A Study of Value-Aware Eigenoptions

by

Harshil Kotamreddy

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

The standard reinforcement learning (RL) paradigm entails an agent taking an action every time step in order to maximize environmental reward. A key aspect of intelligence that is not captured in the standard RL framework is the ability to operate at multiple timescales. A typical RL agent performs an action at every time step, with a decision being made at every time step. Intelligent beings in the real world (such as humans) typically do not make decisions about individual joint or muscle movements and instead operate at larger timescales. The options framework allows for temporal abstraction and allows agents to operate at scales beyond a single time step by abstracting away decisions about single time step actions when an option is initiated. Eigenoptions are a method for option discovery derived from the successor representation. Since the successor representation encodes diffusive information about the environment, eigenoptions are potentially a good candidate for credit assignment. However, in the scope of model-free RL, eigenoptions have been used strictly for exploration. In this thesis, we explore the utility of eigenoptions for credit assignment in tabular and pixel-based grid worlds and demonstrate both the benefits and difficulties of using eigenoptions for credit assignment.

Preface

Much of this thesis is based on a paper presented at the Inductive Biases in Reinforcement Learning Workshop at the Reinforcement Learning Conference (Kotamreddy and Machado, 2025).

To Amma and Nana,

For pushing me to be a better version of myself every day.

Acknowledgements

First of all, I would like to thank my wonderful supervisor, Marlos, for being a firm guiding hand throughout my work on this thesis. He pushed me to do more and even when things did not work, he made me ask and understand *why* things did not work. He taught me what it means to be a good researcher and for that I will be forever grateful. I would also like to thank Matt Taylor, who guided me during my first year and made me carefully consider what kind of research I wanted to do. And while I only spoke to him briefly during classes, I am thankful to Rich Sutton, who, through his many lectures and talks, has shaped my thoughts surrounding artificial intelligence and what an intelligent agent might look like.

I would also like to thank Marcos and Mohamed, my fellow labmates and MSc students, with whom I spent many long nights working on class projects and assignments during my first year and discussing our research during my second year. They have always been there for me when I needed them. I am grateful for Aniket, who was always helpful during classes and dragged me to the gym when I didn't want to go. I would also like to thank my roommate Shivam, who always entertained my unpolished ideas and made me think through them deeply. I am also grateful for Prahbat, who served as my tennis partner during the few months I played, and who always inspired me to dig deeper and think carefully about research questions.

I am grateful to everyone else who made me feel welcome at RLAI: Rick, Siddarth, Diego, Dikshant, Shuai, Haseeb, Kushagra, Srinjoy, Kevin, Parham, and Anna. I would also like to thank those that I met outside of campus who made me feel at home in Edmonton: Alex, Dan, Morpheus, Deni, Mar, Wendy, Gabrielle, and Sobayer.

Finally, I would like to acknowledge my parents, who have always pushed me to better myself and have always supported me regardless of circumstance.

Contents

List of Algorithms

List of Figures

Chapter 1

Introduction

Reinforcement learning (RL) typically involves an agent interacting with an environment by taking actions. During this process, the agent learns a policy over actions that maximizes its sum of discounted rewards. Although the standard RL paradigm has achieved many successes recently (e.g., Berner et al., 2019; Degrave et al., 2022; Ouyang et al., 2022; Silver et al., 2016, 2018), the use of temporal abstractions, such as options (Sutton et al., 1999b), is relatively underexplored. Options take a slightly different approach to the RL paradigm by introducing the notion of multi-step actions. Actions are now not just limited to a single time step, but can be taken over several time steps by following an option policy that is different from the agent's own policy. This enables the agent to act over multiple timescales, similar to the way humans plan and make decisions. Although the use of options has not seen as much success as the usual RL paradigm, options handcrafted by human experts were used in several high-profile cases and proved instrumental to the system's success (e.g., Bellemare et al., 2020; Vinyals et al., 2019). These successes highlight the potential of the options framework.

Moreover, there is substantial room for improvement with option-based methods. In particular, autonomous discovery of such options continues to be a rich and active area of research (e.g., Bacon et al., 2017; Bagaria and Konidaris, 2019; Dayan and Hinton, 1992; Eysenbach et al., 2019; Harb et al., 2018; Konidaris and Barto, 2009; Vezhnevets et al., 2017). The diversity of these methods highlights areas in the field of RL where option-based methods have the potential to help, including credit assignment (Solway et al., 2014; Sutton et al., 1999b), exploration (Jinnai et al., 2019;

Jong et al., 2008), and generalization (Konidaris and Barto, 2007).

A particularly promising method of option discovery is through eigenoptions (Machado et al., 2017; Machado and Bowling, 2016). Eigenoptions are learned through the successor representation (Dayan, 1993), which encodes the temporal distance between states. They were originally introduced to address exploration challenges and have been shown to scale, achieving state-of-the-art performance in 3D navigation tasks and Atari 2600 games (Klissarov and Machado, 2023). However, even though the successor representation encodes information about how information diffuses through the environment, which is closely related to temporal credit assignment, few works have used eigenoptions for credit assignment (e.g., Liu et al., 2017; Sutton et al., 2023). Furthermore, these works did not explicitly investigate whether eigenoptions used for credit assignment provided additional benefit over simply using them for exploration.

1.1 Thesis Statement

In this work, we explicitly investigate whether learning a policy over both primitive actions and eigenoptions (i.e., assigning credit to eigenoptions in addition to primitive actions) in model-free RL provides additional benefit over using them strictly for exploration. Specifically, we investigate whether learning a policy over eigenoptions and primitive actions leads to faster learning. We find that when eigenoptions are pre-computed, learning a policy over options leads to significantly faster learning. However, when eigenoptions are learned online, the benefits are not nearly as significant. We also investigate whether learning a policy over options provides learning benefits in the function approximation setting. We find that when eigenoptions are approximated, correctly estimating when an eigenoption should terminate plays an important role in the effectiveness of credit assignment.

1.2 Contributions

In this thesis we:

1. Demonstrate the benefits of learning option-values for pre-computed eigenop-

tions in MDPs.

2. Introduce Value-Aware Covering Eigenoptions (VACE), a method to learn option-values for eigenoptions learned online.

3. Introduce Deep Value-Aware Eigenoptions (DVAEO), a deep RL method to learn option-values for eigenoptions.

1.3 Thesis Structure

In Chapter 2, we provide the necessary background to understand the topics covered in the thesis. In Chapter 3, we conduct experiments in the tabular setting. Specifically, we investigate eigenoptions for credit assignment when they are provided in advance and when they are learned online. We find that eigenoptions provided in advance significantly speed up learning in the tabular setting. However, when eigenoptions are learned online, we find that early inaccuracies in the learned options and their option values skews the agent's behavior in a way that hinders its ability to fully explore the environment. This, in turn, hinders credit assignment as the agent is not always able to find goal states quickly. In Chapter 4, we analyze the implications of approximating both option-values and option policies in the non-linear function approximation setting. We introduce a method to learn option-values in the non-linear function approximation setting. We find that when approximating options, approximating termination conditions well is especially important for eigenoptions to be effective when used for credit assignment. In Chapter 5, we summarize our contributions and discuss potential future work. We provide experimental details and information about hyperparameter sweeps in Appendix A.

Chapter 2

Background

In this chapter, we provide background and context to better understand the algorithms and experiments described in later chapters. We use capital letters to designate random variables, calligraphic font for sets, lowercase bold letters for vectors, and uppercase bold letters to denote matrices. $\Delta(\mathcal{X})$ is the set of probability distributions over the set $\mathcal{X}$.

2.1 Reinforcement Learning

In the reinforcement learning (RL) setting, an agent interacts with an environment with the aim of maximizing reward. RL problems are typically represented by a Markov Decision Process (MDP). An MDP is defined as a tuple $\langle \mathcal{S}, \mathcal{A}, p, r \rangle$ where $\mathcal{S}$ is the set of all states, $\mathcal{A}$ is the set of all actions, $p(s'|s,a) = \mathbb{P}[S_{t+1} = s'|S_t = s, A_t = a]$ is the transition probability kernel, and $r(s,a) = \mathbb{E}[R_{t+1}|S_t = s, A_t = a]$ is the expected reward given the agent takes action a in state s. At every time step t, the agent is in state S_t and interacts with the environment by taking action A_t; then the agent moves to state S_{t+1} according to p and receives a reward R_{t+1}.

The agent learns a policy $\pi : \mathcal{S} \to \Delta(\mathcal{A})$ that maximizes the expected discounted return, $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$, where $\gamma \in [0,1)$ is the discount factor. There are two main paradigms for learning the optimal policy: value-based methods and policy gradient methods.

2.1.1 Value-based Methods

Value based methods estimate either the state value function $v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$ or the state-action value function $q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$ and calculate the policy using the value function. In this work, we focus on value-based methods that estimate the state-action value function q_π for the optimal policy, π^*.

Q-learning

Q-learning (Watkins and Dayan, 1992) is a temporal difference method that estimates q_{π^*} using the update rule

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left(R_{t+1} + \gamma \max_{a \in \mathcal{A}} Q(S_{t+1}, a) - Q(S_t, A_t) \right), \tag{2.1}$$

where α is the step size. In this work, we use ϵ-greedy exploration. At each time step, the agent takes a random exploratory action with probability ϵ. Otherwise, the agent takes an action according to the policy π, which is defined as $\pi(s) = \text{argmax}_{a \in \mathcal{A}} Q(s, a)$. Q-learning tends to overestimate action-values due to maximization bias because it takes the max over all possible future actions in its update rule (van Hasselt, 2010). The maximum over any set of samples is always greater than or equal to the expectation over the set of samples. The same is true in Q-learning where we take the max over all possible future actions instead of simply using the value of the next state-action pair in our update. Over a large enough set of updates, Q-learning will tend to overestimate values due to the maximization bias in the target.

Double Q-learning

Double Q-learning (van Hasselt, 2010) aims to reduce overestimation by learning two separate state-action value functions, Q^A and Q^B, concurrently with half the samples being used to update the first value function and half the samples being used to

update the second. Each action-value function is used to update the other:

$$\begin{aligned} Q^A(S_t, A_t) \leftarrow & Q^A(S_t, A_t) + \alpha(R_{t+1} + \\ & \gamma Q^B(S_{t+1}, \operatorname*{argmax}_a Q^A(S_{t+1}, a)) - Q^A(S_t, A_t)), \end{aligned} \tag{2.2}$$

$$\begin{aligned} Q^B(S_t, A_t) \leftarrow & Q^B(S_t, A_t) + \alpha(R_{t+1} + \\ & \gamma Q^A(S_{t+1}, \operatorname*{argmax}_a Q^B(S_{t+1}, a)) - Q^B(S_t, A_t)). \end{aligned} \tag{2.3}$$

In practice, one of the updates is chosen randomly at every time step.

Watkins' Q(λ)

The n-step return,

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n Q_{t+n-1}(S_{t+n}, A_{t+n}), \tag{2.4}$$

is the target for an arbitrary n-step TD update, where Q_k is the estimate of the state-action value function at time step k. The λ-return, $G_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} G_{t:t+n}$, averages all n-step updates in an exponentially weighted manner. Watkins' $Q(\lambda)$ (Watkins, 1989) computes the λ-return online through the use of eligibility traces. The eligbility trace for a state-action pair is an accumulating trace:

$$e(s,a) \leftarrow \mathbb{1}_{S_t=s, A_t=a} + \begin{cases} \gamma\lambda e(s,a) & \text{if } Q(S_t, A_t) = \max_a Q(S_t, a) \\ 0 & \text{otherwise,} \end{cases} \tag{2.5}$$

where $\mathbb{1}_{S_t=s, A_t=a}$ is an indicator function that is equal to 1 if $S_t = s$ and $A_t = a$, and 0 otherwise. The Q-learning update then becomes:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') - Q(S_t, A_t) \right) e(s,a). \tag{2.6}$$

This allows Watkins' $Q(\lambda)$ to calculate a λ-return that looks ahead until the first exploratory action is taken, after which the trace is set to zero.

