UNIVERSIDADE DO VALE DO RIO DOS SINOS
UNIDADE ACADÊMICA DE GRADUAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

MATEUS BEGNINI MELCHIADES

DYNAMIC OPTION CREATION FOR OPTION-CRITIC REINFORCEMENT LEARNING

São Leopoldo
2023

Mateus Begnini Melchiades

DYNAMIC OPTION CREATION FOR OPTION-CRITIC REINFORCEMENT LEARNING

Article presented as partial requirement for obtaining the title of Bachelor in Computer Science, from the Computer Science course at Universidade do Vale do Rio dos Sinos (UNISINOS)

Advisor: Prof. Dr. Gabriel de Oliveira Ramos

São Leopoldo
2023

DYNAMIC OPTION CREATION FOR OPTION-CRITIC REINFORCEMENT LEARNING

Mateus Begnini Melchiades[1]
Gabriel de Oliveira Ramos[2]

Abstract: Reinforcement Learning (RL) is an increasingly popular technique in the field of machine learning due to its ability to learn by interacting directly with the environment. Unlike traditional supervised learning, RL agents do not rely on labelled examples, thus following a more natural approach to the concept of learning. The options framework proposed by Sutton *et al.* introduces the concept of temporal abstraction in MDPs by combining high level courses of action that may span over multiple time steps with primitive, single-step actions, which can greatly improve planning and learning speeds (SUTTON; PRECUP; SINGH, 1999). Throughout the past two decades, there has been active interest in autonomous option discovery, as well as determining what characterizes a good option. Given the fact that the number of instructions a sequential machine needs to run for executing planning is directly related to the number of options it has to consider at each time step, it is important to have a small but comprehensive set of options. Wan and Sutton (WAN; SUTTON, 2022) proposed the Fast Planning Option-Critic (FPOC) algorithm that prioritizes discovering general options for faster planning; however, their implementation is limited by the fact that it requires manually specifying the number of desired options. In the present work, we improve upon the FPOC implementation by proposing an approach for creating options dynamically in training time. The dynamic option creation algorithm analyzes the current options' variance to determine whether the learning process would benefit from a new option, resulting in the minimum number of options for maximum return in any given problem. Our method manages to not only minimize re-training efforts by always converging to the optimal number of options, but also achieve a higher cumulative per-episode reward in certain cases when compared to FPOC. The proposed method can also be adapted to other Option-Critic algorithms, solving a major limitation of the original architecture, which requires multiple runs with different parameters to determine the ideal number of options for the task.

Keywords: Reinforcement Learning. Options. Option adjust. Option discovery.

1 INTRODUCTION

In the field of Artificial Intelligence, Reinforcement Learning stands out as the approach that most closely resembles actual learning. While traditional machine learning techniques focus on generalizing outcomes given a set of input-output examples, RL algorithms can learn to execute tasks solely by interacting with its environment (SUTTON; BARTO, 2020). This *learning by interaction* makes Reinforcement Learning algorithms much more scalable than Supervised Learning, given that in most cases it is not possible to easily obtain examples of good behavior for training. For instance, while it is relatively easy to create a dataset with pictures of houses, cars, and sidewalks to train a neural network to recognize objects around a self-driving car, the

[1]Computer Science Undergraduate at Unisinos. Email: mateusbme@edu.unisinos.br
[2]Graduate Program in Applied Computing Professor at Unisinos. Email: gdoramos@unisinos.br

same cannot be said for making a dataset that maps the position of these objects to the car's ideal steering and velocity. A Reinforcement Learning agent, on the other hand, can learn to control the vehicle inside the safety of a simulated environment, and later only fine-tune its behavior to adjust itself to the real world.

Despite the demonstrated potential, RL still faces a plethora of challenges, especially when dealing with complex environments, where learning a good set actions can take a prohibitively long time. Temporal abstraction in Reinforcement Learning has been a major topic of research in the field over the past decades, which attempts to improve performance by delegating sub-tasks of a problem to separate components, thus reducing the overall complexity of learning extended courses of action. The Options framework proposed by SUTTON; PRECUP; SINGH (1999) introduces temporal abstraction by combining single step actions with options, which represent longer courses of action with variable time. Options have been used as the base for many subsequent methods—some of which are presented in Section 3. However, efficiently working with options is still an open problem in literature, as discovering options autonomously, as well as what defines a good option, is still broadly debated.

