July 2020

Improving Reinforcement Learning Techniques by Leveraging Prior Experience

Francisco M. Garcia

IMPROVING REINFORCEMENT LEARNING TECHNIQUES BY LEVERAGING PRIOR EXPERIENCE

A Dissertation Presented

by

FRANCISCO M. GARCIA

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2020

College of Information and Computer Sciences

IMPROVING REINFORCEMENT LEARNING TECHNIQUES BY LEVERAGING PRIOR EXPERIENCE

A Dissertation Presented

by

FRANCISCO M. GARCIA

Approved as to style and content by:

Philip S. Thomas, Chair

Erik Learned-Miller, Member

Shlomo Zilberstein, Member

Melinda D. Dyar, Outside Member

James Allan, Department Chair
College of Information and Computer Sciences

ABSTRACT

IMPROVING REINFORCEMENT LEARNING TECHNIQUES BY LEVERAGING PRIOR EXPERIENCE

MAY 2020

FRANCISCO M. GARCIA

B.Sc., UNIVERSITY OF THE SCIENCES IN PHILADELPHIA

M.Sc., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Philip S. Thomas

In this dissertation we develop techniques to leverage prior knowledge for improving the learning speed of existing reinforcement learning (RL) algorithms. RL systems can be expensive to train, which limits its applicability when a large number of agents need to be trained to solve a large number of tasks; a situation that often occurs in industry and is often ignored in the RL literature. In this thesis, we develop three methods to leverage the experience obtained from solving a small number of tasks to improve an agent's ability to learn on new tasks the agent might face in the future.

First, we propose using compression algorithms to identify *macros* that are likely to be generated by an optimal policy. Because compression techniques identify sequences that occur frequently, they can be used to identify action patterns that are often required to solve a task.

Second, we address some of the limitations present in the first method by formalizing an optimization problem that allows an agent to learn a set of *options* that are appropriate for the tasks. Specifically, we propose an analogous objective to compression by minimizing the number of decisions an agent has to make to generate the observed optimal behavior. This technique also addresses a question that is often ignored in the option literature: how many options are needed?

Finally, we show that prior experience can also be leveraged to address the *exploration-exploitation dilemma*; a central problem in RL. We propose a framework in which a small number of tasks are used to train a meta-agent on *how* to explore. After being trained, any agent facing a new task can query the meta-agent on what action it should take for exploration.

We show empirically that, when facing a large number of tasks, leveraging prior experience can be an effective way of improving existing reinforcement learning techniques. At present, the application of RL in the industry setting remains rather limited. One of the reasons being how costly and time consuming training large scale systems can be. We hope this work provides some guidance for future work, and that it inspires new research in exploiting existing knowledge to make RL a practical alternative to tackle large scale real-world problems.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The concept of reinforcement learning (RL) as a method for training control systems can be traced back to the origins of artificial intelligence (AI) as a field. Although perhaps a matter of contention, many researchers agree that the Summer Research Project on Artificial Intelligence, held at Dartmouth College in 1956, was the starting point of AI. To make this event possible, several researchers submitted a proposal stating their intended line of study; in it Marvin Minsky writes:

> *The machine is provided with input and output channels and an internal means of providing varied output responses to inputs in such a way that the machine may be "trained" by a "trial and error" process...when placed in an appropriate environment and given a criterion for "success" or "failure" can be trained to exhibit "goal-seeking" behavior...*

This is the same principle underlying RL as a field (Sutton and Barto, 1998). For many years, RL was mostly considered a matter of academic curiosity; however, recent technological advances have led to a surge of interest in RL as a potential solution to many practical applications. Some examples of these include ad placement (Theocharous et al., 2015), autonomous navigation (Kuderer et al., 2015) and insulin pump control (Thomas et al., 2017), among others.

RL focuses on the problem of learning (by trial an error) how a decision maker, called an *agent*, should behave in order to achieve a desired result. Traditionally, the goal of RL techniques is to learn a policy (a mechanism by which an agent makes decisions) to solve a specific problem assuming that no prior experience is available. The majority of RL research focuses on this setting, where the agent learns from *scratch* (Watkins and Dayan, 1992; Silver et al., 2014; Mnih et al., 2015; Schulman et al., 2017). In some circumstances this might be a reasonable assumption, but there are occasions where an agent may need to solve a large number of tasks or where expert demonstrations might be available. In these situations, ignoring the information contained in demonstrations or the information obtained from solving earlier tasks would be wasteful.

The use of prior experience has been studied in the RL literature to some extent under the settings of *transfer learning* (Taylor and Stone, 2009), *curriculum learning* (Bengio et al., 2009) or *learning from demonstration* (Argall et al., 2009). In the context of RL, transfer learning is generally concerned with how the behavior of an agent trained on a previous task can be adapted to a new problem. For example, Bou-Ammar et al. (2015) showed how control policies for UAVs learned in simulation can be adapted using transfer learning techniques to work well for real world altitude control of quadrotors. Curriculum learning, on the other hand, refers to a learning framework in which an agent learns how to solve a series of tasks of incremental difficulty. This strategy is based on insights stemming from the developmental psychology literature (Elman, 1993); it has been noted that initially solving simple tasks allows humans to eventually solve problems that they were unable to solve before. This same principle applies to RL agents. Lastly, learning from demonstration has been shown to be a viable approach for applying RL to humanoid robots. A robot can be shown a few examples demonstrating the behavior needed to complete a task, learn how to imitate those behaviors by using PD controllers, and apply RL techniques to adapt the learned behaviors to new tasks (Mülling et al., 2010).

Even though leveraging prior experience to speed up learning for novel tasks has been considered by others researchers to some extent, there is no consensus on how to best use prior experience. As we will discuss in the next chapter, a number of different approaches have been tried with some success: directly adapting policies to new tasks, learning a general initial policy that could be fine-tuned quickly, learning a general value function (defined in the next chapter) that would allow for fast learning, etc.

1.1 Contributions

In this thesis, we develop techniques to improve an agent's learning abilities in a set of related tasks by leveraging existing experience in two ways: discovering *temporal abstractions* and learning how to *explore*. Throughout this thesis we make the following contributions:

1. We develop a compression-based approach for identifying recurring action patterns that form the basis of optimal behavior. These action patterns are given by fixed length sequences of actions known as *macros* (McGovern and Sutton, 1998). Research in animal behavior suggests that, in nature, the complex behavior needed to achieve a difficult goal is obtained through simple building blocks that are often reused (i Cancho and Lusseau, 2009), and it has been argued that this is a consequence of compression in the information-theoretical sense (i Cancho et al., 2013).

2. We extend the ideas that we developed in the first contribution to the more general formulation of *options* (Precup, 2000). Options are a generalization of macros in that, instead of executing a fixed sequence of actions, they maintain an *option policy*, that governs the behavior of the agent while the option is executing, and a *termination function*, that determines when the option terminates. Because options are able to terminate when they deem necessary and are able to select the action that seems appropriate given the current context, they

are a more flexible alternative to macros. Based on insights obtained from compression, we propose an objective, the optimization of which results in options that are capable of generating optimal behavior.

3. Finally, we present a *meta-learning* framework for learning how an agent should *explore* when solving a novel task. When an agent first faces a new task, it does not know what behavior would lead to the desired result; thus the agent must choose between acting for the sake of learning more about the task (exploration) and acting according to what it believes to be best (exploitation). This trade-off is a central problem in RL and is known as the *exploration-exploitation dilemma*. The techniques introduced in Chapters 3 and 4 *indirectly* address this problem by biasing the behavior of the agent towards those action patterns that were found in previous optimal policies. Our last contribution *directly* addresses this trade-off by exploiting the experience obtained from solving a small number of tasks and learning a policy tailored for exploration that the agent can use whenever it faces a new task.

1.2 Layout

The remainder of this dissertation is structured as follows:

- Chapter 2 (Background and Related Work): This chapter introduces concepts and notations that are relevant to all chapters, and discusses research that is relevant to the overarching topic of leveraging prior knowledge in RL.

- Chapter 3 (A Heuristic Approach to Macro Discovery through Compression Techniques): This chapter argues that compression provides a means for identifying reusable behavioral patterns and presents a framework for learning macro actions based on compression techniques.

- Chapter 4 (A Probabilistic Approach to Option Discovery): This chapter builds on the insights obtained in the previous chapter and extends these ideas to derive an optimization problem whose solution results in a set of reusable options.
- Chapter 5 (A Meta-MDP Formulation to Improve Exploration): This chapter introduces a meta-learning framework to learn an exploration strategy that allows an agent to explore efficiently in a distribution of related tasks.
- Chapter 6 (Conclusion and Future Research Directions): This chapter concludes our work by summarizing the contributions of this thesis and proposes avenues for future research.

CHAPTER 2

BACKGROUND AND RELATED WORK

This chapter presents background and an overview of existing research relevant to the overarching topic of leveraging prior knowledge in RL. We present standard notation used throughout this thesis and, in addition, each chapter contains background and discusses related work tailored to its content.

2.1 Notation

We standardize notation in this dissertation as follows. Sets are denoted by calligraphic capital letters, e.g., $\mathcal{X}$, and elements of the set by lowercase letters, e.g., $x \in \mathcal{X}$. We use brackets when defining a set and parentheses to denote a sequence. For example, $\mathcal{X} = \{x, x'\}$ refers to a set composed of elements x and x', whereas $X = (x, x')$ refers to a sequence beginning with the element x followed by the element x'. We also denote as $(x_i)_{i=1}^n$ the sequence $(x_1, \ldots, x_n)$, where x_i refers to the i^{th} element in the sequence.

Random variables are denoted by capital letters, e.g., X, and scalar values by lowercase letters, e.g., x. Lastly, in some cases equations might be too long, and must therefore be split into multiple lines; in those circumstances we use $\times$ to denote scalar multiplication being split across multiple lines.

2.2 Markov Decision Processes

This dissertation focuses on sequential decision making processes; that is, processes where an agent needs to optimize the sequence of decisions it makes with respect to the environment with which it interacts. In this thesis, we formulate these problems as Markov decision processes (MDPs): a mathematical framework for modeling decision making where outcomes are, in part, the result of an external source of stochasticity and, in part, the results of the actions taken by a decision maker.

The diagram in Figure 2.1 depicts how an agent interacts with its environment in the MDP framework. The agent chooses an action a (based on the current state of the environment) and interacts with the environment by executing a. After the action is executed, the environment transitions to a new state, s, and the agent receives a reward, r.

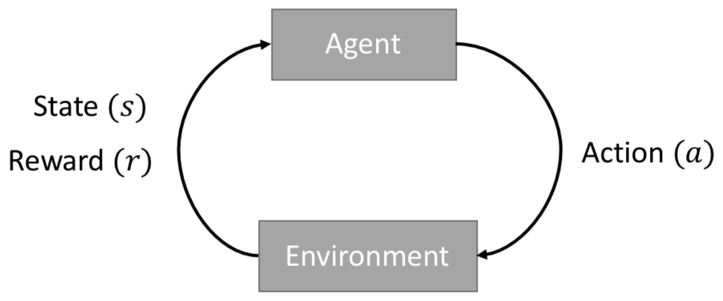


Figure 2.1: Diagram of agent-environment interaction in the MDP framework.

We use t to denote the time-step since the beginning of the process, such that $t = 0$ refers to the initial time-step. The behavior of an MDP can be described as follows: at time-step t the environment is in some state S_t, the agent takes an action A_t, receives a reward R_t, and the environment transitions to a new state S_{t+1}. This process repeats a number of times and terminates when the agent reaches an absorbing state (a state which, once entered, cannot be left) or $t = t_{\max}$, where $t_{\max}$

denotes the maximum number of time-steps allowed. We call the entire sequence of decisions, from the beginning of the process until termination, an *episode*. After each episode terminates, the environment is reset to an initial state S_0 and the process is repeated.

A policy, π, describes the mechanism by which the agent decides what action to take given the current state of the environment. It determines the probability of the agent selecting an action a at any given state s; that is, $\pi(s, a) = \Pr(A_t = a|S_t = s)$. The performance of a policy is defined as: $J^\pi = \mathbf{E}\left[\sum_{t=0}^{t_{\max}} \gamma^t R_t \middle| \pi\right]$, where $t_{\max}$ can be finite or $t_{\max} = \infty$, and $\gamma \in [0, 1]$ is a discount factor. Notice that if $t_{\max} = \infty$ and $\gamma = 1$, J^π could be unbounded, so we assume that both conditions are not true simultaneously.

In intuitive terms, the performance J^π of a policy π is the expected discounted sum of rewards obtained by the agent in a particular MDP given that the agent acts according to a policy π. The sum of rewards obtained during an episode is also called the *return*. Throughout this thesis we will refer to the discounted sum of rewards as the discounted return whenever it helps in simplifying discussion of the concepts being presented.

A transition function, P, describes how the environment transitions between states. A trajectory of length l, defined as $H_l = (S_0, A_0, R_0, \ldots, S_{l-1}, A_{l-1}, R_{l-1}, S_l)$, denotes the sequence of states, actions, and rewards the agent experiences by interacting with the environment for l time-steps. The objective of the agent is to learn a policy that achieves the maximum discounted return possible. Such a policy is referred to as an *optimal policy*, denoted as π^*, and is defined as:

$$\pi^* \in \arg\max_{\pi} \mathbf{E}\left[\sum_{t=0}^{t_{\max}} \gamma^t R_t \middle| \pi\right]$$

.

Formally, an MDP is defined as a tuple $(\mathcal{S}, \mathcal{A}, P, d_0, R, \gamma)$, where:

- $\mathcal{S}$ is the state set of the environment. It defines the set of all possible states the system could be in. An element of $\mathcal{S}$ is denoted by s, and we use S_t to refer to a random variable representing the state of the environment at time-step t.

- $\mathcal{A}$ is the set of actions that the agent can take. An element of $\mathcal{A}$ is denoted by a, and A_t refers to the random variable denoting the action taken at time-step t.

- R is the reward function which characterizes the distribution over rewards at time t given S_t, A_t, and S_{t+1}. We also use R_t to denote a random variable representing the reward the agent receives at time-step t.

- P denotes the transition function that determines how states transition in the MDP. The distribution over states at time $t+1$ is fully determined by the state of the environment and the action taken at time t. This property is referred to as the *Markov property*. Formally $P(s, a, s')$ is the probability that the environment transitions to state s' after the agent takes action a at state s:

  $$P(s, a, s') = \Pr(S_{t+1} = s' | S_t = s, A_t = a).$$

  Some algorithms, such as value or policy iteration (Sutton and Barto, 1998), assume that P is known by the agent. When this is not the case, there are two options: 1) to use algorithms that do not explicitly model P (e.g., Q-learning, SARSA, Actor Critic methods) or 2) to estimate the transition dynamics by making assumptions about the family of probability distributions to which P belongs, and learning the parameters of that distribution through interactions with the environment.

- d_0 is the initial state distribution. That is, at time-step $t = 0$ the environment will be in some state S_0, sampled according to the distribution d_0.

- $\gamma \in [0, 1]$ is a discount factor used to balance preference between long-term and short-term rewards. A smaller value of γ means that the agent will try to maximize shorter term reward.

A central concept to RL is the Q-function, defined as

$$Q^\pi(s, a) = \mathbf{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a, \pi\right].$$

Intuitively, this function returns the value of taking action a in state s at time-step t and then behaving according to the policy π thereafter; this value is referred to as the Q-value. A useful property of the Q-function is given by the *Bellman equation*:

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

This property forms the basis of several popular algorithms; it implies that the Q value associated with 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, given A_t and S_t. Similarly, the *value function* is used to calculate the value of a state s under a policy π. It is defined as $V^\pi(s) = \mathbf{E}\ \big[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}\big|S_t = s, \pi\big]$, and its Bellman equation is given by:

$$V^\pi(s) = \mathbf{E}\big[R_t + \gamma\, V^\pi(S_{t+1})\big|S_t = s, \pi\big].$$

These two equations are of particular importance because many algorithms such as Q-learning, value iteration or policy iteration, aim to approximate these two functions in order to learn an optimal policy. There are two techniques that are commonly used to approximate the value function or Q-function: tabular form and function approximation. For an in-depth treatment of the topic we refer the reader to the work by Sutton and Barto (2018).

The tabular form refers to the practice of keeping in memory an estimate of $V(s)$ or $Q(s, a)$ for all states s or state-action pairs (s, a) the agent has encountered thus far. Because this technique maintains one value per state or state-action pair, algorithms that use this approach are able to obtain accurate estimates of these values; in fact, some algorithms like value iteration or Q-learning have been shown to converge to the true value-function or Q-functions for an optimal policy when using a tabular representation (Szepesvári, 1997).

A limitation of this approach is that it scales poorly as the number of states to be considered increases. This drawback is often addressed by using function approximation. The value function or Q-function are estimated from models V_θ or Q_θ, with parameters θ (e.g., neural networks), and the parameters can be learned from samples as the agent interacts with the environment. When taking this approach, the convergence guarantees provided by the tabular form are lost, however it works well in practice and allows an agent to learn complex policies that would be too computationally expensive to learn in tabular form (Mnih et al., 2015; Lillicrap et al., 2015; Mnih et al., 2016).

Finally, in this thesis we consider the scenario where an agent needs to learn a policy for several related tasks. We define a *problem class* to be a set, $\mathcal{C}$, of tasks c (MDPs), where $c = (\mathcal{S}, \mathcal{A}, P_c, d_0^c, R_c, \gamma)$. That is, each element of a problem class is an MDP defining its own transition function P_c, initial state distribution d_0^c , and reward function R_c. However, all tasks in $\mathcal{C}$ share the same state set, action set, and discount parameter γ.

2.3 Experimental Environments

To study the behavior of the methods developed in this work and to illustrate some of their properties, we selected a number of standard benchmarks problems used across the RL literature. Below we introduce each class of problems used throughout this thesis, and describe how tasks vary within $\mathcal{C}$. Each chapter will also present specific problems used to test the methods developed therein.

2.3.1 Gridworld

In a Gridworld task, the environment can be represented by a grid of size $n \times m$. The agent is randomly placed in an initial state and the objective is to reach a specific goal state. The state is represented by the (x, y) position in the environment and the agent has four primitive actions at its disposal: move right, move left, move up or move down. When testing for robustness to *stochastic* environments we consider the case where, after executing an action, with probability $1 - \alpha$ the agent moves in the intended direction and, with probability α the agent moves in any of the other directions at random. This is in contrast to *deterministic* environments, where the agent moves in the intended direction with probability 1.0. 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.

Figure 2.2 shows two variations of a Gridworld problem. The agent is depicted in red, the goal in green and obstacles are shown in black.

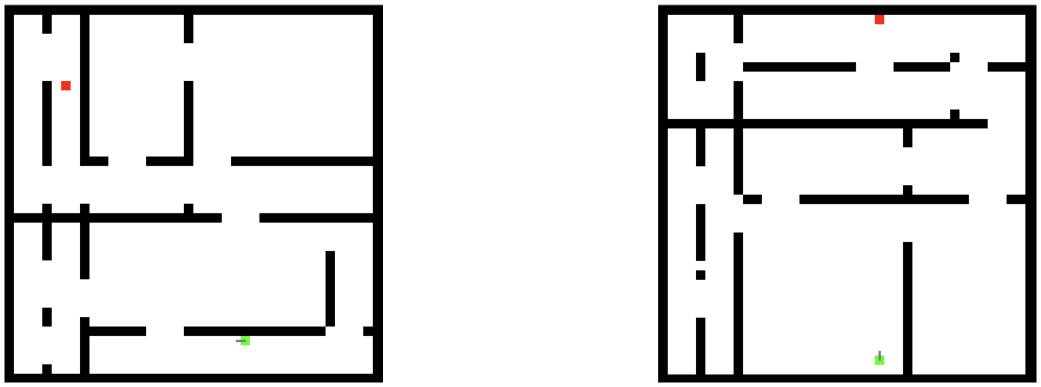


Figure 2.2: Example of two task variations in Gridworld.

2.3.2 Animat

This type of problem was first introduced by Thomas and Barto (2011). The agent is a circular creature that moves in a continuous state space with eight 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 moves according to the following mechanics. Let d be the total displacement given by an actuator β with angle θ_β, the displacement of the agent for a set of active actuators, $\mathcal{B}$, is given by: $\Delta_x = \sum_{\beta \in \mathcal{B}} d \cos(\theta_\beta)$ and $\Delta_y = \sum_{\beta \in \mathcal{B}} d \sin(\theta_\beta)$. After taking an action, the new state is perturbed by standard normal noise to simulate stochasticity.

The agent is tasked with moving to a goal location; it receives a small negative reward at each time-step and a large positive reward upon reaching the goal state. The different variations of the tasks correspond to randomized start and goal positions in different environments.

Two task variations for Animat are shown in Figure 2.3, showing the creature with active actuators highlighted and the goal shown as a red circle.

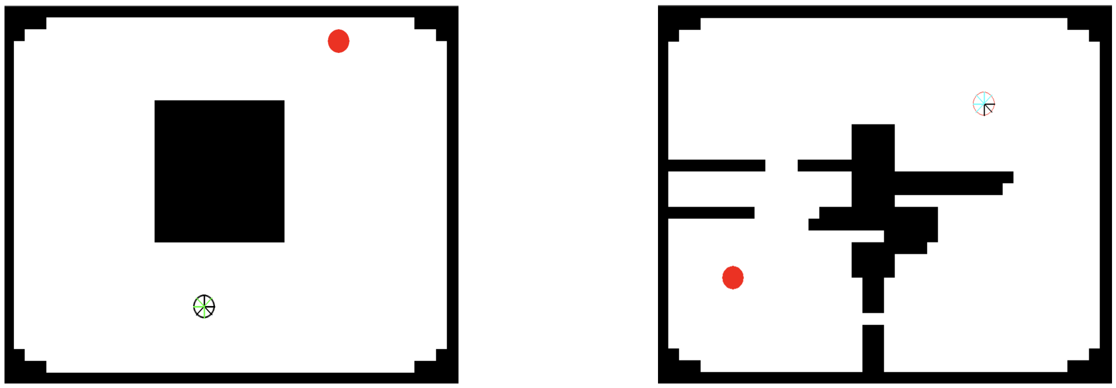


Figure 2.3: Example of task variations in Animat.

2.3.3 OpenAI Gym Suite

OpenAI Gym (Brockman et al., 2016) is a suite of benchmark problems specifically devised for testing RL algorithms. The domain of available problems include robotic simulations, video games, traditional control problems, among others. We modified the implementation of several existing environments within Gym to allow us to define task variations that would differ significantly in terms of their transition function, reward function and initial state distribution. We will specify in each chapter when and how we use problems from this suite to benchmark our methods.

2.4 Related Work

As mentioned in the introduction, we identified three main settings that result in leveraging prior knowledge as a natural side effect of the problem formulation: transfer learning, curriculum learning, and learning from demonstration. In this section we review the relevant literature in more detail and present a discussion on *temporal abstraction*—a prevalent technique for gaining useful insights from prior experience that forms the basis of two of the main contributions in this work.

2.4.1 Transfer Learning in Reinforcement Learning

Pan and Yang (2010) defined the general transfer learning problem as follows:[1]

> Given a source problem class $\mathcal{C}_S$ and source tasks $c_S \in \mathcal{C}'_S$, where $\mathcal{C}'_S \subset \mathcal{C}_S$, a target problem class $\mathcal{C}_T$ and target tasks $c_T \in \mathcal{C}'_T$, where $\mathcal{C}'_T \subset \mathcal{C}_T$, *transfer learning* aims to help improve the learning of optimal policies $\pi^*_{c_T}$, for $c_T \in \mathcal{C}'_T$ using knowledge obtained from learning optimal policies $\pi^*_{c_S}$ for $c_S \in \mathcal{C}'_S$.

[1]For clarity we adapt their definition to the notation used in this dissertation.

For example, one might first learn optimal policies, $\pi^*_{c_S}$, for a set of source tasks, $c_S \in \mathcal{C}'_S$, in a source problem class and use information gathered from learning each $\pi^*_{c_S}$ to quickly learn optimal policies for a set of target tasks $c_T \in \mathcal{C}'_T$. There are two main type of problems that transfer learning is concerned with: inter-domain transfer, where $\mathcal{C}_S \neq \mathcal{C}_T$, and intra-domain transfer, where $\mathcal{C}_S = \mathcal{C}_T$. From this point of view, this thesis focuses on the setting where $\mathcal{C}_S = \mathcal{C}_T$; that is, the tasks the agent uses to gather experience are assumed to belong to the same problem class as the tasks the agent faces later on.

In the context of RL, we can consider the work by Driessens and Džeroski (2004) as a concrete example of transfer learning. The authors propose guiding the agent during early stages of learning by using demonstrations or solutions to a number of related tasks to fit a regression model to estimate the Q-function. When facing a new problem, instead of randomly initializing the Q-function estimates, it is initialized using the regression model; thus, transferring knowledge of the Q-functions of related problems, and guiding the agent at the beginning stages of learning.

In a thorough survey of transfer learning for RL, Taylor and Stone (2009) identified five metrics of interest to evaluate the benefits of transfer:

1. *Jumpstart:* The initial performance of an agent on a target task.

2. *Asymptotic Performance:* The performance of the final policy learned by an agent in a target task.

3. *Total Reward:* The total reward accumulated by an agent over a fixed period of time. In episodic tasks, this is the cumulative return of a learning agent and can be visualized by the area under the learning curve—the curve depicting the return obtained by an agent as a function of episodes.

4. *Transfer Ratio:* The ratio of the total reward accumulated by an agent using transfer learning to that of an agent without transfer.

5. *Time to Threshold:* The time needed by an agent to learn a policy that achieves some pre-defined performance threshold.

For the problems we study, we evaluate the ability to generalize prior knowledge by considering *total reward* or *time to threshold* (whichever is appropriate) as our evaluation metrics.

2.4.2 Learning from Demonstration

Another approach for using prior knowledge to help an agent learn policies quickly is *learning from demonstration* (LfD) (Argall et al., 2009). In this formulation, it is assumed that an agent is shown examples of optimal behavior and the goal is to derive policies that are capable of mimicking the demonstrated behavior. Naturally, the derived policies are only able to adapt to regions of the state space that were encountered during the demonstrations in a specific task. However, when an agent encounters an unknown state or unknown task, the policy derived from demonstration may provide a suitable initial guess which can be quickly fine-tuned.

The applications of LfD in RL are vast. It has been successfully used in real-time systems to teach a robot how to play table tennis (Mülling et al., 2010) and to play with a soccer ball (Grollman and Jenkins, 2007). It has also been used in more abstract RL problems as a way of teaching an agent to better explore the state space (Subramanian et al., 2016) or as a means to transfer human demonstrations into baseline policies to improve an agent's performance on a specific task (Taylor et al., 2011).

2.4.3 Curriculum Learning

In the context of machine learning, curriculum learning was popularized by Bengio et al. (2009). The authors proposed a framework for training neural networks inspired by developmental psychology principles: start by training on simple problems and gradually increase the difficulty as the network reaches satisfactory performance. This strategy is what makes it possible for humans to learn things that would otherwise seem impossible to learn. The same principles apply to RL.

If the agent has a way of assessing the difficulty of a task, given a set of tasks it is required to solve, it could choose to solve the simpler ones first and leverage the policy learned to help in solving more difficult problems. In this regard, curriculum learning is a general technique for reusing prior knowledge.

Although simple techniques for curriculum learning have been shown to be effective in practice, in the context of RL this area has not been studied as in depth as transfer learning or LfD.

2.4.4 Temporal Abstraction

Temporal abstraction in RL (Precup, 2000) is a technique that oftentimes facilitates learning an optimal policy for difficult problems. In the original MDP formulation, at each time-step the agent's policy selects an action for execution, which last for a single time-step. These are called *primitive actions*. Temporal abstraction refers to the ability of an agent to execute *temporally extended* actions—actions that can last for several time-steps. This framework is formalized in the formulation of semi-Markov decision process (SMDP) (Sutton et al., 1999).

There are several formulations and algorithms that have been proposed for temporal abstractions, like hierarchies of abstract machines (HAMs) or the MAXQ algorithm (Parr and Russell, 1998; Dietterich, 1998). However, in this thesis we will focus on two specific formulations: *macros* and *options*.

Macros are open-loop temporally extended actions: that is, sequences of actions that are executed to termination regardless of the states encountered during execution. Formally, a macro of length k is a sequence of actions, $m = (a^1, a^2, \ldots, a^k)$, where a^i represents the i^{th} action in the sequence. If carefully constructed, macros can dramatically improve the speed with which an agent learns to solve a task. This is because they bias the behavior of an agent towards certain action patterns; if these action patterns are appropriate for the task at hand, they would allow the agent to reach high-value states early on (McGovern and Sutton, 1998).

Options are a generalization of macros to closed-loop policies. An option, $o = (\mathcal{I}_o, \mu_o, \beta_o)$, is a tuple in which $\mathcal{I}_o \subseteq \mathcal{S}$ is the set of states in which option o can be executed (the *initiation set*), μ_o is a policy that governs the behavior of the agent while executing o, and $\beta_o : \mathcal{S} \to [0, 1]$ is a termination function that determines the probability that o terminates in a given state. Options allow an agent to execute high-level behaviors without having to consider each primitive action at each state encountered during execution. For example, a mobile robot trying to exit a room in an indoor environment could choose to execute a `go-to-door` option instead of having to consider each individual action that is needed to exit the room. Because they simplify the decision making process of the agent, options that are well suited for an MDP usually lead to significant improvements in learning speed (Precup, 2000). In this thesis, we assume that $\mathcal{I}_o = \mathcal{S}$ for all options o; that is, the options are available at every state.

When an agent is able to plan by selecting options in an MDP, the process is considered a SMDP. This formulation enables the agent to not only estimate the Q-function of an option upon executing it, but also the Q-value of the primitives executed by the option policy through *intra-option learning* (Sutton et al., 1999). Whenever working with macros or options, we make use of intra-option learning to speed up learning.

The definitions presented for macros and options, however, only provide a formalism for how to implement temporal abstraction in RL, but do not give us any guidance about how those abstractions (macros or options) should be obtained, how many are needed, or how an agent should use them. As we will show in this work, one viable approach is to use prior experience to find abstractions that are useful for novel problems an agent might face.

In this work, we will use two state-of-the-art methods in temporal abstraction for comparison: eigen-options, (Marlos C. Machado, 2017), and the option-critic architecture, (Bacon et al., 2017).