2.1.2 Policy Gradient Methods

Policy gradient methods estimate the policy directly by defining a policy, $\pi(s|a, \boldsymbol{\theta})$, that is differentiable with respect to its parameters, $\boldsymbol{\theta}$. Such methods perform gradient ascent updates to maximize the objective function $J(\boldsymbol{\theta}) = v_{\pi_{\boldsymbol{\theta}}}(s_0)$, which is the value

of the policy at the start state. The policy gradient theorem (Sutton et al., 1999a) tells us that

$$\nabla_\theta J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla_\theta \pi(s|a,\boldsymbol{\theta}) \tag{2.7}$$

$$= \mathbb{E}_\pi \left[\sum_a q_\pi(S_t,a) \nabla_\theta \pi(a|S_t,\boldsymbol{\theta}) \right] \tag{2.8}$$

$$= \mathbb{E}_\pi \left[\sum_a \pi(a|S_t,\boldsymbol{\theta}) q_\pi(S_t,a) \frac{\nabla_\theta \pi(a|S_t,\boldsymbol{\theta})}{\pi(a|S_t,\boldsymbol{\theta})} \right] \tag{2.9}$$

$$= \mathbb{E}_\pi \left[q_\pi(S_t,A_t) \nabla_\theta \ln \pi(a|S_t,\boldsymbol{\theta}) \right] \tag{2.10}$$

$$= \mathbb{E}_\pi \left[G_t \nabla_\theta \ln \pi(a|S_t,\boldsymbol{\theta}) \right], \tag{2.11}$$

where $\mu(s) = \sum_{t=0}^{\infty} \gamma^t \mathbb{P}[S_t = s | S_0 = s_0, A_{0:t\sim\pi}]$ is the discounted on-policy distribution. This leads to the REINFORCE (Williams, 1992) update for $\boldsymbol{\theta}$:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha G_t \nabla_\theta \ln \pi(a|S_t,\boldsymbol{\theta}). \tag{2.12}$$

Using a baseline in the update reduces variance:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha (G_t - b(s)) \nabla_\theta \ln \pi(a|S_t,\boldsymbol{\theta}). \tag{2.13}$$

Actor-critic methods use parameterized value functions to compute the one-step return rather than the full return. They also use a parametrized value function as the baseline in order to reduce variance. An actor-critic method with a state-value function $\hat{v}(s,\boldsymbol{w})$ parameterized with weights $\boldsymbol{w}$ would use the following update:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \left(R_{t+1} + \hat{v}(S_{t+1},\boldsymbol{w}) - \hat{v}(s,\boldsymbol{w}) \right) \nabla_\theta \ln \pi(a|S_t,\boldsymbol{\theta}). \tag{2.14}$$

The value functions can also be updated using the temporal difference error:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha \left(R_{t+1} + \hat{v}(S_{t+1},\boldsymbol{w}) - \hat{v}(s,\boldsymbol{w}) \right) \nabla_{\boldsymbol{w}} \hat{v}(S_t,\boldsymbol{w}). \tag{2.15}$$

2.2 Options

Options in RL are a framework for temporal abstraction, letting agents take "actions" that last longer than a single time step. (Sutton et al., 1999b). An option is defined as a tuple, $o = \langle \mathcal{I}_o, \pi_o, \beta_o \rangle$, where $\mathcal{I}_o \subseteq \mathcal{S}$ is the option's initiation set, $\pi_o : \mathcal{S} \to \Delta(\mathcal{A})$ is

the option's policy, and $\beta_o : \mathcal{S} \to [0, 1]$ is the option's termination function. An option can be taken in any state within the initiation set, $\mathcal{I}_o$, after which it acts according to the option policy, π_o until it terminates according to β_o. Note that actions in $\mathcal{A}$ are one-step options.

Given a set of options Ω (which may or may not include one-step actions), agents can learn a policy, $\pi : \mathcal{S} \to \Delta(\Omega)$, over options. The reward for following an option o at state s is defined as:

$$r(s, o) = \mathbb{E}_{\pi_o, \beta_o}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{k-1} R_{t+k} | S_t = s], \tag{2.16}$$

where k is the number of steps the option took to terminate. The option-value function is defined as:

$$q_\pi(s, o) = \mathbb{E}[G_t | S_t = s, O_t = o] \tag{2.17}$$

$$= r(s, o) + \sum_{s'} p(s'|s, o) \sum_{o' \in \Omega_{s'}} \pi(o'|s') q_\pi(s', o'), \tag{2.18}$$

where $p(s'|s, o)$ is the probability that the agent reaches a state s' given that it followed an option o that was initiated in state s, and Ω_s is the set of all options that can be initiated in state s.

2.2.1 Constructing Options

While having a framework for temporal abstraction is nice, not all options constructed using this framework are going to be useful. Finding methods to construct useful options has been and continues to be an active field of research.

Bottleneck States

Traditional methods for constructing options involved creating options that lead to bottleneck states. Bottleneck states are states that the agent has to go through in order to reach new parts of the state space that are potentially more interesting. Because the agent must go through them to reach different parts of the state space, they occur often in trajectories that lead to different goal states.

Sutton et al. (1999b) define eight *hallway* options that lead to the bottleneck states. Two options are defined for each room. The options can be initiated anywhere

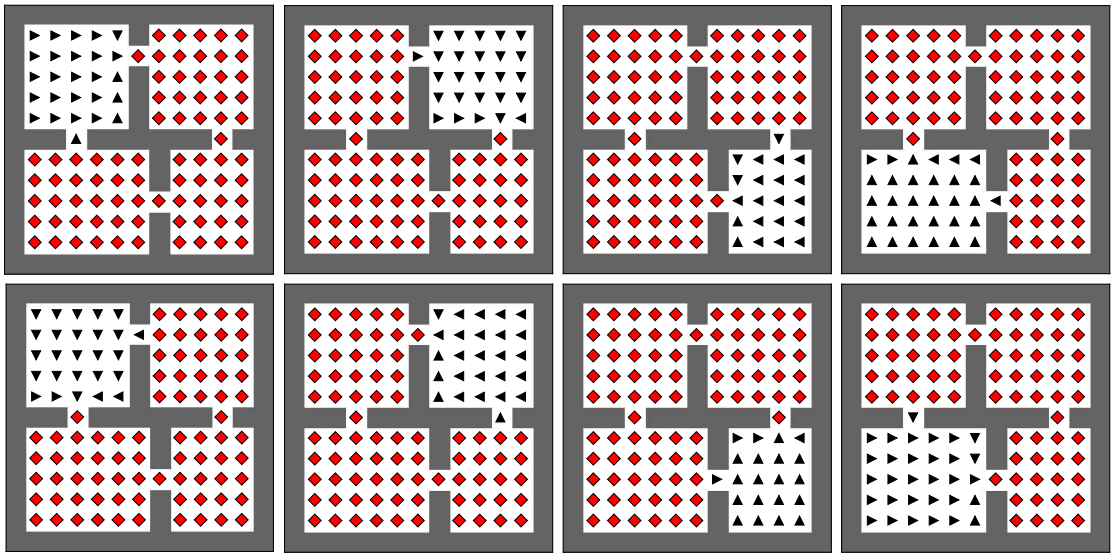

Figure 2.1: The eight hallway options in the four rooms domain. Red diamonds represent termination states.

within a room and the options' policies direct the agent toward each of the bottleneck states, terminating when the agent reaches them. This is made clearer in Figure 2.1, which shows each of the eight hallway options. We use these options as a baseline in Section 3.1 since they have been shown to be particularly good for credit assignment, minimizing the geometric mean number of trial-and-error attempts necessary for the agent to discover the optimal policy for any task (Solway et al., 2014).

A wide range of methods have been explored to discover bottleneck states. Several methods use graph-based approaches to construct a graph of the MDP and identify bottleneck states (Menache et al., 2002; Şimşek and Barto, 2008; Şimşek et al., 2005). Other approaches use diverse density (McGovern and Barto, 2001b; Stolle and Precup, 2002).

Eigenoptions

Eigenoptions (Machado et al., 2017, 2018) are options discovered through the eigenvectors of the successor representation (SR) (Dayan, 1993). The SR encodes the "temporal distance" between states. It represents each state s as an $|\mathcal{S}|$ dimensional vector that contains the expected discounted visitation from s to every state $s' \in \mathcal{S}$. The SR matrix, $\mathbf{\Psi}_\pi$, with respect to a policy π, can be calculated using $\mathbf{P}_\pi$, the transition probability matrix induced by π:

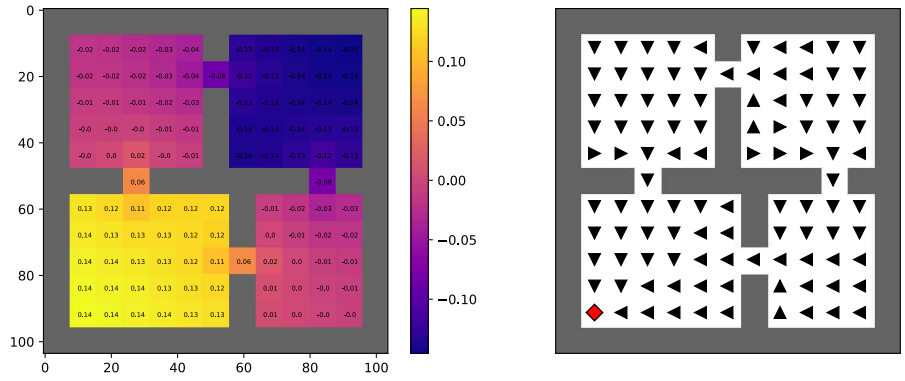

Figure 2.2: The top eigenvector of the SR and its corresponding option policy. The learned option policy leads to the state with the highest value.

$$\boldsymbol{\Psi}_\pi = (\mathbf{I} - \gamma \mathbf{P}_\pi)^{-1}, \tag{2.19}$$

where $\mathbf{I}$ is the identity matrix. When $\mathbf{P}_\pi$ is unknown, given a step size η, the SR can be estimated from samples using the temporal difference update rule for state S_t at time step t:

$$\hat{\Psi}(S_t, j) \leftarrow \hat{\Psi}(S_t, j) + \eta \left[\mathbb{1}_{\{S_t = j\}} + \gamma \hat{\Psi}(S_{t+1}, j) - \hat{\Psi}(S_t, j) \right] \quad \forall j \in \mathcal{S}. \tag{2.20}$$

Eigenoptions are options that maximize an intrinsic reward function defined by the eigenvectors of the SR. The intrinsic reward function used to generate an option policy using eigenvector $\boldsymbol{e}$ of $\boldsymbol{\Psi}_\pi$ is:

$$r^{\boldsymbol{e}}(s, s') = \boldsymbol{e}^\top (\boldsymbol{\phi}(s') - \boldsymbol{\phi}(s)) \quad \forall s, s' \in \mathcal{S}, \tag{2.21}$$

where $\boldsymbol{\phi}(s)$ is the feature representation of state s. In the tabular case, when $\boldsymbol{\phi}$ is a one-hot vector, the reward function is defined as:

$$r^{\boldsymbol{e}}(s, s') = \boldsymbol{e}(s') - \boldsymbol{e}(s) \quad \forall s, s' \in \mathcal{S}. \tag{2.22}$$

We can use a learning algorithm such as Q-learning to learn an option policy using the intrinsic reward function. The option policy learned from such a reward function directs the agent toward states with the highest value. Since the reward function only provides negative reward for leaving these high-valued states, such states are added

to the termination set. Thus, an eigenoption defined on an eigenvector of the SR takes the agent toward the highest-valued states of its respective eigenvector. This is shown more clearly in Figure 2.2, which shows the option policy corresponding to the top eigenvector of the SR.

Eigenoptions can also be discovered online through the Representation-Driven Option Discovery (ROD) cycle (Machado, 2025; Machado et al., 2023). The cycle starts with an RL agent gathering data by interacting with its environment and then using this data to learn a representation of the environment. In this case, the agent learns the SR and its eigenvectors. The agent then uses the learned representation to create an intrinsic reward function (see Eq. 2.21), and then learns an option policy to maximize it. Finally, the option's initiation set is defined as all states with positive Q-values, and the complement is defined as termination states. This can be done because the intrinsic reward function is naturally defined such that the agent receives negative rewards when it exits what should be a termination state.

Once an option is created, it is added to the agent's option set, and the cycle repeats. In this work, we use the top eigenvector of the SR to generate an option during each cycle. This option tends to direct the agent toward the edge of the region currently explored. Following such an option allows the agent to skip the random dithering required to reach new unexplored regions, leading to better exploration. So far, such options have only been used for exploration. Specifically, in an ϵ-greedy step, the agent might sample an option and act according to its policy until termination. Through this process, the agent explores areas of the environment that are less frequently visited, enabling much faster and more efficient exploration. This algorithm is called covering eigenoptions (CEO) (Machado et al., 2023).