The Option-Critic Architecture (BACON; HARB; PRECUP, 2017) strives to address these issues by discovering options using policy gradient techniques, which provides a more scalable solution when compared to using subgoals like its predecessors. However, Option-Critic—as well as its derivatives—is limited by the fact that the number of discovered options are fixed. When starting the training process, the user must provide the number of desired options, which is not ideal given that in most scenarios it is not clear prior to execution how many options the algorithm will be able to benefit from. On the one hand, defining too few options can hinder the agent's performance, and, on the other, too many options can have an impact on training and planning speeds. WAN; SUTTON (2022) propose the Fast Planning Option-Critic (FPOC) as an extension of Option-Critic that focuses on improving planning speed by only allowing a handful of options to be considered at any given time. Despite mitigating part of the problem, FPOC would still perform better if it was able to create new options as needed.

In this paper, we propose an extension of the FPOC algorithm for dynamically creating options in training time. Under the assumption that the variance in an option's episodic return represents its uncertainty, we analyze the rate at which the variance of every option increases by calculating the slope of its integral in a small window of time. If the slope if too steep, determined by a user-provided threshold value, the algorithm will create a new option whose goal is to complement the existing options, focusing on what the current set of options are struggling to learn. The complementary option uses a replay buffer to learn from the episodes where the previous options performed badly, thus ensuring diversity in the learned options. We tested our method in the four-rooms environment, also used by the base algorithm. As a result, our method manages to create just the necessary amount of options for the given problem, while also resulting in a slightly higher average return when compared to FPOC. Although not covered in this paper, the proposed solution can be easily adapted to other variations of the Option-Critic

architecture.

Our method improves upon the state-of-the-art by providing two major contributions to the Option-Critic architecture: 1. A method for determining whether the learning process would benefit from more options, which can be used either in training time or in future experiments as an indication that the current set of options could not encompass the entire problem. 2. A novel technique for creating new options in training time by using experience replay, which can not only avoid expensive re-runs of the training process, but also achieve grater reward and planning speed by discovering the minimum number of options necessary for learning the dynamics of any given environment.

The rest of this document is organized as follows: Section 2 provides an overview of the main components behind Reinforcement Learning, as well as a brief explanation of the Options framework and the Option-Critic architecture. Section 3 presents the related work found in literature, as well as their gaps and opportunities. In Section 4, we present our proposed method, explaining the logic behind some of our decisions. Section 5 contains the results obtained by our algorithm, as well as comparisons with the base FPOC. Finally, Section 6 provides a brief discussion on the limitations of our work, as well as concluding remarks.

2 BACKGROUND

2.1 Reinforcement Learning

Reinforcement learning is a field of machine learning that focuses on goal-oriented learning, in which a computational agent learns by interacting with its surrounding environment. Unlike traditional Supervised Learning methods where the system's behavior is generalized from examples of good outcomes—the labels—given a set of inputs, a reinforcement learning agent only requires interaction with the environment or a similar enough model to learn an optimal behavior (SUTTON; BARTO, 2020). The idea of learning by interaction is much more aligned with the traditional concept of the word, and is also the method humans and animals use to acquire skills in any task. For instance, a toddler learns how to walk by repetitive imperfect attempts, iterating on their experience before trying again. A reinforcement learning agent, for instance, can apply the same logic to learn how to control a bipedal robot, where it can use its past experience—i.e. how many steps it applied to a motor—and the feedback from the environment—i.e. how far it got before eventually falling over—to adjust its internal parameters in the direction of walking even farther in the next attempt.

2.1.1 Elements of Reinforcement Learning

As previously mentioned, the reinforcement learning framework defines an agent and the environment it is in. However, there are four other main elements behind the learning process: a

policy, a reward signal, a value function and, optionally, a model of the environment (SUTTON; BARTO, 2020), which shall be explained below.

A policy, represented by the symbol π, is the definition of how the agent will behave at any given time. If we define the agent's current situation as the state s, and the action it takes sampled from a finite set of available actions as $a \in \mathcal{A}$, then $\pi(a|s)$ represents the probability (represented by the symbol "|") that the agent will select action a when faced with state s. An agent's policy can be computationally represented as a mapping from states to actions inside a lookup table, but it is also possible to represent it with more complex search functions and gradients.