Eigen-options build on the ideas of proto-value functions (Mahadevan, 2005); by using the graph Laplacian for the transition graph of a MDP, this method finds the k eigen-vectors with largest eigen-values and learns options that allow an agent to reach extreme states in the MDP by treating the eigen-vectors as pseudo-rewards. The intuition behind this idea is that the identified options would allow an agent to move quickly to the extremes of the transition graph, allowing it to better explore the state space and quickly learn an effective policy. However, there are two major drawbacks: it is not clear how to find the right number of options to learn, and each option is obtained by solving a new MDP corresponding to each eigen-vector.

The option-critic architecture, on the other hand, directly learns options and a policy over options while solving a new MDP. The options and policy over options are defined by a set of parameters which are updated in order to optimize the expected return the agent receives. As we will see in subsequent chapters, the ability of the options learned by the option-critic to generalize to new problems is limited.

2.4.5 Generalization in the Context of Planning

Related to the problem of RL is that of classical planning. In this framework, the goal is to identify a policy that optimizes some utility; unlike the case of RL, the problems are generally specified by a description of pre-conditions, actions, post-conditions, and goal states. Pre-conditions are characteristic of the current state that must be true for certain actions to be available, and post-conditions are the possible effects that a given action might have on the current state. In other words, this framework usually requires one to be able to specify goal states, and to have a complete model of the world (assumptions not met in this thesis). When these assumptions hold, and the search space is not prohibitively large, planning techniques tend to yield better solutions than RL.

In the context of planning, the use of prior knowledge has been studied more extensively than in RL. One early technique tackling generalization in robot planning was presented by Fikes et al. (1993); the authors extend the original STRIPS planning specification language to be able to generalize some of the solutions produced within this framework. They propose replacing problem-specific constants found in STRIPS plans with problem-independent parameters. This extension allows for the creation of macro actions by reusing plans (or partial plans) found when solving similar tasks.

Another related approach is to extend the problem definition with partial plans, or macros, found from a number of related problems (Newton et al., 2007). The authors propose using genetic algorithms to generate a large pool of candidate macros, evaluate each one of them in a set of simple problems, and incorporate the top k performing macros to the problem definition.

There has been considerable amount of work that focuses on generalization, and leveraging prior knowledge in the planning community (Botea et al., 2005; Srivastava et al., 2010, 2011); however, most of these techniques take advantage of the structure present in the planning formulation, and are therefore not directly applicable to the RL setting treated in this thesis.

CHAPTER 3

A HEURISTIC APPROACH TO MACRO DISCOVERY THROUGH COMPRESSION TECHNIQUES

In many potential RL applications, an agent might have access to a set of trajectories obtained from expert demonstrations for related tasks. These trajectories might be provided by the user (perhaps handcrafted behaviors showing how to solve some tasks) or, if the agent is required to solve a large number of tasks, solving a small number of tasks could provide demonstrations of optimal behavior for a specific class of problems.

In such a scenario, when facing a new task, it would be wasteful for an agent to ignore this prior knowledge and learn how to solve a subsequent problem completely from scratch. As mentioned in the introduction, research in animal behavior indicates that complex behavior is derived from simple building blocks. For example, i Cancho and Lusseau (2009) analyzed the behavior of bottlenose dolphins in the wild and identified a set of simple behavioral patterns which, as a whole, describe the entire behavioral repertoire displayed at the surface. They also found that the simpler a building block is, the more often it tends to be used. This observation draws a connection to compression; shorter encodings are used to represent the symbols that are used more often. Although some researchers consider the fact that simpler behaviors are used more often as a natural result of energy efficiency, it has also been argued that this is a *consequence* of compression in the information-theoretical sense (i Cancho et al., 2013)

One way to improve an agent's ability to learn a new task is through temporally-extended actions, provided they are adequate for the task. In this chapter, we show that ideas taken from the compression literature can be used to identify macros (akin to behavioral building blocks) from recurring action patterns in trajectories obtained from (near)-optimal demonstrations.

We consider the scenario where an agent is required to solve a large number of different but related tasks, belonging to the same problem class. After an agent learns optimal (or near-optimal) policies for a set of training tasks, trajectories from these policies are sampled to generate demonstrations. From these trajectories we generate, evaluate, and select macros that will be effective in solving subsequent tasks.

In this chapter, we present a general framework for identifying macro actions appropriate to the problem class. We posit that useful macros are building blocks of recurring behaviors and can be identified through compression techniques. We introduce the notion of the *utility of a macro*, which we call the U-value, and introduce a novel approach for evaluating distances between macros. Of particular importance is the fact that this framework naturally addresses a common problem that arises in popular option and macro discovery techniques: including too many macros in the action set hinders the ability of an agent to learn an optimal policy (as the agents has too many decisions to consider), while including too few forgoes the opportunity to leverage useful macros.

While many approaches in the temporal abstraction literature require to pre-define the number of macros or options to be considered, the framework presented in this chapter naturally results in a heuristic for finding an appropriate number of macros. Our framework allow the agent to incorporate a small and diverse subset of effective macros from a large number of candidates. The agent is then able to leverage the identified macros to quickly learn a policy in new problems.

This chapter is organized as follows. In Section 3.1 we provide a formal problem description for this chapter. We present related work relevant to the content in Section 3.2 and present an introduction to compression in Section 3.3. In Section 3.4 we describe the proposed framework and describe how compression can be used to identify macros using one compression technique, LZW (Welch, 1984). We show the benefits of our approach in Section 3.5 and conclude with closing remarks in Section 3.6.

3.1 Problem Description

We consider the setting where an agent is required to solve a set of tasks $c \in \mathcal{C}$ sampled from a distribution over the problem class $\mathcal{C}$. To evaluate our method, we will use n sampled tasks to construct a set of training tasks, $\mathcal{C}_{train}$, that will allow us to identify a set of macros. We will then evaluate the performance of these macros in a set of m tasks, $\mathcal{C}_{test}$, sampled from the same distribution; we assume that when solving a particular task, the agent can interact with it for I episodes. Our goal is to identify patterns in trajectories obtained from optimal policies for problems already solved to improve learning in novel problems. We assume that macros identified in trajectories of optimal policies in a set of representative training tasks will be useful to solve other related tasks obtained from the same problem class, as they represent recurrent behavioral building blocks. This assumption is supported by our experimental results.

As a concrete example of this type of problem, consider the case of an agent tasked with allocating cloud resources as described by Liu et al. (2017). 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 were able to improve its effectiveness in learning, it would be able to quickly reach an efficient resource allocation policy in any new problem it might face. In such a scenario, the agent could learn optimal policies for a small sub-set of resource allocation problems, $\mathcal{C}_{train}$, analyze trajectories from these policies, and identify macros to quickly obtain efficient policies for a set of novel problems, $\mathcal{C}_{test}$.

We define the performance of a set of macros, $\mathcal{M}$, in a particular task c to be the expected average return an agent receives over I episodes on a task c using an extended action set $\mathcal{A}_{\mathcal{M}} = \mathcal{A} \cup \mathcal{M}$ (an action set composed of both primitives and macros). Formally, we define the performance as $\rho(\mathcal{M}, c) = \mathbf{E}\left[\frac{1}{I}\sum_{i=0}^{I}\sum_{t=0}^{T} R_t^i \middle| \mathcal{A}_{\mathcal{M}}, c\right]$, where R_t^i is the reward at time step t during the i^{th} episode, the implication being 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 increase its expected return during training.

Formally speaking, let C be a random variable denoting a task drawn from $\mathcal{C}$; our objective is to find one (of the many possible) optimal set of macros $\mathcal{M}^*$ for $\mathcal{C}$ according to the following criterion:

$$\mathcal{M}^* \in \arg\max_{\mathcal{M}} \quad \mathbf{E}\left[\rho(\mathcal{M}, C)\right]. \tag{3.1}$$

Unfortunately, the domain of the objective (3.1) is discrete, making it non-differentiable, and ρ cannot be easily evaluated, so directly solving for $\mathcal{M}^*$ is a difficult problem. As previously argued, compression techniques could be used to identify recurring behaviors; if applied to demonstrations obtained from optimal policies, these techniques would allow an agent to identify macros that are efficient for learning how to solve new related tasks. In this chapter we will show one heuristic method in which compression can be used to generate a set of candidate macros, and how to approximate the set $\mathcal{M}^*$ by incorporating the set of top performing and diverse macros, $\mathcal{M}'$, to the agent action-set.

For the framework developed in this chapter, we assume that we are able to estimate the Q-values for all actions $a \in \mathcal{A}$ and states $s \in \mathcal{S}$, and also the transition function P_c by assuming a family of distributions to which it belongs. We also consider the case where π^* is greedy with respect to Q; that is, at any state the optimal policy always chooses the action with highest value. As we will see in the rest of this chapter, these assumptions allow us to efficiently estimate the *utility* of a macro.

Although the method proposed in this chapter is motivated by principled methods related to compression, it is worth noting that in practice we do not optimize the objective in (3.1) directly. Instead, in this chapter we focus on developing a practical method motivated by the compression literature.

3.2 Related Work

In this section, we review work related to the discovery of macro actions and provide a brief introduction to the concepts of compression relevant to this chapter.

3.2.1 A Motivating Example for the Use of Macro Actions

We justified the benefits of macros by discussing an example occurring in nature; however, we would now like to motivate macros by looking at a concrete example of a study performed by McGovern and Sutton (1998). In their work, the authors analyzed the effect macros have on learning performance and how they can either improve or hinder learning efforts depending on how macros are constructed and what specific task an agent is trying to solve. One experiment in particular demonstrates these effects clearly.

Figure 3.1 shows the effect that certain macros might have in a problem class when using the learning algorithm Q-learning (Watkins and Dayan, 1992). The objective in this problem is for the agent to navigate to the goal state, marked with the letter

G, starting from the state marked with the letter S. The agent has four primitive actions: move left, right, up and down, and the macros are defined by executing one of the actions repeatedly enough times to reach the edge of the grid.

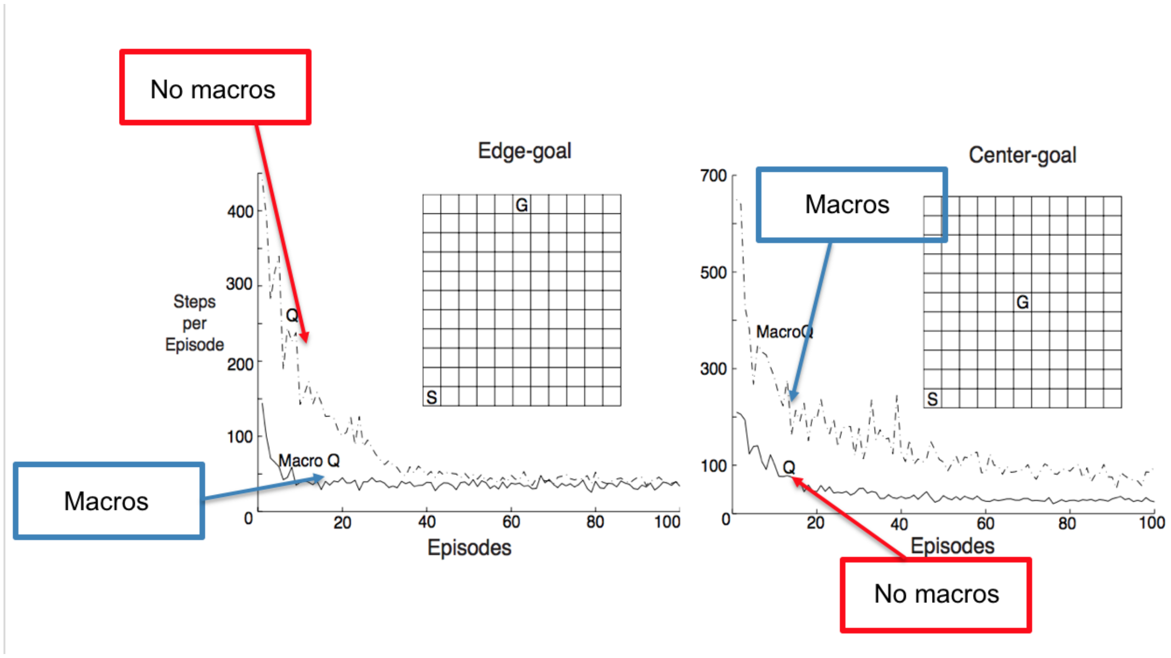


Figure 3.1: Effect of macros on learning. The plot on the left shows that Q-learning with macros is able to learn a policy to reach the goal much quicker than without using macros for this goal location. The plot indicates that the opposite effect is possible if the goal is located in a different position. Figure adapted with permission by McGovern and Sutton (1998).

The figure compares the learning performance of two different agents in two different tasks: the curve marked macro-Q shows the learning performance of an agent using macro actions and the curve marked Q refers to the learning performance of an agent using only primitive actions. In the figure on the left, the goal was placed at the top of the grid. The learning curves indicate that using macros clearly speed up learning for this particular task. On the other hand, the plots corresponding to the figure on the right indicate that, when the goal is being placed at the center of the grid, the use of these macros actually hinders the ability of the agent to quickly learn a good policy.

The reason for this difference in learning efficiency becomes evident when looking at Figure 3.2. The plots show the distribution of state visitations from 500,000 steps of the agent when there is no goal. The graph on the left refers to an agent using only primitive actions and shows that the distribution of state visitation is roughly uniform throughout the state space. On the other hand, the graph on the right shows the same plot when the agent is allowed to use the macros defined previously. These macros bias the behavior of the agent to visit states at the edge of the grid, making these areas easier to reach, but states in the center harder, which is consistent with the results shown in the previous figure.

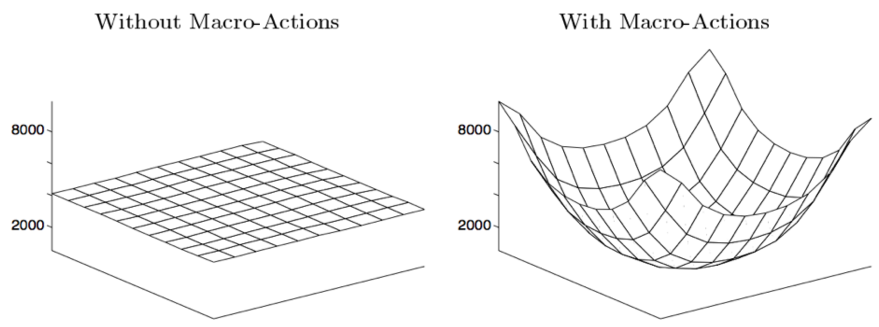


Figure 3.2: State visitation distribution for the environment when randomly selecting primitive actions (left) and both primitive and macros (right). Figure adapted with permission from work done by McGovern and Sutton (McGovern and Sutton, 1998).

These experiments intend to show that the bias induced by a set of macros can be highly beneficial if crafted correctly. If the macros encode behavioral building blocks that occur frequently in optimal behaviors for many related tasks, they should allow the agent to learn quicker in novel tasks that related to the ones from which macros were obtained.

3.2.2 Option and Macro Discovery

As mentioned earlier, macros and options have the potential of simplifying learning (near)-optimal policies for difficult tasks. These techniques allow the agent to “commit” to some behavior for an extended period of time, simplifying the decision making process of the agent’s policy, which can result in improved learning speed. This potential learning improvement has led to the development of methods for identifying useful options or macros to become an active area of research under the name of *skill discovery*.

One approach for discovering options 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. McGovern and Barto (2001) propose 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 bottlenecks. A more recent approach based on a similar principle is the eigen-options method presented by Machado et al. (2017). The authors show that by analyzing the transition graph of an MDP—the graph obtained by representing states as vertices and possible state transitions as edges—it is possible to obtain options, called *eigen purposes*, that allow an agent to efficiently move to distant regions of the state space. The intuition is that by learning options that can reach the extreme states of the transition graph, the agent is able to better explore the state space. These methods, however, exhibit a major drawback: they require the full transition graph to be known in advance in order to be efficient. For complex problems, this is an unattainable requirement.

Many other techniques for discovering macros or options do not suffer from this drawback of relying on identifying specific states (Randl, 1998; Krishnamurthy et al., 2016; Menache et al., 2002; Bacon et al., 2017); however, many are still limited in their application because they assume that the transition graph will be the same in future tasks.

Lastly, another interesting approach is that of Hansen et al. (1996), where macros emerge naturally from optimizing the proposed objective. The authors consider the case where there is a cost associated with sensing; for example, the power drawn from a battery in a mobile robot when observing its environment. In this setting, it is possible for an agent to learn when to act in an open-loop fashion, and when to act close-loop (i.e., when to sense), thus, minimizing the power drawn from the battery and prolonging the life of the agent. The authors show that by including an action of "sensing" that incurs in a cost every time it is called, it is possible to use RL techniques to learn when and how long to act open-loop, and what actions to take while doing so; essentially, learning open-loop macros.

In contrast to the techniques mentioned, the framework we propose in this chapter makes minimal assumptions about the structure of the problem, and potential macros are evaluated offline based on the Q-values estimated for primitive actions.

3.3 A Brief Introduction to Compression

The goal of compression is to represent messages or data in a compact manner by reducing the number of symbols needed to express that information. Many compression algorithms share the same building blocks and their differences lie in how those elements are constructed and used. For the reader interested in a detailed treatment of the topic we refer them to the work by Salomon and Motta (2009). In this section we introduce the concepts relevant to the framework presented in this chapter.

Many compression algorithms pre-define or incrementally build a set of symbols called an alphabet; we refer to this alphabet as Σ. Accompanying the alphabet there is also a *codebook* defining a unique binary representation for each symbol in Σ;

this representation is referred to as a *codeword*. The alphabet contains all symbols that might be present in a message and the goal of compression is to represent the original message using the smallest number of bits possible. Compression techniques can be classified in two types: *lossy compression*, where compression is achieved through approximate techniques, thus losing part of the original information, and *lossless compression*, where the original message can be recovered exactly from the compressed representation. We only consider the case of lossless compression.

Once the codebook is obtained, 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 two alphabets $\Sigma_1 = \{a, i, h\}$ and $\Sigma_2 = \{a, i, h, hi\}$, with corresponding codebooks $A = \{00, 01, 10\}$ and $B = \{00, 01, 10, 11\}$. Furthermore, consider encoding a message $\alpha =$ "hi" under each different codebook. The shortest binary representation of α under codebook A would be 1001 ($h = 01$, $i = 10$); however, under codebook B, it would be represented as 11 ($hi = 11$). This is a simple example of how compression techniques are able to find a compact representation to express messages. If we had to consider compressing many messages, the compression techniques we use would have to create a short binary representation for those sequences of symbols that occur more frequently.

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

The proposed framework can be summarized by the diagram shown in Figure 3.3. We assume the agent has trained in a set of tasks $\mathcal{C}_{train} \subset \mathcal{C}'$ and has obtained an optimal policy π^*_c for each task. The agent then 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.

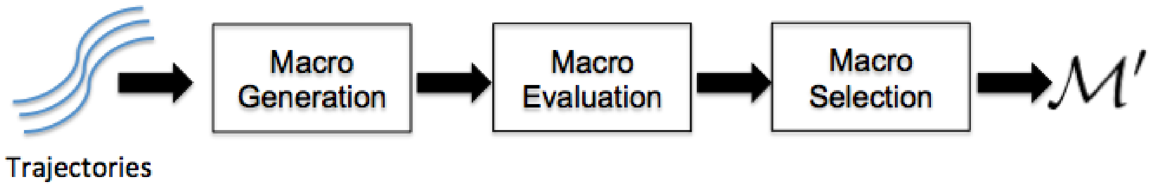


Figure 3.3: Diagram depicting proposed framework.

3.4.1 Macro Generation - A Compression Approach for Identifying 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 be an expensive procedure and 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 a trajectory h obtained by following the sequence of primitive actions (a, b, c, d) obtained from some trajectory h. We will refer to the sequence of actions that originated a trajectory h as an *action-trajectory* h_a. From the perspective of compression, we view h_a as a message to be compressed and the primitive actions a, b, c, d as the symbols initially available in the alphabet. Some compression algorithms, like Huffman Coding (Huffman, 1952), require one to specify the codebook with all available symbols before compressing a message. In our context, this corresponds to specifying beforehand all possible macros to be considered, which would defeat the purpose of using compression to identify macros. For that reason, we consider algorithms which incrementally build the codebook from an initial set of symbols, such as LZW (Welch, 1984). Using this type of algorithm to compress an action-trajectory, h_a, would result in building repeating sequences of symbols that are incorporated into the alphabet; allowing us to identify new macros during compression. In other words, initially the symbols in the alphabet represent only primitive actions and, after compression, the alphabet will also contain symbols representing macros.[1] A similar application has been proposed by Elliott and Huber (2005), where the authors proposed using compression for discovering sequences of actions that are often performed by a user in desktop applications, so that the system can assist a user in completing a task.

Following this intuition, we compress the sampled action-trajectories and, from the symbols defined in the final codebook, we obtain 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) frequently recurring in those trajectories. In other words, they constitute behavioral building blocks that are often present in optimal trajectories in tasks belonging to a specific class of problems. For this reason, they represent good candidates for reusable macros.

In our implementation, we selected LZW (Welch, 1984) 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.

3.4.2 Macro Evaluation - The Utility of a Macro

Algorithm 1 would possibly generate a very large set of candidate macros, $\mathcal{M}$, that were identified from action patterns in the sampled trajectories. However, we do not have a sense of how useful they are in relation to each other. One way of evaluating the macros would be to re-train the agent on the training tasks, adding

[1]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

```
1: $\Sigma = \mathcal{A}$
2: macro $m = ()$
3: for each action-trajectory $h_a$ do
4:     for $i = 1 \dots |h_a|$ do
5:         $a_i = i^{\text{th}}$ element in $h_a$
6:         $m = m + a_i$
7:         if $m \notin \Sigma$ then
8:             $\Sigma = \Sigma \cup \{m\}$
9:             $m = ()$
```

each macro in turn to the action-set and assessing the resulting improvement with respect to only using primitive actions. However, for large primitive action sets there could be thousands of macros identified, making this a computationally expensive strategy. To address this problem, we define 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 primitive actions.

Recall that in a task c, the Q-value for a state s and action a under a policy π can be calculated by the *Bellman equation*, defined as:

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

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, given A_t and S_t.

We can generalize the definition of the Q-value for a primitive action to an arbitrary macro m of length l. We denote by $m_{(i)}$ the i^{th} action in macro m, and define the Q-value for a macro m in a state s as:

$$Q_c^\pi(s,m) = \mathbf{E}\ \big[\sum_{i=t}^{l} \gamma^{i-t} R_t + \gamma^{t+l} Q_c^\pi(S_{t+l}, A_{t+l}) | S_t = s, A_t = m_{(1)}, \cdots, A_{t+l-1} = m_{(l)}\big].$$

With this definition in hand, we propose determining the *utility* of a macro m over a problem class $\mathcal{C}$ by the U-function defined as:

$$U_{\mathcal{C}}(m) = \mathbf{E}\left[Q_{C}^{\pi^*}(S, m)\right], \tag{3.2}$$

where the expectation is defined over both tasks C and states S sampled from the *on-policy distribution* (Sutton and Barto, 2018) of the optimal policy for each corresponding task. In other words, we defined the utility of a macro m to be the expected Q-value of m over all states in a problem class $\mathcal{C}$ assuming the agent behaves according to the optimal policy for all tasks $c \in \mathcal{C}$.

Assuming that compressing action-trajectories generates a large number of candidate macros, learning the true utility of macros as defined in Eq. (3.2) for each candidate becomes computationally expensive, particularly if the size of $\mathcal{S}$ is large. However, given our stated assumption that π^* is greedy with respect to Q and we have an estimate for the Q-functions and transition functions, this formulation allows us to compute the U-values offline for all macros in closed-form. This property is shown in Theorem 1. For clarity we write $P_c(s^{(t)}, a, s^{(k)})$,where $k > t$, in place of $\Pr(S_k = s^{(k)} | A_t = a, S_t = s^{(t)})$. We also denote by $s^{(k)}$ the state visited after executing k actions from state s, and denote by $R_c(s^{(t)}, m, s^{(k)})$ the reward accumulated by executing macro m at state $s^{(t)}$ in task c.

Theorem 1. Let π be a policy, $c \in \mathcal{C}$ a task in problem class $\mathcal{C}$, and $Q_{\mathcal{C}}^{\pi}(s, a)$ be the Q-value of executing action $a \in \mathcal{A}$ in state $s \in \mathcal{S}$. The Q-value, $Q_{\mathcal{C}}^{\pi}(s, m)$, of a macro m of length l (and consequently $U_{\mathcal{C}}^{\pi}(m)$) can be computed in closed-form by:

$$\begin{aligned}Q_c^{\pi}(s, m) = &\sum_{k=1}^{l}\Bigg[\sum_{s^{(1)}\in\mathcal{S}} \cdots \sum_{s^{(l)}\in\mathcal{S}} \bigg(Q(s^{(k-1)}, m_{(k)}) - \gamma^k \sum_{a'\in\mathcal{A}} \pi(a', s^{(k)}) Q_c^{\pi}(s^{(k)}, a')\bigg) \\ &\times \frac{\prod_{i=1}^{l} P_c(s^{(i-1)}, m_{(i)}, s^{(i)})}{P_c(s^{(k-1)}, m_{(k)}, s^{(k)})}\Bigg] \\ &+ \prod_{i=1}^{l} P_c(s^{(i-1)}, m_{(i)}, s^{(i)}) \gamma^{(l)} \sum_{a'\in\mathcal{A}} \pi(a', s^{(l)}) Q_c^{\pi}(s^{(l)}, a').\end{aligned} \tag{3.3}$$

Proof. Assume we know the true Q function for a policy π in task c. Let $s^{(i)}$ denote the i^{th} state encountered by executing macro m at state s. We can calculate the Q-value of a macro m of length l at state s as follows:

$$
\begin{aligned}
Q_c^\pi(s,m) &= \sum_{s^{(l)}\in\mathcal{S}} P_c(s^{(0)},m,s^{(l)})\left(R_c(s^{(0)},m,s^{(l)}) + \gamma^l \sum_{a'\in\mathcal{A}} \pi(a',s^{(l)})Q_c^\pi(s^{(l)},a')\right)\\
&= \sum_{s^{(1)},\ldots,s^{(l)}\in\mathcal{S}} \left(P_c(s^{(0)},m_{(1)},s^{(1)}) \times \cdots \times P_c(s^{(l-1)},m_{(l)},s^{(l)})\right)\\
&\quad \times \left(R_c(s^{(0)},m,s^{(l)}) + \gamma^l \sum_{a'\in\mathcal{A}} \pi(a',s^{(l)})Q_c^\pi(s^{(l)},a')\right)\\
&= \sum_{s^{(1)},\ldots,s^{(l)}\in\mathcal{S}} \left(P_c(s^{(0)},m_{(1)},s^{(1)}) \times \cdots \times P_c(s^{(l-1)},m_{(l)},s^{(l)})\right)\\
&\quad \times \Bigg(R_c(s^{(0)},m_{(1)},s^{(1)}) + \cdots + \gamma^{(l-1)} R_c(s^{(l-1)},m_{(l)},s^{(l)})\\
&\quad + \gamma^l \sum_{a'\in\mathcal{A}} \pi(a',s^{(l)})Q_c^\pi(s^{(l)},a')\Bigg)\\
&= \sum_{s^{(1)},\ldots,s^{(l)}\in\mathcal{S}} \prod_{i=1}^{l} P_c(s^{(i-1)},m_{(i)},s^{(i)})\\
&\quad \times \Bigg(R_c(s^{(0)},m_{(1)},s^{(1)}) + \cdots + \gamma^{(l-1)} R_c(s^{(l-1)},m_{(l)},s^{(l)})\\
&\quad + \gamma^l \sum_{a'\in\mathcal{A}} \pi(a',s^{(l)})Q_c^\pi(s^{(l)},a')\Bigg)\\
&= \sum_{s^{(1)},\ldots,s^{(l)}\in\mathcal{S}} \prod_{i=1}^{l} P_c(s^{(i-1)},m_{(i)},s^{(i)}) R_c(s^{(0)},m_{(1)},s^{(1)})\\
&\quad + \cdots + \prod_{i=1}^{l} P_c(s^{(i-1)},m_{(i)},s^{(i)}) \gamma^{(l-1)} R_c(s^{(l-1)},m_{(l)},s^{(l)})\\
&\quad + \prod_{i=1}^{l} P_c(s^{(i-1)},m_{(i)},s^{(i)}) \gamma^{(l)} \sum_{a'\in\mathcal{A}} \pi(a',s^{(l)})Q_c^\pi(s^{(l)},a')\\
&= \sum_{s^{(1)},\ldots,s^{(l)}\in\mathcal{S}} P_c(s^{(0)},m_{(1)},s^{(1)}) R_c(s^{(0)},m_{(1)},s^{(1)}) \frac{\prod_{i=1}^{l} P_c(s^{(i-1)},m_{(i)},s^{(i)})}{P_c(s^{(0)},m_{(1)},s^{(1)})}\\
&\quad + \cdots + P_c(s^{(l-1)},m_{(l)},s^{(l)}) \gamma^{(l-1)} R_c(s^{(l-1)},m_{(l)},s^{(l)}) \frac{\prod_{i=1}^{l} P_c(s^{(i-1)},m_{(i)},s^{(i)})}{P_c(s^{(l-1)},m_{(l)},s^{(l)})}
\end{aligned}
$$

$$
\begin{aligned}
&+ \prod_{i=1}^{l} P_c(s^{(i-1)}, m_{(i)}, s^{(i)})\gamma^{(l)} \sum_{a'\in\mathcal{A}} \pi(a', s^{(l)}) Q_c^{\pi}(s^{(l)}, a') \\
=& \sum_{s^{(1)},\dots,s^{(l)}\in\mathcal{S}} \Bigg(P_c(s^{(0)}, m_{(1)}, s^{(1)}) R_c(s^{(0)}, m_{(1)}, s^{(1)}) + \gamma \sum_{a'\in\mathcal{A}} \pi(a', s^{(1)}) Q_c^{\pi}(s^{(1)}, a') \\
&- \gamma \sum_{a'\in\mathcal{A}} \pi(a', s^{(1)}) Q_c^{\pi}(s^{(1)}, a') \Bigg) \frac{\prod_{i=1}^{l} P_c(s^{(i-1)}, m_{(i)}, s^{(i)})}{P_c(s^{(0)}, m_{(1)}, s^{(1)})} \\
&+ \cdots + \Bigg(\gamma^{(l-1)} P_c(s^{(l-1)}, m_{(l)}, s^{(l)}) R_c(s^{(l-1)}, m_{(l)}, s^{(l)}) \\
&+ \gamma^{l} \sum_{a'\in\mathcal{A}} \pi(a', s^{(l)}) Q_c^{\pi}(s^{(l)}, a') - \gamma^{l} \sum_{a'\in\mathcal{A}} \pi(a', s^{(l)}) Q_c^{\pi}(s^{(l)}, a') \Bigg) \\
&\times \frac{\prod_{i=1}^{l} P_c(s^{(i-1)}, m_{(i)}, s^{(i)})}{P_c(s^{(l-1)}, m_{(l)}, s^{(l)})} \\
&+ \prod_{i=1}^{l} P_c(s^{(i-1)}, m_{(i)}, s^{(i)})\gamma^{(l)} \sum_{a'\in\mathcal{A}} \pi(a', s^{(l)}) Q_c^{\pi}(s^{(l)}, a') \\
=& \sum_{s^{(1)},\dots,s^{(l)}\in\mathcal{S}} \Bigg(Q(s^{(0)}, m_{(1)}) - \gamma \sum_{a'\in\mathcal{A}} \pi(a', s^{(1)}) Q_c^{\pi}(s^{(1)}, a') \Bigg) \\
&\times \frac{\prod_{i=1}^{l} P_c(s^{(i-1)}, m_{(i)}, s^{(i)})}{P_c(s^{(0)}, m_{(1)}, s^{(1)})} \\
&+ \cdots + \Bigg(\gamma^{(l-1)} Q(s^{(l-1)}, m_{(l)}) - \gamma^{l} \sum_{a'\in\mathcal{A}} \pi(a', s^{(l)}) Q_c^{\pi}(s^{(l)}, a') \Bigg) \\
&\times \frac{\prod_{i=1}^{l} P_c(s^{(i-1)}, m_{(i)}, s^{(i)})}{P_c(s^{(l-1)}, m_{(l)}, s^{(l)})} \\
&+ \prod_{i=1}^{l} P_c(s^{(i-1)}, m_{(i)}, s^{(i)})\gamma^{(l)} \sum_{a'\in\mathcal{A}} \pi(a', s^{(l)}) Q_c^{\pi}(s^{(l)}, a') \\
=& \sum_{k=1}^{l} \Bigg[\sum_{s^{(1)}\in\mathcal{S}} \cdots \sum_{s^{(l)}\in\mathcal{S}} \Bigg(Q(s^{(k-1)}, m_{(k)}) - \gamma^{k} \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] \\
&+ \prod_{i=1}^{l} P_c(s^{(i-1)}, m_{(i)}, s^{(i)})\gamma^{(l)} \sum_{a'\in\mathcal{A}} \pi(a', s^{(l)}) Q_c^{\pi}(s^{(l)}, a')
\end{aligned}
\tag{3.4}
$$

□

Notice that this expression is given in terms of the Q-values of primitives and does not depend on the reward function (other than through the Q-values of primitives), and consequently, the U-values can be calculated in closed form offline. Having access to the Q-values of primitives for a set of tasks is a reasonable assumption; considering that the agent is required to solve a number of MDPs anyway and that algorithms like Q-learning, (Watkins and Dayan, 1992) and DQN (Mnih et al., 2015) approximate the optimal Q-function. 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(s,a) readily available for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$.

