGovern, A., and Sutton, R. 1998. Macro actions in reinforcement learning: An empirical analysis. Technical report, University of Massachusetts - Amherst, Massachusetts, USA.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; Petersen, S.; Beattie, C.; Sadik, A.; Antonoglou, I.; King, H.; Kumaran, D.; Wierstra, D.; Legg, S.; and Hassabis, D. 2015. Human-level control through deep reinforcement learning. *Nature* 518(7540):529–533.

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

Thomas, P. S., and Barto, A. G. 2011. Conjugate markov decision processes. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, 137–144. USA: Omnipress.

Watkins, C. J. C. H., and Dayan, P. 1992. Q-learning. In *Machine Learning*, 279–292.

Welch, T. A. 1984. A technique for high-performance data compression. *Computer* 17(6):8–19.

Whitehead, S. D. 1991. Complexity and cooperation in q-learning. In *Proceedings of the Eighth International Workshop (ML91), Northwestern University, Evanston, Illinois, USA*, 363–367.