If a policy represents what action to take at a certain state, then the reward signal is the consequence of following this state-action pair. On each time step, the environment returns to the agent a numeric value determining how good the performed action was for the current state; this value is called the *reward*. The ultimate goal of a reinforcement learning agent is to accumulate the largest amount of reward as possible until termination—that is, to take action A_t at each time step t given the current state S_t so that it achieves the largest sum of rewards from $t = 0$ until $t = T$, where T represents the final time step. The sum of all rewards starting from some time step t until the final time step is called the *return*, represented as $G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T$ or, as shown in Equation 1:

$$G_t \doteq \sum_{k=t+1}^{T} R_k \tag{1}$$

Although the above equation is suitable for *episodic* tasks—where there is a clearly defined termination condition in the form of a terminal state—, it cannot be directly applied in *continuous* tasks, where the concept of an episode is not clearly defined and it might even continue forever. If $T = \infty$, Equation 1 diverges, thus making the approximation of G_t impossible. A more practical (and common) representation for the return is the *discounted return* where each future reward, as the name implies, has a discount rate $\gamma \in [0, 1)$ applied to it, as shown in Equation 2. This ensures that the return converges for every $R_t \in \mathbb{R}$ for continuous tasks.

$$G_t \doteq \sum_{k=0}^{+\infty} \gamma^k R_{t+k+1} \tag{2}$$

Now that we have established that the goal of a reinforcement learning agent is to maximize its return, we can define the value function of a state as being its expected return, starting from that state, and following some policy until termination. A value function is always written with respect to a policy π and a state s as $v_\pi(s)$, which represents how much reward an agent expects to accumulate when starting in s following π thereafter or, in other words, how good it is to be at s at any given time. By combining the value function of every state under a policy π, we define the state-value function v_π. Value functions are a vital part of learning, as they can be used to compare different policies in terms of their expected reward—if $v_{\pi'}(s) > v_\pi(s)$, then

following π' when faced with state s will lead to more reward than continuing to follow π. By also keeping track of the value of taking an action a, starting from state s, and following π thereafter, we can define the action-value function $q_\pi(s, a)$ as how good it is to take action a when in s and follow π afterwards. Similarly to the state-value function, we can define q_π as the action-value function for policy π.

Lastly, a model of the environment can be used for estimating the outcomes of state-action transitions without having to interact directly with it. Models can be used for *planning*—further explained in Section 2.1.3—which can be briefly described as the process of deciding what action to take by considering its possible outcomes. Reinforcement Learning methods can be either *model-based* or *model-free*, depending on whether they use a model of the environment or direct experience with it to learn state-action transitions, respectively.

2.1.2 Markov Decision Processes

A Markov Decision Process, MDP for short, is the formalization of the sequential decision making problem, where taking an action in the current time step affects not only its immediate reward, but also the subsequent state, and, consequently, their future rewards as well (SUTTON; BARTO, 2020). We can mathematically define a finite MDP as a tuple $\mathcal{M} \doteq \langle \mathcal{S}, \mathcal{A}, \gamma, r, p \rangle$, where $\mathcal{S}$ and $\mathcal{A}$ denote the state and action sets, respectively, γ the discount factor, r the reward function, and $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ the probability distribution over next states given that an action $a \in \mathcal{A}$ is taken at state $s \in \mathcal{S}$ (HARB et al., 2018). This formalization combines all concepts introduced in Section 2.1.1 into a cyclical representation where, given a state $s \in \mathcal{S}$, the agent takes action $a \in \mathcal{A}$, receives a reward $r \in \mathbb{R}$, and transitions to the next state $s' \in \mathcal{S}^+$ ($\mathcal{S}$ plus a terminal state) with probability $p \in [0, 1]$, represented by Equation 3:

$$p(s', r|s, a) \doteq Pr\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\} \tag{3}$$

When a policy π interacts with an MDP $\mathcal{M}$, we say that π induces a Markov process over the states, actions, and rewards of $\mathcal{M}$ over which we can expect a discounted return defined by the value function $v_\pi(s)\, \forall s \in \mathcal{S}$:

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi \left[G_t | S_t = s\right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma G_{t+1} \mid S_t = s\right] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \Big[r + \gamma \mathbb{E}_\pi \left[G_{t+1} \mid S_{t+1} = s'\right]\Big] \\ &= \sum_a \pi(a|s) \sum_{s', r} p(s', r|s, a) \left[r + \gamma v_\pi(s')\right], \ \forall s \in \mathcal{S} \end{aligned} \tag{4}$$

For each combination of (a, s', r), we compute its probability $\pi(a|s)\, p(s', r|s, a)$ and use it

to give weight to the reward plus discounted value function of each successor state. Equation 4 is the Bellman equation for v_π, which is a cornerstone of Dynamic Programming and can be interpreted as the average over all future possibilities for s, weighted by its probability of occurring (BELLMAN, 1957).