We also need to consider how introducing a new element to the action set affects the terms $\sum_{a \in \mathcal{A}} \pi(a, s) Q_c^{\pi}(s, a)$, for an action a and state s in Eq 3.3. Specifically, introducing a new element to the action set changes the distribution given by π. There are two cases we need to consider that will affect how we can compute the Q-value of a macro from the Q-values of primitives: when the policy is greedy (at any state, π always takes the action with largest Q-value) and when the policy is stochastic (at any state, there is a non-zero probability that π will pick any action).

In the case of a greedy policy notice that: $\sum_{a' \in \mathcal{A}} \pi(a', s) Q_c^{\pi}(s, a') = \max\limits_{a' \in \mathcal{A}} Q_c^{\pi}(s, a')$. Furthermore, by definition, there is always a state-primitive pair whose Q-value is no smaller than the largest state-macro pair Q-value[2]. Consequently, we have that $\max\limits_{a' \in \mathcal{A}} Q_c^{\pi}(s, a') = \max\limits_{a' \in \mathcal{A}_{\mathcal{M}}} Q_c^{\pi}(s, a')$ 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 does affect the summation over all actions. When introducing a new element to the action set, the probability distribution defined by π will necessarily change as well and the value of a macro can no longer be calculated in closed form.

[2] A macro, m, is composed of a sequence of primitives, so at every state there is always a primitive, p, that would allow to reach the same state as m. Therefore, the Q-value of p under a greedy policy cannot be less than the Q-value of m.

However, we can use the Q-values calculated for primitives as a starting point and apply the *Bellman equation* to quickly obtain the correct Q-values.

3.4.3 Macro Selection - Encouraging Macro Diversity

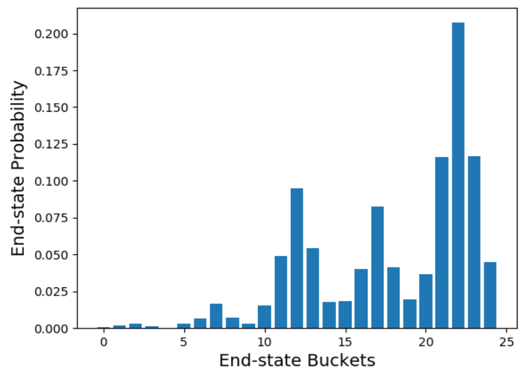
We can use the U-value to estimate which macros we generally believe will lead to higher rewards. However, there is a trade-off we must account for when extending the agent's action set. If too few macros are included in the action set, the agent might miss out on the ability to better explore the state 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. (2017). We tackle this problem by defining a distance function between macros and only including those that are dissimilar enough to the rest of the action-set. Note that in what follows, we make the assumption that a distance measure for the state space is available.

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 $X = d(S_{t+l_m}, S_t)$ denote a random variable, with support $\mathcal{X}$, describing the change in state caused by the execution of m, where d is a distance measure for the state space, and let p_m be the distribution of X 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:

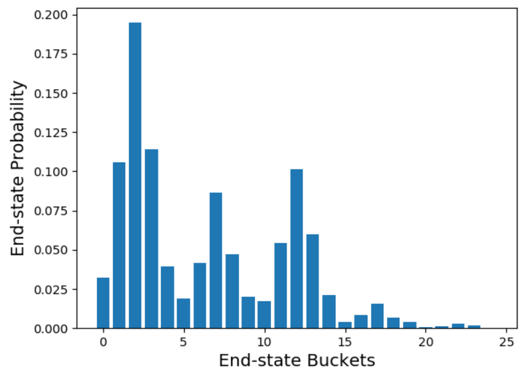
$$D_{KL}(p_{m_1}||p_{m_2}) = -\sum_{x\in\mathcal{X}} p_{m_1}(X=x)\ \log\left(\frac{p_{m_2}(X=x)}{p_{m_1}(X=x)}\right).$$

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

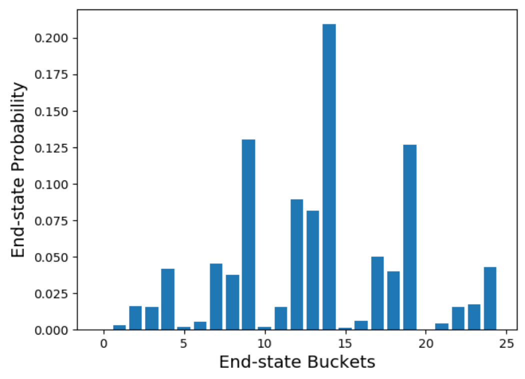
Figure 3.4 shows the empirical end-state distribution calculated for four macros in the maze navigation problem class (introduced in the next section), where the macros m_1, m_2, m_3, m_4 are defined by repeating the same primitive action five times. The possible primitive actions are given by r, l, 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 use the distance between end-state distributions that results from executing two macros as a measure of the similarity between those macros.



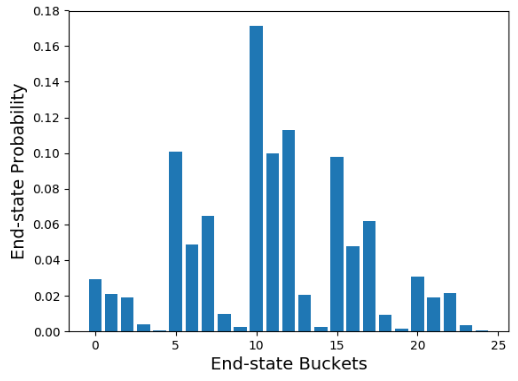
(a) End-state distribution for macro m_1



(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 3.4: 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 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.

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 U-value, the U-function defines a preference criterion by which macros can be selected.

Pseudocode describing our macro discovery framework is given in Algorithm 3.

Algorithm 2 Macro discovery framework

```
1: **1. Macro Generation**
2: Learn optimal policy $\pi_c^*$ for all $c \in \mathcal{C}_{train}$.
3: Collect action-trajectories $h_a$ from each $\pi_c^*$ in task $c$.
4: Generate macros $\mathcal{M}$ from all $h_a$ by Algorithm 1
5:
6: **2. Macro Evaluation**
7: Sort all $m \in \mathcal{M}$ by $U_{\mathcal{C}}^{\pi_c^*}(m)$ in descending order.
8:
9: **3. Macro Selection**
10: $\mathcal{A}_{\mathcal{M}'} = \mathcal{A}$
11: for $m \in \mathcal{M}$ do
12:     if $\min D_{KL}(p_m || p_{m'}) > \delta, \forall m' \in \mathcal{A}_{\mathcal{M}'}$ then
13:         $\mathcal{A}_{\mathcal{M}'} = \mathcal{A}_{\mathcal{M}'} \cup \{m\}$
14: return $\mathcal{A}_{\mathcal{M}'}$
```

3.5 Experimental Results

In this section we present experimental results demonstrating that the identified macros lead to improved learning speed. In this section we aim to answer the following questions: **1)** what is the effect that each stage of our framework has on the macros that are identified? **2)** how does the learning speed of an agent using the identified macros compare to baselines and competing methods? and **3)** how does our method scale in complex problems where the transition function and Q-values must be estimated from data?

We also show through experiments that, because they are simple and general, the identified macros can be beneficial when the tasks used for evaluation may have a different transition graph than the ones used for training.

3.5.1 Does Each Stage Serve a Purpose?

In the previous section we introduced an intuitive 3-stage framework for obtaining generally useful macros, but we have not shown what the effect of each component is in identifying the final set of macros. To better understand the behavior of our framework, we evaluated the results obtained at each stage of the process on several tasks of Gridworld, introduced in Section 2.3.1. This experiment is meant to show how at each step of the process, the identified macros improve the performance of a learning agent and how it compares to the performance of an agent using primitive actions only. Figure 3.5 shows average learning curves (return obtained by the agent as a function of training episodes) and standard error over ten different Gridworlds.

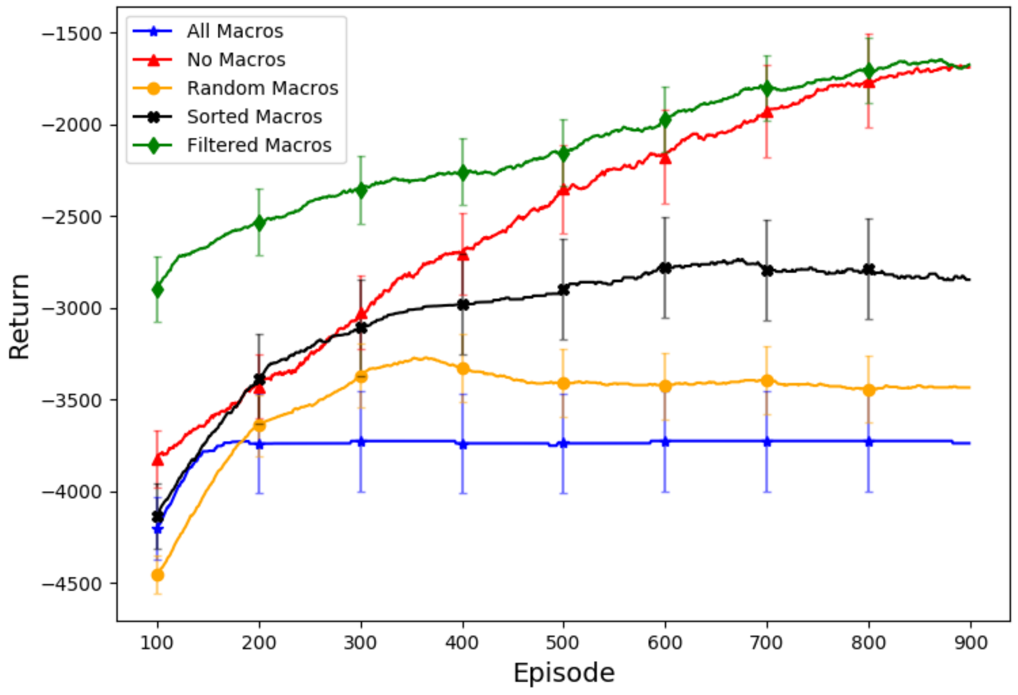


Figure 3.5: Stage evaluation in the macro discovery framework.

The curve in red is our baseline, an agent learning using only primitive actions. The curve in blue shows the performance by including all generated macros in the action set. In this experiment the first stage of our framework generated 500 possible macros. As expected, including such a large number of potential actions is detrimental and so performance suffers significantly.

The curves in black and orange provide an evaluation of the second stage. In orange, the plot shows the mean performance over ten trials of 20 randomly sampled macros generated at the first stage. In black, the plot shows the performance of the top 20 generated macros ranked according to their U-value. This performance improvement compared to randomly selecting macros gives a clear indication that, on average, using the U-value to assess the utility of a macro is a reasonable indicator of how useful a macro might be. However, similar macros would have similar utility, and solely basing the selection on utility limits how diverse the identified macros will be.

Finally, in green we show the performance of the macros selected, based not only on their utility, but also on their diversity as proposed on our framework. The final set of macros shows a substantial improvement over each individual stage, showing that encouraging diversity of high-utility macros works well in practice.

3.5.2 Evaluation on Benchmark Problems

Through the previous experiments, we provided evidence that each step in our framework accomplishes the desired result. We now discuss results showing that the identified macros can lead to large performance improvement on a wide range of RL benchmarks.

We first present tests comparing the performance between our framework and using only primitive actions in the chain problem (introduced in the next section). This is a simple class of problems that allows us to easily assess the behavior of our approach and understand the type of macros that it identifies.

We then evaluate the quality of the macros identified in more complex problems. For these cases, we contrast our approach to using eigen options (Marlos C. Machado, 2017) and the Option-Critic architecture (Bacon et al., 2017); two state-of-the-art methods for learning temporal abstractions in RL. These techniques work in a similar setting to ours, where the agent first interacts with a number of tasks and those experiences can then be leveraged to facilitate learning in novel, but related problems.

As learning algorithms we used Q-learning when representing the Q-function in tabular form (Whitehead, 1991) and, when dealing with more complex environments that require the use of function approximation, we used deep Q-networks (DQN) (Mnih et al., 2015). Q-learning is an algorithm that approximates the Q-function for an optimal policy; the agent can then follow an optimal policy by taking the action with highest Q-value at each state it encounters. When implementing Q-learning in tabular form, the algorithm is guaranteed to converge to the optimal Q-function (under certain assumptions). DQN is a variation of Q-learning where the Q-function is approximated by deep neural networks. Even though the convergence guarantees of Q-learning are lost when using neural networks, DQN is capable dealing with with very large state spaces and works well in practice.

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. A description of each problem used for experimentation can be found in Chapter 2.

3.5.3 Chain Domain

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 small reward of +10 at the closest end of the chain and a large reward of +1,000 at the end that is farther away. Two different different tasks within the chain problem class are shown in Figure 3.6. 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. Each cell indicates a different state in the chain.

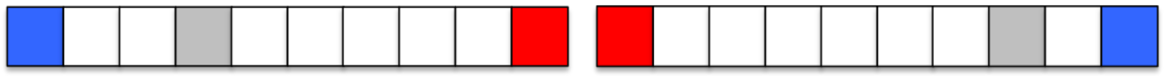

(a) Chain task example 1 (b) Chain task example 2

Figure 3.6: Example tasks for chain problem class. The agent starts in the location shown as a gray square within the chain. High and low reward states shown in red and blue, respectively.

For our implementation of the chain problem class, the agent receives a reward of -1 at each time-step, a small reward of $+10$ upon reaching the closest end of the chain and a large reward of $+1{,}000$ when reaching the farthest end of a chain. We found that setting the value $\delta = 2.0$, to filter macros, worked well for this problem. As a reminder of what constitutes a task in this type of environments, two different different tasks within the chain problem class are shown in Figure 3.6. 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 study this domain as an intuitive example of the type of problems where simple open-loop macros can lead to a substantial improvement in an agent's performance. In this problem class, if the agent only has access to primitive actions, oftentimes the policy will converge to a sub-optimal policy. One reason for this is that primitive actions are often not enough for an agent to visit all ends of the chain in early stages of learning, and so the agent settles for a sub-optimal behavior. However, when given access to an appropriate set of macros, the agent is able to reach both ends of the chain early in its lifetime and learn the optimal policy for a specific task.

Figure 3.7 depicts the mean reward accumulated by the agent while learning how to solve 20 different test chains after using only four tasks for training to generate candidate macros. For training we used chains of 10 states, which are short enough that the optimal policies could be learned easily, and tested on chains of 50 states.

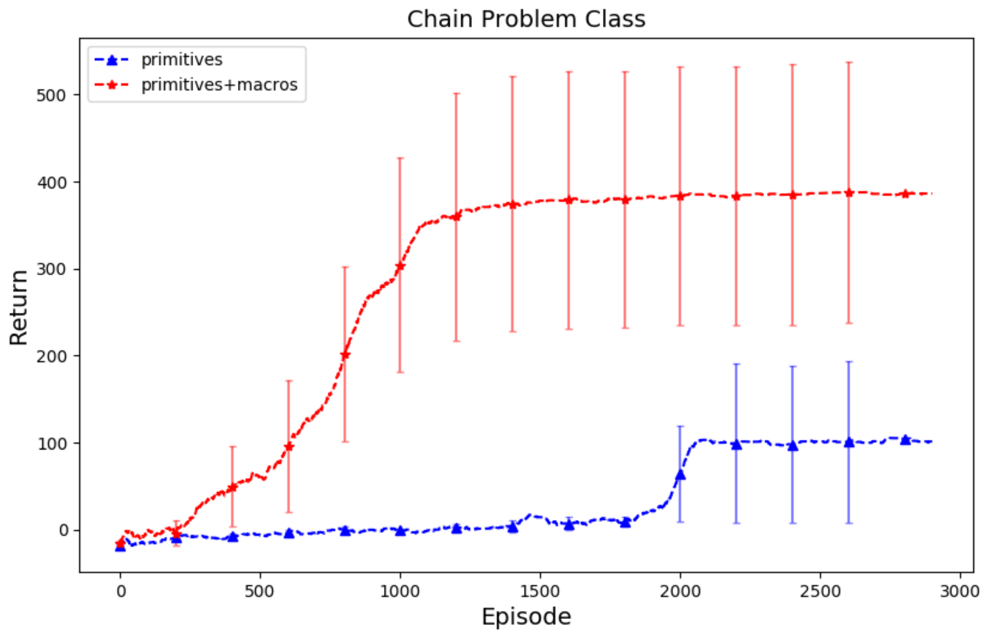


Figure 3.7: Comparison of mean learning curve (and standard error) over 20 randomly generated chains.

The chains were generated by randomly sampling an integer between 0 and 10 for training and between 0 and 50 for testing corresponding to the initial state of the agent. Upon reaching the end of the chain that is closer to the initial state the agent received a reward of +10 and a reward of +1,000 when it reaches the farther end of the chain.

The results show that, on average, the policy of an agent equipped only with primitive actions converges to a sub-optimal behavior, since it hardly ever discovers the farthest end of the chain containing the largest reward. As the action-set of the agent is augmented with macros, the agent's behavior is biased by known well-performing action patterns; thus, facilitating finding an optimal solution to the different tasks.

3.5.4 Gridworld

The previous experiment allowed us to see that using an appropriate set of macros could enable an agent to reach an optimal policy when only having access to primitive actions would fail to do so. We now consider a Gridworld problem (introduced in Section 2.3.1), where the state space is much larger than in the previous tests. We randomly generated a Gridworld of size 60×60 and created task variations by changing the goal and start state of the agent. In this implementation, the agent receives a reward of -1 after each step and a reward of $+100$ upon reaching the goal state. We test robustness to stochastic environments by introducing noise into each executed action: after selecting an action, with probability 0.8 the agent transitions to the intended state and with probability 0.2 it transitions to a state as if an action has been executed at random.

The agent trained on six tasks to generate candidate macros and tested on 20 different tasks, and we set the threshold for filtering macros to $\delta = 2.0$. In each task, the agent trained for $2,000$ episodes, each of which ends when the agent reaches the goal state or after $5,000$ time-steps, whichever happens first.

In this experiment we compare the performance of our method to eigen options and the option-critic architecture, assuming we know the true transition function and the true Q-values on training tasks. This is a strong assumption that we relax in later experiments. To accurately calculate the Q-values, we used Q-learning in tabular form as our learning algorithm.

One key point to notice in this experiment is that the task variations were created by changing the location of the goal and start state, but not the layout of the Gridworld itself. This means the transition graph of the MDP is maintained across tasks, which is the setting assumed by eigen options and the option-critic. We will later show that when this assumption is not true, the benefits of our method become more evident.

Figure 3.8 shows the mean performance and standard error of the agent in the 20 testing tasks. This experiment highlights the benefits for learning of each method over using only primitives and shows that our method results in comparable performance to the competing methods.

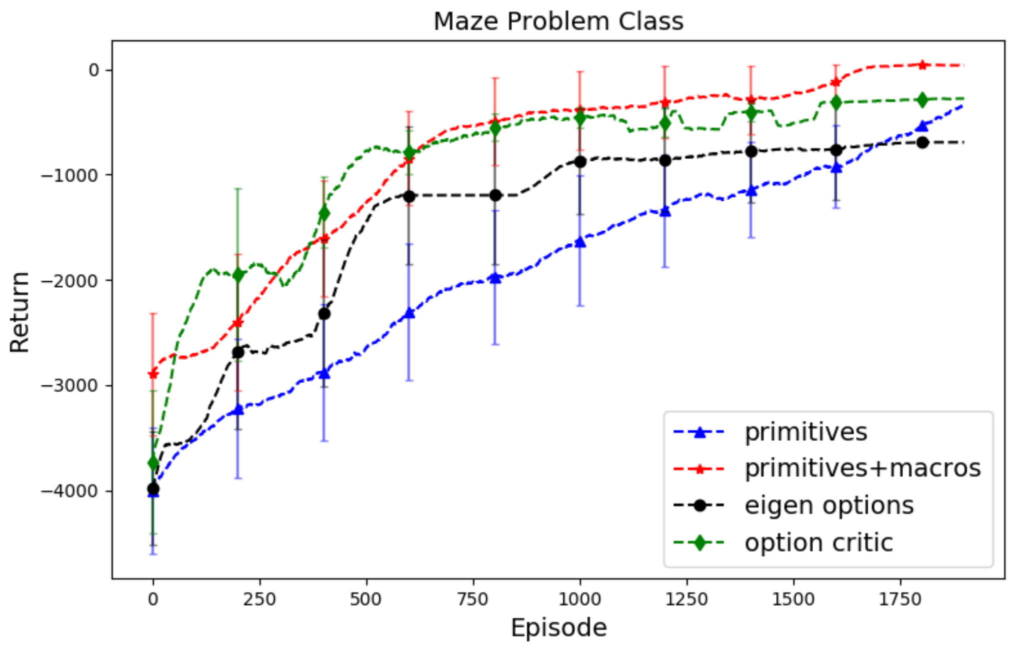


Figure 3.8: Mean performance on 20 testing tasks on maze navigation problem class. Macros evaluated using true Q function and transition function.

To understand why our method is able to outperform simply learning at the primitive level, we visualized the trajectory followed by an agent when choosing actions at random for a period of 1,000 time-steps in a simple Gridworld. This is shown in Figure 3.9. 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. Because many algorithms choose action randomly at early stages of learning, this latter approach allows the agent to learn at a “global” scale early in its lifetime.

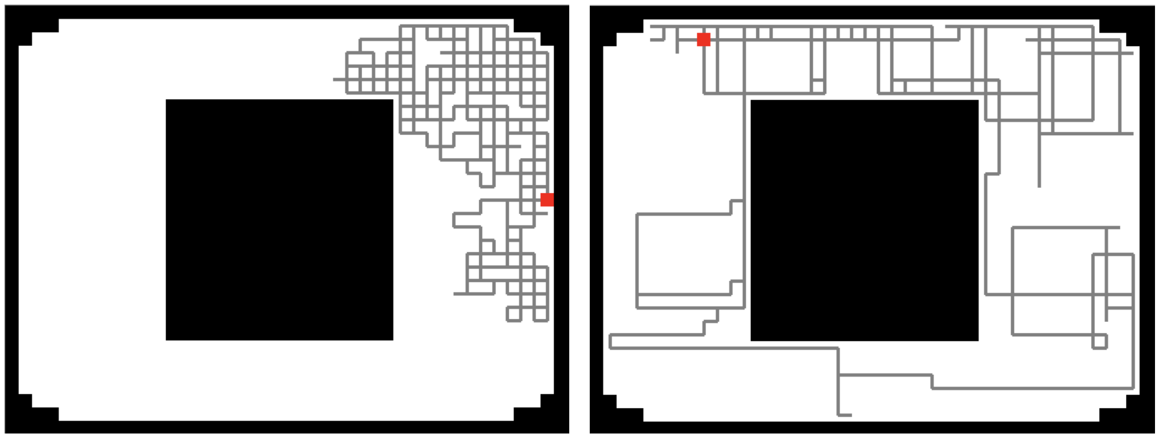


Figure 3.9: Trajectories obtained from pure exploration after 1000 steps using action-sets $\mathcal{A}$ (left) and $\mathcal{A}_{\mathcal{M}'}$ (right). The path taken is shown in gray and the agent's final position is shown in red

3.5.5 Scaling up Results to Large State Spaces

We now evaluate our method in the more interesting case where the agent needs to identify action patterns 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 evident.

In the previous experiments, we were able to precisely calculate the U-values of each macro since the transition probabilities and true Q-values for primitives were known. We now show that the proposed framework is a viable alternative when we use function approximation to estimate the Q-function and we approximate the transition model from data. In all of these experiments, the agent collected (s, a, s') transitions during learning on training tasks, and they were used to fit a model to estimate the transition probabilities $P(s, a, s')$. It is worth noting that in the following problems the transition graphs are not maintained across tasks, making the competing methods not well suited or capable of dealing with this more general learning setting.

(1) Gridworld (approximate): We revisited the gridworld problem class, this time approximating the true Q-values for primitives for the training tasks using DQN and varying tasks, not only by changing the start and goal location, but also randomly generating Gridworld layouts. Because the environments were randomly generated, each task had a different transition graph, preventing the competing methods from exploiting this property.

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 $P(s, a, s')$ by looking at the frequency count.

Figure 3.10 shows the mean learning curve and standard error over 20 randomly sampled environments. This case highlights how effective simple but general macros can be compared to the competing methods.

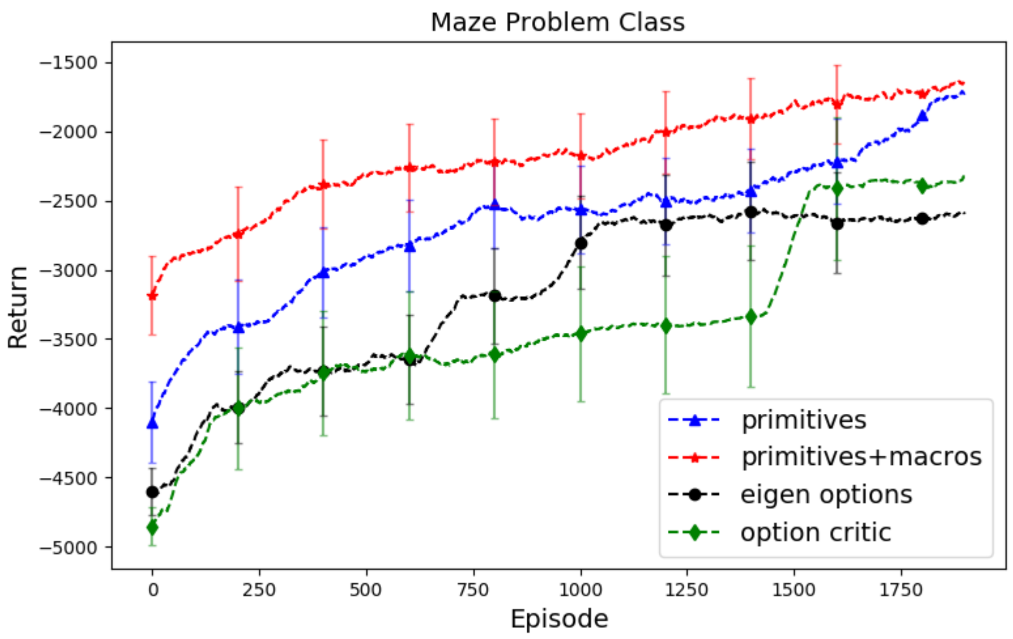


Figure 3.10: Mean performance on 20 testing tasks on maze navigation. The macros were identified by approximating the Q-function with a neural network and approximating the transition function from data.