Option Critic

Methods using the option critic framework (Bacon et al., 2017) learn options through the environmental reward rather than an intrinsic reward. The option policy and termination functions are learned using gradient ascent through the intra-option policy gradient and termination gradient theorems.

The intra-option policy gradient theorem gives us the gradient of the state-option

value function $q_\pi(s, o)$ with respect to $\boldsymbol{\theta}$, the parameters of option o:

$$\frac{\partial q_\pi(s,o)}{\partial \boldsymbol{\theta}} = \sum_{s,o} \mu_\pi(s, o|s_0, o_0) \sum_a \frac{\partial \pi_{o,\boldsymbol{\theta}}(a|s)}{\partial \boldsymbol{\theta}} q_u(s, o, a), \tag{2.23}$$

where $\mu_\pi(s, o|s_0, o_0) = \sum_{t=0}^{\infty} \gamma^t \mathbb{P}(s_t = s, o_t = o|s_0, o_0)$ is the discounted occupancy of state-option pairs along trajectories starting from (s_0, o_0), $\pi_{o,\boldsymbol{\theta}}$ is the option policy of option o with parameters $\boldsymbol{\theta}$, and $q_u(s, o, a)$ is the state-action-option value function. This leads to the following update rule:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha_{\boldsymbol{\theta}} \frac{\partial \ln \pi_{o,\boldsymbol{\theta}}(A_t|S_t)}{\partial \boldsymbol{\theta}} Q_u(S_t, O_t, A_t), \tag{2.24}$$

where $\alpha_{\boldsymbol{\theta}}$ is the step size for $\boldsymbol{\theta}$, the parameters of option o, and Q_u is the estimate of q_u. Updating $\boldsymbol{\theta}$ in this manner modifies the option policy and maximizes the state-option value function.

The termination gradient theorem gives us the gradient of the option value of option o upon arrival in state s, $U(s, o) = (1-\beta(s, o))q_\pi(s, o)+\beta(s, o)\sum_o \pi(o|s)q_\pi(s, o)$, with respect to $\boldsymbol{\vartheta}$, the parameters of the termination function of option o, β_o:

$$\frac{\partial U(s', o)}{\partial \boldsymbol{\vartheta}} = \sum_{o,s'} \mu_\pi(s', o|s_1, o_0) \frac{\partial \beta_{o,\boldsymbol{\vartheta}}(s')}{\partial \boldsymbol{\vartheta}} A_\pi(s', o), \tag{2.25}$$

where $\mu_\pi(s, o|s_1, o_0) = \sum_{t=0}^{\infty} \gamma^t \mathbb{P}(s_t = s, o_t = o|s_1, o_0)$ is the discounted occupancy of state-option pairs along trajectories starting from (s_1, o_0), $\beta_{o,\boldsymbol{\vartheta}}$ is the termination function of option o with parameters $\boldsymbol{\vartheta}$, and $A_\pi(s, o) = Q_\pi(s, o) - V_\pi(s)$ is the advantage function. This leads to the following update rule:

$$\boldsymbol{\vartheta} \leftarrow \boldsymbol{\vartheta} - \alpha_{\boldsymbol{\vartheta}} \frac{\beta_{o,\boldsymbol{\vartheta}}(s')}{\partial \boldsymbol{\vartheta}} \left(Q_\pi(s', o) - V_\pi(s')\right), \tag{2.26}$$

where $\alpha_{\boldsymbol{\vartheta}}$ is the step size for $\boldsymbol{\vartheta}$. Updating $\boldsymbol{\vartheta}$ in this manner modifies the termination function of option o and maximizes the value function of option o upon arrival.

We use option critic as a baseline against value-aware eigenoptions when we learn option policies online. Option-critic is a well-established method designed for credit assignment with options. Option critic learns a policy over options while the option policies are learned online. This is quite similar to the ROD cycle, where eigenoptions are learned online and added to the set of options as they are learned. The main

difference between the two is that option critic uses a set number of options that are improved over time while the number of options used in the ROD cycle increases as new options are discovered. Option critic also tends to learn options that emulate bottleneck options (Bacon et al., 2017), so they serve as a good baseline for credit assignment.

Other Methods

While the methods discussed so far are the ones we use and discuss in the remainder of this work, there are several other methods for option discovery. Skill chaining methods construct options backwards from a goal state in a manner that allows future options to reach their respective sub-goals (Konidaris and Barto, 2009). Skill chaining has also been extended to the deep RL setting (Bagaria and Konidaris, 2019). While the options generated through skill chaining can be used for rapid credit assignment, its main limitation is that options can only be constructed once the goal is reached, making it dependent on good exploration.

Feudal reinforcement learning (Dayan and Hinton, 1992) involves agents that operate on multiple hierarchies. Agents are composed of managers and workers where managers set subgoals that workers must learn to reach. Managers at the top of the hierarchy maximize environmental reward by setting goals while the workers maximize an intrinsic reward to reach the subgoals set by managers. This idea was extended to deep RL through feudal networks (Vezhnevets et al., 2017), which was further improved upon by several works (Levy et al., 2019; Nachum et al., 2018).

Empowerment maximization methods learn diverse options by maximizing the mutual information between an agent's actions and its future observations (Klyubin et al., 2005). Gregor et al. (2017) maximize mutual information between skills using a variational inference objective (Mohamed and Rezende, 2015). Eysenbach et al. (2019) improves upon this objective by learning options with high entropy.

Our descriptions of the methods in this subsection are very brief, and there are many more works that we did not discuss here. Klissarov et al. (2025) conducted a survey that discusses these methods and provides a more in-depth overview of the methods we discuss here.

2.2.2 Learning Option Values

In the model-based setting, agents have the flexibility to plan using options while never following an option. This enables the agent to propagate credit much more quickly without having to act according to an option policy. One such model-based technique is goal-space planning (Lo et al., 2024). Given a set of options that lead to subgoals, a model is learned over an abstract MDP constructed using the given options. The model is used to plan over subgoals and propagate subgoal values quickly, which can then be used to update state values that the agent uses to act in the environment.

In the model-free setting, which is the focus of this work, most option-based methods learn a policy over options for reusability and in order to learn more quickly. For instance, methods using the option critic framework learn a policy over options while learning the option policies. This is true for nearly all of the methods for option discovery that we have discussed so far with the notable exception of eigenoptions and most empowerment maximization methods, which were designed for exploration.

Learning a policy over options is very similar to learning a policy over primitive actions. When using value-based methods to learn a policy over options, we estimate the state-option value function. We can learn the state-option value function using the SMDP Q-Learning update rule upon option termination (Sutton et al., 1999b):

$$Q(S_t, O_t) \leftarrow Q(S_t, O_t) + \alpha \left[R + \gamma^K \max_{o \in \mathcal{O}_{S_{t+K}}} Q(S_{t+K}, o) - Q(S_t, O_t) \right], \tag{2.27}$$

where $o \in \mathcal{O}$ is an option, $\mathcal{O}$ is the set of all options, $Q(s, o)$ is the estimate of the optimal state-option value function, α is the step size, K is the number of time steps the option took before termination, R represents the cumulative discounted reward through the duration of the option, and $\mathcal{O}_S$ is the set of all options that contain state S within their initiation set, $\mathcal{I}_o$.

Intra-option Q-learning (Sutton et al., 1998) improves on SMDP Q-learning by updating $Q(s, o)$ while the agent is still acting according to an option's policy, in contrast to applying the update rule only upon an option's termination:

$$Q(S_t, O_t) \leftarrow Q(S_t, O_t) + \alpha \left[(R_{t+1} + \gamma U(S_{t+1}, O_t)) - Q(S_t, O_t) \right], \tag{2.28}$$

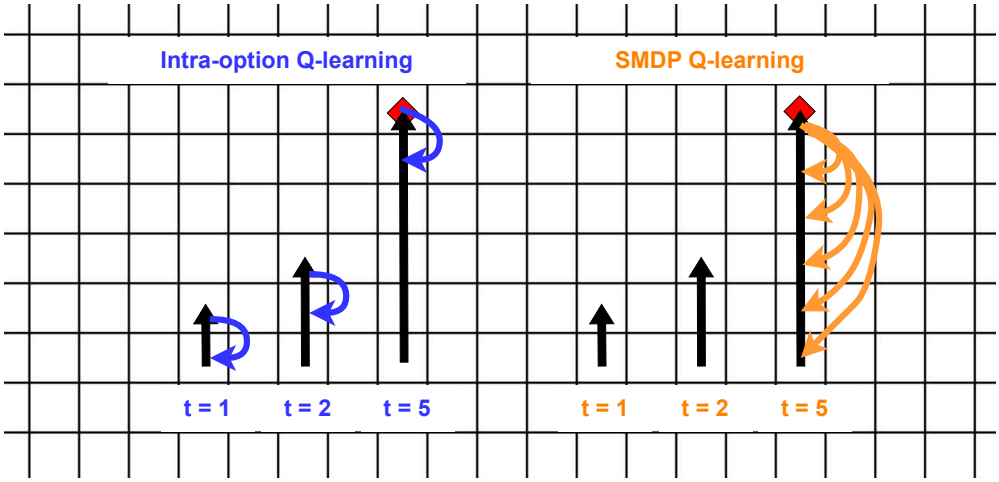

Figure 2.3: Option value updates with intra-option Q-learning (blue) and SMDP Q-learning (orange). Option termination is represented by a red diamond. Intra-option Q-learning performs a single step update during every time step. SMDP Q-learning performs updates for all state-action pairs visited while the option was being executed but only performs the updates upon termination.

where $U(s,o) = (1-\beta_o(s))\,Q(s,o) + \beta_o(s)\max_{o'\in\mathcal{O}_s} Q(s,o')$ is the option-value upon arrival. This allows us to perform state-option updates to all options if their policies take the same action as the current option and/or action.

Figure 2.3 shows the difference between intra-option Q-learning and SMDP Q-learning. Intra-option Q-learning performs a single step update every time step. However, it updates all other options along with the current option. SMDP Q-learning, on the other hand, updates all visited state-option pairs upon termination but only updates the option that was executed.

Similar to learning a policy over primitive actions, the policy over options can be defined based on the learned state-option value function:

$$\pi(s) = \operatorname*{argmax}_{o\in\mathcal{O}} Q(s,o). \tag{2.29}$$

2.3 Function Approximation and Deep RL

When the observation space is small and can be stored in memory, it is easy to represent value functions in tables. However, when observations are more complex (for

instance with the use of pixel-based observations) it is significantly more difficult, and sometimes impossible, to represent all possible observations in a table. To deal with this, we must approximate the value function by parameterizing it with some weights $\boldsymbol{\theta}$.

Linear function approximation methods represent the value function linearly with respect to the weights, for instance, $q(s, a; \boldsymbol{\theta}) = \boldsymbol{\theta}^\top \boldsymbol{x_t}$ where $\boldsymbol{x_t}$ is the feature vector. However, linear methods often require features to be hand-crafted for complex observation spaces. To deal with such observation spaces, deep neural networks are often used as non-linear function approximators.

2.3.1 Deep Q-Network

Deep Q-Network (DQN) was the first algorithm to incorporate deep neural networks into the RL domain to solve tasks with high-dimensional sensory input (Mnih et al., 2015). The algorithm, inspired by Q-learning, uses a neural network to approximate the Q-function instead of representing it with a tabular matrix. The loss function for the neural network on the i'th iteration is defined as:

$$L_i = \mathbb{E}_{s,a\sim\rho(\cdot)} \left[\left(y_i - Q(s, a; \boldsymbol{\theta}_i) \right)^2 \right], \tag{2.30}$$

where $y_i = \mathbb{E}\left[r + \gamma \max_{a'} Q\left(s', a'; \boldsymbol{\theta}^-\right) \middle| s, a \right]$ is the target, $\rho(s, a)$ is the behavior distribution (i.e. the state-action distribution based on the agent's behavior) and $\boldsymbol{\theta}$ are the parameters of the neural network that are optimized through gradient descent. In practice, the Q-network used in the target, also called the target network, typically uses the frozen parameters $\boldsymbol{\theta}^-$ of the main Q-network parameters $\boldsymbol{\theta}$ to maintain stability in the loss function. It is updated periodically and, each time it is updated, a new frozen copy of $\boldsymbol{\theta}$ replaces $\boldsymbol{\theta}^-$. A replay buffer is also used to control the sample distribution and make learning easier for the neural network.