We can assume for finite MDPs that there exists a policy that is better than all others for a given task, which is denoted as the optimal policy. Policies can be partially ordered by their respective value functions. A policy π is perceived as better than some other policy π' if its expected return is greater or equal to the one in π' for all states. In other words, $\pi \geq \pi'$ if and only if $v_\pi(s) \geq v_{\pi'}(s)\ \forall s \in \mathcal{S}$. An optimal policy π_* therefore has an optimal state-value function v_* and an optimal action-value function q_* defined as, respectively:

$$
\begin{aligned}
v_*(s) &\doteq \max_\pi v_\pi(s) \qquad && \forall s \in \mathcal{S} \\
q_*(s,a) &\doteq \max_\pi q_\pi(s,a) \qquad && \forall s \in \mathcal{S}, a \in \mathcal{A}(s)
\end{aligned}
\tag{5}
$$

2.1.3 Models and planning

In Reinforcement Learning, a model of the environment is some form of representation the agent can use to predict how the environment will react to an action in a given state. We use the term *planning* to describe when an agent interacts with a model to predict the outcome of its actions in the current state—and possibly future states—for improving its policy. The opposite of planning, *learning*, is when an agent adjusts its policy by interacting directly with the environment. Methods that rely on planning are called *model-based*, while models that rely on learning are known as *model-free*.

Planning can also be useful at decision time—that is, the agent can simulate the scenario of taking a certain action in the current state before making a decision. This form of planning is called *decision-time planning*, as the simulation returns the action A_t the agent should take at the current state. Decision-time planning can be particularly useful in more complex scenarios where thinking multiple steps ahead is more important than speed, such as in a game of chess. However, planning can suffer from latency issues, as simulating multiple future scenarios at every state transition can be prohibitively expensive in terms of computational cost. As a matter of fact, the more actions available at any given state, the more expensive planning becomes, as more state-action permutations must be computed.

2.1.4 Dynamic Programming in the context of Reinforcement Learning

Dynamic Programming algorithms, although of limited use in RL due to their assumption of a perfect model of the environment as an MDP, still provide much of the theory behind learning optimal policies. The goal of DP is to compute better policies through the use of value functions

in a way that, by calculating the value function of a learned policy, we can use it to find a better policy. This creates a loop between *policy evaluation* and *policy improvement* operations which, as shall be demonstrated in the following paragraphs, ensures us that every iteration will return a new policy that is equal or better to the previous policy.

Using Equation 4 as a base, we can start with an approximation v_0 chosen arbitrarily and apply the Bellman equation for v_π to obtain a new value function approximation v_1, then use v_1 to obtain v_2, and so on. This process is also known as *iterative policy evaluation*, which converges to v_π as $k \to \infty$. We can generalize the process of approximating v_{k+1} from v_k as:

$$\begin{aligned} v_{k+1}(s) &\doteq \mathbb{E}_\pi \left[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s\right] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_k(s')\right], \ \forall s \in \mathcal{S} \end{aligned} \tag{6}$$

Now that we know how good it is to follow a given policy π starting from the state s, that is, $v_\pi(s)$, we can use this information to determine whether a change in behavior would be beneficial or not. Let us consider the value of selecting action a in s and follow the existing policy π thereafter, which can be represented by:

$$\begin{aligned} q_\pi(s, a) &\doteq \mathbb{E}\left[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a\right] \\ &= \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_\pi(s')\right] \end{aligned} \tag{7}$$

If $q_\pi(s, a)$ is greater than $v_\pi(s)$, it means that it is better to select a once in s and continue to follow π afterwards than to simply follow π all the time. With this in mind, one can assume that selecting a for every occurrence of s would be even better, which would mean that a new policy π' where $\{\pi'(a|s) = 1|S_t = s\}$ is better than the current policy, or equal to it if the current policy is already optimal. As shown in (SUTTON; BARTO, 2020), this assumption is true and is a special case of a general result called the *policy improvement theorem*.

Given the evidence above, we can extend policy improvement to all states by selecting at each state the action that gives us the largest action-value estimate $q_\pi(s, a)$. This will return a new policy π' that is greedy with respect to the value function of the original policy, as shown in Equation 8:

$$\begin{aligned} \pi' &\doteq \underset{a}{\text{argmax}}\, q_\pi(s, a) \\ &= \underset{a}{\text{argmax}}\, \mathbb{E}\left[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a\right] \\ &= \underset{a}{\text{argmax}} \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_\pi(s')\right], \end{aligned} \tag{8}$$

where argmax_a is the value of a that maximizes the expression.