The eigen option approach fails to generalize to new domains and actually prevents the agent from learning an optimal policy. This seems to validate our intuition; given that the options learned by the eigen option method are specific to the transition graph in which they were learned, the options fail to generalize to new environments. The option critic architecture shows an interesting behavior, improving performance abruptly in a step-wise fashion. This behavior is the result of how the option critic adapts to new tasks; the algorithm learns options based on a specific task (MDP) and, when facing a new task, it needs to unlearn the options that are not useful and adapt them to the new MDP. Because each task has a different transition graph, the options learned for a previous task are not immediately helpful in solving a new one. Our method, on the other hand, identifies macros that are agnostic to the transition graph, leading to temporal abstractions that generalize to the overall problem class.

(2) Animat: In this experiment the agent receives a reward of -1 at each time-step and a reward of $+100$ at the goal state. Given that the state space is continuous, we estimated the transition function by fitting a Gaussian linear model from $10,000$ sampled transitions (s, a, s') that were obtained while learning training tasks. We used four training tasks and tested 10 task variations corresponding to different environments with distinct transition graphs.

It is worth pointing out that this type of problem can be particularly difficult to solve because there are actuator combinations that cancel each other out, preventing the agent from moving from its current state. At early stages of learning, the agent is likely to select such actuator combinations that would not form part of optimal policies. Macros identified using well-performing trajectories are unlikely to include such actions, and so, they will an agent to reach the goal more easily.

(3) Lunar Lander is a control problem taken from OpenAI Gym suite. The agent is tasked with landing a rocket in a platform on an uneven terrain. It has four primitive actions at its disposal: thrust left, thrust right, thrust up or do nothing. The state is represented by the (x, y) position and the (x, y) acceleration in the 2D environment. The task variations vary in the landing location, terrain geometry and the thrust force of the rocket. Because the state is given by continuous variables, but updated at discrete time intervals, the change in thrust results in a change of the transition graph: certain states will never be visited.

Two task variations for lunar lander are shown in Figure 3.11. Notice how the designated landing sites vary in size and sit at different elevations in the terrain; changing considerably the difficulty level of the tasks.

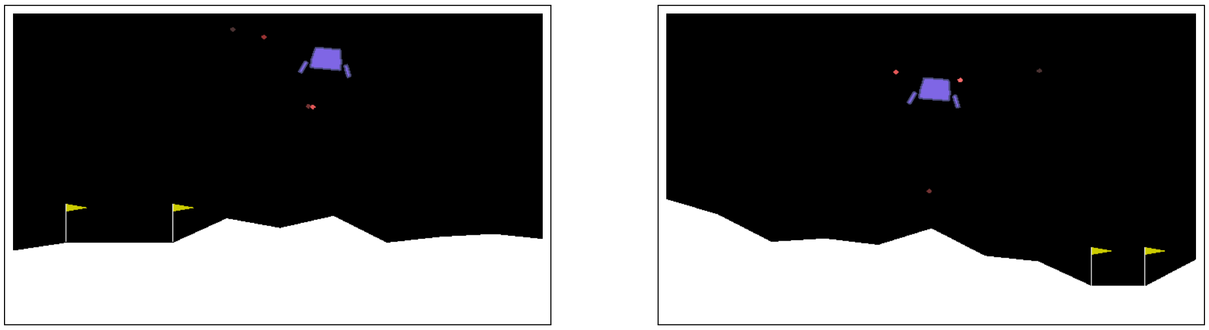


Figure 3.11: Example of task variations in Lunar Lander.

Performance results (according to Eq (3.1)) for each method, in the three domains described above, are reported in Table 3.1. These experiments reflect the scenario where the agent may face many tasks whose transition graphs vary between them. Although we observed an increased variance in performance compared to competing methods, our method results in a clear improvement of the average performance.

Problem Class	Primitives	Primitives+Macro	Eigen-Options	Option-Critic
Maze Nav. (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 3.1: Average performance on test tasks with large state spaces. Larger values are better; they indicate more reward accumulated by the agent during learning.

3.6 Conclusions

In this chapter we have presented a general framework for identifying reusable macros. Inspired by the postulate that the building blocks observed in animal behavior could be the *consequence* of compression, we proposed using compression techniques as a means to identify such building blocks in observed optimal policies for a class of MDPs. Assuming that an agent will be required to solve a large number of related tasks, we can identify repeating action patterns that are associated with high rewards. However, some practical considerations need to be taken into account, as too many macros would hinder an agent's ability to learn. To address this issue we propose ranking the identified macros according to their *utility* for a class of problems by having the agent solve a small number of training tasks. The proposed formulation can be efficiently computed in closed-form; making it a feasible method to be used in practice. We further proposed using KL divergence of the *end-state distribution* of macros as a filter to include only a diverse set of macros in an agent's action set. This ensures that is it possible to represent very diverse behaviors in a compact manner while maintaining a balance on the size of the action set.

The results of our experiments show that, even though macros constitute a simple approach for creating temporal abstraction, the proposed approach is able to identify macros that are sufficiently general for the class of problems that they are able to perform comparably to complex state-of-the-art methods for option discovery when the transition graph is maintained across tasks. If tasks are allowed to differ not only in their reward structure but also in terms of their transition dynamics, however, our framework is able to significantly outperform those competing methods.

One clear limitation of this approach is that it works well under the assumption that actions in a trajectory can be matched exactly, and the sequence of actions are executed blindly; that is, ignoring information obtained in states encountered during execution. In the next chapter, we deal with these limitations by extending the ideas developed in this chapter to the options framework and present an optimization problem analogous to the compression objective proposed for learning macros.

CHAPTER 4

A PROBABILISTIC APPROACH TO OPTIONS DISCOVERY

We have shown that using compression algorithms can be a viable option for identifying macros that define building blocks observed in optimal behaviors. Macros, however, have limitations: they are open-loop (i.e., they ignore information in the states encountered during execution) and, by construction, they must execute the entire sequence of actions until completion. Options, on the other hand, provide more flexibility. While executing an option, the action A_t executed at time t is selected based on the state S_t, according to the option policy. Similarly, the probability of the option terminating execution depends on the state S_t.

This added flexibility makes options an appealing alternative to macros, but it is not clear how one could apply the principles of compression described in the previous chapter to options. Compression techniques assume that it is possible to obtain an *exact* match between the symbols in the message and the ones in the codebook, but since options are defined by a stochastic policy and termination function, it is not clear how compression techniques could be used in this setting.

When considering using compression to identify macros, we notice that in representing a trajectory h, each symbol used in the compressed representation corresponds to a decision made by the agent's policy, π. From this point of view, compressing a trajectory is equivalent to minimizing the number of decisions made by the agent while still representing the original trajectory.

In this chapter we present an optimization objective for learning a set of options that are able to compress a set of trajectories in expectation. To do so, we propose minimizing the expected number of decisions an agent's policy has to make to represent a set of optimal trajectories, while simultaneously maximizing the probability of generating those trajectories.

The experiments in this chapter show that the derived options result in behaviors that specialize to a specific sub-problem of the overall task. This property is highly desirable, since it allows an agent to quickly learn that, in certain situations, some options are not particularly useful while they are highly effective in others.

4.1 Problem Description

Recall that an option $o = (\mathcal{I}_o, \mu_o, \beta_o)$, is a tuple composed by an initiation set $\mathcal{I}_o \subseteq \mathcal{S}$, option policy μ_o, and termination function β_o. In the options framework, at each time-step, t, the agent chooses an action, A_t, based on the current option, O_t, being executed. Let T_t be a Bernoulli random variable, where $T_t = 1$ if the previous option, O_{t-1}, terminated at time t, and $T_t = 0$ otherwise. If $T_t = 1$, O_t is chosen using the policy π. If $T_t = 0$, then the previous option continues, that is, $O_t = O_{t-1}$. We consider the case where the agent's policy π is a policy over options; that is, when the agent makes a decision π selects which option will be run and the action executed is selected by the option's policy. To ensure that we can represent any trajectory, we consider primitive actions to be options which always select one specific action and then terminate; that is, for an option, o, corresponding to a primitive, a, for all $s \in \mathcal{S}$, the termination function would be given by $\beta_o(s) = 1$, and the policy by $\mu_o(s, a') = 1$ if $a' = a$ and 0 otherwise.

Notice that in this formulation, the number of decisions made by the policy over options corresponds to the number of times options terminate, as each termination requires the policy to subsequently choose a new option. Using this perspective, compressing trajectories over options is analogous to minimizing the number of decisions needed to represent those trajectories.

Let $\mathcal{O} = \mathcal{O}_\mathcal{A} \cup \mathcal{O}_\mathcal{O}$ denote a set of options, $\{o_1, \ldots, o_n\}$, where $\mathcal{O}_\mathcal{A}$ refers to the set of options corresponding to primitive actions and $\mathcal{O}_\mathcal{O}$ to the set of options corresponding to temporal abstractions. Furthermore, let H be random variable denoting a trajectory of length $|H|$ generated by a near-optimal policy, and let H_t be a random variable denoting the sub-trajectory of H up to the state encountered at time-step t. We seek to find a set, $\mathcal{O}^* = \{o_1^*, \ldots, o_n^*\}$, that maximizes the following objective:

$$J(\pi, \mathcal{O}) = \mathbf{E}\left[\sum_{t=1}^{|H|} \Pr(T_t = 0, H_t|\pi, \mathcal{O}) + \lambda_1 g(H, \mathcal{O}_{\mathcal{O}})\right], \tag{4.1}$$

where $g(h, \mathcal{O}_{\mathcal{O}})$ is a regularizer that encourages a diverse set of options, and λ_1 is a scalar hyper-parameter. If we are also free to learn the parameters of π, then $\mathcal{O}^* \in \arg\max_{\mathcal{O}} \max_{\pi} J(\pi, \mathcal{O})$. One choice for g is the average KL divergence on a given trajectory over the set of options being learned:

$$g(h, \mathcal{O}_{\mathcal{O}}) = \frac{1}{|\mathcal{O}_{\mathcal{O}}|(|\mathcal{O}_{\mathcal{O}}| - 1)} \sum_{o, o' \in \mathcal{O}_{\mathcal{O}}} \sum_{t=0}^{|h|-1} D_{\mathrm{KL}}\left(\, \mu_o(s_t) || \, \mu_{o'}(s_t)\right).$$

Note that this term is only defined when we consider two or more options; when that is not the case we set this term to 0.

Intuitively, with this objective we seek to find a set of options that terminate as infrequently as possible while still generating near-optimal trajectories with high probability. Given a set of options, a policy over options, and a set sampled trajectories, we can calculate these probabilities *exactly*, allowing us to optimize this objective using gradient-based methods.

4.2 Related Work

In the previous chapter we introduced the option-critic architecture (Bacon et al., 2017) as a strategy for learning options. In this formulation, the options and policy over options are trained to maximize the expected return, but nothing encourages the options to be used for extended periods of time. An issue that often arises with this formulation is that the termination functions of the options collapse to "always terminate", essentially becoming primitive actions. To address this issue, subsequent work proposed associating a cost with switching options (Harb et al., 2018). This proposition allowed the agent to not only learn a set of options, but also encourage their use. However, the performance of this framework relied heavily on finding the right cost to associate with switching options.

Another attempt to address this issue was proposed by Harutyunyan et al. (2019). In this line of work, the authors considered the problem of learning the termination function of options; instead of associating a cost with switching options, they optimized the termination function so that options would only terminate in a localized region of the state space. The authors achieve this objective by minimizing the entropy of the termination function of an option over the state space.

These techniques are inline with the overall idea discussed in this chapter: we want to minimize the number of decisions an agent has to make to solve a task. Nevertheless, these methods do not consider the case where prior experience may exist, and focus on learning options from scratch. Furthermore, the user needs to pre-define the number of options that will be used. In contrast, we assume that we have access to existing knowledge and present a technique for learning options that are applicable to a class of problems and not specialized to a specific task. In addition, we propose a procedure that allows an agent to learn the necessary number of options; avoiding the need to have a user pre-define how many options should be learned.

4.3 Offline Option Learning

The method we propose can be summarized as follows: first, we obtain a set of sample trajectories $\mathcal{H}$ assumed to have been generated by optimal policies (this can be obtained from demonstration or by learning an optimal policy to a number of tasks. Then, we introduce a set of options and a policy over options, and optimize our objective with respect to their parameters. Finally, the resulting options are incorporated to the agent's action set and leveraged to solve new tasks. Given that the options are learned to generate the demonstrated optimal trajectories, we assume that they encode behaviors that are likely to be found in novel but related tasks.

To achieve this goal, one questions remains to be answered: how can we optimize the objective in (4.1)? One approach is to rewrite the objective in terms of the probability of generating a trajectory given a set of options and a policy over options, and in terms of the expected number of terminations given that the agent follows a given trajectory.

To see this, recall that $J(\pi, \mathcal{O}) = \mathbf{E}\left[\sum_{t=1}^{|h|} \Pr(T_t = 0, H_t | \pi, \mathcal{O})\right]$ (we ignore the regularization term for ease of notation). Assuming access to a set $\mathcal{H}$ of sample trajectories, we can estimate J from sample averages and expand an approximation, J_{approx} as follows:

$$\begin{aligned}
J_{approx}(\pi, \mathcal{O}, \mathcal{H}) =& \frac{1}{|\mathcal{H}|} \sum_{h \in \mathcal{H}} \sum_{t=1}^{|h|} \Pr(T_t = 0, H_t = h_t | \pi, \mathcal{O}) \\
=& \frac{1}{|\mathcal{H}|} \sum_{h \in \mathcal{H}} \sum_{t=1}^{|h|} \big(1 - \Pr(T_t = 1 | H_t = h_t, \pi, \mathcal{O})\big) \Pr(H_t = h_t | \pi, \mathcal{O}) \\
=& \frac{1}{|\mathcal{H}|} \sum_{h \in \mathcal{H}} \sum_{t=1}^{|h|} \Big(1 - \mathbf{E}\big[T_t = 1 | H_t = h_t, \pi, \mathcal{O}\big]\Big) \Pr(H_t = h_t | \pi, \mathcal{O})
\end{aligned}$$

It can be seen that to maximize the above expression $\mathbf{E}\big[T_t | H_t = h_t, \pi, \mathcal{O}\big]$ should be minimized while $\Pr(H = h | \pi, \mathcal{O})$ should be maximized. Given that for long trajectories the expected number of terminations increases while the probability of generating the trajectories goes to 0, we normalize the number of terminations by the length of the trajectory, $|h|$, and adjust a hyperparameter, λ_2, to prevent one term from dominating the other during optimization. Based on this observation we propose optimizing the following objective:

$$\hat{J}(\pi, \mathcal{O}, \mathcal{H}) = \frac{1}{\mathcal{H}} \sum_{h_{|h|} \in \mathcal{H}} \lambda_2 \Pr(H = h | \pi, \mathcal{O}) - \frac{\sum_{t=1}^{|h|} \mathbf{E}\left[T_t | H_t = h_t, \pi, \mathcal{O}\right]}{|h|} + \lambda_1 g(H, \mathcal{O}_{\mathcal{O}}). \tag{4.2}$$

Without the scaling λ_2 and $|h|$ (the length of the trajectory), gradient-based method techniques might ignore one of the terms while optimizing $\hat{J}$.

Equation (4.2) can be expressed entirely in terms of π, options $\mathcal{O} = \{o_1, \ldots, o_n\}$ and the transition function, P, allowing us to compute the value of this objective exactly. The two theorems that follow, show that the proposed objective can be expressed in terms of known quantities or estimated quantities (i.e., π, $\mathcal{O}$, and P), allowing us to efficiently calculate and differentiate $\hat{J}$ in order to optimize $\mathcal{O}$.

Theorem 2. Given a set of options $\mathcal{O}$ and a policy π over options, the expected number of terminations for a trajectory h of length $|h|$ is given by:

$$\begin{aligned}\sum_{t=1}^{|h|} \mathbf{E}\left[T_t = 1 | H_t = h_t, \pi, \mathcal{O}\right] = &\sum_{t=1}^{|h|} \Bigg(\sum_{o \in \mathcal{O}} \beta_o(s_t) \big(\mu_o(s_{t-1}, a_{t-1}) \Pr(O_{t-1} = o | H_{t-1} = h_{t-1}, \pi, \mathcal{O})\big) \\ &\times \big(\sum_{o' \in \mathcal{O}} \mu_{o'}(s_{t-1}, a_{t-1}) \Pr(O_{t-1} = o' | H_{t-1} = h_{t-1}, \pi, \mathcal{O})\big)^{-1} \Bigg),\end{aligned}$$

where,

$$\begin{aligned}\Pr(O_{t-1} = o | H_{t-1} = h_{t-1}, \pi, \mathcal{O}) = &\Bigg[\Bigg(\pi(s_{t-1}, o) \beta_o(s_{t-1}) \Bigg) \Bigg(P(s_{t-2}, a_{t-2}, s_{t-1}) \mu_o(s_{t-2}, a_{t-2}) \\ &\times \Pr(O_{t-2} = o | H_{t-2} = h_{t-2}, \pi, \mathcal{O}) (1 - \beta_o(s_{t-1})) \Bigg) \Bigg],\end{aligned}$$

and $\Pr(O_0 = o | H_0 = h_0, \pi, \mathcal{O}) = \pi(s_0, o)$.

Proof. Notice that $\sum_{t=1}^{|h|} \mathbf{E}\left[T_t = 1 | H_t = h_t, \pi, \mathcal{O}\right] = \sum_{t=1}^{|h|} \Pr(T_t = 1 | H_t = h_t, \pi, \mathcal{O})\, 1$, so if we find an expression for $\Pr(T_t = 1 | H_t = h_t, \pi, \mathcal{O})$, we can calculate the expectation exactly. We define $\Pr(T_0 = 1 | H_1 = h_1, \pi, \mathcal{O}) = 1$ for ease of derivation even though there is no option to terminate at T_0. For convenience, as a shorthand notation to denote dependency on $H_t = h_t$, π and $\mathcal{O}$, we will write $\Omega_t = \omega_t$

$$\begin{aligned}\Pr(T_t = 1 | \Omega_t = \omega_t) &= \sum_{o \in \mathcal{O}} \Pr(T_t = 1 | O_{t-1} = o, \Omega_t = \omega_t) \Pr(O_{t-1} = o | \Omega_t = \omega_t) \\ &= \sum_{o \in \mathcal{O}} \beta_o(s_t) \Pr(O_{t-1} = o | \Omega_t = \omega_t) \\ &= \sum_{o \in \mathcal{O}} \beta_o(s_t) \Pr(O_{t-1} = o | \Omega_{t-1} = \omega_{t-1}, A_{t-1} = a_{t-1}, S_t = s_t)\end{aligned}$$

$$
\begin{aligned}
&= \sum_{o\in\mathcal{O}} \beta_o(s_t) \frac{\Pr(S_t = s_t|\Omega_{t-1} = \omega_{t-1}, A_{t-1} = a_{t-1}, O_{t-1} = o)}{\Pr(S_t = s_t|\Omega_{t-1} = \omega_{t-1}, A_{t-1} = a_{t-1})} \\
&\quad \times \Pr(O_{t-1} = o|\Omega_{t-1} = \omega_{t-1}, A_{t-1} = a_{t-1}) \\
&= \sum_{o\in\mathcal{O}} \beta_o(s_t) \frac{\Pr(S_t = s_t|\Omega_{t-1} = \omega_{t-1}, A_{t-1} = a_{t-1})}{\Pr(S_t = s_t|\Omega_{t-1} = \omega_{t-1}, A_{t-1} = a_{t-1})} \\
&\quad \times \Pr(O_{t-1} = o|\Omega_{t-1} = \omega_{t-1}, A_{t-1} = a_{t-1}) \\
&= \sum_{o\in\mathcal{O}} \beta_o(s_t) \Pr(O_{t-1} = o|\Omega_{t-1} = \omega_{t-1}, A_{t-1} = a_{t-1}) \\
&= \sum_{o\in\mathcal{O}} \beta_o(s_t) \big(\Pr(A_{t-1} = a_{t-1}|\Omega_{t-1} = \omega_{t-1}, O_{t-1} = o) \\
&\quad \times \Pr(O_{t-1} = o|\Omega_{t-1} = \omega_{t-1})\big) \big(\Pr(A_{t-1} = a_{t-1}|\Omega_{t-1} = \omega_{t-1})\big)^{-1} \\
&= \sum_{o\in\mathcal{O}} \beta_o(s_t) \frac{\mu_o(s_{t-1}, a_{t-1}) \Pr(O_{t-1} = o|\Omega_{t-1} = \omega_{t-1})}{\Pr(A_{t-1} = a_{t-1}|\Omega_{t-1} = \omega_{t-1})} \\
&= \sum_{o\in\mathcal{O}} \beta_o(s_t) \frac{\mu_o(s_{t-1}, a_{t-1}) \Pr(O_{t-1} = o|\Omega_{t-1} = \omega_{t-1})}{\sum_{o'\in\mathcal{O}} \Pr(A_{t-1} = a_{t-1}, O_{t-1} = o'|\Omega_{t-1} = \omega_{t-1})} \\
&= \sum_{o\in\mathcal{O}} \beta_o(s_t) \mu_o(s_{t-1}, a_{t-1}) \Pr(O_{t-1} = o|\Omega_{t-1} = \omega_{t-1}) \\
&\quad \times \big(\sum_{o'\in\mathcal{O}} \Pr(A_{t-1} = a_{t-1}|O_{t-1} = o', \Omega_{t-1} = \omega_{t-1}) \\
&\quad \times \Pr(O_{t-1} = o'|\Omega_{t-1} = \omega_{t-1})\big)^{-1} \\
&= \sum_{o\in\mathcal{O}} \beta_o(s_t) \frac{\mu_o(s_{t-1}, a_{t-1}) \Pr(O_{t-1} = o|\Omega_{t-1} = \omega_{t-1})}{\sum_{o'\in\mathcal{O}} \mu_{o'}(s_{t-1}, a_{t-1}) \Pr(O_{t-1} = o'|\Omega_{t-1} = \omega_{t-1})}.
\end{aligned}
$$

We now need to find an expression for $\Pr(O_{t-1} = o|\Omega_{t-1} = \omega_{t-1})$ in terms of known probabilities.

$$
\begin{aligned}
\Pr(O_{t-1} = o|\Omega_{t-1} = \omega_{t-1}) =& \big[\Pr(O_{t-1} = o, T_{t-1} = 1|\Omega_{t-1} = \omega_{t-1}) \\
&+ \Pr(O_{t-1} = o, T_{t-1} = 0|\Omega_{t-1} = \omega_{t-1})\big] \\
=& \Big[\big(\Pr(O_{t-1} = o|\Omega_{t-1} = \omega_{t-1}, T_{t-1} = 1) \\
&\times \Pr(T_{t-1} = 1|\Omega_{t-1} = \omega_{t-1})\big) \\
&+ \big(\Pr(O_{t-1} = o|\Omega_{t-1} = \omega_{t-1}, T_{t-1} = 0) \\
&\times (1 - \Pr(T_{t-1} = 1|\Omega_{t-1} = \omega_{t-1}))\big) \Big]
\end{aligned}
$$

$$\begin{aligned}
&= \Big[\big(\pi(s_{t-1}, o) \Pr(T_{t-1} = 1 | \Omega_{t-1} = \omega_{t-1}) \big) \\
&\quad + \big(\Pr(O_{t-1} = o | \Omega_{t-1} = \omega_{t-1}, T_{t-1} = 0) \\
&\quad \times (1 - \Pr(T_{t-1} = 1 | \Omega_{t-1} = \omega_{t-1})) \big) \Big] \\
&= \Big[\big(\pi(s_{t-1}, o) \beta_o(s_{t-1}) \big) \\
&\quad + \big(\Pr(O_{t-1} = o | \Omega_{t-1} = \omega_{t-1}, T_{t-1} = 0)(1 - \beta_o(s_{t-1})) \big) \Big].
\end{aligned}$$

We are now left with figuring out how to calculate $\Pr(O_{t-1} = o | \Omega_{t-1} = \omega_{t-1}, T_{t-1} = 0)$.

$$\begin{aligned}
\Pr(O_{t-1} = o | T_{t-1} = 0, \Omega_{t-1} = \omega_{t-1}) &= \Pr(O_{t-2} = o, A_{t-2} = a_{t-2}, S_{t-1} = s_{t-1} | \Omega_{t-1} = \omega_{t-1}) \\
&= \Pr(A_{t-2} = a_{t-2}, S_{t-1} = s_{t-1} | O_{t-2} = o, \Omega_{t-1} = \omega_{t-1}) \\
&\quad \times \Pr(O_{t-2} = o | \Omega_{t-1} = \omega_{t-1}) \\
&= \Pr(S_{t-1} = s_{t-1} | A_{t-2} = a_{t-2}, O_{t-2} = o, \Omega_{t-1} = \omega_{t-1}) \\
&\quad \times \Pr(A_{t-2} = a_{t-2} | O_{t-2} = o, \Omega_{t-1} = \omega_{t-1}) \\
&\quad \times \Pr(O_{t-2} = o | \Omega_{t-1} = \omega_{t-1}) \\
&= P(s_{t-2}, a_{t-2}, s_{t-1}) \mu_o(s_{t-2}, a_{t-2}) \\
&\quad \times \Pr(O_{t-2} = o | \Omega_{t-1} = \omega_{t-1}) \\
&= P(s_{t-2}, a_{t-2}, s_{t-1}) \mu_o(s_{t-2}, a_{t-2}) \\
&\quad \times \Pr(O_{t-2} = o | \Omega_{t-2} = \omega_{t-2}),
\end{aligned}$$

where $\Pr(O_0 = o | \Omega_0 = \omega_0) = \pi(s_0, o)$.

Using the recursive function $\Pr(O_{t-1} = o | \Omega_{t-1} = \omega_{t-1})$, the expected number of terminations for a given trajectory is given by:

$$\sum_{t=1}^{|h|} \mathbf{E}\left[T_t = 1 | \Omega_t = \omega_t\right] = \sum_{t=1}^{|h|} \left(\sum_{o \in \mathcal{O}} \beta_o(s_t) \frac{\mu_o(s_{t-1}, a_{t-1}) \Pr(O_{t-1} = o | \Omega_{t-1} = \omega_{t-1})}{\sum_{o' \in \mathcal{O}} \mu_{o'}(s_{t-1}, a_{t-1}) \Pr(O_{t-1} = o' | \Omega_{t-1} = \omega_{t-1}} \right),$$

$\square$

Theorem 3. Given a set of options $\mathcal{O}$ and a policy π over options, the probability of generating a trajectory h of length $|h|$ is given by:

$$\Pr(H_{|h|} = h_{|h|}|\pi, \mathcal{O}) = d_0(s_0)\big[\sum_{o \in \mathcal{O}} \pi(s_0, o)\mu_o(s_0, a_0) f(h_{|h|}, o, 1)\big] \prod_{k=0}^{|h|-1} P(s_k, a_k, s_{k+1}),$$

where f is a recursive function defined as:

$$f(h_t, o, i) = \begin{cases} 1, & \text{if } i = t \\ \Bigg[\Bigg(\beta_o(s_i) \sum_{o' \in \mathcal{O}} \pi(s_{i+1}, o')\mu_{o'}(s_{i+1}, a_{i+1}) f(h_t, o', i+1)\Bigg) & \\ +\Bigg((1 - \beta_o(s_i))\mu_o(s_{i+1}, a_{i+1}) f(h_t, o, i+1)\Bigg)\Bigg] & \text{otherwise.} \end{cases}$$

Proof. We define $H_{i,t}$ to be the history from time i to time t, that is, $H_{i,t} = (S_i, A_i, S_{i+1}, A_{i+1}, \ldots, S_t)$, where $i < t$. If $i = t$, the history would contain a single state.

$$\begin{aligned}
\Pr(H_t = h_t|\pi, \mathcal{O}) =& \Pr(S_0 = s_0|\pi, \mathcal{O}) \Pr(H_{1,t} = h_{1,t}, A_0 = a_0|S_0 = s_0, \pi, \mathcal{O}) \\
=& d_0(s_0) \Pr(H_{1,t} = h_{1,t}, A_0 = a_0|S_0 = s_0, \pi, \mathcal{O}) \\
=& d_0(s_0) \sum_{o \in \mathcal{O}} \Pr(H_{1,t} = h_{1,t}, A_0 = a_0, O_o = o|S_0 = s_0, \pi, \mathcal{O}) \\
=& d_0(s_0) \sum_{o \in \mathcal{O}} \Pr(O_0 = o|S_0 = s_0, \pi, \mathcal{O}) \\
& \times \Pr(H_{1,t} = h_{1,t}, A_0 = a_0|S_0 = s_0, O_0 = o, \pi, \mathcal{O}) \\
=& d_0(s_0) \sum_{o \in \mathcal{O}} \pi_(s_0, o) \Pr(H_{1,t} = h_{1,t}, A_0 = a_0|S_0 = s_0, O_0 = o, \pi, \mathcal{O}) \\
=& d_0(s_0) \sum_{o \in \mathcal{O}} \pi_(s_0, o) \Pr(A_0 = a_o|S_0 = s_0, O_0 = o, \pi, \mathcal{O}) \\
& \times \Pr(H_{1,t} = h_{1,t}|S_0 = s_0, O_0 = o, A_0 = a_0, \pi, \mathcal{O}) \\
=& d_0(s_0) \sum_{o \in \mathcal{O}} \pi_(s_0, o)\mu_o(s_0, a_o) \\
& \times \Pr(H_{1,t} = h_{1,t}|S_0 = s_0, O_0 = o, A_0 = a_0, \pi, \mathcal{O}).
\end{aligned}$$

We now need to find an expression to calculate $\Pr(H_{1,t} = h_{1,t}|S_0 = s_0, O_0 = o, A_0 = a_0, \pi, \mathcal{O})$. Consider the probability of seeing history $h_{i,t}$ given the previous state, s, the previous option, o, and the previous action, a:

$$
\begin{aligned}
&\Pr(H_{i,t} = h_{i,t}|S_{i-1} = s, O_{i-1} = o, A_{i-1} = a)\\
=&\Pr(S_i = s_i|S_{i-1} = s, O_{i-1} = o, A_{i-1} = a)\\
&\times \Pr(H_{i+1,t} = h_{i+1,t}, A_i = a_i|S_{i-1} = s, O_{i-1} = o, A_{i-1} = a, S_i = s_i)\\
=&P(s, a, s_i) \Pr(H_{i+1,t} = h_{i+1,t}, A_i = a_i|S_{i-1} = s, O_{i-1} = o, A_{i-1} = a, S_i = s_i)\\
=&P(s, a, s_i) \Pr(H_{i+1,t} = h_{i+1,t}, A_i = a_i|O_{i-1} = o, A_{i-1} = a, S_i = s_i)\\
=&P(s, a, s_i)\big[\Pr(T_i = 1|O_{i-1} = o, A_{i-1} = a, S_i = s_i)\\
&\times \Pr(H_{i+1,t} = h_{i+1,t}, A_i = a_i|O_{i-1} = o, A_{i-1} = a, S_i = s_i, T_i = 1)\\
&+ \Pr(T_i = 0|O_{i-1} = o, A_{i-1} = a, S_i = s_i)\\
&\times \Pr(H_{i+1,t} = h_{i+1,t}, A_i = a_i|O_{i-1} = o, A_{i-1} = a, S_i = s_i, T_i = 0)\big]\\
=&P(s, a, s_i)\big[\beta_o(s_i)\\
&\times \Pr(H_{i+1,t} = h_{i+1,t}, A_i = a_i|O_{i-1} = o, A_{i-1} = a, S_i = s_i, T_i = 1)\\
&+ (1 - \beta_o(s_i))\\
&\times \Pr(H_{i+1,t} = h_{i+1,t}, A_i = a_i|O_{i-1} = o, A_{i-1} = a, S_i = s_i, T_i = 0)\big].
\end{aligned}
$$

Even though the equation above might seem complicated, there are only two cases we need to consider: either the current option terminates and a new one must be selected (the first term), or the current option does not terminate (the second term). Let's consider each of them separately.