2.3.2 Double Deep Q-Network

Double Deep Q-Network (DDQN) (Hasselt et al., 2016) aims to reduce DQN's overestimation bias by selecting the greedy action using the main network rather than

the target network when calculating the target. The target is now defined as:

$$y_i = \mathbb{E}\left[r + \gamma Q\left(s', \underset{a'}{\operatorname{argmax}}\, Q\left(s', a'; \boldsymbol{\theta}_i\right); \boldsymbol{\theta}_i^-\right) \middle| s, a\right]. \tag{2.31}$$

It is important to note that unlike Double Q-learning, which uses and learns a second Q-function, DDQN does not use an additional Q-function and simply uses the target network in the modified update rule. This allows for minimal change to the DQN architecture while attaining the benefits of Double Q-learning.

Chapter 3

Learning Option-Values for Eigenoptions in the Tabular Setting

In the tabular case, eigenoptions defined using the first n eigenvectors of the SR have been shown to significantly reduce the number of time steps required to visit all states of an environment (Machado et al., 2018). Additionally, perhaps because they capture information on how rewards diffuse in the environment, they have also been shown to be quite effective when used for planning (Sutton et al., 2023). Thus, a very natural question in the model-free setting is whether we should see even faster learning if we were to learn option-values for eigenoptions; using them not only for exploration but also for credit assignment. We call this approach value-aware eigenoptions (VAEO).

In Section 3.1, we consider bottleneck options as a baseline because they are the most traditional approach for credit assignment (McGovern and Barto, 2001a; Şimşek and Barto, 2008) and they can seen as forming an optimal behavioral hierarchy (Solway et al., 2014). Furthermore, they act as a fair comparison to pre-computed eigenoptions because they are also provided ahead of time and not learned online. In Section 3.2, we consider option critic as a baseline because it learns options online and the options that are learned are similar to bottleneck options Bacon et al., 2017.

In this work, we run all our experiments in modified versions of the Minigrid environment (Chevalier-Boisvert et al., 2023) using Gym (Brockman et al., 2016). The original Minigrid environment included actions to rotate the agent left or right by 90 degrees, and move the agent in the direction of its current orientation. We use

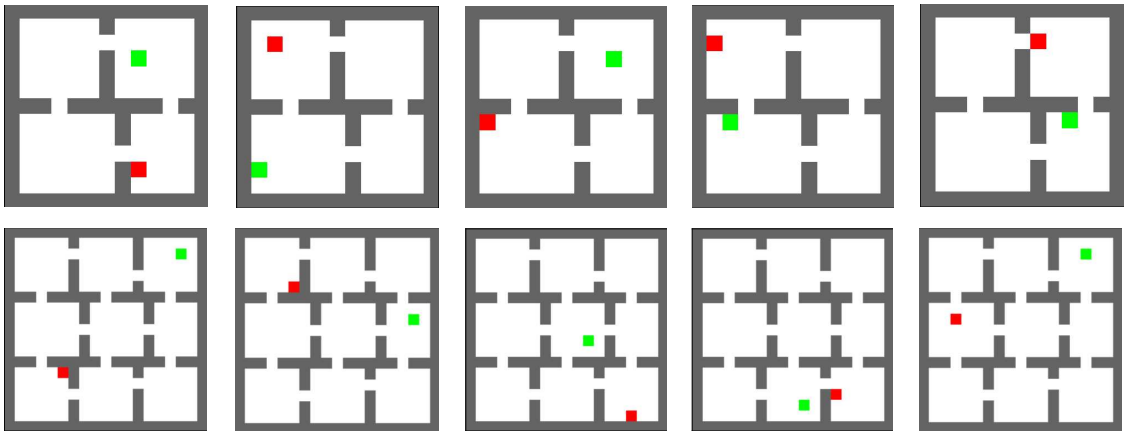

Figure 3.1: Environment configurations in the four rooms and nine rooms domains. Green squares represent start states and red squares represent goal states. The start and goal positions within these configurations were randomly generated. The agent can move in cardinal directions (up, down, left and right) at each time step.

an action set where the agent can move up, down, right and left. Episodes have a maximum length of 1,000 steps and environment transitions are deterministic. We detail the four rooms and nine rooms configurations that we use in our experiments in Figure 3.1. The agent receives zero reward at every transition except those that lead to the goal state, which lead to +1 reward.

3.1 Acting and Learning with Eigenoptions

We first focus on the benefits of value-aware eigenoptions (VAEO) in the best case scenario where options are provided ahead of time. Thus, we first consider options obtained through the closed form of the SR (Eq. 2.19). We demonstrate that in the tabular setting, learning option-values for pre-computed eigenoptions leads to faster learning when compared to using pre-computed eigenoptions strictly for exploration. We perform experiments in the four rooms and nine rooms domains with randomly generated start and goal state configurations (see Figure 3.1).

We generate eigenoptions using the top n eigenvectors of the SR. We learn the option policies using Q-learning with samples from random exploration of the environment, where the agent starts at a random location every episode. Episodes are 1,000 steps long, and in this phase, we do not treat goal states as absorbing states but as regular states. To learn option-values for these pre-computed eigenoptions,

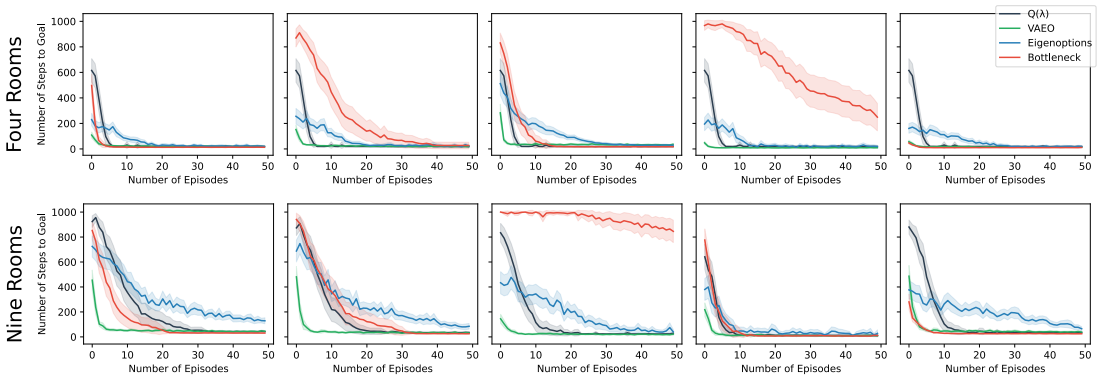

Figure 3.2: Performance of value-aware eigenoptions (VAEO) compared to baselines. VAEO outperforms the other algorithms in most configurations. We use the environment configurations shown in Figure 3.1. We use six and 24 eigenoptions in the four rooms and nine rooms domains, respectively. We use all bottleneck options, which totals to eight in the four rooms domain and 24 in the nine rooms domain. We evaluate all algorithms for 100 independent runs and the shaded region represents a 99% confidence interval.

we use SMDP Q-learning to update the values of the option currently being followed and intra-option Q-learning to update the values of all other options and actions. Elaborating further, the values of the option currently being followed are updated upon termination. And, if an action a is taken in a state s, regardless of how it was selected, we update all options that would have taken action a in state s. We also perform action updates for primitive actions at every time step. For clarity, VAEO is detailed in Algorithm 1.

We report the number of time steps it takes the agent to reach the goal state during every episode. We compare the performance of VAEO to Q-learning with eigenoptions used for exploration (Machado et al., 2023, 2018) and intra-option Q-learning with bottleneck options (Sutton et al., 1998). We performed a hyperparameter sweep to determine the number of eigenoptions to use in each environment. Further details on hyperparameters are provided in Appendix A.

As shown in Figure 3.2, VAEO outperforms eigenoptions and bottleneck options in nearly all configurations of the environment. This suggests that credit assignment provides additional benefit on top of the exploration bonus provided by eigenoptions. As initially predicted, learning option-values allows the agent to exploit options that it previously found to be useful to maximize its return.

Algorithm 1 Value-Aware Eigenoptions (VAEO)

Input: α; ▷ Step-size for action and option updates
γ; ▷ Discount factor
n; ▷ Number of pre-computed eigenoptions to use
ϵ ▷ Epsilon for epsilon-greedy exploration

$\Omega \leftarrow n$ pre-computed eigenoptions
$Q \leftarrow |\mathcal{S}| \times |\mathcal{A} \cup \Omega|$ matrix
for $i = 1$ **to** T **do**
  $s \leftarrow$ current state
  if Uniform$(0, 1) < \epsilon$ **then**
    Choose a randomly from $\mathcal{A} \cup \Omega$
  else
    $a \leftarrow \text{argmax}_a\, Q(s, a)$; break ties randomly
  end if
  $\mathcal{D}_o \leftarrow \emptyset$
  if $a \in \mathcal{A}$ **then**
    Take action a, receive r, s'
    Append (s, a, r, s') to $\mathcal{D}_o$
    $Q(s, a) \leftarrow Q(s, a) + \alpha\,(r + \gamma \max_{a' \in \mathcal{A} \cup \Omega} Q(s', a') - Q(s, a))$
  else
    while option has not terminated **do**
      Take action a_o according to option policy, receive s', r
      $Q(s, a_o) \leftarrow Q(s, a_o) + \alpha\,(r + \gamma \max_{a' \in \mathcal{A} \cup \Omega} Q(s', a') - Q(s, a_o))$
      Append (s, a_o, r, s') to $\mathcal{D}_o$
    end while
  end if
  UpdateOptionValues$(a, \mathcal{D}_o, s')$ (see Algorithm 2)
end for

Algorithm 2 Update Option Values

Input: $o \in \Omega \cup \mathcal{A}$ ▷ Option or Action
$\mathcal{D}_o$ ▷ Sequence of samples collected while taking the option
s_{next} ▷ State the agent is in after taking option o

return $\leftarrow 0$
for (s, a, r, s') **in** reverse$(\mathcal{D}_o)$ **do**
  return $\leftarrow r + \gamma \cdot$ return
  $Q(s, o) \leftarrow Q(s, o) + \alpha\,[(\text{return} + \gamma\, \text{argmax}_{o' \in \mathcal{O}}\, Q(s_{\text{next}}, o')) - Q(s, o)]$
  for $\hat{o}$ **in** $\Omega \setminus \{o\}$ **do**
    if $\pi_{\hat{o}}(s) = a$ **then**
      $Q(s, \hat{o}) \leftarrow Q(s, \hat{o}) + \alpha\,[(r + \gamma U(s', \hat{o})) - Q(s, \hat{o})]$ (see Equation 2.28)
    end if
  end for
end for

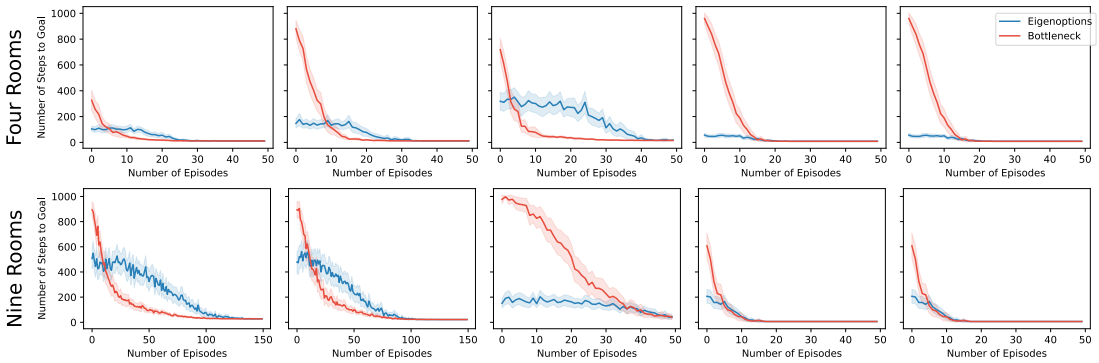

Figure 3.3: Credit assignment through an evaluation phase. We evaluate both algorithms for 100 seeds and report the 99% confidence interval. Environment configurations are the same as in Figure 3.1.

3.1.1 Disentangling Credit Assignment

While these results seem to indicate that credit assignment through options improves performance, when we learn a policy over options, we are essentially treating options as additional actions. Thus, the results above still confound the benefits of using eigenoptions for credit assignment and exploration.

To factor out the impact of the exploration benefits induced by eigenoptions, we constrained the agent to select only primitive actions while performing an intra-option update at every time step to learn option-values through the agent's actions. This eliminates any and all exploration benefits the agent gets from acting according to an option's policy. At the end of every episode, we evaluate the agent's time-to-goal when also considering options in its action space. In this case, the performance we see is strictly due to credit being assigned to the eigenoptions.

As shown in Figure 3.3, eigenoptions initially assign credit much more quickly than bottleneck options. However, they can stay at suboptimal trajectories longer. These results are not too surprising. Bottleneck options tend to be shorter, and the solution to any start/goal configuration unavoidably leads the agent through a bottleneck state, making bottleneck options part of the optimal policy (Solway et al., 2014; Sutton et al., 2023). In fact, it is somewhat surprising that eigenoptions are so competitive. We conjecture that this might be due to how eigenoptions are obtained through the environment's diffusive information flow, and that eigenoptions operate at different time scales, potentially allowing some of them to be a part of the optimal policy.

3.2 Discovering Options Online

In this section, we propose an algorithm to learn option-values for eigenoptions when eigenoptions are discovered online through the ROD cycle. In Section 3.2.1, we describe the algorithm and show its performance against baselines in the four rooms and nine rooms settings. We also show through an example run (see Figure 3.6) that when options are exploited and treated as additional actions, their temporal persistence hinders exploration. In Section 3.2.2, we show the different methods we attempted in order to try and solve this issue.

3.2.1 Learning Option Values with Online Option Discovery

We now discuss learning option-values when options are discovered online. We start by introducing option-values to CEO. Every time a new option is added to the option set, we initialize an additional vector in our Q-function to store option-values for the newly created option. As the agent interacts with the environment, we use the same updates as in Section 3.1 to learn option-values. We call this method Value-Aware Covering Eigenoptions (VACE). Algorithm 3 shows pseudocode for VACE.

To see how much of a benefit value-aware eigenoptions provide when eigenoptions are discovered online, we compare the performance of VACE to CEO, Q(λ), and option critic. In CEO, the agent can randomly sample an eigenoption and follow it during the ϵ-exploration phase. This is also true in VACE, but we also learn option-values and allow the agents to greedily follow options during the greedy phase. For both CEO and VACE, we learn a new option from the samples collected every 1,000 time steps. We perform a 100 sweeps over the dataset collected over the 1,000 timesteps to update the SR. We then use the intrinsic reward calculated from the top eigenvector of the SR to learn the option policy.

In order to make a fair comparison with option critic, we learn option critic's policy over options using intra-option Q-learning. Furthermore, we collect a dataset of transitions as the agent interacts with the environment, and after every 1,000 steps, we perform 100 sweeps over the dataset to update the policy over options and the option policies. This emulates the computational overhead of learning eigenoptions and

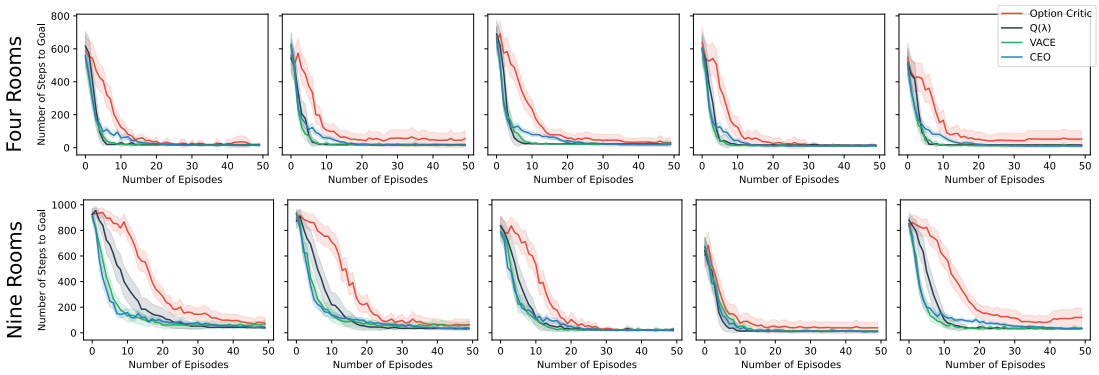

Figure 3.4: Performance of VACE and baselines when discovering options online every 1,000 time steps. Environment configurations are the same as in Figure 3.1. We evaluate all algorithms on 100 independent runs, and the shaded region represents a 99% confidence interval.

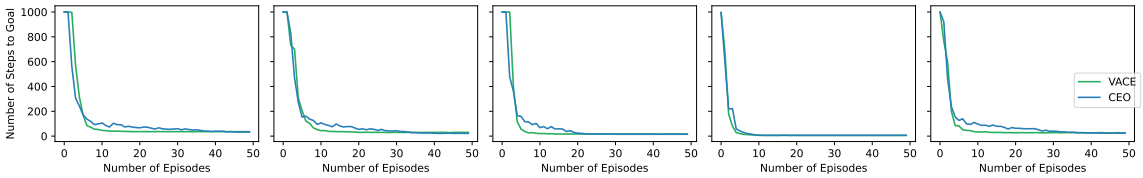

Figure 3.5: Median over 100 independent runs for VACE and baselines in the nine rooms environment.

allows for a fair comparison between the two methods. The rest of the experimental setup is the same as when using pre-computed options. We plot the time-to-goal for every episode and compare how quickly each algorithm is able to learn a good policy in each environment configuration.

As shown in Figure 3.4, VACE outperforms option critic and CEO in the four rooms environment and is on-par with Q(λ). In the nine rooms environment, both CEO and VACE seem to learn quicker than Q(λ) and option critic in most settings because of better exploration afforded by eigenoptions. However, the performance gain for VACE when compared to CEO is much less apparent. VACE still outperforms CEO in median performance, though (see Figure 3.5). This is because VACE outperforms CEO on most runs, but it performs significantly worse in a few runs. When options are used for both credit assignment and exploration, inaccurate estimates of an option's value might make a suboptimal option be selected much more often than it should be in the argmax. Such an option has a much more persistent behaviour, with long-term consequences, when compared to a wrongly selected primitive action.

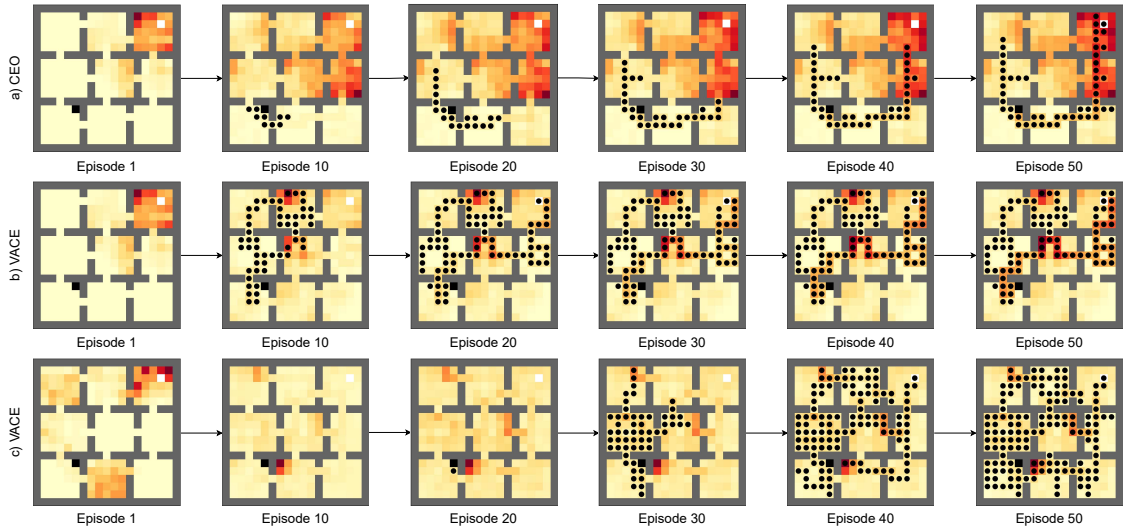

Figure 3.6: Value propagation in a) a typical run of CEO, b) a typical run of VACE and c) a bad run of VACE. The start state is highlighted in white, while the goal state is highlighted in black. Black dots represent states with a value greater than zero. Darker red states represent states frequently visited, while lighter yellow squares represent states visited infrequently.

Thus, in VACE, the discovered options do help the agent visit underexplored regions in the environment, but because the discovered options are treated as actions, they have a much bigger impact on the agent, sometimes making it harder for the agent to find the goal state.

Consistently acting according to option's policies can sometimes hinder performance. In CEO, value propagation is relatively slow as value is propagated one state at a time, but the values start to be propagated more quickly because of effective exploration (see Figure 3.6a). Figure 3.6b shows a typical run of VACE. VACE propagates value much faster and, in this particular run, is able to create a value trace to the goal by episode 20. Figure 3.6c shows a bad run of VACE where it struggles to find the goal because the agent hops from one option to another, and even though it initially gets close to the goal state, it does not use enough random actions to get to the goal state because another option "takes over" before that. Thus, it takes much longer to start propagating values back.

Algorithm 3 Value-Aware Covering Eigenoptions (VACE)

Input: η, α_o, α ; ▷ Step-sizes for the SR and option policies and intra-option Q-learning
γ_o, γ ; ▷ Discount factor for option policies and intra-option Q-learning
N_{steps} ; ▷ Number of samples per option created
N_{sweeps} ; ▷ Number of sweeps to learn SR from dataset
N_{iter} ; ▷ Number of iterations of VACE
ϵ ▷ Epsilon for epsilon-greedy exploration
$\mathcal{D} \leftarrow \emptyset$
$\Omega \leftarrow \emptyset$
$Q \leftarrow |\mathcal{S}| \times |\mathcal{A}|$ matrix
for $i = 1$ **to** N_{iter} **do**
  for $j = 1$ **to** N_{steps} **do**
    $s \leftarrow$ current state
    if Uniform$(0, 1) < \epsilon$ **then**
      Choose a randomly from $\mathcal{A} \cup \Omega$
    else
      $a \leftarrow \text{argmax}_a\, Q(s, a)$; break ties randomly
    end if
    $\mathcal{D}_o \leftarrow \emptyset$
    if $a \in \mathcal{A}$ **then**
      Take action a, receive r, s'
      Append (s, a, r, s') to $\mathcal{D}$ and $\mathcal{D}_o$
      $Q(s, a) \leftarrow Q(s, a) + \alpha\,(r + \gamma \max_{a' \in \mathcal{A} \cup \Omega} Q(s', a') - Q(s, a))$
    else
      while option has not terminated **do**
        Take action a_o according to option policy, receive s', r
        $Q(s, a_o) \leftarrow Q(s, a_o) + \alpha\,(r + \gamma \max_{a' \in \mathcal{A} \cup \Omega} Q(s', a') - Q(s, a_o))$
        Append (s, a_o, r, s') to $\mathcal{D}$ and $\mathcal{D}_o$
      end while
    end if
    UpdateOptionValues$(a, \mathcal{D}_o, s')$ (see Algorithm 2)
  end for
  Learn SR by applying Equation 2.20 sweeping through $\mathcal{D}$ N_{sweeps} times.
  Get top eigenvector of SR, create intrinsic reward function using Equation 2.21.
  Learn an option policy to maximize intrinsic reward using Q-learning.
  Define termination states as all states with $Q(s, a) \leq 0$.
  Append option to Ω, append column of zeros to Q.
end for

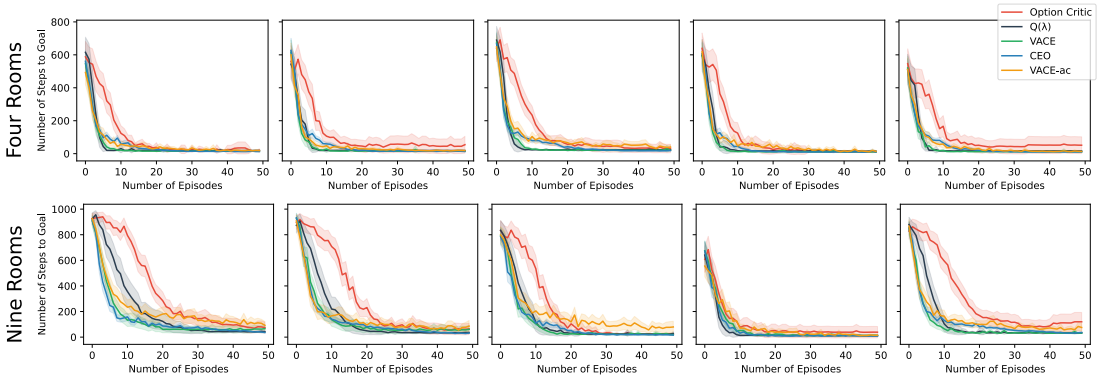

Figure 3.7: Performance of VACE with actor-critic (VACE-ac) compared to VACE, CEO, Q(λ), and option critic. Environment configurations are the same as in Figure 3.1. We evaluate all algorithms on 100 independent runs, and the shaded region represents a 99% confidence interval.