Case 1 - option terminates: If we terminate, we sum over new options:

$$
\begin{aligned}
&\Pr(H_{i+1,t} = h_{i+1,t}, A_i = a_i|O_{i-1} = o, A_{i-1} = a, S_i = s_i, T_i = 1)\\
=&\sum_{o' \in \mathcal{O}} \Pr(O_i = o'|O_{i-1} = o, A_{i-1} = a, S_i = s_i, T_i = 1)\\
&\times \Pr(H_{i+1,t} = h_{i+1,t}, A_i = a_i|O_{i-1} = o, A_{i-1} = a, S_i = s_i, T_i = 1, O_i = o')
\end{aligned}
$$

$$
\begin{aligned}
&= \sum_{o' \in \mathcal{O}} \pi(s_i, o') \Pr(H_{i+1,t} = h_{i+1,t}, A_i = a_i | O_{i-1} = o, A_{i-1} = a, S_i = s_i, T_i = 1, O_i = o') \\
&= \sum_{o' \in \mathcal{O}} \pi(s_i, o') \Pr(H_{i+1,t} = h_{i+1,t}, A_i = a_i | S_i = s_i, O_i = o') \\
&= \sum_{o' \in \mathcal{O}} \pi(s_i, o') \Pr(A_i = a_i | S_i = s_i, O_i = o') \Pr(H_{i+1,t} = h_{i+1,t} | S_i = s_i, O_i = o', A_i = a_i) \\
&= \sum_{o' \in \mathcal{O}} \pi(s_i, o') \mu_{o'}(s_i, a_i) \Pr(H_{i+1,t} = h_{i+1,t} | S_i = s_i, O_i = o', A_i = a_i).
\end{aligned}
$$

Note that $\Pr(H_{i,t} = h_{i,t} | S_{i-1} = s, O_{i-1} = o, A_{i-1} = a)$ has the same form as the expanded probability.

Case 2 - option does not terminate: This tells us that $O_i = o$, so we may drop the dependency on the $i-1$ terms:

$$
\begin{aligned}
&\Pr(H_{i+1,t} = h_{i+1,t}, A_i = a_i | S_{i-1} = s, O_{i-1} = o, A_{i-1} = a, S_i = s_i, T_i = 0) \\
&= \Pr(H_{i+1,t} = h_{i+1,t}, A_i = a_i | S_i = s_i, O_i = o) \\
&= \Pr(A_i = a_i | S_i = s_i, O_i = 0) \Pr(H_{i+1,t} = h_{i+1,t} | S_i = s_i, O_i = o, A_i = a_i) \\
&= \mu_o(s_i, a_i) \Pr(H_{i+1,t} = h_{i+1,t} | S_i = s_i, O_i = o, A_i = a_i).
\end{aligned}
$$

Plugging these two cases back into our earlier equation yields:

$$
\begin{aligned}
&\Pr(H_{i,t} = h_{i,t} | S_{i-1} = s, O_{i-1} = o, A_{i-1} = a) \\
&= P(s, a, s_i) \big[\beta_o(s_i) \sum_{o' \in \mathcal{O}} \pi(s_i, o') \mu_{o'}(s_i, a_i) \Pr(H_{i+1,t} = h_{i+1,t} | S_i = s_i, O_i = o', A_i = a_i) \\
&\quad + (1 - \beta_o(s_i)) \mu_o(s_i, a_i) \Pr(H_{i+1,t} = h_{i+1,t} | S_i = s_i, O_i = o, A_i = a_i) \big].
\end{aligned}
$$

Note that each term contains an expression of the same form, $\Pr(H_{i,t} = h_{i,t} | S_{i-1} = s, O_{i-1} = o, A_{i-1} = a)$. We can therefore compute the probability recursively. Our recursion will terminate when we consider $i = t$, as $H_{t,t}$ contains a single state, and we adopt the convention of its probability to be 1. Notice that for every recursive step, both inner terms will produce a $P(s, a, s_i)$ term. Consider the result when we factor every recursive $P(s, a, s_i)$ term to the front of the equation. We define the following recursive function:

$$f(h_t, o, i) = \begin{cases} 1, & \text{if } i = t \\ \Big[\beta_o(s_i) \sum_{o' \in \mathcal{O}} \pi(s_{i+1}, o')\mu_{o'}(s_{i+1}, a_{i+1}) f(h_t, o', i+1) & \\ +(1 - \beta_o(s_i))\mu_o(s_{i+1}, a_{i+1}) f(h_t, o, i+1)\Big], & \text{otherwise.} \end{cases}$$

Notice that this is the recursive probability described above, but with the $P(s, a, s')$ terms factored out. We now see that:

$$\Pr(H_{i,t} = h_{i,t} | S_{i-1} = s_{i-1}, O_{i-1} = o, A_{i-1} = a_{i-1}) = f(h_t, o, i) \prod_{k=i-1}^{t-1} P(s_k, a_k, s_{k+1}).$$

Plugging this all the back into our original equation for $\Pr(H_t = h_t | \pi, \mathcal{O})$ gives us the desired result:

$$\Pr(H_{|h|} = h_{|h|} | \pi, \mathcal{O}) = d_0(s_0) \left[\sum_{o \in \mathcal{O}} \pi(s_0, o)\mu_o(s_0, a_o) f(h_{|h|}, o, 1) \right] \prod_{k=0}^{t-1} P(s_k, a_k, s_{k+1}).$$

□

The theorems above tell us that we are able to express the objective in (4.2) entirely in terms of the option policies, μ, termination functions, β, and policy over options, π. Given a parametric representation of these functions we can differentiate (4.2) with respect to their parameters and optimize with any standard optimization technique.

4.3.1 Learning Options Incrementally

One common issue that often arises in option discovery is identifying how many options are needed for a given problem. Oftentimes this number is pre-defined by the user based on intuition. In such a scenario, one could simply initialize a pre-defined number of options and adapt the parameters of those options by optimizing the proposed objective in (4.2). Instead, we propose not only learning options, but also an appropriate number of options, by following the procedure shown in Algorithm 3. In the pseudocode, we assume a parametric model for the policy, π_θ, over options with parameters θ, for the option policy, μ_ϕ, with parameters ϕ and for the termination function β_ψ. We update the models with gradient ascent using a learning rate α for N epochs, before attempting to include a new option.

Algorithm 3 Option Learning Framework - Pseudocode

```
1: Collect set of trajectories $\mathcal{H}$
2: Initialize option set $\mathcal{O}$ with primitive options
3: done = false
4: $\hat{J}_{prev} = -\infty$
5: while done == false do
6:     Initialize new option $o' = (\mu'_\phi, \beta'_\psi)$, initializing parameters for $\phi$ and $\psi$.
7:     $\mathcal{O}' = \mathcal{O} \cup o'$
8:     Initialize parameters $\theta$ of policy $\pi_\theta$
9:     for k=1,...,n do
10:        $\hat{J}_k = \hat{J}(\pi_\theta, \mathcal{O}', \mathcal{H})$
11:        $\theta = \theta + \alpha \frac{\partial \hat{J}_k}{\partial \theta}$
12:        $\phi = \phi + \alpha \frac{\partial \hat{J}_k}{\partial \phi}$
13:        $\psi = \psi + \alpha \frac{\partial \hat{J}_k}{\partial \psi}$
14:    if $\hat{J}_N - \hat{J}_{prev} < \Delta$ then
15:        done = true
16:    else
17:        $\mathcal{O} = \mathcal{O}'$
18:        $\hat{J}_{prev} = \hat{J}_N$
19: Return new option set $\mathcal{O}$
```

The algorithm introduces one option at a time and updates the policy π, and the termination function and option policy of the newly introduced option, o', for n. Any previously introduced option is kept fixed. After the new option is trained, we measure how much $\hat{J}$ has improved. If it fails to improve above some threshold, Δ, the procedure terminates. This results in a natural way of obtaining an appropriate number of options to represent the observed trajectories.

4.4 Experimental Results

In this section we describe experiments used to evaluate the proposed offline option learning approach. To be able to visualize and understand the options produced by our algorithm, we will show results in the "four rooms" domain—a gridworld composed of four large open areas connected by a small doorway—and we compare the options discovered by our method against options generated by the option-critic architecture (Bacon et al., 2017) and eigen options (Marlos C. Machado, 2017). We then extend our experiments to assess the performance of the technique in a few select problems from the Atari 2600 emulator provided by OpenAI Gym. These experiments demonstrate that the options learned by our method focus on specific sub-tasks within the larger problem, and are capable of improving performance over primitive actions and options produced by competing approaches.

4.4.1 Empirical Validation of Derived Equations

Before delving into the experiments, we corroborate that the derivation of the proposed objective and its implementation is correct. We conducted a simple empirical test to compare the calculated expected number of decisions in a trajectory and the probability of generating each trajectory for a set of 10 trajectories on 10 MDPs. The MDPs are chains of seven states with different transition functions. We randomly initialized four options and a policy over options, and estimated the probability of generating each trajectory and the expected number of terminations, for each sampled trajectory, by Montecarlo sampling $10,000$ trials. Table 4.1 presents results for the 10 trajectories verifying empirically that the equations were correctly derived and implemented. The table compares the empirical and true probability of generating a given trajectory, $\hat{\Pr}(H|\cdot)$ and $\Pr(H|\cdot)$ (first term in (4.2)), and the empirical and true sum of expected number of decisions an agent has to make to generate those trajectories, $\sum_{t=1}^{|H|} \hat{\mathbf{E}}\left[T_t|\cdot\right]$ and $\sum_{t=1}^{|H|} \mathbf{E}\left[T_t|\cdot\right]$ (second term in (4.2)), where the "hat" operator indicates an empirical estimate of the indicated quantity.

H	$\hat{\Pr}(H\|\pi, \mathcal{O})$	$\Pr(H\|\pi, \mathcal{O})$	$\sum_{t=1}^{\|H\|} \hat{\mathbf{E}}\left[T_t\|H_t, \pi, \mathcal{O}\right]$	$\sum_{t=1}^{\|H\|} \mathbf{E}\left[T_t\|H_t, \pi, \mathcal{O}\right]$
h_1	0.0932	0.0957	3.060	3.178
h_2	0.0158	0.0173	4.139	4.154
h_3	0.2149	0.2122	1.965	2.178
h_4	0.0995	0.0957	2.979	3.178
h_5	0.0962	0.0957	3.024	3.178
h_6	0.1354	0.1384	2.9579	3.1596
h_7	0.00040	0.00038	9.750	8.794
h_8	0.1854	0.1881	2.820	3.072
h_9	0.0379	0.0368	4.2612	4.4790
h_{10}	0.1864	0.1881	2.8404	3.0723

Table 4.1: Validation of equations and implementation.

Note that the cases with largest discrepancy between the estimated and calculated number of terminations occur when the probability of generating a trajectory is low. This happens because we are estimating this number via Monte Carlo sampling; since this quantity is conditioned on the trajectory being generated, a low probability trajectory makes accurate estimation difficult given the small number of samples being produced.

4.4.2 Experiments on Four Rooms Environment

We tested our approach in the four rooms domain: a gridworld of size 40×40, in which the agent is placed in a randomly selected start state and needs to reach a randomly selected goal state. At each time-step, the agent executes one of four available actions: moving left, right, up or down, and receives a reward of -1. Upon reaching the goal state, the agent receives a reward of $+10$. We generated 30 different task variations (by changing the goal and start location) and collected six sample trajectories from optimal policies, learned using Q-learning, from six different start and goal configurations. We evaluated our method on the remaining 24 tasks.

Each option was represented as a two-layer neural network, with 32 neurons in each layer, and two output layers: a softmax output layer over the four possible actions representing μ, and a separate sigmoid layer representing β. We implemented our objective using PyTorch which simplifies gradient calculation through automatic differentiation, and we used the tabular form of Q-learning with ϵ-greedy exploration.

Figure 4.1 shows the change in the average expected number of terminations and average probability of generating the observed trajectories while learning options, as new options are introduced and adapted to the sampled trajectories. Options were learned over the six sampled optimal trajectories and every 50 epochs a new option was introduced to the option set, for a total of 4 options. For every new option, the change in probability of generating the observed trajectories as well as the change in expected number of decisions reaches a plateau after 30 or 40 training epochs. When a new option is introduced, there is a large jump in the loss because a new policy, π, is initialized arbitrarily to account for the new option set being evaluated. However, after training the new candidate option, the overall loss improves beyond what it was possible before introducing the new option.

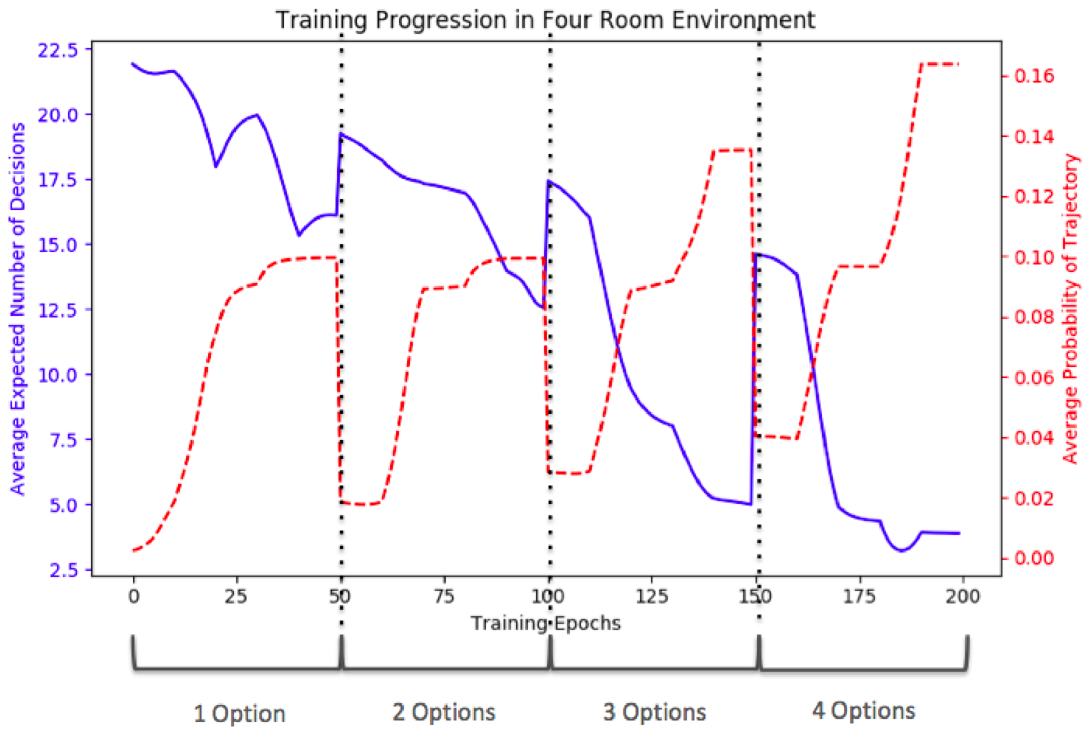


Figure 4.1: Visualization of loss for sampled trajectories in four rooms domain over 200 training epochs. Every 50 training epochs a new option is introduced. For a given set of sampled trajectories, the decreasing average number of decisions made by π is shown in blue and the increasing probability of generating the observed trajectories is shown in red.

In Figure 4.2, we compare the performance of Q-learning on 24 novel test tasks (randomly selected start and goal states) using options discovered from offline option learning (with and without regularization using KL divergence), eigenoptions, and option critic. We allowed each competing method to learn options from the same six training tasks and, to ensure a fair comparison, we used the original code provided by the authors. As baselines, we also compare against primitive actions and randomly initialized options.

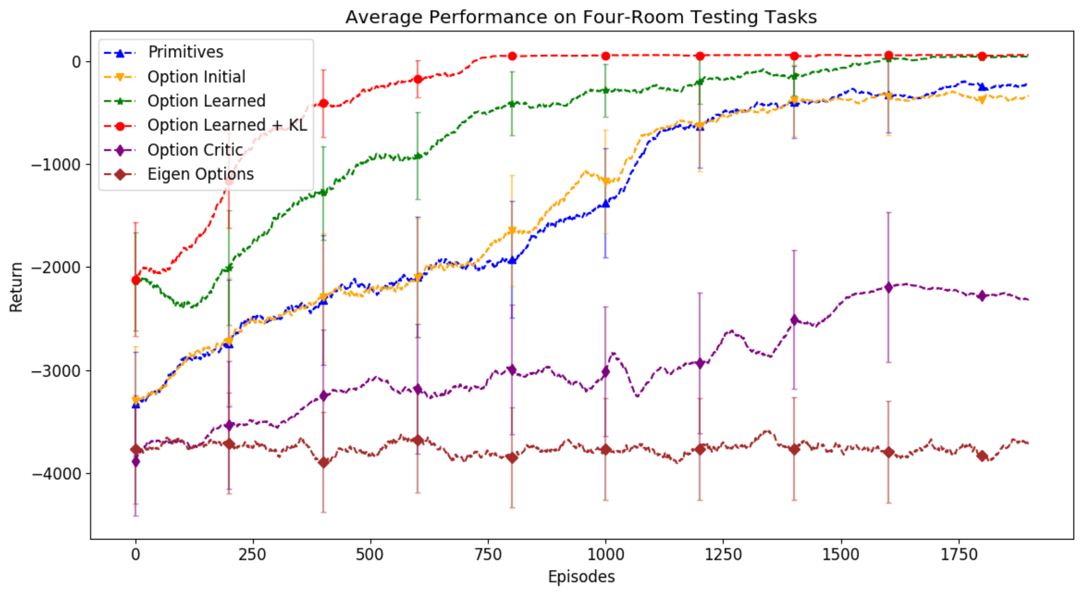


Figure 4.2: Performance Comparison on four rooms domain. Six tasks were used for training and 24 different tasks for testing. The plot shows the average return (and standard error) on the y-axis as a function of the episode number on the test tasks. For our proposed method we set $\lambda_2 = 100.0$ and $\lambda_1 = 0.001$ when using KL regularization.

It might seem surprising that both eigen-options and the option-critic failed to reach an optimal policy when they were shown to work well in this type of problem; for that we offer the following explanation. Our implementation of four rooms is defined in a much larger state space than the ones where these methods were originally tested, making each individual room much larger. Since the options identified by these methods tend to lead the agent from room to room, it is possible that, once in the correct room, the agent executes an option leading to a different room before it had the opportunity to find the goal. When testing our approach in the smaller version of the four room problem, we found no clear difference to the performance of the competing methods. In this setting, the options learned by our method found an optimal policy in all testing tasks. We set the threshold Δ for introducing a new option to 10% of $\hat{J}$ at the previous iteration and the hyperparameter $\lambda_2 = 100.0$. When adding KL regularization, we set $\lambda_1 = 0.001$.

To understand the reason behind the improvement in performance resulting from offline option learning, we turn the reader's attention to Figure 4.3.

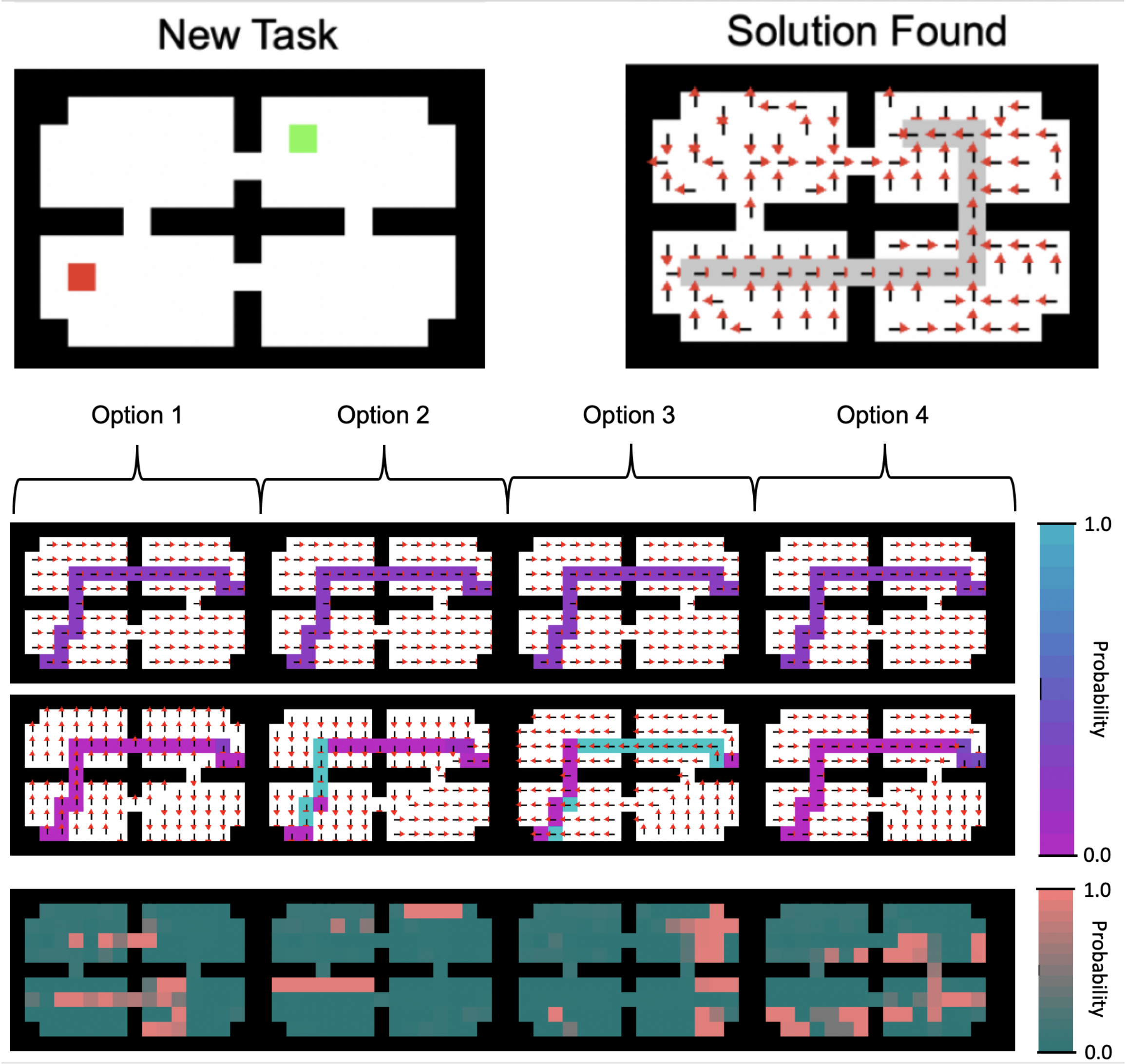


Figure 4.3: Visualization of our framework for learning options in four rooms domain.

The figure shows a visualization of how our method learns options from sampled trajectories and is able to reuse them in new problems. A novel task is seen in the top left, where the agent (red) has to navigate to a goal (green). On the top right, we show the solution found by the agent. The three rows below show how the options were learned and exploited in the new task. The highlighted area in the top two rows show a sample trajectory and the color corresponds to the probability that the option would take the demonstrated action. Notice that this trajectory does not start from the start state of the new task, since the trajectory was generated on a previous task. The arrows show the action that is most likely at each state. Before training (first row), each option is randomly initialized, but after training (second row) each option specializes in a specific skill (a navigation pattern). In this example, the demonstrated trajectory can be generated with high probability by using option 3 and 2. The last row shows a heat-map indicating where each options is likely to be called by the policy learned in the new task. As can be seen, the agent learns to use each option in very specific situations; that is, they are specialized. For example, option 1 is likely to be called to make the agent move up, if it is located in one of the bottom rooms.

4.4.3 Experiments on Atari Domain

We evaluated the quality of the options learned by our framework in two different Atari 2600 games: Breakout and Amidar. Because the games did not easily provide a way of modifying the environments, we created task variations by performing a random walk before the agent was allowed to start learning, thus changing the initial state distribution, and varying the number of frames a time-step in the MDP lasted. These variations result in different tasks varying in their transition function and initial state distributions.

We trained the policy over options using the algorithm Asynchronous Advantage Actor Critic (A3C) (Mnih et al., 2016). This is a parallel algorithm that uses function approximation and has been shown to work well for large scale problems. The state of the agent was represented as grayscale images and 12 trajectories were sampled for training. Each trajectory lasted until a life was lost, not for the entire duration of the episode. The options policy and termination functions were implemented as two-layer neural networks, where the input was give as a stack of grayscale images of the last two frames. We ran 32 training agents in parallel on CPUs, the learning rate was set to 0.0001 and the discount factor γ was set to 0.99.

Because the options can only learn from the states observed in the trajectories, it is possible that when using them, they will be executed in previously unseen states. When this happens, the termination function may decide to never terminate, as it has not been trained in that region of the state space before. To address this issue, we add a value of 0.05 to the predicted probability of termination for each time-step that the option has been running. Therefore, in our experiments an option cannot run for more than 20 time-steps in total.

4.4.3.1 Breakout

In this type of problem, the agent's goal is to control a paddle to prevent the ball from falling while knocking down every block. After the ball makes contact with one of the blocks the agent receives a positive reward of $+1.0$ and a large penalty of -10.0 if it fails to catch the ball. If the agent drops the ball it loses a life and, after three lives are lost, the episode terminates and the paddle is placed in a random initial position. There are three possible actions: move right, move left and fire the ball off the paddle (this action is only useful at the beginning of a life).

Figure 4.4 shows an example of this class of problems.

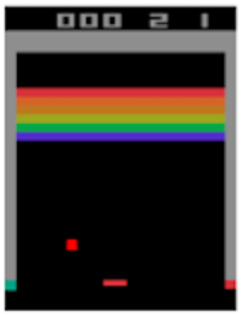


Figure 4.4: Example observation in a task of Breakout. A state is represented by the last four observations in grayscale.

Figure 4.5 shows the performance of an agent as a function of training time in Breakout. Given appropriate hyperparameters, the options learned by our method allow the agent to solve tasks much more quickly than if it only had access to primitive actions. Furthermore, notice that using the options as initialized (before being trained) decreases performance; showing that our objective is capable learning appropriate options. In this scenario setting the hyperparameters to $\lambda_2 = 5,000$ and $\lambda_1 = 0.01$ led to the best improvement in performance.

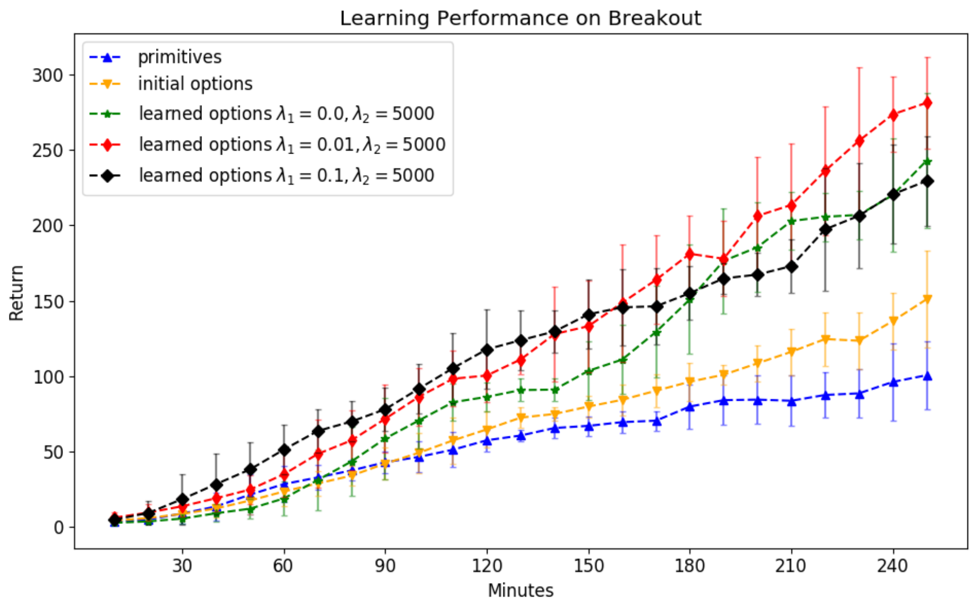


Figure 4.5: Average returns on Breakout comparing primitives (blue), options before training (orange) and learned options for different values of λ_1 and λ_2. The shaded region indicates the standard error.

Figure 4.6 depicts the behavior for one of the learned options on Breakout. The option is efficient at catching the ball after it bounces on the left wall, and then terminates with high probability before the ball has to be caught again. Bear in mind that the option remains active for many time-steps, significantly reducing the number of decisions made by the policy over options. However, it does not maintain control for so long that the agent is unable to respond to changing circumstances.

This option is only useful in this specific case; for example, it was not helpful in returning a ball bounced off the right wall. That is to say, the option specialized in a specific sub-task within the larger problem: a highly desirable property for options to be able to generalize well to new environments.