3.2.2 Trying to Address the Drawbacks of Option Persistence

We attempt to address the persistence of options, leading to worse exploration when options are used for exploitation. To do this, we use actor-critic approach with a softmax policy and linear function approximation to learn a policy over options. Because a softmax policy is stochastic, we conjecture that it could allow the agent to potentially explore using primitive actions even when the likelihood of initiating an option is very high. This should address the persistence of options by lowering the likelihood that the agent takes multiple options in sequence. We plot the time-to-goal of VACE with actor critic (VACE-ac) after each episode in the four rooms and nine rooms environment and compare it against CEO, VACE, Q(λ) and option-critic.

As shown in Figure 3.7, initially, VACE-ac seems promising and seems to match the performance of the other methods. However, it converges to a suboptimal policy with much higher variance. Why does this happen? When learning options online using the ROD cycle, the option set grows. When using a value-based approach, we could simply initialize a new row of zeros in the Q-table. However, using a similar approach and initializing new weights with a value of zero means that if weights are close to zero, a newly added option is much more likely to be initiated. This makes the persistence issue worse and leads to more variance as options are added.

Another method to potentially address the persistence of options is to add a k time step delay after an option is taken, preventing another option from being taken until k

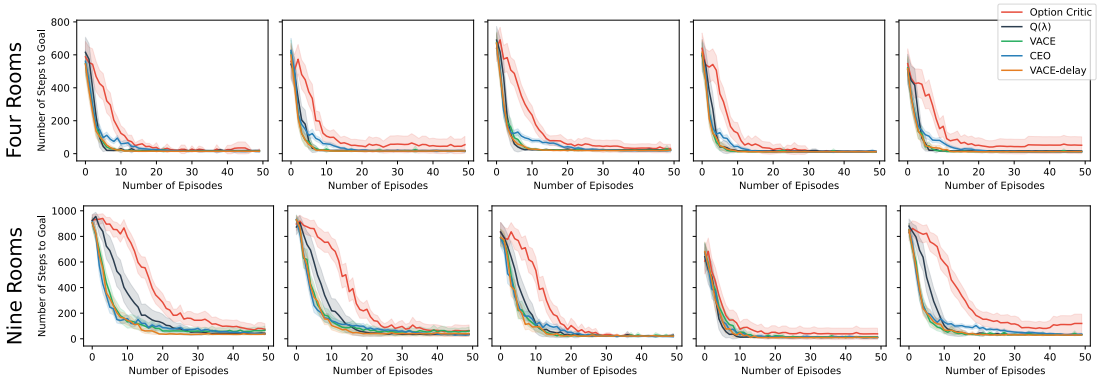

Figure 3.8: Performance of VACE with a 10-step delay compared to VACE, CEO, Q(λ), and option critic. Environment configurations are the same as in Figure 3.1. We evaluate all algorithms on 100 independent runs, and the shaded region represents a 99% confidence interval.

time steps have occurred. This should prevent options from being taken consecutively during the initial exploration phase, which should allow the agent to explore nearby states after option termination, and lead to better exploration of states that are not in the path of an option. During initial tests with a k time step delay, incorrect action-value estimates caused significantly worse performance. With the k time step delay, when an option terminates, actions must be taken for the next k-steps. Because action-values are updated as the option is being taken and not recursively upon an option's termination, action-value estimates do not propagate at the same rate as option-value updates. This can lead to situations where an option led to a high value state and propagated option values, but the next time the agent gets to one of the option's initiation states after taking a different option, it is unable to use the option again due to the k-step delay. And since action-values have not propagated, it cannot use actions to follow the same path either.

To address this, we make a slight modification to the VACE algorithm in addition to the delay by ensuring action updates are done recursively upon option termination. We remove the action-value update from the option execution loop and instead update option and action values using the full trace we collect while executing the option (see Algorithm 4). We again run the algorithm in the nine rooms and four rooms environments and compare the time-to-goal per episode against VACE, CEO, Q(λ), and option critic. As shown in Figure 3.8, adding the delay and changing the order

Algorithm 4 Update Option and Action Values

Input: $o \in \Omega \cup \mathcal{A}$; ▷ Option or Action
$\mathcal{D}_o$; ▷ Sequence of samples collected while taking the option
s_{next} ; ▷ State the agent is in after taking option o

```
return ← 0
for (s, a, r, s') in reverse(D_o) do
    Q(s,a) ← Q(s,a) + α[(r + γ argmax_{o'∈O} Q(s', o')) − Q(s,a)]
    return ← r + γ · return
    Q(s,o) ← Q(s,o) + α[(return + γ argmax_{o'∈O} Q(s_next, o')) − Q(s,o)]
    for ô in Ω \ {o} do
        if π_ô(s) = a then
            Q(s,ô) ← Q(s,ô) + α[(r + γU(s', ô)) − Q(s,ô)]
        end if
    end for
end for
```

of action-value updates provided little improvement over VACE.

It is important to ask, however, whether there is additional performance to be gained from recursively updating action-values in VACE and CEO. To test this, we modify VACE and CEO to include the recursive action-value updates and again run the algorithm in the nine rooms and four rooms environments and compare the time-to-goal per episode against VACE, CEO, and Q(λ). As shown in Figure 3.9, improvements to VACE are marginal and only occur in a few environment configurations. However, performance for CEO with recursive action-value updates is on-par with

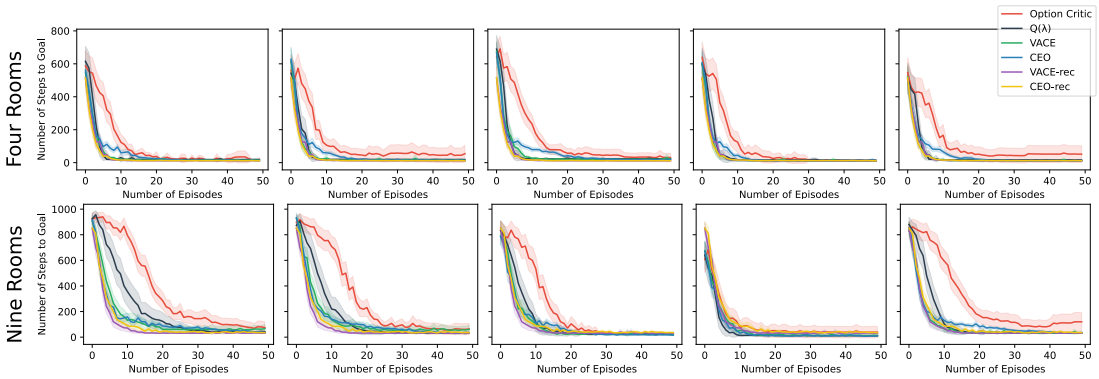

Figure 3.9: Performance of VACE and CEO with recursive action-value updates (VACE-rec and CEO-rec) compared to VACE, CEO, Q(λ), and option critic. Environment configurations are the same as in Figure 3.1. We evaluate all algorithms on 100 independent runs, and the shaded region represents a 99% confidence interval.

VACE in both environments. This suggests that using eigenoptions strictly to assign credit to actions provides the same benefit as learning a policy over eigenoptions.

Chapter 4

Approximating Option-Values with Non-Linear Function Approximation

Since value-aware eigenoptions lead to faster learning in the tabular setting when options are pre-computed, it is natural to ask whether they would also help in the function approximation setting. We investigate this question in the four rooms and nine rooms environments with pixel inputs. We again use a modified version of the Minigrid environment (Chevalier-Boisvert et al., 2023) with cardinal actions. We first focus on using neural networks to approximate option-values, using tabular methods to compute the eigenvectors of the SR and the options themselves. Solutions to approximating the eigenvectors of the SR (Gomez et al., 2024; Wang et al., 2021; Wu et al., 2018) and the eigenoptions themselves (Klissarov and Machado, 2023) have already been discussed in the literature. We discuss nuances around option termination when using them for credit assignment in Section 4.2.

In all experiments, we use a two-layer convolutional network with 32 channels each, a 3×3 kernel, and a stride of 2, followed by a fully connected layer with 256 nodes and then the output layer with 4 nodes (one for each action) in the action-value network and 1 node in the option-value network. We implemented our algorithms with the PFRL library (Fujita et al., 2021).

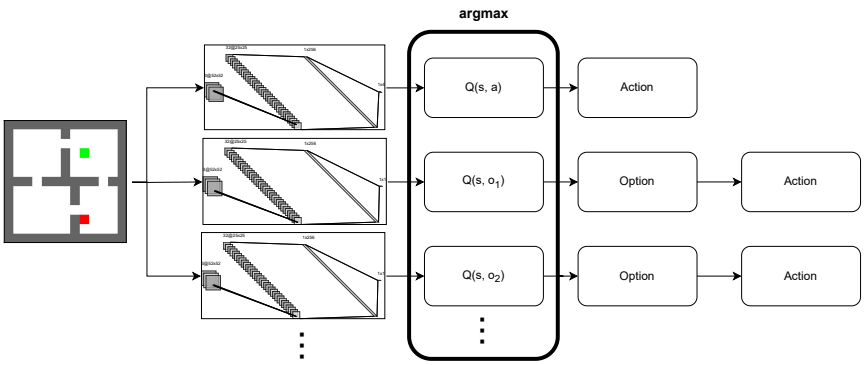

Figure 4.1: Hierarchical DQN architecture. We include a separate network to predict the value of each option. If an option has the highest option-value, its respective option policy will choose an action.

4.1 Tabular Option Policies

To learn option-values in the function approximation setting, we use a hierarchical DQN architecture (Kulkarni et al., 2016). We use a separate neural network to learn each of the option-value functions on top of the value function for primitive actions (see Figure 4.1).

The network parameters are represented as $\boldsymbol{\theta}^{o_k} \in \boldsymbol{\theta}^o$, where $k = 0$ represents the network that learns primitive action-values, and $k > 0$ represents the network that learns the option-values for option k. The target networks are represented as $\boldsymbol{\theta}^{o_k^-} \in \boldsymbol{\theta}^{o^-}$. The action or option with the highest value is selected during a greedy step. We use the DQN (Mnih et al., 2015) loss function $L_i = \mathbb{E}_{s,a\sim p(\cdot)}\left[(y_i - Q(s,a;\boldsymbol{\theta}_i))^2\right]$ with a modified Double DQN (DDQN) (Hasselt et al., 2016) target:

$$y_i = \mathbb{E}\left[r + \gamma U(s', o; \boldsymbol{\theta}_i^o, \boldsymbol{\theta}_i^{o^-}) \middle| s, a\right], \tag{4.1}$$

where

$$\begin{aligned} U(s', o; \boldsymbol{\theta}_i^o, \boldsymbol{\theta}_i^{o^-}) = &(1 - \beta_o(s))\, Q(s, o; \boldsymbol{\theta}_i^{o^-}) \\ &+ \beta_o(s) Q\left(s, \operatorname*{argmax}_{o'} Q\left(s, o'; \boldsymbol{\theta}_i^o\right); \boldsymbol{\theta}_i^{o^-}\right). \end{aligned} \tag{4.2}$$

Similar to the experiments we performed in the tabular setting, we evaluate deep value-aware eigenoptions (DVAEO) against deep eigenoptions (DDQN with eigenoptions for exploration) and DDQN with ϵ-greedy exploration. We use the same number

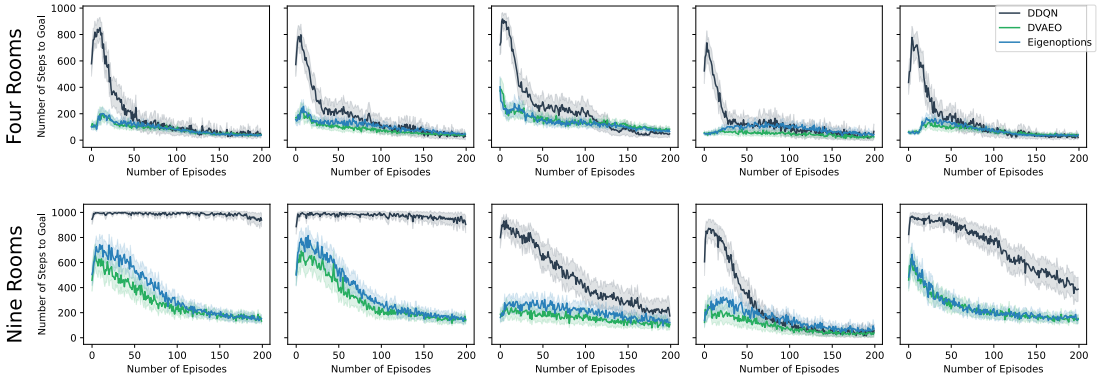

Figure 4.2: Performance of DVAEO and baselines with pixel observations. Environment configurations are the same as in Figure 3.1. We use six tabular eigenoptions in the four rooms domain and 24 tabular eigenoptions in the nine rooms domain for both DVAEO and deep eigenoptions. We train all algorithms for 100 independent runs and the shaded region represents a 99% confidence interval.

of eigenoptions from the tabular setting, six eigenoptions in the four rooms domain and 24 eigenoptions in the nine rooms domain, for both DVAEO and deep eigenoptions. This hyperparameter was not swept in this setting. We populate the replay buffer with 1,000 random transitions in the environment before starting the learning process.

As shown in Figure 4.2, DVAEO performs slightly better than deep eigenoptions in the nine rooms environment, while there is no difference in performance in the four rooms environment. The performance gains are not nearly as significant as those in the tabular setting, though. We hypothesize that this is because learning with neural networks is much slower than in the tabular case. This could imply that the exploration benefits from eigenoptions outweigh credit assignment in these experiments. However, the fact that there is a larger performance gain in the bigger environment might suggest that the benefits of DVAEO would become more evident in more complex environments.

4.2 Approximating Option Policies

We now extend the previous architecture by also approximating the option's policies and termination conditions. We use the DDQN algorithm to learn the option policies. Unlike the tabular setting, the method for determining whether to terminate

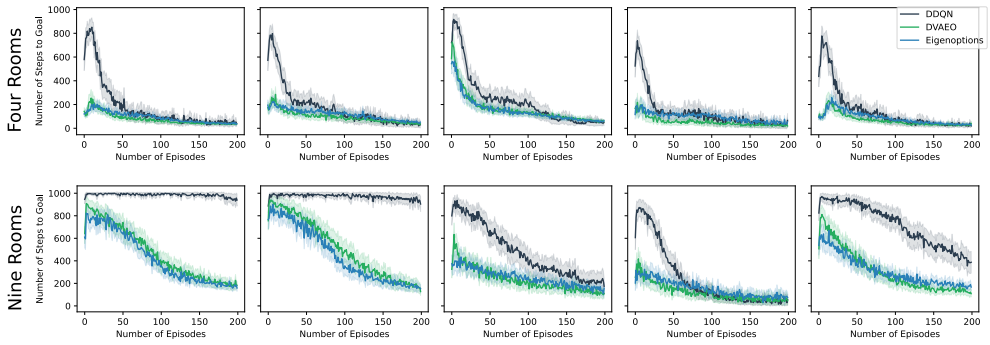

Figure 4.3: Performance of DVAEO (with approximated option policies) and baselines with pixel observations. Environment configurations are the same as in Figure 3.1. We use six eigenoptions in the four rooms environment and 24 eigenoptions in the nine rooms environment. We train all algorithms for 100 independent trials, and the shaded region represents a 99% confidence interval.

an option at a given state is much more brittle because the generalization of neural networks leads to estimates of many states changing with a single update, making termination conditions based on specific thresholds ineffective. Klissarov and Machado (2023) successfully utilized option policies with nondeterministic termination using a constant $\beta = 0.1$. While this makes sense for exploration, given that randomized termination still leads the agent toward underexplored states, our initial experiments showed extremely high variance and slow learning when using a constant β. To reduce stochasticity, we terminate options 10 time steps after initiation; we obtained this value through a hyperparameter sweep over both environments we considered.

As before, we evaluate DVAEO against deep eigenoptions (now with deep option policies) and DDQN with ϵ-greedy exploration. The option policies were trained on a version of the environment without start or goal states for 500 episodes, ensuring that the learned policies matched the tabular policies. We again use six eigenoptions in the four rooms environment and 24 eigenoptions in the nine rooms environment for both DVAEO and deep eigenoptions, and we populate the replay buffer with 1,000 random samples from the environment before starting the learning process.

DVAEO with DDQN option policies performs similarly to deep eigenoptions in most configurations (see Figure 4.3). Using eigenoptions for exploration led to similar

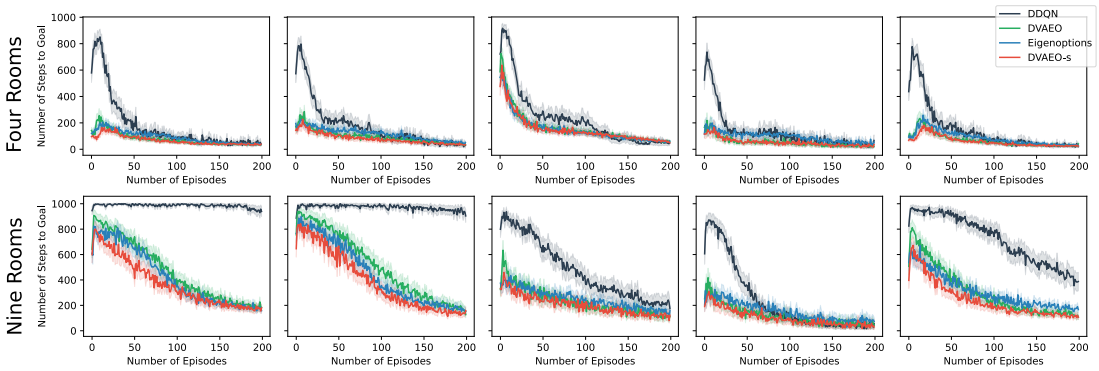

Figure 4.4: Performance of DVAEO (with approximated option policies) with same-state termination (DVAEO-s) and without same-state termination (DVAEO) compared to baselines with pixel observations. Environment configurations are the same as in Figure 3.1. We use six eigenoptions in the four rooms environment and 24 eigenoptions in the nine rooms environment. We train all algorithms for 100 independent trials, and the shaded region represents a 99% confidence interval.

performance as when using them for both exploration and credit assignment. This can be attributed to suboptimal termination conditions. The DDQN option policies were functionally the same as the tabular ones, except for the termination condition. When termination states are not well defined, there are some states where the option's policy leads the agent into a loop and/or a wall. Since the agent cannot terminate at such states, it has to continue taking the option until it terminates either through chance or after n steps.

To confirm that the poor performance of DVAEO with DDQN option policies is due to poor termination, we run the exact same algorithm, but instead of terminating options after n steps, we terminate when the agent visits the same state in two consecutive time steps. That is, we terminate an option if the agent stays in its current state instead of moving to a new state.

Keeping everything else the same, we can see that when terminating after staying in the same state, DVAEO with DDQN option policies performs better than DCEO in the nine rooms setting (see Figure 4.4). This confirms that suboptimal termination conditions hamper learning when approximating option policies. This issue is exacerbated if there is poor network initialization or bad updates due to neural network generalization. Interrupting such options might be a promising avenue for future work.

Chapter 5

Conclusion

In this work, we studied value-aware eigenoptions. We (1) demonstrated the benefits of learning option-values for pre-computed eigenoptions, (2) introduced VACE, a method that builds on CEO to learn option-values for eigenoptions learned online, and (3) introduced DVAEO, a deep RL method to learn option-values for eigenoptions. Maybe even more important than the methods themselves were the discussions around the impact of termination conditions in the function approximation setting and the challenges of learning option-values for eigenoptions that are discovered online.

Learning accurate termination for options is vital to learning accurate option-values. Without accurate termination, the agent may suffer from significantly higher variance (if termination is random), or get stuck until the option terminates (if using n-step termination). When learning option-values for eigenoptions that are discovered online, option persistence hinders exploration. When initial option-value estimates are inaccurate, eigenoptions might be initiated much more frequently, making it difficult for the agent to explore nearby states after option termination.

Future work may involve methods that learn a well-defined termination set for eigenoptions in the function approximation setting. Potential approaches might involve thresholding (Jinnai et al., 2020) and clustering (Ramesh et al., 2019) based on the learned value function. Additionally, developing techniques to maintain the exploration benefits of eigenoptions once they are added to an agent's action space will be crucial to using option-values effectively when eigenoptions are learned online.

More generally, this work outlines the potential and challenges associated with using eigenoptions for credit assignment in model-free RL. We have demonstrated

that eigenoptions exhibit potential for credit assignment, despite being initially introduced for exploration. However, their potential for credit assignment seems to be overshadowed by interactions with the state visitation distribution they induce when they are learned online, or when used in the function approximation setting.

References

Bacon, Pierre-Luc, Jean Harb, and Doina Precup (2017). "The option-critic architecture." In: *AAAI Conference on Artificial Intelligence*. 1, 11, 13, 18

Bagaria, Akhil and George Konidaris (2019). "Option discovery using deep skill chaining." In: *International Conference on Learning Representations*. 1, 13

Bellemare, Marc G., Salvatore Candido, Pablo Samuel Castro, Jun Gong, Marlos C. Machado, Subhodeep Moitra, Sameera S. Ponda, and Ziyu Wang (2020). "Autonomous navigation of stratospheric balloons using reinforcement learning." In: *Nature* 588 (7836), pp. 77–82. 1

Berner, Christopher, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique P. d. O. Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang (2019). "Dota 2 with large scale deep reinforcement learning." In: *CoRR* abs/1912.06680. 1

Brockman, Greg, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba (2016). "OpenAI gym." In: *CoRR* abs/1606.01540. 18

Chevalier-Boisvert, Maxime, Bolun Dai, Mark Towers, Rodrigo de Lazcano, Lucas Willems, Salem Lahlou, Suman Pal, Pablo Samuel Castro, and Jordan Terry (2023). "Minigrid & Miniworld: Modular & Customizable Reinforcement Learning Environments for Goal-Oriented Tasks." In: *CoRR* abs/2306.13831. 18, 31

Dayan, Peter (1993). "Improving generalization for temporal difference learning: The successor representation." In: *Neural Computation* 5 (4), pp. 613–624. 2, 9

Dayan, Peter and Geoffrey E. Hinton (1992). "Feudal reinforcement learning." In: *Neural Information Processing Systems*. 1, 13

Degrave, Jonas, Federico Felici, Jonas Buchli, Michael Neunert, Brendan D. Tracey, Francesco Carpanese, Timo Ewalds, Roland Hafner, Abbas Abdolmaleki, Diego de Las Casas, Craig Donner, Leslie Fritz, Cristian Galperti, Andrea Huber, James Keeling, Maria Tsimpoukelli, Jackie Kay, Antoine Merle, Jean-Marc Moret, Seb Noury, Federico Pesamosca, David Pfau, Olivier Sauter, Cristian Sommariva, Stefano Coda, Basil Duval, Ambrogio Fasoli, Pushmeet Kohli, Koray Kavukcuoglu, Demis Hassabis, and Martin A. Riedmiller (2022). "Magnetic control of tokamak plasmas through deep reinforcement learning." In: *Nature* 602 (7897), pp. 414–419. 1

Eysenbach, Benjamin, Abhishek Gupta, Julian Ibarz, and Sergey Levine (2019). "Diversity is all you need: Learning skills without a reward function." In: *International Conference on Learning Representations*. 1, 13

Fujita, Yasuhiro, Prabhat Nagarajan, Toshiki Kataoka, and Takahiro Ishikawa (2021). "ChainerRL: A deep reinforcement learning library." In: *Journal of Machine Learning Research* 22 (77), pp. 1–14. 31, 44

Gomez, Diego, Michael Bowling, and Marlos C. Machado (2024). "Proper Laplacian representation learning." In: *International Conference on Learning Representations*. 31

Gregor, Karol, Danilo Jimenez Rezende, and Daan Wierstra (2017). "Variational intrinsic control." In: *International Conference on Learning Representations Workshop Track*. 13

Harb, Jean, Pierre-Luc Bacon, Martin Klissarov, and Doina Precup (2018). "When waiting is not an option: Learning options with a deliberation cost." In: *AAAI Conference on Artificial Intelligence*. 1

Hasselt, Hado van, Arthur Guez, and David Silver (2016). "Deep reinforcement learning with double Q-learning." In: *AAAI Conference on Artificial Intelligence*. 16, 32

Jinnai, Yuu, Jee Won Park, David Abel, and George Konidaris (2019). "Discovering options for exploration by minimizing cover time." In: *International Conference on Machine Learning*. 1