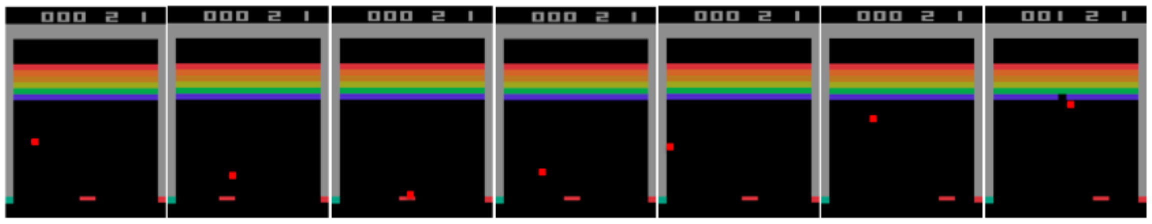


Figure 4.6: Visualization of a learned option executed until termination on Breakout. The option learned to catch the ball bouncing off the left wall and terminates with high probability before the ball bounces a wall again (ball size increased for visualization).

4.4.3.2 Amidar

In this problem, the agent's goal is to traverse every segment of a maze populated with enemies and avoid coming into contact with the enemies, which makes the agent lose a life. The agent receives a positive reward for each segment the it covers and a penalty if it makes contact with an enemy. After losing a life the agent resets to an initial position and after losing three lives the episode terminates. The state is represented as a stack of grayscale images of the last four time-steps. Figure 4.7 shows an example of a task in Amidar.

Results for Amidar are shown in Figure 4.8. Just as was the case with Breakout, an appropriate setting of hyperparameters allowed the agent to learn options that resulted in a clear improvement in learning performance. In this scenario, setting the hyperparameters to $\lambda_2 = 5,000$ and $\lambda_1 = 0.1$ resulted in a substantial improvement over simply using primitive actions. Notice, again, how using the options as they were initialized does not lead to any improvement in performance.

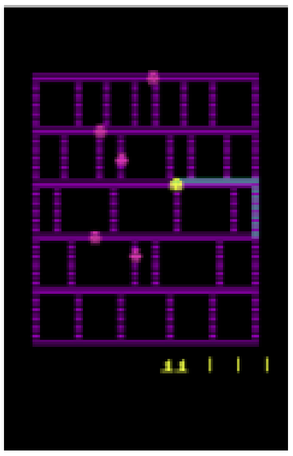


Figure 4.7: Example observation in a task of Amidar. A state is represented by the last four observations in grayscale.

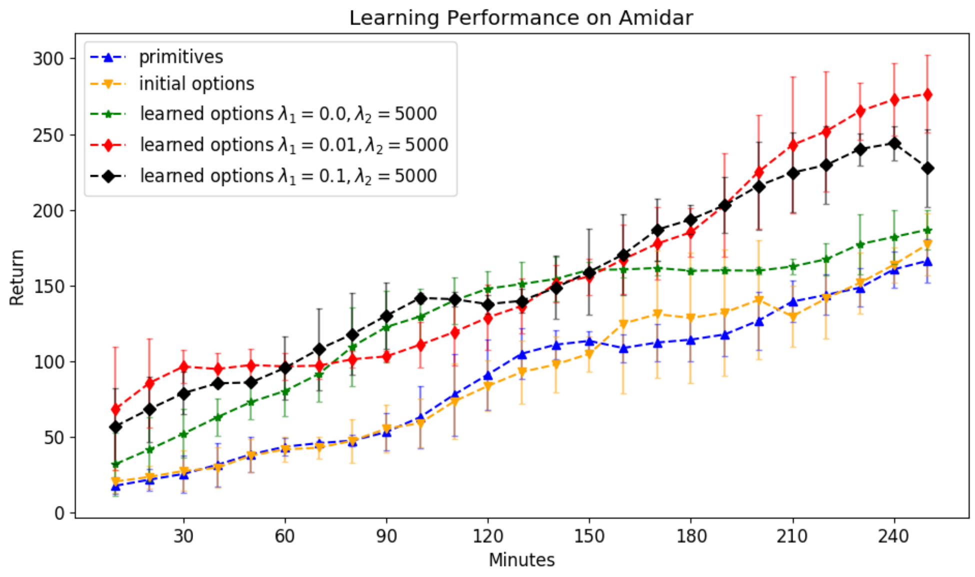


Figure 4.8: Average returns on Amidar comparing primitives (blue), options before training (orange) and learned options for different values of λ_1 and λ_2. The shaded region indicates the standard error.

Figure 4.9 shows the selection of two of the options learned for Amidar when starting a new game. At the beginning of the game, option 1 is selected, which takes the agent to a specific intersection before terminating. The agent then selects option 2, which chooses a direction at the intersection, follows the resulting path, and terminates at the next intersection. Note that the agent does not need to repeatedly select primitive actions in order to simply follow a previously chosen path. Having access to these types of options enable an agent to easily replicate known good behaviors, allowing for faster and more meaningful exploration of the state space.

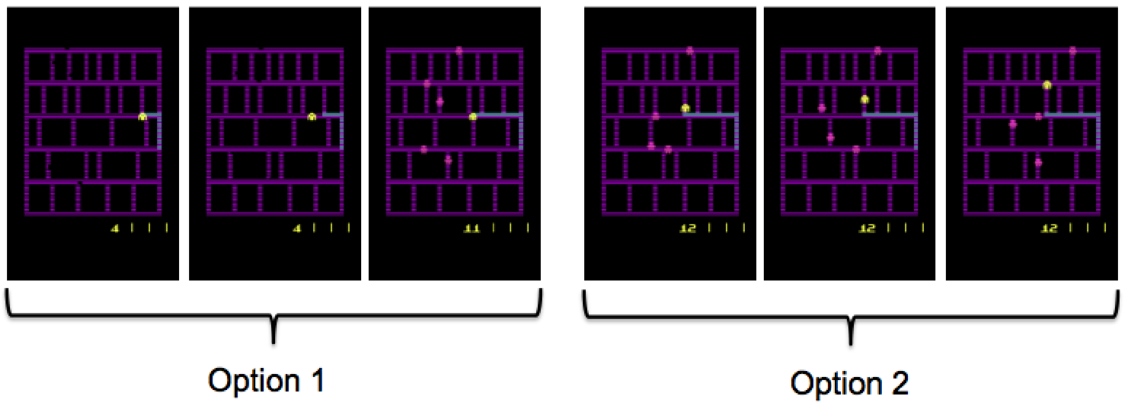


Figure 4.9: Visualization of two learned options on Amidar. The agent is shown in yellow and enemies in pink. Option 1 learned to move up, at the beginning of the game, and turn left until getting close to an intersection. Option 2 learned to turn in that intersection and move up until reaching the next one.

Note that these results do not necessarily show that the options allow the agent to learn a better final policy, but rather that the learned options are able to guide early in its lifetime resulting in much more effective learning.

4.5 Conclusions

In this chapter we have used the insights obtained in the previous chapter and addressed the limitations discussed for macros. By using options, we are able to learn temporal abstractions that are capable of selecting actions and when to terminate based on the state of the agent. We showed that specialized behaviors can be learned by adapting the notion of compression, developed in the previous chapter, to options and presented an optimization objective for learning options offline from demonstrations. Optimizing the objective results in a set of options that allows an agent to reproduce the observed behavior while minimizing the number of decisions made by the policy over options. The experiments show that the learned options not only are able to improve the learning ability of the agent on new tasks, but are also able to outperform competing methods.

There are some clear directions for future development. While we have shown that our method is capable of discovering powerful options, properly tuning the hyperparameters, λ_1 and λ_2, is necessary for learning appropriate options. In complex environments, this is not an easy task. Future work could study methods for automatically finding the right balance between hyperparameters or, if possible, eliminate the need for such hyperparameters altogether. Another possible dimension of improvement is to study how to extend the proposed ideas to the online setting; an agent may be able to sample trajectories as it is learning a task and progressively use them to continuously improve its option set.

In the last two chapters we showed how prior experience can be used to obtain meaningful temporal abstractions for a class of problems. One of the reasons why these techniques are able to provide such an improvement is that they bias the behavior of the agent towards known well-performing actions in early stages of learning. During this period the agent mainly learns about the environment, since it has not yet gathered enough information to learn a good policy; in other words, the agent is exploring the environment. Therefore, guiding exploration can provide a substantial improvement in performance.

In the next chapter we provide a different approach to leveraging existing knowledge. We present a formulation that allows an agent to exploit existing knowledge to learn a policy that can be used specifically for exploration for a set of related tasks.

CHAPTER 5

A META-MDP FORMULATION TO IMPROVE EXPLORATION

So far we have shown that we can use prior knowledge to learn temporally extended actions and that extending the original action set with these temporal abstractions generally leads to improved learning performance. This improvement is a consequence of the agent biasing its behavior towards executing known well-performing actions in early stages of learning; when the agent is mostly exploring the environment.

In this chapter we present an alternative approach to using temporal abstraction for leveraging existing experience. We present a framework for learning a policy specifically learned for exploration by introducing a second agent whose objective is to guide exploration in a specific problem class. We introduce the concept of an *optimal exploration policy*, μ^*, and present a MDP formulation to learn such a policy.

We consider the scenario where, if an agent solving a task $c \in \mathcal{C}$ decides to explore, it takes an action according to the policy of an *advisor*—a different agent whose policy is tailored for exploration. We formulate the problem of learning μ^* as a new MDP and allow the agent to use any RL algorithm to learn a policy for solving a given task c. We show that, in our formulation, an optimal policy for the MDP the advisor faces is an optimal exploration policy according to our definition, and that our framework allows us to leverage the existing RL literature to learn an optimal exploration policy.

5.1 Problem Description

We assume that an RL agent has to solve a series of tasks $c \in \mathcal{C}$ belonging to a problem class $\mathcal{C}$ and consider a process that proceeds as follows. First, a task,

$c \in \mathcal{C}$ is sampled from some distribution, $d_\mathcal{C}$, over $\mathcal{C}$. Next, the agent uses some pre-specified RL algorithm (e.g., Q-learning or Sarsa) to approximate an optimal policy on the sampled task, c. At each time-step, the agent makes a decision whether it should explore or exploit. If the agent decides to explore, it behaves according to an *exploration policy* μ, otherwise according to an *exploitation policy* π. The exploration policy is learned by another RL agent, called an *advisor*, whose task is to tell the agent what action to take (if it decided to explore) in the specific task it is trying to solve.

We assume that the agent interacts with for I episodes lasting T time-steps each. We define the performance of an exploration policy, μ, for a specific task $c \in \mathcal{C}$ to be

$$\rho(\mu, c) = \mathbf{E}\left[\sum_{i=0}^{I}\sum_{t=0}^{T} R_t^i \middle| \mu, c\right], \tag{5.1}$$

where R_t^i is the reward at time step t during the i^{th} episode.

Let C be a random variable that denotes a task sampled from $d_\mathcal{C}$. Our objective is to find an *optimal exploration policy*, μ^*, which we define to be any policy satisfying:

$$\mu^* \in \underset{\mu}{\arg\max} \quad \mathbf{E}\left[\rho(\mu, C)\right]. \tag{5.2}$$

This objective seeks to maximize the cumulative return of an agent on the problem class in expectation. Intuitively, assuming an agent always uses the same learning algorithm to learn an optimal policy π_c^* for each task $c \in \mathcal{C}$ and assuming its behavior always begins with same initial policy, a more efficient exploration policy will lead to higher expected cumulative return throughout the problem class. When presented with a new task from the same problem class, an agent using this learned exploration policy should be able to learn how to solve the task efficiently.

In this chapter, we seek to approximate the behavior of an optimal exploration policy μ^* from an agent's interaction with tasks sampled from $\mathcal{C}$. In particular, we present a solution by modeling the problem as an MDP, which allows us to leverage existing work in RL, and show that the optimal policy for this new MDP results in an optimal exploration policy μ^*,

We contrast our framework with another meta learning approach called Model Agnostic Meta Learning (MAML) (Finn et al., 2017). Unlike our approach, this method seeks to find a good set of initial parameters from where to beginning optimizing on new problems; in the case of RL, MAML learns a general initial policy that can be optimized quickly to a specific task. Our empirical results show a clear improvement over MAML and demonstrate that our framework is a viable solution for learning an efficient exploration strategy.

5.2 Related Work

There is a large body of work discussing the problem of *how* an agent should behave during exploration *when faced with a single MDP*. Simple strategies, such as ϵ-greedy with random action-selection, *Boltzmann action-selection* or *softmax action-selection*, make sense when an agent has no prior knowledge of the problem that it is facing. The performance of an agent exploring with random action-selection reduces drastically as the size of the state-space increases (Whitehead, 1991). The performance of Boltzmann or softmax action-selection hinges on the accuracy of the action-value estimates. When these estimates are poor (e.g., early during the learning process), it can have a drastic negative effect on the overall performance of the agent.

Recent research concerning exploration has taken the approach of adding an exploration "bonus" to the reward function. VIME (Houthooft et al., 2016) takes a Bayesian approach by maintaining a model of the dynamics of the environment, obtaining a posterior of the model after taking an action, and using the KL divergence between these two models as a bonus. The intuition behind this approach is that encouraging actions that make large updates to the model allows the agent to better explore areas where the current model is inaccurate. Another approach based on exploration bonus was proposed by Pathak et al. (2017). The authors propose using a neural network to predict state transitions based on the action taken and provide an intrinsic reward proportional to the prediction error. The agent is therefore encouraged to take state transitions that are not modeled accurately. In general, these methods guide the agent into selecting actions that lead to updating certain states and not others. A more efficient approach to achieve a similar effect is to create symbolic representation of states, and update the value of several states after taking an action as opposed to the value of a single state (Feng et al., 2003).

Although these techniques can be efficient when an agent is dealing with a single MDP assuming no prior experience, they do not provide a means to guide an agent to explore intelligently when faced with a novel, but related task.

In this chapter we consider the idea of meta-learning, or learning to learn, which has been a recent area of focus. Although there is no standard definition of what meta-learning is, roughly speaking it refers to the use of results from machine learning experiments, to learn models that can generalize to different tasks. For example, Andrychowicz et al. (2016) show how to learn a model that specifies the update rule for a class of optimization problems. Given an objective function f and parameters θ, the authors proposed learning a model, g_ϕ, such that the update to parameters θ_k, at iteration k are given according to $\theta_{k+1} = \theta_k + g_\phi(\nabla f(\theta_k))$. Another example was presented by Rosenbaum et al. (2017), where the authors show how RL can be used in meta-learning to learn efficient neural network architectures. However, even though one can draw a connection to our work through meta-learning, these methods are not concerned with the problem of exploration.

In the context of RL, a similar idea can be applied by defining a meta-MDP, i.e., considering the agent as part of the environment in a larger MDP (Liu et al., 2012; Thomas and Barto, 2011). Even though there is a considerable amount of work that has been done with meta-learning in RL, most research has not addressed the problem of exploration. In this chapter, we show how meta-learning can be used to address the problem of exploration by considering an agent's exploration strategy as its own independent policy. To the best of our knowledge, meta-learning approaches have not been used for this purpose and this chapter aims to explore this idea.

5.3 A Framework for Learning an Exploration Policy

We consider the case where there are two agents interacting with each other. One agent, to which we refer as *the agent*, maintains a task-specific policy that is used for exploitation; given a specific task c, the agent's goal is to learn an *optimal exploitation policy* defined by $\pi_c^* \in \underset{\pi}{\operatorname{argmax}} \mathbf{E}\left[\sum_{t=0}^{T} R_t | \pi, c\right]$. Thus, the agent's goal is to maximize the standard expected return objective.

The second agent, which we call *the advisor*, maintains a policy, μ, which is used by *the agent* for exploration. The goal of the advisor is to learn an optimal exploration policy μ^* for the problem class $\mathcal{C}$ as defined in Eq. (5.2)

The agent will have $K = IT$ time-steps of interactions with each of the sampled tasks. Hereafter we use i to denote the index of the current episode on the current task, t to denote the time step within that episode, and k to denote the number of time steps that have passed on the current task, i.e., $k = iT + t$, and we refer to k as the *advisor time-step*.[1] At every advisor time-step, k, the advisor suggests an action, U_k, to the agent, where U_k is sampled according to μ. Whenever the agent decides to explore, it uses an action provided by the *advisor* according to its policy, μ. After the agent completes I episodes on the current task, a new task is sampled from $\mathcal{C}$ and the agent's policy is reset to an initial policy. Notice that the goals of the advisor and agent solving a specific task are different: the agent solving a specific task tries to optimize the expected return on the task at hand, while the advisor searches for an exploration policy that causes the agent to learn quickly across all tasks. As such, the advisor may learn to suggest bad actions if that is what the agent needs to see to learn quickly.

In this work, we assume an ϵ-greedy exploration schedule, i.e., with probability ϵ_i the agent explores and with probability $1 - \epsilon_i$ the agent exploits, where $(\epsilon_i)_{i=1}^{I}$ is a sequence of exploration rates $\epsilon_i \in [0, 1]$ and i refers to the episode number in the current task.

[1]We assume that if the agent finishes an episode in less than T time-steps, it enters an absorbing state and receives rewards equal to zero for the remainder of the episode.

5.3.1 The Meta-MDP Framework

Our framework can be viewed as a meta-MDP—an MDP within an MDP. From the point of view of the agent, the environment is the current task, c (an MDP). However, from the point of view of the advisor, the environment contains both the task, c, and the agent. At every time-step, the advisor selects an action U and the agent an action A. The selected actions go through a selection mechanism which executes action A with probability $1 - \epsilon_i$ and action U with probability ϵ_i at episode i. Figure 5.1 depicts the proposed framework with action A (exploitation) being selected. Even though one time step for the agent corresponds to one time step for the advisor, one episode for the advisor constitutes a lifetime of the agent (all of its interactions with a sampled task). From this perspective, wherein the advisor is merely another RL algorithm, we can take advantage of the existing body of work in RL to optimize the exploration policy, μ.

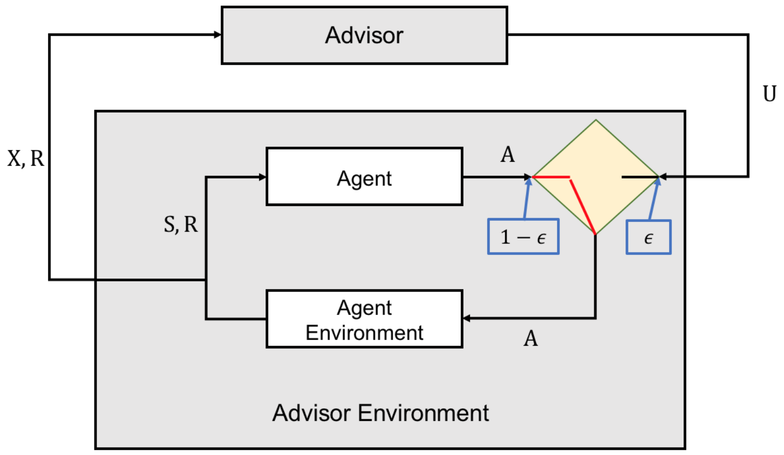


Figure 5.1: MDP view of interaction between the advisor and agent. At each time-step, the advisor selects an action U and the agent an action A. With probability ϵ the agent executes action U and with probability $1-\epsilon$ it executes action A. After each action the agent and advisor receive a reward R, the agent and advisor environment transitions to states S and X, respectively.

5.3.2 Theoretical Results

Below, we formally define the meta-MDP faced by the advisor and show that an optimal policy for the meta-MDP optimizes the objective in (5.2). Recall that R_c, P_c, and d_0^c denote the reward function, transition function, and initial state distribution of the MDP $c \in \mathcal{C}$.

To formally describe the meta-MDP, we must capture the property that the agent can implement an arbitrary RL algorithm. To do so, we assume the agent maintains some memory, M_k, that is updated by some learning rule l (an RL algorithm) at each time step, and write π_{M_k} to denote the agent's policy given that its memory is M_k. In other words, M_k provides all the information needed to determine π_{M_k} and its update is of the form $M_{k+1} = l(M_k, S_k, A_k, R_k, S_{k+1})$ (this update rule can represent popular RL algorithms like Q-Learning and actor-critics). We make no assumptions about which learning algorithm the agent uses (e.g., it can use Sarsa, Q-learning, REINFORCE, and even batch methods like Fitted Q-Iteration), and consider the learning rule to be unknown and a source of uncertainty.

Proposition 1. *Consider an advisor policy, μ, and episodic tasks $c \in \mathcal{C}$ belonging to a problem class $\mathcal{C}$. The problem of learning μ can be formulated as an MDP, $M_{\text{meta}} = (\mathcal{X}, \mathcal{U}, T, Y, d_0')$, where $\mathcal{X}$ is the state space, $\mathcal{U}$ the action space, T the transition function, Y the reward function, and d_0' the initial state distribution.*

Proof. To show that M_{meta} is a valid MDP we need to characterize the MDP's state set, X, action set, U, transition function, T, reward function, Y, and initial state distribution d_0'. We assume that when facing a new task, the agent memory, M, is initialized to some fixed memory M_0 (defining a default initial policy and/or value function). The following definitions capture the intuition provided previously:

- $\mathcal{X} = \mathcal{S} \times \mathcal{I} \times \mathcal{C} \times \mathcal{M}$. That is, the state set $\mathcal{X}$ is a set defined such that each state, $x = (s, i, c, M)$ contains the current task, c, the current state, s, in the current task, the current episode number, i, and the current memory, M, of the agent.

- $\mathcal{U} = \mathcal{A}$. That is, the action set is the same as the action set of the problem class, $\mathcal{C}$.

- T is the transition function, and is defined such that $T(x, u, x')$ is the probability of transitioning from state $x \in \mathcal{X}$ to state $x' \in \mathcal{X}$ upon taking action $u \in \mathcal{U}$. Assuming the underlying RL agent decides to explore with probability ϵ_i and to exploit with probability $1 - \epsilon_i$ at episode i, then T is defined as follows:

$$T(x, u, x') = \begin{cases} \mathrm{d}_0^c(s')\mathbf{1}_{c'=c, i'=i+1, M'=l(M,s,a,r,s')} & \text{if } s \text{ is terminal and } i \neq I-1 \\ \mathrm{d}_{\mathcal{C}}(c')d_0^{c'}(s')\mathbf{1}_{i'=0, M'=M_0} & \text{if } s \text{ is terminal and } i = I-1 \\ \Big(\quad \epsilon_i P_c(s, u, s') & \\ +(1-\epsilon_i)\sum_{a \in A_c} \pi_M(s, a) P_c(s, a, s') \Big) & \\ \times \mathbf{1}_{c'=c, i'=i, M'=l(M,s,a,r,s')} & \text{otherwise.} \end{cases} \tag{5.3}$$

- Y is the reward function, and defines the reward obtained after taking action $u \in \mathcal{U}$ in state $x \in \mathcal{X}$ and transitioning to state $x' \in \mathcal{X}$. Notice that from the point of view of the meta-MDP, the reward function if a probability distribution and taking an action u effectively samples from this distribution. Let R be a random variable denoting the reward received by the meta agent, then Y is given by:

$$Y(x, u, x') = \begin{cases} \Pr(R = R_c(s, u, s')) = \epsilon_i + (1 - \epsilon_i)\pi_M(u, s) \\ \Pr(R = R_c(s, a, s')) = (1 - \epsilon_i)\ \pi_{Mk}(a, s), \forall a \in \mathcal{A}/\{u\} \end{cases} \tag{5.4}$$

Also, notice that $\mathbf{E}\left[Y(x, u, x')\right] = \epsilon_i R_c(s, u, s') + (1-\epsilon_i)\sum_{a \in \mathcal{A}} \pi_M(a, s) R_c(s, a, s')$

- d_0' is the initial state distribution and is defined by: $d_0'(x) = d_{\mathcal{C}}(c) d_0^c(s)\mathbf{1}_{i=0}$.

$\square$

Now that we have fully described M_{meta}, we are able to show that the optimal policy for this new MDP corresponds to an optimal exploration policy.

Theorem 4. An optimal policy for M_{meta} is an optimal exploration policy, μ^*, as defined in (5.2). That is, $\mathbf{E}\left[\rho(\mu, C)\right] = \mathbf{E}\left[\sum_{k=0}^{K} Y_k \middle| \mu, \right]$.

Proof. We can show that an optimal policy of M_{meta} is an optimal exploration policy as defined in Eq. (5.2). To do so, it is sufficient to show that maximizing the return in the meta-MDP is equivalent to maximizing the expected performance. That is, $\mathbf{E}\left[\rho(\mu, C)\right] = \mathbf{E}\left[\sum_{k=0}^{K} Y_k \middle| \mu, \right]$.

$$\begin{aligned}
\mathbf{E}\left[\rho(\mu, C)\right] =& \sum_{c \in \mathcal{C}} \Pr(C = c)\, \mathbf{E}\left[\sum_{i=0}^{I}\sum_{t=0}^{T} R_t^i \middle| \mu, C = c\right] \\
=& \sum_{c \in \mathcal{C}} \Pr(C = c) \sum_{i=0}^{I}\sum_{t=0}^{T}\ \mathbf{E}\left[R_t^i \middle| \mu, C = c\right] \\
=& \sum_{c \in \mathcal{C}} \Pr(C = c) \sum_{i=0}^{I}\sum_{t=0}^{T}\sum_{s \in \mathcal{S}} \Pr(S_{iT+t} = s | C = c, \mu)\, \mathbf{E}\left[R_t^i \middle| \mu, C = c, S_{iT+t} = s\right] \\
=& \sum_{c \in \mathcal{C}} \Pr(C = c) \sum_{i=0}^{I}\sum_{t=0}^{T}\sum_{s \in \mathcal{S}}\sum_{a \in \mathcal{A}} \Pr(S_{iT+t} = s | C = c, \mu) \\
& \times \Pr(A_{iT+t} = a | S_{iT+t}, \mu)\, \mathbf{E}\left[R_t^i \middle| \mu, C = c, S_{iT+t} = s, A_{iT+t} = a\right] \\
=& \sum_{c \in \mathcal{C}} \Pr(C = c) \sum_{i=0}^{I}\sum_{t=0}^{T}\sum_{s \in \mathcal{S}}\sum_{a \in \mathcal{A}}\sum_{s' \in \mathcal{S}} \Pr(S_{iT+t} = s | C = c, \mu) \Pr(A_{iT+t} = a | S_{iT+t} = s, \mu) \\
& \times \Pr(S_{iT+t+1} = s' | S_{iT+t} = s, A_{iT+t} = a, \mu) \\
& \times \mathbf{E}\left[R_t^i \middle| \mu, C = c, S_{iT+t} = s, A_{iT+t} = a, S_{iT+t+1} = s'\right] \\
=& \sum_{c \in \mathcal{C}} \Pr(C = c) \sum_{i=0}^{I}\sum_{t=0}^{T}\sum_{s \in \mathcal{S}}\sum_{a \in \mathcal{A}}\sum_{s' \in \mathcal{S}} \Pr(S_{iT+t} = s | C = c, \mu) \Pr(A_{iT+t} = a | S_{iT+t} = s, \mu) \\
& \times \left(\epsilon_i P_c(s, a, s') + (1 - \epsilon_i) \sum_{a' \in \mathcal{A}} \pi_{M_{iT+t}}(a', s) P_c(s, a', s')\right) \\
& \times \left(\epsilon_i R_c(s, a, s') + (1 - \epsilon_i) \sum_{a' \in \mathcal{A}} \pi_{M_{iT+t}}(a', s) R_c(s, a', s')\right) \\
=& \sum_{c \in \mathcal{C}} \Pr(C = c) \sum_{i=0}^{I}\sum_{t=0}^{T}\sum_{s \in \mathcal{S}}\sum_{a \in \mathcal{A}}\sum_{s' \in \mathcal{S}} \Pr(S_{iT+t} = s | C = c, \mu) \Pr(A_{iT+t} = a | S_{iT+t} = s, \mu) \\
& \times T(x = (s, i, c, M_{iT+t}), a, x = (s', i, c, M_{iT+t})) \\
& \times \mathbf{E}\left[Y(x = (s, i, c, M_{iT+t}), a, x = (s', i, c, M_{iT+t}))\right]
\end{aligned}$$

$$
\begin{aligned}
=&\sum_{c\in\mathcal{C}}\Pr(C=c)\sum_{k=0}^{K}\sum_{s\in\mathcal{S}}\sum_{a\in\mathcal{A}}\sum_{s'\in\mathcal{S}}\Pr(S_k=s|C=c,\mu)\Pr(A_k=a|S_k=s,\mu)\\
&\times T(x=(s,i,c,M_k),a,x'=(s',i,c,M_k))\\
&\times\mathbf{E}\left[Y(x=(s,i,c,M_k),a,x'=(s',i,c,M_k)|\mu,C=c,S_k=s,A_k=a,S_{k+1}=s')\right]\\
=&\sum_{k=0}^{K}\sum_{x\in\mathcal{X}}\sum_{a\in\mathcal{A}}\sum_{x'\in\mathcal{X}}\Pr(X_k=x|\mu)\\
&\times\Pr(U_k=a|X_k=x)T(x,u,x')\,\mathbf{E}\left[Y_k|X_k=s,U_k=a,X_{k+1}=x',\mu\right]\\
=&\sum_{k=0}^{K}\sum_{x\in\mathcal{X}}\sum_{a\in\mathcal{A}}\sum_{x'\in\mathcal{X}}\Pr(X_k=x|\mu)\\
&\times\Pr(U_k=a|X_k=x)\Pr(X_{k+1}=x'|U_k=a,X_k=x)\\
&\times\mathbf{E}\left[Y_k|X_k=s,U_k=a,X_{k+1}=x',\mu\right]\\
=&\sum_{k=0}^{K}\sum_{x\in\mathcal{X}}\sum_{a\in\mathcal{A}}\Pr(X_k=x|\mu)\ \Pr(U_k=a|X_k=x)\,\mathbf{E}\left[Y_k|X_k=x,U_k=a,\mu\right]\\
=&\sum_{k=0}^{K}\sum_{x\in\mathcal{X}}\Pr(X_k=x|\mu)\,\mathbf{E}\left[Y_k|X_k=x,\mu\right]\\
=&\sum_{k=0}^{K}\mathbf{E}\left[Y_k|\mu\right]\\
=&\mathbf{E}\left[\sum_{k=0}^{K}Y_k|\mu\right]
\end{aligned}
$$

$\square$

Given that M_{meta} is an MDP for which an optimal exploration policy is an optimal policy, it follows that the convergence properties of RL algorithms apply to the search for an optimal exploration policy. For example, in some experiments the advisor uses the REINFORCE algorithm (Williams, 1992), the convergence properties of which have been well-studied (Phansalkar and Thathachar, 1995).