Jinnai, Yuu, Jee Won Park, Marlos C. Machado, and George Konidaris (2020). "Exploration in reinforcement learning with deep covering options." In: *International Conference on Learning Representations*. 36

Jong, Nicholas K., Todd Hester, and Peter Stone (2008). "The utility of temporal abstraction in reinforcement learning." In: *International Joint Conference on Autonomous Agents and Multiagent Systems*. 1

Kingma, Diederik P. and Jimmy Ba (2015). "Adam: A method for stochastic optimization." In: *International Conference on Learning Representations*. 44

Klissarov, Martin, Akhil Bagaria, Ziyan Luo, George Dimitri Konidaris, Doina Precup, and Marlos C. Machado (2025). "Discovering temporal structure: An overview of hierarchical reinforcement learning." In: *CoRR* abs/2506.14045. 13

Klissarov, Martin and Marlos C. Machado (2023). "Deep Laplacian-based options for temporally-extended exploration." In: *International Conference on Machine Learning*. 2, 31, 34

Klyubin, Alexander S., Daniel Polani, and Chrystopher L. Nehaniv (2005). "All else being equal be empowered." In: *European Conference on Artificial Life*. 13

Konidaris, George and Andrew Barto (2007). "Building portable options: Skill transfer in reinforcement learning." In: *International Joint Conference on Artificial Intelligence*. 2

— (2009). "Skill discovery in continuous reinforcement learning domains using skill chaining." In: *Neural Information Processing Systems*. 1, 13

Kotamreddy, Harshil and Marlos C. Machado (2025). "A study of value-aware eigenoptions." In: *Inductive Biases in Reinforcement Learning Workshop at RLC*. iii

Kulkarni, Tejas D., Karthik Narasimhan, Ardavan Saeedi, and Josh Tenenbaum (2016). "Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation." In: *Neural Information Processing Systems*. 32

Levy, Andrew, George Konidaris, Robert Platt, and Kate Saenko (2019). "Learning multi-level hierarchies with hindsight." In: *International Conference on Learning Representations*. 13

Liu, Miao, Marlos C. Machado, Gerald Tesauro, and Murray Campbell (2017). "The eigenoption-critic framework." In: *CoRR* abs/1712.04065. 2

Lo, Chunlok, Kevin Roice, Parham Mohammad Panahi, Scott M. Jordan, Adam White, Gabor Mihucz, Farzane Aminmansour, and Martha White (2024). "Goal-space planning with subgoal models." In: *Journal of Machine Learning Research* 25 (330), pp. 1–57. 14

Machado, Marlos C. (2025). "Representation-driven option discovery in reinforcement learning." In: *AAAI Conference on Artificial Intelligence*. 11

Machado, Marlos C., Andre Barreto, Doina Precup, and Michael Bowling (2023). "Temporal abstraction in reinforcement learning with the successor representation." In: *Journal of Machine Learning Research* 24 (80), pp. 1–69. 11, 20

Machado, Marlos C., Marc G. Bellemare, and Michael Bowling (2017). "A Laplacian Framework for Option Discovery in Reinforcement Learning." In: *International Conference on Machine Learning*. 2, 9

Machado, Marlos C. and Michael Bowling (2016). "Learning purposeful behaviour in the absence of rewards." In: *CoRR* abs/1605.07700. 2

Machado, Marlos C., Clemens Rosenbaum, Xiaoxiao Guo, Miao Liu, Gerald Tesauro, and Murray Campbell (2018). "Eigenoption discovery through the deep successor representation." In: *International Conference on Learning Representations*. 9, 18, 20

McGovern, Amy and Andrew Barto (2001a). "Accelerating reinforcement learning through the discovery of useful subgoals." In: *International Symposium on Artificial Intelligence, Robotics, and Automation in Space*. 18

— (2001b). "Automatic discovery of subgoals in reinforcement learning using diverse density." In: *International Conference on Machine Learning*. 9

Menache, Ishai, Shie Mannor, and Nahum Shimkin (2002). "Q-Cut—dynamic discovery of sub-goals in reinforcement learning." In: *European Conference on Machine Learning*. 9

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis (2015). "Human-level control through deep reinforcement learning." In: *Nature* 518 (7540), pp. 529–533. 16, 32

Mohamed, Shakir and Danilo Jimenez Rezende (2015). "Variational information maximisation for intrinsically motivated reinforcement learning." In: *Neural Information Processing Systems*. 13

Nachum, Ofir, Shixiang Shane Gu, Honglak Lee, and Sergey Levine (2018). "Data-efficient hierarchical reinforcement learning." In: *Neural Information Processing Systems*. 13

Ouyang, Long, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe (2022). "Training language models to follow instructions with human feedback." In: *Neural Information Processing Systems*. 1

Ramesh, Rahul, Manan Tomar, and Balaraman Ravindran (2019). "Successor options: An option discovery framework for reinforcement learning." In: *International Joint Conference on Artificial Intelligence*. 36

Silver, David, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis (2016). "Mastering the game of Go with deep neural networks and tree search." In: *Nature* 529 (7587), pp. 484–489. 1

Silver, David, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis (2018). "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play." In: *Science* 362 (6419), pp. 1140–1144. 1

Şimşek, Özgür and Andrew Barto (2008). "Skill characterization based on betweenness." In: *Neural Information Processing Systems*. 9, 18

Şimşek, Özgür, Alicia P. Wolfe, and Andrew Barto (2005). "Identifying useful subgoals in reinforcement learning by local graph partitioning." In: *International Conference on Machine Learning*. 9

Solway, Alec, Carlos Diuk, Natalia Córdova, Debbie Yee, Andrew Barto, Yael Niv, and Matthew M. Botvinick (2014). "Optimal behavioral hierarchy." In: *PLOS Computational Biology* 10 (8), pp. 1–10. 1, 9, 18, 22

Stolle, Martin and Doina Precup (2002). "Learning options in reinforcement learning." In: *International Symposium on Abstraction, Reformulation, and Approximation*. 9

Sutton, Richard S., Marlos C. Machado, G. Zacharias Holland, David Szepesvari, Finbarr Timbers, Brian Tanner, and Adam White (2023). "Reward-respecting subtasks for model-based reinforcement learning." In: *Artificial Intelligence* 324, p. 104001. 2, 18, 22

Sutton, Richard S., David A. McAllester, Satinder Singh, and Yishay Mansour (1999a). "Policy gradient methods for reinforcement learning with function approximation." In: *Neural Information Processing Systems*. 7

Sutton, Richard S., Doina Precup, and Satinder Singh (1998). "Intra-option learning about temporally abstract actions." In: *International Conference on Machine Learning*. 14, 20

— (1999b). "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning." In: *Artificial Intelligence* 112 (1), pp. 181–211. 1, 7, 8, 14

van Hasselt, Hado (2010). "Double Q-learning." In: *Neural Information Processing Systems*. 5

Vezhnevets, Alexander Sasha, Simon Osindero, Tom Schaul, Nicolas Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu (2017). "Feudal networks for hierarchical reinforcement learning." In: *International Conference on Machine Learning*. 1, 13

Vinyals, Oriol, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander Sasha Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom Le Paine, Çaglar Gülçehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy P. Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver (2019). "Grandmaster level in StarCraft II using multi-agent reinforcement learning." In: *Nature* 575 (7782), pp. 350–354. 1

Wang, Kaixin, Kuangqi Zhou, Qixin Zhang, Jie Shao, Bryan Hooi, and Jiashi Feng (2021). "Towards better Laplacian representation in reinforcement learning with generalized graph drawing." In: *International Conference on Machine Learning*. 31

Watkins, Christopher J. C. H. (1989). "Learning from delayed rewards." PhD thesis. King's College. 6

Watkins, Christopher J. C. H. and Peter Dayan (1992). "Q-learning." In: *Machine Learning* 8 (3), pp. 279–292. 5

Williams, Ronald J. (1992). "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning." In: *Machine Learning* 8, pp. 229–256. 7

Wu, Yifan, George Tucker, and Ofir Nachum (2018). "The Laplacian in RL: Learning Representations with Efficient Approximations." In: *International Conference on Learning Representations*. 31

Appendix A

Hyperparameters and Other Experimental Details

For all hyperparameter sweeps, we run each configuration for 10 seeds each. We then pick the best configuration by picking the configuration with the lowest AUC (where the curve is the mean of the 10 seeds).

For the algorithms in Figure 3.2, we use $\gamma_o = 0.99$ and $\alpha_o = 0.1$ for VACE and CEO, where γ_o and α_o correspond to the discount factor and step size in the option updates. We swept the following hyperparameters for all algorithms in all environment configurations: $\epsilon = \{0.01, 0.05, 0.1, 0.15, 0.2\}$, $\alpha = \{1.0, 0.3, 0.1, 0.03, 0.01\}$, $\gamma = \{0.99, 0.95, 0.9\}$ and chose the best set of hyperparameters based on the AUC of the mean time-to-goal per episode. For Q(λ), we ran a sweep on $\lambda = \{0.5, 0.6, 0.75, 0.9, 1\}$.

We also conducted a hyperparameter sweep on the number of eigenoptions in each environment, leading to the choice of six eigenoptions in the four rooms domain and 24 eigenoptions in the nine rooms domain. We swept over $N_{\text{options}} = \{2, 4, 6, 8, 16, 24, 32\}$ in both domains. To learn eigenoption policies, we use Q-learning with $\gamma = 0.9$ and $\alpha = 0.1$. We run Q-learning for 100 episodes of 1,000 steps each and use the intrinsic reward function. We use all bottleneck options in both environments. In the four rooms domain there are a total of eight bottleneck options, while there are 24 bottleneck options in the nine rooms domain.

For the algorithms in Figure 3.3, we use the same values as in Figure 3.2. Intra-option and action updates happen only in the training phase. However, options cannot be used in the training phase and all intra-option updates are made when

primitive actions are taken.

For the CEO and the VACE variants in Section 3.2, we use $\eta, \alpha_o = 0.1$; $\gamma_o = 0.99$, and $N_{\text{steps}} = 1000$. We swept the following hyperparameters for the CEO and VACE variants: $\epsilon = \{0.01, 0.05, 0.1, 0.15, 0.2\}$, $\alpha = \{1.0, 0.3, 0.1, 0.03, 0.01\}$, $\gamma = \{0.99, 0.95, 0.9\}$ in the first configuration of both the four room and nine room settings and use the best hyperparameters from the first configuration for the rest of the configurations. We also ran a hyperparameter sweep over $N_{\text{steps}} = \{100, 500, 1000, 10000\}$ in both the four rooms and nine rooms domains. To learn eigenoption policies, we store all samples in a dataset and sweep over the dataset $N_{\text{sweeps}} = 100$ times using Q-learning with $\gamma = 0.9$ and $\alpha = 0.1$. We use the same hyperparameters as we did in Section 3.1 for Q(λ). For option-critic, we ran a sweep over the following hyperparameters: $\alpha, \alpha_{\boldsymbol{\theta}}, \alpha_{\boldsymbol{\vartheta}} = \{0.01, 0.03, 0.1, 0.3, 1.0\}$, $\epsilon = \{0.01, 0.05, 0.1, 0.15, 0.2\}$, temperature $= \{0.001, 0.01, 0.1\}$.

For the algorithms in Figures 4.2 and 4.3, we use modified implementations from PFRL (Fujita et al., 2021). Pixel observations from the environments were of size $52 \times 52 \times 3$. For all algorithms, we use $\gamma = 0.9$, an update interval of 1 step, a target update interval of 2,000 steps, a replay buffer of size 20,000, and the Adam optimizer (Kingma and Ba, 2015) with a step size of 10^{-5}. We use linear ϵ decay with $\epsilon = 0.1$ decaying to $\epsilon = 0.05$ over 180,000 steps. We also use six eigenoptions in the four rooms domain and 24 eigenoptions in the nine rooms domain just as we did with our tabular experiments. However, we did not perform a hyperparameter sweep over the number of options in the deep RL setting.

To learn eigenoption policies, we train a DDQN model, with $\epsilon = 1$ and a replay buffer of size 100,000, on each eigenvector of the SR for 500 episodes of length 1,000. When eigenoption policies are being used in either deep eigenoptions or DVAEO, they are no longer updated and are used in evaluation mode.

For all deep RL algorithms (including eigenoption policies), we use a two-layer convolutional network with 32 channels each, a 3×3 kernel, and a stride of 2, followed by a fully connected layer with 256 nodes and then the output layer with 4 nodes (one for each action) in the action-value network and 1 node in the option-value network.