Although the framework presented thus far is intuitive and results in nice theoretical properties (e.g., methods that guarantee convergence to at least locally optimal exploration policies), each episode corresponds to a new task $c \in \mathcal{C}$ being sampled. This means that training the advisor may require solving a large number of tasks (episodes of the meta-MDP), each one potentially being an expensive procedure. To address this issue, we sampled a small number of tasks $c_1, \ldots, c_n$, where each $c_i \sim d_{\mathcal{C}}$ and train many episodes on each task in parallel. By taking this approach, every update to the advisor is influenced by several simultaneous tasks and results in an practical approach to obtain a general exploration policy to be used in subsequent tasks.

5.3.3 Pseudocode

We developed two different training procedures based on two popular RL algorithms: REINFORCE, (Williams, 1992), and Proximal Policy Optimization (PPO), (Schulman et al., 2017). While REINFORCE is a more stable learning algorithm than PPO, it relies on Monte Carlo sampling and can make learning a slow process. PPO, on the other hand, is a temporal difference (TD) method, which addresses some of the computational limitations of REINFORCE. Below we describe our proposed algorithms in more detail.

REINFORCE is a Monte Carlo method that assumes the agent has a parameterized policy and the parameters are updated to maximize the expected return. By sampling trajectories from the current policy, the agent estimates the expected return of the policy and updates the parameters in the direction of the gradient of the policy's expected return.

Algorithm 4 presents pseudocode for an implementation of our method training the advisor using REINFORCE. The algorithm runs for I_{meta} episodes for the advisor and I episodes of the agent per advisor episode. At the end of each agent episode, the agent's policy π parameterized by θ is updated via REINFORCE with step size α, lines [10-12]. At the end of each advisor episode, every trajectory recorded by every agent episode is used to update the exploration policy μ parameterized by ϕ with REINFORCE using step size β, lines [13-15].

Algorithm 4 Agent + Advisor - REINFORCE

```
Initialize advisor policy $\mu$ randomly
for $i_{meta} = 0, 1, \ldots, I_{meta}$ do
    Sample task $c$ from $d_c$
    for $i = 0, 1, \ldots, I$ do
        Initialize $\pi$ to $\pi_0$
        $s_t \sim d_0^c$
        for $t = 0, 1, \ldots, T$ do
            $a_t \sim \begin{cases} \mu \text{ with probability } \epsilon_i \\ \pi \text{ with probability } (1 - \epsilon_i) \end{cases}$
            take action $a_t$, observe $s_t$, $r_t$
        for $t = 0, 1, \ldots, T - 1$ do
            G $= \sum_{k=t+1}^{T} R_k$
            $\theta = \theta + \alpha \, G \, \nabla \log \pi(a_t, s_t)$
    for $t = 0, 1, \ldots, TI - 1$ do
        G $= \sum_{k=t+1}^{TI} R_k$
        $\phi = \phi + \beta \, G \, \nabla \log \mu(a_t, s_t)$
```

In Algorithm 5 we show how PPO can be used to train both the agent and advisor. PPO maintains two parameterized policies for an agent, π_θ and $\pi_{\theta'}$ (with parameters θ and θ') and a model $\hat{V}_\phi$ (with parameters ϕ) used to estimate the value function. The policy $\pi_{\theta'}$ refers to the previous policy while π_θ refers to the latest updated policy. The algorithm runs $\pi_{\theta'}$ for l time-steps and computes the generalized advantage estimates (GAE), (Schulman et al., 2016), $\hat{A}_1, \ldots, \hat{A}_l$; where $\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots + (\gamma\lambda)^{l-t+1}\delta_{l-1}$ and $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$.

At a give time-step t, the objective of PPO seeks to maximize the following objective:

$$J(\theta, \theta', \phi) = \mathbf{E}_t \left[\min(r_t \hat{A}_t, \text{clip}(r_t, 1 - \alpha, 1 + \alpha)\hat{A}_t) \right] - (R_t + \gamma \, \hat{V}_\phi(s_{t+1}) - \hat{V}_\phi(s_t))^2$$

where $r_t = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta'}(a_t|s_t)}$, and $\hat{V}_\phi(s)$ is an estimate of the value for state s. The updates are done in mini-batches that are stored in a buffer of collected samples.

To train the agent and advisor with PPO we defined two separate sets of policies: μ_ϕ and $\mu_{\phi'}$ for the advisor, and π_θ and $\pi_{\theta'}$ for the agent. The agent collects samples of trajectories of length l to update its policy, while the advisor collects trajectories of length n, where $l < n$. So, J (the objective of the agent) is computed with l samples while J_{meta} (the objective of the advisor) is computed with n samples. In our experiments, setting $n \geq 2l$ seemed to give the best results.

Notice that the presence of a buffer to store samples, means that the advisor will be storing samples obtained from many different tasks, which prevents it from over-fitting to one particular problem.

Algorithm 5 Agent + Advisor - PPO

```
1: Initialize advisor policy $\mu_\psi$, $\mu_{\psi'}$ randomly
2: for $i_{meta} = 0, 1, \ldots, I_{meta}$ do
3:     Sample task $c$ from $d_c$
4:     for $i = 0, 1, \ldots, I$ do
5:         Initialize $\pi_\theta$ and $\pi_{\theta'}$
6:         $s_t \sim d_0^c$
7:         for $t = 0, 1, \ldots, T$ do
8:             $a_t \sim \begin{cases} \mu_{\psi'} \text{ with probability } \epsilon_i \\ \pi_{\theta'} \text{ with probability } (1 - \epsilon_i) \end{cases}$
9:             take action $a_t$, observe $s_t$, $r_t$
10:            if $t \,\%\, l = 0$ then
11:                compute $\hat{A}_1, \ldots, \hat{A}_l$
12:                optimize $J$ w.r.t $\pi_\theta$
13:                $\pi_{\theta'} = \pi_\theta$
14:            if $t \,\%\, n = 0$ then
15:                compute $\hat{A}_1, \ldots, \hat{A}_n$
16:                optimize $J_{meta}$ w.r.t $\mu_\psi$
17:                $\mu_{\psi'} = \mu_\psi$
```

5.4 Experiments

In this section we present experiments for discrete and continuous control tasks. We tested the training of advisor policies using the algorithms described above based on REINFORCE, (Williams, 1992), and Proximal Policy Optimization (PPO), (Schulman et al., 2017).

5.4.1 Discrete Control Experiments

For discrete control tasks we present two different classes of problems: Pole-balancing, (Sutton and Barto, 1998), and Animat (Thomas and Barto, 2011). We chose these problems because any tasks belonging to these domain share a common structure that is intuitive and easy to interpret; therefore, it allows us to study in detail the behavior of our algorithm.

5.4.1.1 Pole Balancing

(Sutton and Barto, 1998), is a classical control problem commonly used to benchmark RL algorithms. The agent is tasked with applying force to a cart to prevent a pole that balancing on it from falling. States are represented by 4-D vectors describing the position and velocity of the cart, and angle and angular velocity of the pendulum, i.e., $s = [x, v, \theta, \dot{\theta}]$. We consider two different versions of this problem; one with a discrete action set and another one with a continuous action set. In the discrete case the agent has 2 actions at its disposal: apply a force f in the positive or negative x direction. In the continuous case, the agent picks a value in the range $[0, 1]$ which is mapped to an action in the range $[-f, f]$.

The distinct variations in tasks were constructed by modifying four variables: pole mass, m_p, pole length, l, cart mass, m_c, and force magnitude, f. Figure 5.2 depict two variations of this type of problem.

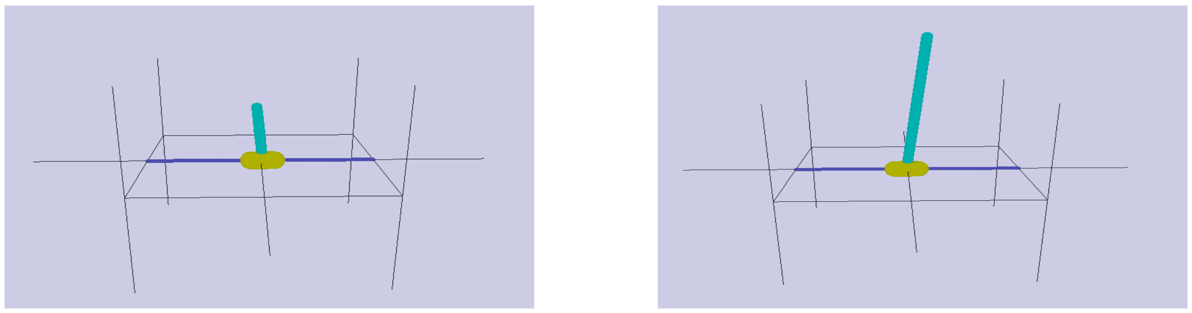


Figure 5.2: Example of task variations in Pole-balancing.

Figure 5.3, contrasts the cumulative return of an agent using the advisor for exploration (in blue) with the cumulative return obtained by an agent using ϵ-greedy random exploration (in red) during training over 6 training tasks. The exploitation policy, π, was trained using REINFORCE for $I = 1{,}000$ episodes and the exploration policy, μ, was trained using REINFORCE for 500 iterations. In the figure, the horizontal axis corresponds to iterations, which are episodes for the adviser and entire lifetimes for the agent. The horizontal red line denotes an estimate (with standard error bar) of the expected cumulative reward that an agent will obtain during its lifetime if it samples actions uniformly when exploring. Notice that this is not a function of the training iteration, as the random exploration is not updated. The blue curve (with standard error bars from 15 trials) shows how the expected cumulative reward that the agent will obtain during its lifetime changes as the advisor learns to improve its policy. Here the horizontal axis shows the number of training iterations—the number of episodes of the meta-MDP. By the end of the plot, the agent is obtaining roughly 30% more reward during its lifetime than it was when using a random exploration.

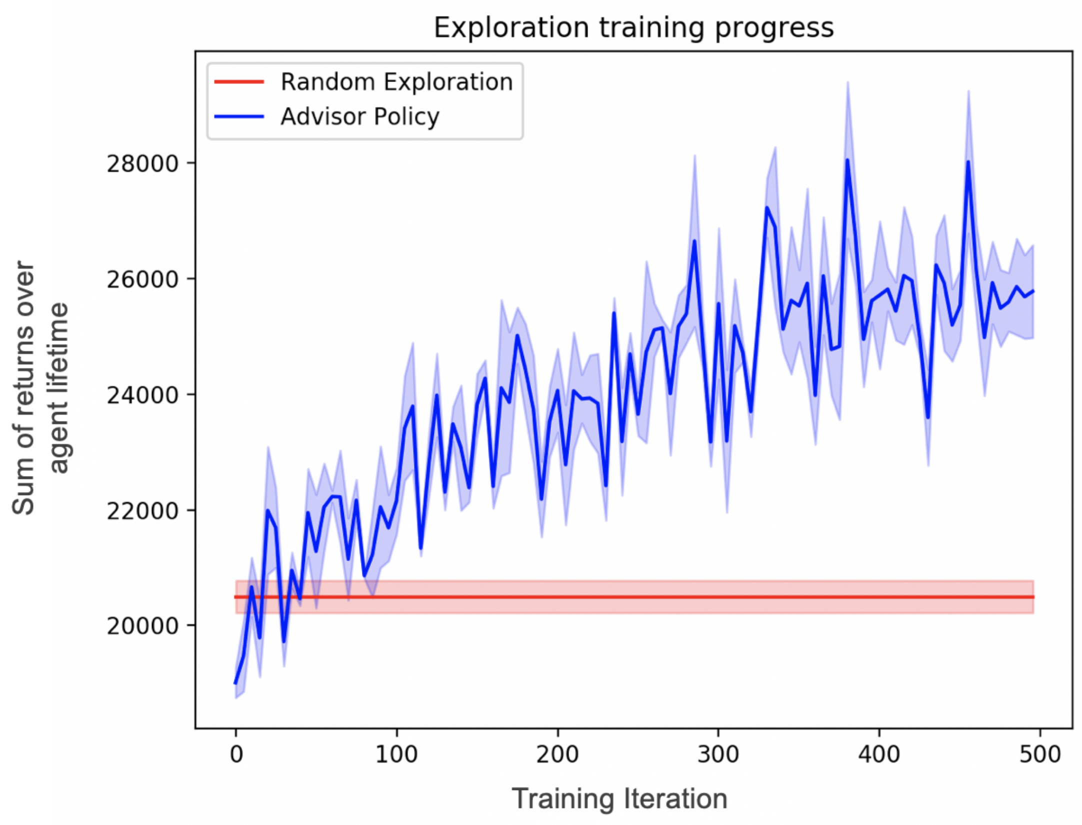


Figure 5.3: Performance curves during training comparing advisor policy (blue) and random exploration policy (red).

To better visualize this difference, Figure 5.4 shows the mean *learning curves* (episodes of an agent's lifetime on the horizontal axis and average return for each episode on the vertical axis) during the first and last 50 iterations. The mean cumulative reward were 25,283 and 30,552 respectively. Notice that, although the final performance obtained is similar, using a trained advisor allows the agent to reach this level of performance faster; thus achieving a larger cumulative return.

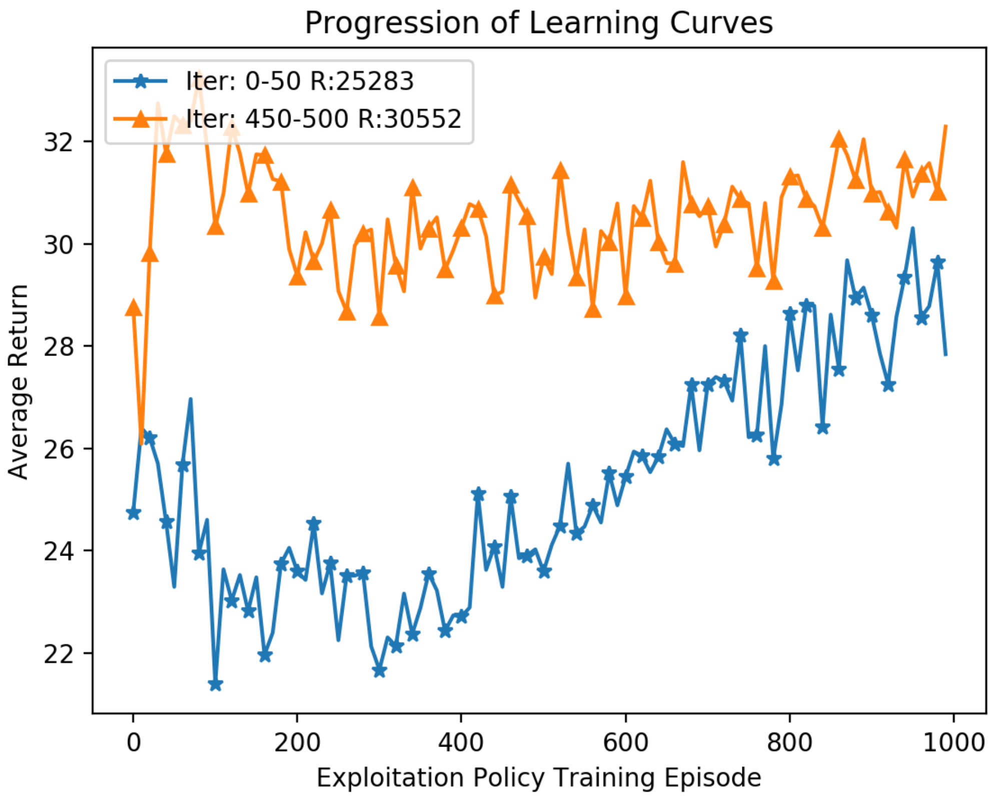


Figure 5.4: Average learning curves on training tasks over the first 50 advisor episodes (blue) and the last 50 advisor episodes (orange).

One might think that the learned policy for exploration might simply be a policy that works well in general. So how do we know that the advisor is learning a policy that is useful for exploration and not simply a policy for exploitation? To answer this question, we generated three distinct unseen tasks for both pole-balancing and compare the performance of an agent using only the learned exploration policy with the performance obtained by an exploitation policy trained to solve each specific task.

The plot in Figure 5.5 compares the mean return (and standard deviation) of the exploration policy (blue) and the exploitation policy (green) on each task variation. From the figure we can see that the exploration policy fails to achieve a comparable performance to a task-specific policy; providing evidence that the policy learned by the advisor is not simply a general exploitation policy.

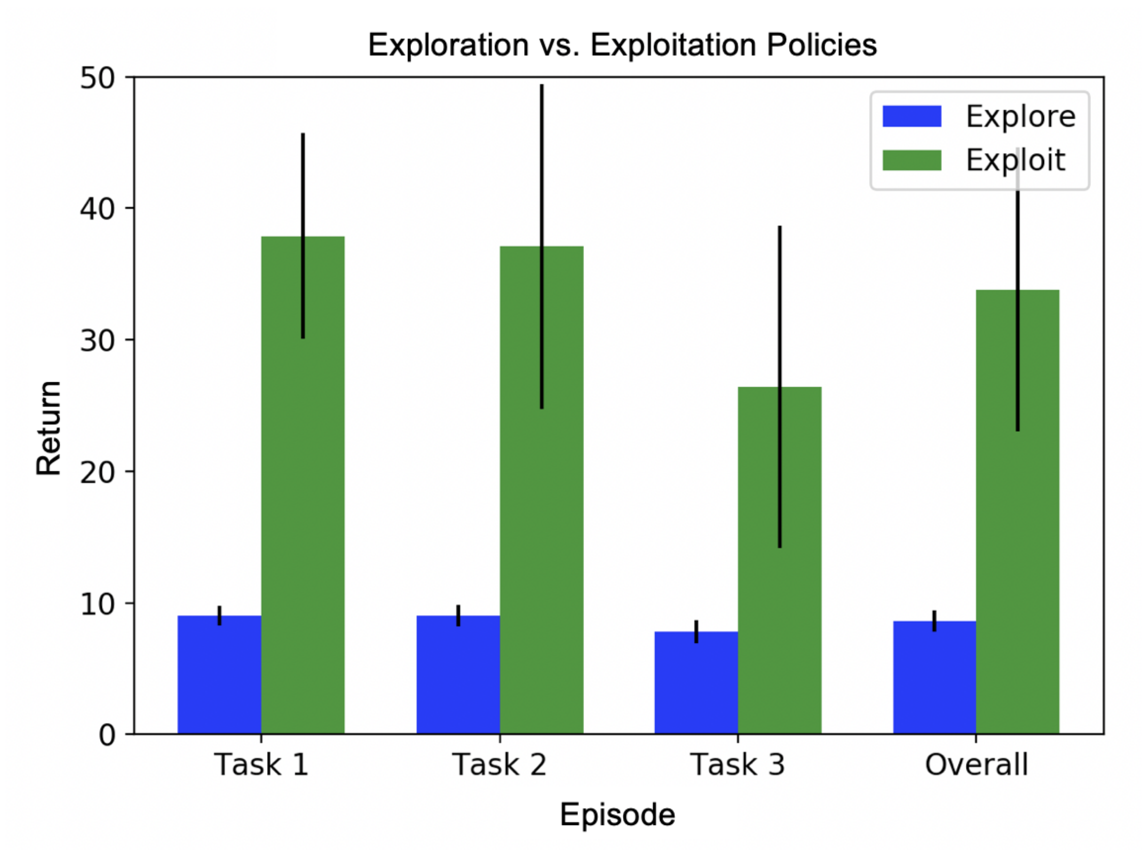


Figure 5.5: Average returns obtained on test tasks when using the advisor's exploration policy (blue) and a task-specific exploitation (green)

5.4.1.2 Animat

We revisited the Animat problem introduced in Chapter 2. The different variations of the tasks correspond to randomized start and goal positions in different environments. An interesting pattern that is shared across tasks in this problem is that there are actuator combinations that are not useful for reaching the goal. For example, activating actuators at $\theta = 0°$ and $\theta = 180°$ would leave the agent in the same position it was before (ignoring the effect of the noise). Even though the environment itself might not provide enough structure for the advisor to leverage previous experiences, the presence of these poor performing actions provide some common patterns that can be leveraged.

Figure 5.6 shows the mean learning curves averaged over all training tasks, where the advisor was trained for 50 iterations. The curve in blue is the average curve obtained from the first 10 iterations of training the advisor and the curve in orange is the average obtained from the last 10 training iterations of the advisor. Each individual task was trained for $I = 800$ episodes. The figure shows a clear performance improvement on average as the advisor improves its policy.

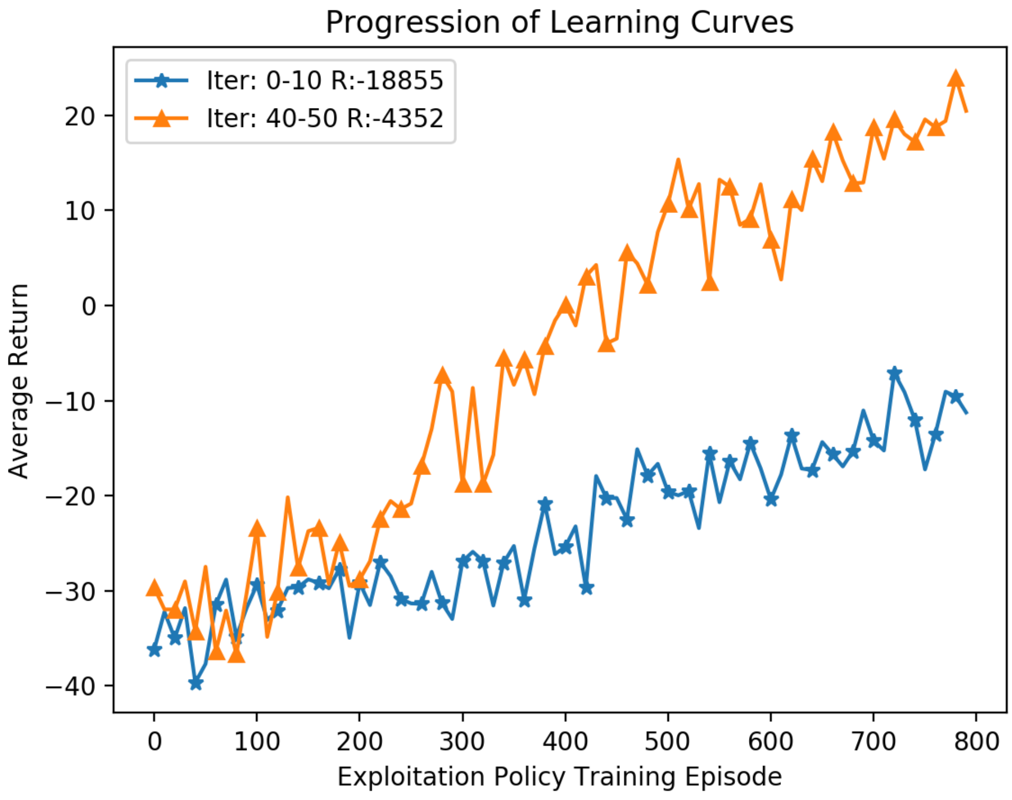


Figure 5.6: Average learning curves for Animat on training tasks over the first 10 iterations (blue) and last 10 iterations (orange).

To test our intuition that an exploration policy would exploit the presence of poor-performing actions, we recorded the frequency with which they were executed on unseen testing tasks when using the learned exploration policy after training and when using a random exploration strategy, over 5 different learned policies. Figure 5.7 helps explain the difference in performance seen in Figure 5.6. It depicts on the y-axis, the percentage of times these poor-performing actions were selected at a given episode, and on the x-axis the agent episode number in the current task. This shows that the agent when using the advisor policy (blue) is encouraged to reduce the selection of known poor-performing actions, compared to an agent choosing actions from a uniform random distribution for exploration (red).

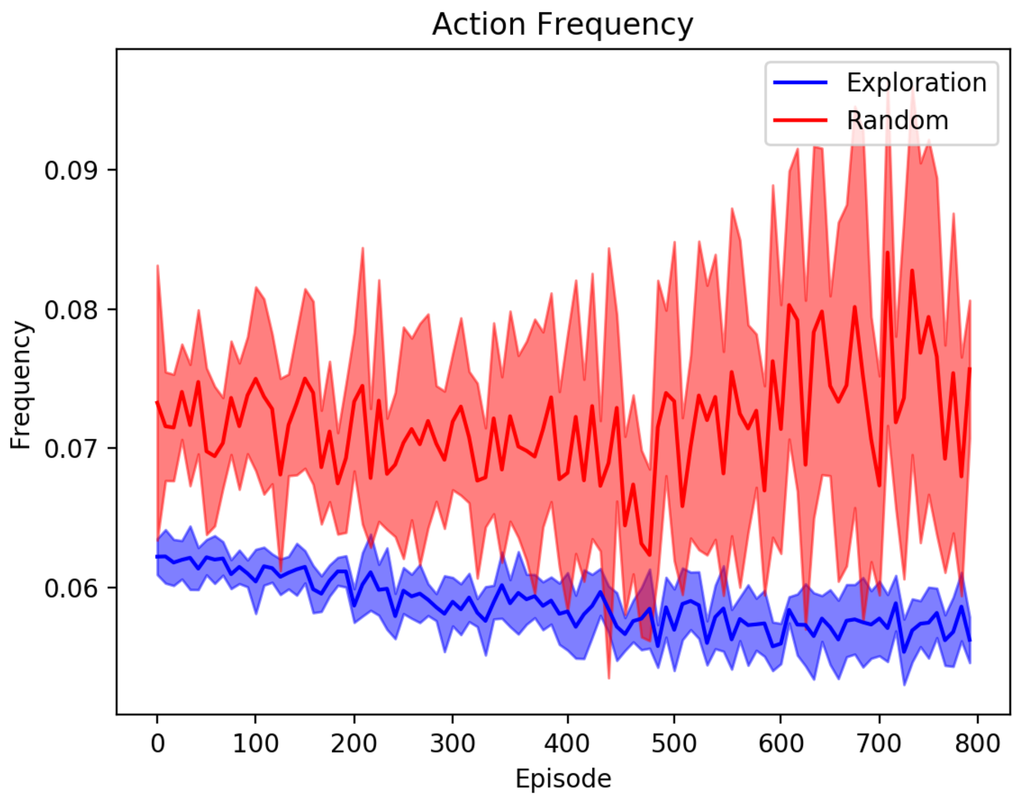


Figure 5.7: Frequency of poor-performing actions in an agent's lifetime with learned (blue) and random (red) exploration.

To verify that the learned exploration policy is not a general exploitation policy, we performed a similar experiment to the one for the Pole-balancing problem. Figure 5.8 shows the average (and standard deviation) number of steps needed to reach the goal in three different tasks. Notice that in this case smaller values are better. Just as before, the performance of a task-specific policy performs considerably better than the exploration policy for any task in particular. Further giving us evidence that the policy learned by the advisor is tailored for exploration.

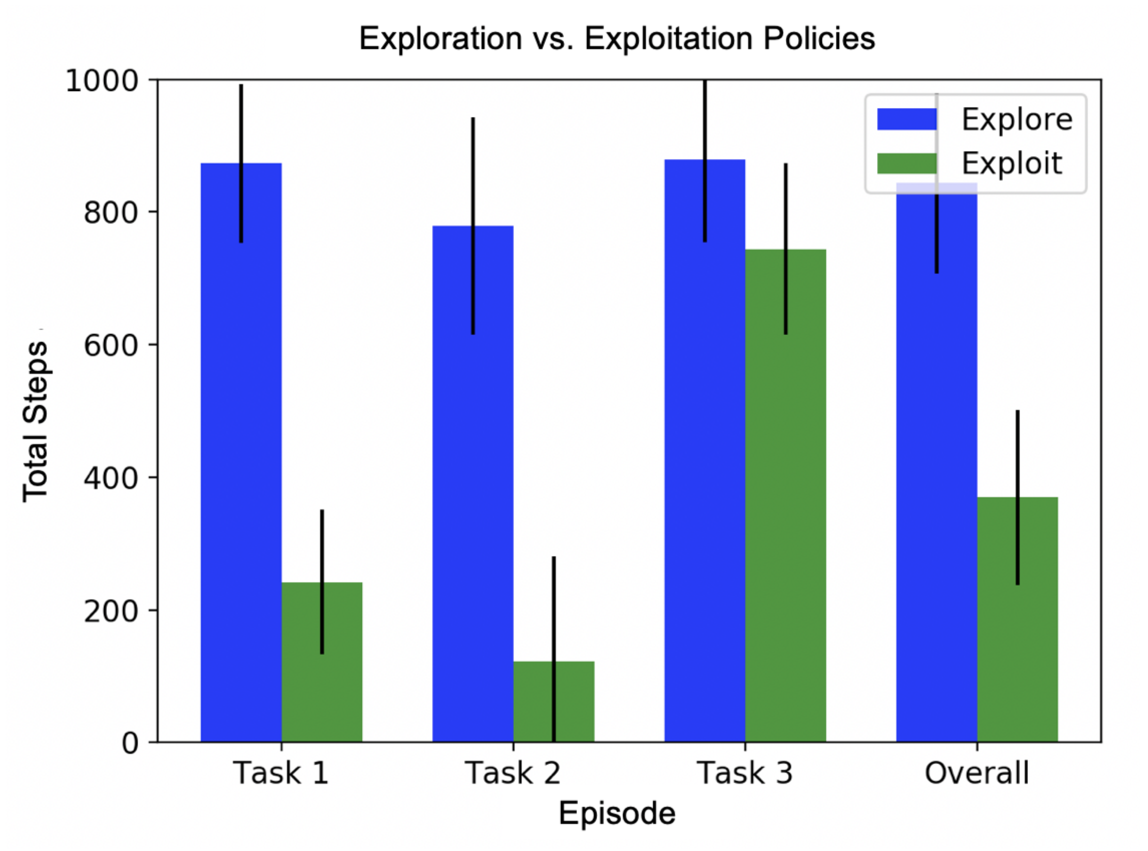


Figure 5.8: Number of steps needed to complete test tasks with advisor policy (blue) and exploitation (green).

5.4.2 Continuous Control Experiments

In this section we examine the performance of our framework on novel tasks, and contrast our method to MAML trained using PPO. In the case of discrete action-sets, we trained each task for 500 episodes and compare the performance of an agent trained with REINFORCE (R) and PPO, with and without an advisor. In the case of continuous tasks, we tested on problem obtained from the OpenAI Gym suite and restrict our experiments to an agent trained using PPO (since it was shown to perform well in continuous control problems), with and without an advisor after training for

500 episodes. In our experiments we set the initial value of ϵ to $\epsilon_0 = 0.8$, and defined the update after each agent episode to be $\epsilon_{i+1} = \max(0.1, 0.995\epsilon_i)$.

5.4.2.1 Pole Balancing

We tested our approach in a variation of the Pole-balancing problem with continuous action set. In this case the agent could apply a force $f \in [-f_{min}, f_{max}]$, where $f_{min} < 0$ and $f_{max} > 0$. The task variations where implemented in the same fashion as in the case of discrete action set.

5.4.2.2 Hopper

The goal of the agent is to reach the end of the track by learning how to jump and balance to move forward. The action set is given by a three dimensional vector denoting the acceleration applied at each of the joins in the agent model, and the state is represented by a vector of 15 dimensions. The tasks vary in the length of the limbs in the model which result in changes in the transition function, initial state distribution and reward function. Two variations are shown in Figure 5.9.

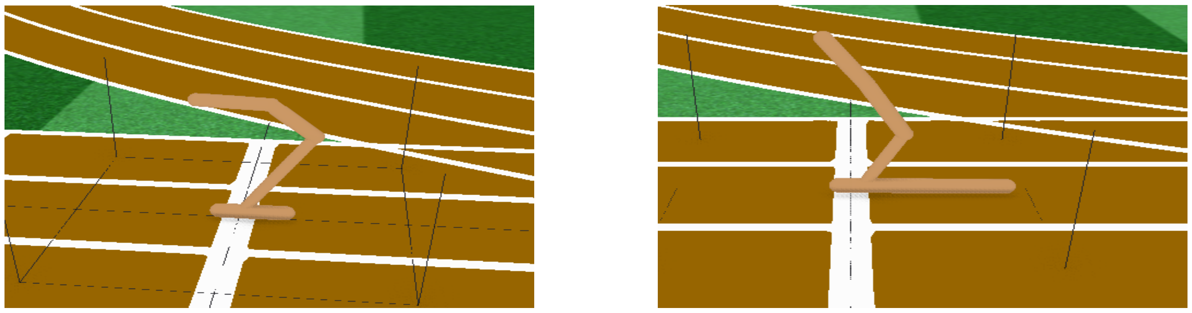


Figure 5.9: Example of task variations in Hopper.

5.4.2.3 Ant

In this problem class the goal of the agent is similar to the previous case. The agent needs to reach the end of track and, at each time step, it receives a positive or negative reward proportional to how closer or farther it got from the end of the track. The action set is given by a eight dimensional vector denoting the acceleration applied at each of the joins in the agent model, and the state is represented by a vector of 28 dimensions. The tasks varied in the models of the agent by varying the size of the body and the length of the limbs; two variations are shown in Figure 5.10.

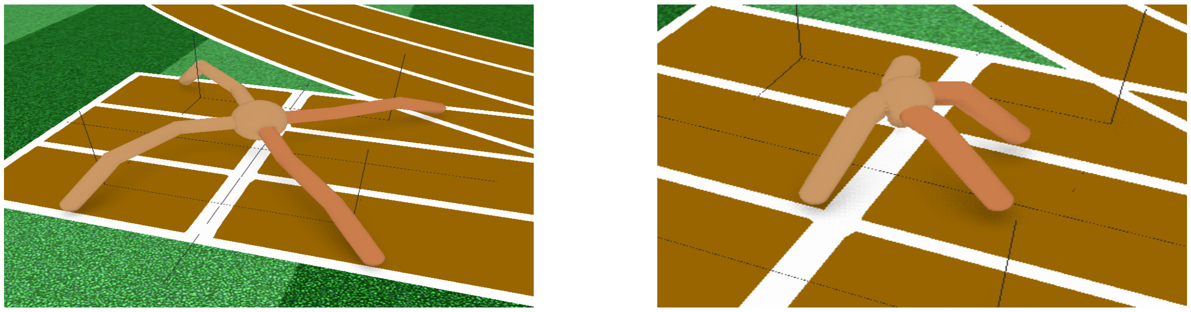


Figure 5.10: Example of task variations in Ant.

Table 5.1 shows the results obtained in all the experiments described in this section. Each novel task was trained 5 times for each method; the table shows the average performance and standard deviation in each task.

Problem Class	R	R+Advisor	PPO	PPO+Advisor	MAML
Pole-balance (d)	20.32 ± 3.15	28.52 ± 7.6	27.87 ± 6.17	$\mathbf{46.29 \pm 6.30}$	39.29 ± 5.74
Animat	-779.62 ± 110.28	$\mathbf{-387.27 \pm 162.33}$	-751.40 ± 68.73	-631.97 ± 155.5	-669.93 ± 92.32
Pole-balance (c)	—	—	29.95 ± 7.90	$\mathbf{438.13 \pm 35.54}$	267.76 ± 163.05
Hopper	—	—	13.82 ± 10.53	$\mathbf{164.43 \pm 48.54}$	39.41 ± 7.95
Ant	—	—	-42.75 ± 24.35	83.76 ± 20.41	$\mathbf{113.33 \pm 64.48}$

Table 5.1: Average performance on discrete and continuous control unseen tasks over the last 50 episodes. In the cases where advisor performs best, the results are statistically significant. For the Ant domain, MAML appears to be better, although the high variance in returns makes this result *not* statistically significant

In the discrete case, we can see that for both pole-balancing and Animat, MAML showed a clear improvement over starting from a random initial policy. However, using the advisor with PPO resulted in a clear improvement in pole-balancing and, in the case of animat, training the advisor with REINFORCE led to an almost 50% improvement over MAML.

In the case of continuous control, the first test corresponds to a continuous version of pole-balancing, where the different variations were obtained by modifying the length and mass of the pole, and the mass of the cart. The second and third set of tasks correspond to the "Hopper" and "Ant" problem classes, where the task variations were obtained by modifying the length and size of the limbs and body.

In all continuous control tasks, both using the advisor and MAML led to significant improvement in performance in the alloted time. In the case of pole-balancing, using the advisor led the agent to accumulate almost twice as much reward as MAML, and in the case of Hopper, the advisor led to accumulating 4 times as much total reward. On the other hand, MAML led to higher average return than the advisor in the Ant problem class, but showing high variance.

An important takeaway from these results is that in all cases, using the advisor resulted in a clear improvement in performance over a limited number of episodes. This does not mean that the agent can reach a better policy over an arbitrarily long period of time, but rather that it is able to reach a certain performance level more quickly.

5.4.3 Self-Driving Simulation

For this experiment we used a simulator implemented by Tawn Kramer from the "Donkey Car" community [2]. The goal of the agent is to control a car moving with a constant acceleration (up to some maximum velocity) and prevent it from leaving the right lane of the road. The action set consist on 15 possible steering angles between angles $\theta_{\min} < 0$ and $\theta_{\max} > 0$ and the state is represented as a stack of the last 4 80×80 images sensed by a front-facing camera. The agent received a reward of 0 at each time-step and a reward of -1 if it leaves the corresponding lane.

[2]The Unity simulator for the self-driving task can be found at `https://github.com/tawnkramer/sdsandbox`

Different tasks vary in the body mass, m, of the car and values of θ_{min} and θ_{max}. These differences imply a change in the transition and reward functions of each different task. Two different variations for these tasks are shown in Figure 5.11.

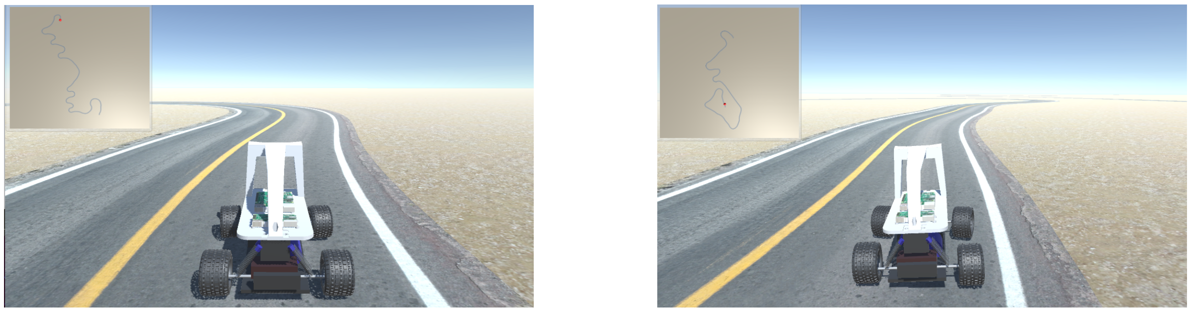


Figure 5.11: Example of task variations in driving simulator.

The experiment, depicted in Figure 5.12, compares an agent who is able to use an advisor during exploration for fine-tuning (blue) vs. an agent who does not have access to an advisor (red). The figure the number of episodes of fine-tuning needed to reach a pre-defined performance threshold ($1,000$ time-steps without leaving the correct lane). The first and second groups in the figure show the average number of episodes needed to fine-tune in the first and second half of tasks, respectively. In the first half of tasks (left), the advisor seems to make fine-tuning more difficult since it has not been trained to deal with this specific problem. Using the advisor took an average of 42 episodes to fine-tune, while it took on average 12 episodes to fine-tune without it. The benefit, however, can be seen in the second half of training tasks. Once the advisor had been trained, it took on average 5 episodes to fine-tune while not using the advisor needed an average of 18 episodes to reach the required performance threshold. When the number of tasks is large enough and each episode is a time-consuming or costly process, our framework could result in important time and cost savings.

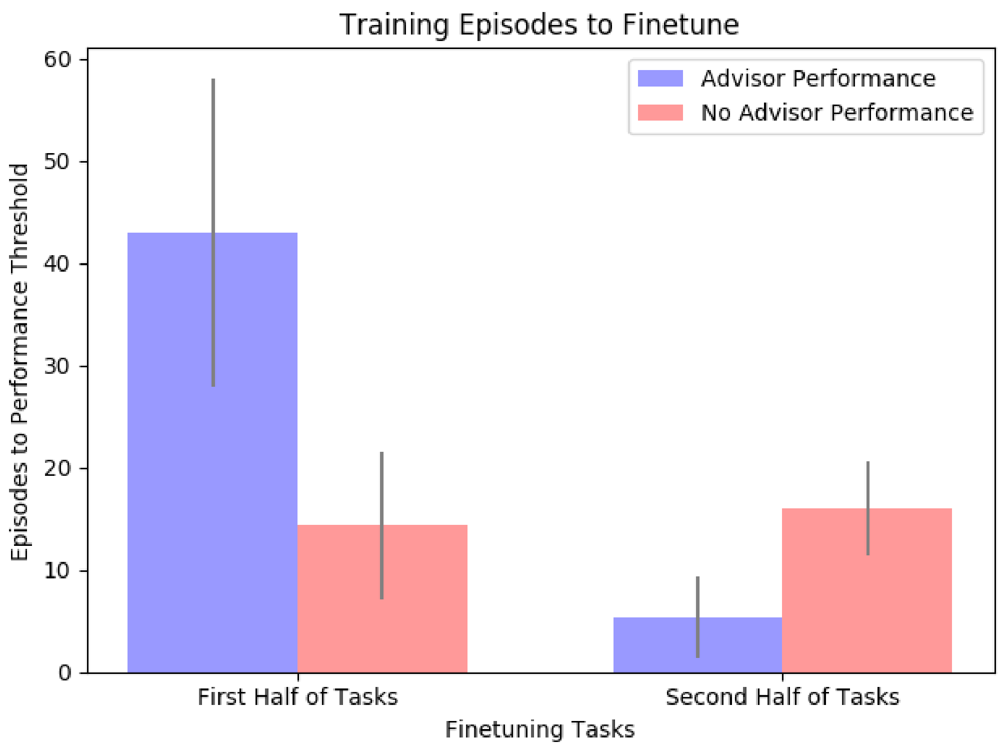


Figure 5.12: Number of episodes needed to achieve threshold performance on self-driving problem (lower is better). Exploration with advisor is shown in blue and random exploration in red. Left group shows mean number of episodes needed to fine-tune on first half of tasks, the right group shows the same on the second half of tasks.

5.5 Discussion and Conclusions

In this chapter, we presented a framework for using prior experience with a set of related tasks to learn an exploration policy for a problem class. We showed that the problem of learning an optimal exploration policy, μ^*, can be formulated as a meta-MDP where the environment is itself defined by an MDP (a given task $c \in \mathcal{C}$). The advisor in this meta-MDP, is an agent whose policy guides exploration in the problem class. We demonstrated through experiments that, after training, using the advisor to guide exploration results in a significant improvement in the time required for an agent to learn how to solve a set of benchmark tasks. In addition, we showed that the policy learned by the advisor *is* an exploration policy and *not* merely a policy that can solve any task; in fact, our experiments show that the policy of the advisor performs poorly if used solely to solve any given task.

This work could be extended in several different directions. In our approach, the probability of whether the agent explores or exploits depends entirely on the current episode; ignoring the possibility that the agent might be behaving reasonably in some regions of the state space. A clear direction for improvement would be to incorporate techniques that would allow an agent to learn *when* it should explore.

Another direction for improvement is addressing the computational cost of training the meta-MDP, which is a problem inherent in meta-learning methods themselves. If the agent were allowed to select which task it will face next, one possible way of addressing this issue would be through curriculum learning techniques. From this perspective, the agent could learn to pick the task that would result in the most improvement for the advisor.

CHAPTER 6

CONCLUSION AND FUTURE WORK

In this dissertation we have presented two different approaches for exploiting existing experience to improve the performance of reinforcement learning algorithms: temporal abstractions and meta-learning.

We began this thesis by motivating the use of prior experience and its relevance to RL; discussing in Chapter 2 different ways in which past knowledge can be used to improve RL algorithms. Inspired by research in the animal behavior literature, in Chapter 3 we developed a framework for obtaining temporal abstractions by using compression techniques. We showed that one can leverage existing knowledge of optimal behavior to obtain recurrent action patterns—macros—that are often present in optimal trajectories. Our method was able to obtain simple open-loop temporally-extended actions and we showed, through empirical evaluation, that they outperform competing methods in many scenarios. However, macros are rather limited in certain situations and are not well suited to represent continuous actions. The options framework provide an alternative to macros that circumvent some of those limitations.

In Chapter 4 we improved on the ideas developed on Chapter 3 and showed how an objective analogous to compression can be posed as an optimization problem whose solution results in a set of options capable of generating optimal trajectories. The options obtained from optimizing the proposed objective, are not only able to improve the performance of a learning agent, but are also specialized for specific sub-tasks within the larger problem: a desirable property within the options framework.

In those two chapters we addressed an issue that is often neglected in the literature: how do we obtain an appropriate number of macros or options for a specific type of problem? In the case of macros, we showed that we can select an appropriate number by filtering out macros that are too similar to each other. In the case of options, we showed that we can iteratively learn one option at a time by optimizing the proposed objective and terminate once the optimization procedure fails to improve beyond some threshold.

Lastly, we took a different route to exploit existing knowledge in Chapter 5. We showed that meta-learning can be used to learn a policy tailored for exploration; a central problem within RL. We formulated the problem as a meta-MDP and showed that the learned policy, not only leads to improved learning performance, but also specializes in exploration and is not merely a general policy to solve any task.

6.1 Future Work

In this thesis we developed a suite of techniques to exploit existing knowledge. Even though the results indicate that we are able to improve learning performance, there are several possibilities for improving and expanding the work presented in this thesis.

1. The methods for temporal abstraction presented here are able to obtain temporally-extended actions in an offline approach. The agent must first obtain a set of optimal trajectories. Our method then obtains a set of macros or options, and finally the agent is able to leverage those macros and options in new problems. This offline approach might be limiting; it is not clear how many trajectories are needed to obtain appropriate abstractions and, once they are obtained, they are not updated after solving subsequent tasks. One way of addressing this limitation would be to derive an online approach for learning temporal abstractions. Instead of learning a fixed set of macros or options after collecting several trajectories, the agent could learn them as new tasks are being solved in an incremental manner.

2. The method developed in Chapter 4 proved to be a particularly efficient approach for learning options. However, for pragmatic reasons, several design de-

cisions were made that were not optimal and resulted in considerable effort in hyperparameter tuning to obtain the reported results. Mainly, the hyperparameters used to balance minimizing the number of terminations and maximizing the probability of generating observed trajectories plays a crucial role in successfully implementing our method. These hyperparameters were introduced to account for the fact that longer trajectories are inherently less likely to occur, so it should be possible to come up a function based on the length of a trajectory that balances these terms. The same issue is true in how we define the threshold used to stop introducing new options. Addressing these two issues in a principled way would get rid of the need for hyperparameters altogether.

3. Our meta-MDP formulation, presented in Chapter 5, suffers from some of the same limitations that plague meta-learning methods in general: the method is computationally expensive and it takes time to see clear benefits. We circumvent some of these limitations by parallelizing training across multiple tasks and using TD methods to update the meta-agent's policy. If we relaxed our problem definition and, instead of assuming the agent is presented with tasks to solve, we consider the case where the agent is *allowed* to pick which tasks it will practice on, we should be able to use techniques from curriculum learning to train on tasks that are simple enough that the agent can solve quickly but also allows the meta-agent to make significant progress. To the best of our knowledge, the use of curriculum learning as a mechanism to efficiently learn in meta-learning problems has not been explored in depth by the community.

4. There is interesting relationship between the work presented in Chapter 5 and Chapter 4. When learning options, an option policy and termination function will learn how to best adapt to the trajectories demonstrated but, naturally, it will not be able to adapt correctly to unseen states (i.e., it was never shown the appropriate behavior for states not present in those trajectories). In those states, not only is the option policy generally not well suited but the termination function might learn to never terminate. To deal with that situation,

we manually increased the probability of termination the longer an option had been running. A more robust approach would be to try to identify in which states the options are not expected to perform well and, upon encountering those states, let the agent behavior be guided by an exploration policy. Going even further, we might be able to apply the Meta-MDP framework to learn an *exploration option*, where both the option policy and termination function are trained to guide the agent in those unknown states.

These are some of the possibilities we have encountered for improving and extending our work, but it is by no means an exhaustive list of ways in which our work can be improved. The takeaway message from this study is that existing knowledge is often available; even if an agent starts learning from scratch, the experience it collects while solving a task can be leveraged to improve learning in subsequent problems. We hope that this thesis helps guide future studies and that, whenever possible, machine learning practitioners will consider leveraging experience as a key step in making RL a practical solution to research problems.

BIBLIOGRAPHY

Marcin Andrychowicz, Misha Denil, Sergio Gomez Colmenarejo, Matthew W. Hoffman, David Pfau, Tom Schaul, and Nando de Freitas. Learning to learn by gradient descent by gradient descent. *CoRR*, abs/1606.04474, 2016. URL `http://arxiv.org/abs/1606.04474`.

Brenna D. Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. A survey of robot learning from demonstration. *Robot. Auton. Syst.*, 57(5):469–483, May 2009. ISSN 0921-8890. doi: 10.1016/j.robot.2008.10.024. URL `http://dx.doi.org/10.1016/j.robot.2008.10.024`.

Pierre-Luc Bacon, Jean Harb, and Doina Precup. The option-critic architecture. In *AAAI*, 2017.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 41–48, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-516-1. doi: 10.1145/1553374.1553380. URL `http://doi.acm.org/10.1145/1553374.1553380`.

Adi Botea, Markus Enzenberger, Martin Müller, and Jonathan Schaeffer. Macro-ff: Improving ai planning with automatically learned macro-operators. *Journal of Artificial Intelligence Research*, 24:581–621, 2005.

Haitham Bou-Ammar, Eric Eaton, Paul Ruvolo, and Matthew E. Taylor. Unsupervised cross-domain transfer in policy gradient reinforcement learning via manifold alignment. In *AAAI*, pages 2504–2510. AAAI Press, 2015.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016. URL `http://arxiv.org/abs/1606.01540`. cite arxiv:1606.01540.

Thomas G. Dietterich. The maxq method for hierarchical reinforcement learning. In *In Proceedings of the Fifteenth International Conference on Machine Learning*, pages 118–126. Morgan Kaufmann, 1998.

Kurt Driessens and Sašo Džeroski. Integrating guidance into relational reinforcement learning. *Mach. Learn.*, 57(3):271–304, December 2004. ISSN 0885-6125. doi: 10.1023/B:MACH.0000039779.47329.3a. URL `https://doi.org/10.1023/B:MACH.0000039779.47329.3a`.

Forrest Elliott and Manfred Huber. Learning macros with an enhanced lz78 algorithm. In *FLAIRS Conference*, 2005.

Jeffrey L. Elman. Learning and development in neural networks: The importance of starting small. cognition. In *Cognition*, page 781–799, 1993.

Zhengzhu Feng, Eric A. Hansen, and Shlomo Zilberstein. Symbolic generalization for on-line planning. In *Proceedings of the Nineteenth Conference on Uncertainty in Artificial Intelligence*, UAI'03, pages 209–216, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers Inc. ISBN 0-127-05664-5. URL `http://dl.acm.org/citation.cfm?id=2100584.2100609`.

Richard E. Fikes, Peter E. Hart, and Nils J. Nilsson. *Learning and Executing Generalized Robot Plans*, page 485–503. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993. ISBN 1558601635.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1126–1135, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR. URL `http://proceedings.mlr.press/v70/finn17a.html`.

D. H. Grollman and O. C. Jenkins. Learning robot soccer skills from demonstration. *2007 IEEE 6th International Conference on Development and Learning*, pages 276–281, 2007.

Eric Hansen, Andrew G. Barto, and Shlomo Zilberstein. Reinforcement learning for mixed open-loop and closed-loop control. In *Proceedings of the Ninth Neural Information Processing Systems Conference*, pages 1026–1032, Denver, Colorado, 1996. URL `http://rbr.cs.umass.edu/shlomo/papers/HBZnips96.html`.

Jean Harb, Pierre-Luc Bacon, Martin Klissarov, and Doina Precup. When waiting is not an option: Learning options with a deliberation cost. In *AAAI*, 2018.

Anna Harutyunyan, Will Dabney, Diana Borsa, Nicolas Heess, Remi Munos, and Doina Precup. The termination critic. In *AISTAT*, 2019.

Rein Houthooft, Xi Chen, Yan Duan, John Schulman, Filip De Turck, and Pieter Abbeel. Curiosity-driven exploration in deep reinforcement learning via bayesian neural networks. *CoRR*, abs/1605.09674, 2016. URL `http://arxiv.org/abs/1605.09674`.

David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, September 1952.

Ramon Ferrer i Cancho and David Lusseau. Efficient coding in dolphin surface behavioral patterns. *Complexity*, 14:23–25, 2009.

Ramon Ferrer i Cancho, Antoni Hernández-Fernández, David Lusseau, Govindasamy Agoramoorthy, Minna J. Hsu, and Stuart Semple. Compression as a universal principle of animal behavior. *Cognitive science*, 37 8:1565–78, 2013.

Ramnandan Krishnamurthy, Aravind S. Lakshminarayanan, Peeyush Kumar, and Balaraman Ravindran. Hierarchical reinforcement learning using spatio-temporal abstractions and deep neural networks. *CoRR*, 2016.

Markus Kuderer, Shilpa Gulati, and Wolfram Burgard. Learning driving styles for autonomous vehicles from demonstration. 2015:2641–2646, 06 2015.

Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *CoRR*, 2015.

Bingyao Liu, Satinder P. Singh, Richard L. Lewis, and Shiyin Qin. Optimal rewards in multiagent teams. In *2012 IEEE International Conference on Development and Learning and Epigenetic Robotics, ICDL-EPIROB 2012, San Diego, CA, USA, November 7-9, 2012*, pages 1–8, 2012. doi: 10.1109/DevLrn.2012.6400862.

Ning Liu, Zhe Li, Jielong Xu, Zhiyuan Xu, Sheng Lin, Qinru Qiu, Jian Tang, and Yanzhi Wang. A hierarchical framework of cloud resource allocation and power management using deep reinforcement learning. *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 372–382, 2017.

Sridhar Mahadevan. Proto-value functions: Developmental reinforcement learning. In *Proceedings of the 22Nd International Conference on Machine Learning*, ICML '05, pages 553–560, New York, NY, USA, 2005. ACM. ISBN 1-59593-180-5. doi: 10.1145/1102351.1102421.

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

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

Amy McGovern and Andrew G. Barto. Automatic discovery of subgoals in reinforcement learning using diverse density. In *Proceedings of the Eighteenth International Conference on Machine Learning*, ICML '01, pages 361–368, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc. ISBN 1-55860-778-1. URL `http://dl.acm.org/citation.cfm?id=645530.655681`.

Ishai Menache, Shie Mannor, and Nahum Shimkin. Q-cut - dynamic discovery of sub-goals in reinforcement learning. In *Proceedings of the 13th European Conference on Machine Learning*, ECML '02, pages 295–306, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-44036-4. URL `http://dl.acm.org/citation.cfm?id=645329.650060`.

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. Human-level control through deep reinforcement learning. *Nature*, 518(7540): 529–533, February 2015. ISSN 00280836. URL `http://dx.doi.org/10.1038/nature14236`.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1928–1937, New York, New York, USA, 20–22 Jun 2016. PMLR. URL `http://proceedings.mlr.press/v48/mniha16.html`.

K. Mülling, J. Kober, and J. Peters. Learning table tennis with a mixture of motor primitives. pages 411–416, Piscataway, NJ, USA, December 2010. Max-Planck-Gesellschaft, IEEE.

M. A. Hakim Newton, John Levine, Maria Fox, and Derek Long. Learning macro-actions for arbitrary planners and domains. In *ICAPS*, 2007.

Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Trans. on Knowl. and Data Eng.*, 22(10):1345–1359, October 2010. ISSN 1041-4347. doi: 10.1109/TKDE.2009.191. URL `http://dx.doi.org/10.1109/TKDE.2009.191`.

Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. In *Proceedings of the 1997 Conference on Advances in Neural Information Processing Systems 10*, NIPS '97, pages 1043–1049, Cambridge, MA, USA, 1998. MIT Press. ISBN 0-262-10076-2. URL `http://dl.acm.org/citation.cfm?id=302528.302894`.

Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. *CoRR*, abs/1705.05363, 2017. URL `http://arxiv.org/abs/1705.05363`.

V. V. Phansalkar and M. A. L. Thathachar. Local and global optimization algorithms for generalized learning automata. *Neural Comput.*, 7(5):950–973, September 1995. ISSN 0899-7667. doi: 10.1162/neco.1995.7.5.950.

Doina Precup. *Temporal Abstraction in Reinforcement Learning.* PhD thesis, 2000. AAI9978540.

Jette Randl. Learning macro-actions in reinforcement learning. In *NIPS*, 1998.

Clemens Rosenbaum, Tim Klinger, and Matthew Riemer. Routing networks: Adaptive selection of non-linear functions for multi-task learning. *CoRR*, abs/1711.01239, 2017. URL `http://arxiv.org/abs/1711.01239`.

David Salomon and Giovanni Motta. *Handbook of Data Compression.* Springer Publishing Company, Incorporated, 5th edition, 2009. ISBN 1848829027, 9781848829022.

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL `http://arxiv.org/abs/1707.06347`.

David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In Tony Jebara and Eric P. Xing, editors, *Proceedings of the 31st International Conference on Machine Learning (ICML-2014)*, pages 387–395. JMLR Workshop and Conference Proceedings, 2014.

Siddharth Srivastava, Neil Immerman, and Shlomo Zilberstein. Merging example plans into generalized plans for non-deterministic environments. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: Volume 1 - Volume 1*, AAMAS '10, page 1341–1348, Richland, SC, 2010. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 9780982657119.

Siddharth Srivastava, Neil Immerman, and Shlomo Zilberstein. A new representation and associated algorithms for generalized planning. *Artificial Intelligence*, 175: 615–647, 02 2011. doi: 10.1016/j.artint.2010.10.006.

Kaushik Subramanian, Charles Lee Isbell, and Andrea Lockerd Thomaz. Exploration from demonstration for interactive reinforcement learning. In *AAMAS*, 2016.

Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998. ISBN 0262193981.

Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 2nd edition, 2018. ISBN 9780262039246.

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

Cs. Szepesvári. The asymptotic convergence-rate of q-learning. In *Proceedings of the 10th International Conference on Neural Information Processing Systems*, NIPS'97, pages 1064–1070, Cambridge, MA, USA, 1997. MIT Press. URL `http://dl.acm.org/citation.cfm?id=3008904.3009053`.

Matthew E. Taylor and Peter Stone. Transfer learning for reinforcement learning domains: A survey. *J. Mach. Learn. Res.*, 10:1633–1685, December 2009. ISSN 1532-4435. URL http://dl.acm.org/citation.cfm?id=1577069.1755839.

Matthew E. Taylor, Halit Bener Suay, and Sonia Chernova. Integrating reinforcement learning with human demonstrations of varying ability. In *AAMAS*, 2011.

Georgios Theocharous, Philip S. Thomas, and Mohammad Ghavamzadeh. Personalized ad recommendation systems for life-time value optimization with guarantees. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, pages 1806–1812. AAAI Press, 2015. ISBN 978-1-57735-738-4. URL http://dl.acm.org/citation.cfm?id=2832415.2832500.

Philip S. Thomas and Andrew G. Barto. Conjugate markov decision processes. In *Proceedings of the 28th International Conference on Machine Learning, ICML 2011, Bellevue, Washington, USA, June 28 - July 2, 2011*, pages 137–144, 2011.

Philip S. Thomas, Bruno Castro da Silva, Andrew G. Barto, and Emma Brunskill. On ensuring that intelligent machines are well-behaved. *CoRR*, abs/1708.05448, 2017. URL http://arxiv.org/abs/1708.05448.

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

T. A. Welch. A technique for high-performance data compression. *Computer*, 17(6): 8–19, June 1984. ISSN 0018-9162. doi: 10.1109/MC.1984.1659158. URL http://dx.doi.org/10.1109/MC.1984.1659158.

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

Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8(3-4):229–256, May 1992. ISSN 0885-6125. doi: 10.1007/BF00992696. URL https://doi.org/10.1007/BF00992696.