As mentioned in the beginning of the section, we can use policy-evaluation and policy-improvement in tandem to achieve the optimal policy. By starting with a policy π, we can use v_π to get a better policy π', then get $v_{\pi'}$ and use it to obtain an even better policy π'', and so on. After a number of iterations, the policy will converge to the optimal policy π_*, with value function v_{π_*}. This process is called *policy iteration*; however, it is not necessary for these two processes to be perfectly alternated. In asynchronous DP methods, for example, there can be several policy evaluation steps before a policy improvement step, and vice-versa. We use the term *generalized policy iteration* (GPI) to describe any interaction between policy-evaluation and policy-improvement, regardless of the granularity.

2.1.5 Temporal Difference Learning

Temporal Difference (TD) learning is, undoubtedly, the most common method used in literature for solving Reinforcement Learning tasks. Unlike Dynamic Programming, TD methods do not require a perfect model of the environment and therefore are much more feasible for solving real-world problems, where we cannot estimate all the dynamics of the environment or computing them would take longer than solving the task itself. We call the way TD algorithms work as *learning from experience*, since they only need to observe outcomes to improve their policy. Temporal Difference is also more efficient than Monte Carlo methods, as they do not need to wait until the current episode terminates before incrementing their target. The simplest TD method, called *one-step* TD or *TD(0)*, is represented in Equation 9:

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right] \ , \tag{9}$$

where $V(S_t)$ is the update target for S_t, and α is a constant step-size parameter.

As we can see, TD(0) only needs the next state and reward to adjust its target, which means that it can make an update right after taking the step $t+1$. We can also see that the elements in brackets represent a subtraction of the current estimated value for S_t from the updated value $R_{t+1} + \gamma V(S_{t+1})$, which denotes the *TD error*, represented by $\delta_t \doteq R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$. The TD error is used in various other algorithms, including the Fast-Planning Option-Critic algorithm—further explained in Section 2.3.2—we use as the base for our work.

One of the first breakthroughs in TD was Q-learning (WATKINS; DAYAN, 1992) which is an *off-policy* learning method, meaning that it learns a target policy while following a different policy—called the behavior policy. The algorithm for Q-learning, shown in Equation 10, approximates the optimal action-value function q_* regardless of the policy being followed.

$$Q(S_t, A_t) \rightarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right] \tag{10}$$

2.2 The options framework

Our work focuses on adjusting the number of *options* in a Reinforcement Learning problem. The options framework is a set of methods for working with temporal abstraction in RL originally proposed by (SUTTON; PRECUP; SINGH, 1999). Options can be described as closed-loop policies that focus on taking high-level actions that span over an extended period of time, be it a few time steps or thousands. For example, when planning a vacation, a human would normally think of steps like searching for a destination, booking a flight, or checking hotel availability. However, as MDPs are only capable of working with discrete time steps, each of these actions would need to be represented as a series of muscle twitches. Even the smallest of actions such as opening a browser to search for good destinations can take thousands of discrete steps, thus making long-term planning and, consequently, goal-oriented behavior, virtually impossible without some form of high-level abstraction. By instead learning sub-tasks as separate policies that can take a non-deterministic number of steps to complete, an agent would only have to iterate over the set of high-level tasks to plan ahead. These temporally extended courses of actions are known as *options*.

As previously stated, MDPs are unable to handle temporally extended actions as they expect that an action a, taken at time t, results in an immediate next state at time $t + 1$, which is not the case for most options. With this in mind, the authors build on the theory of semi-Markov decision processes (SMDPs), which can be described as a variation of the regular MDP that is capable of modeling continuous-time discrete-event systems—in other words, SMDPs allow a model to support temporally-extended courses of action. A fixed set of options $\mathcal{O}$ defines a discrete-time SMDP embedded within the original MDP, where the base system is an MDP with single-step transitions and the options define potentially larger SMDP-like transitions (SUTTON; PRECUP; SINGH, 1999).

An option can be represented by the triple $\langle \mathcal{I}, \pi, \beta \rangle$, where $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ represents its internal policy, $\mathcal{I} \subseteq \mathcal{S}$ is the initiation set denoting in what states the option may be initiated, and $\beta : \mathcal{S}^+ \rightarrow [0, 1]$ is the option's termination condition. Just like in traditional RL, once an option is initiated at time t, an action a_t is selected from the probability distribution $\pi(s_t, \cdot)$, which transitions the environment to the state s_{t+1}. After a state transition, the option can either terminate with probability $\beta(s_{t+1})$ or continue its execution by determining a_{t+1} from $\pi(s_{t+1}, \cdot)$, transitioning to s_{t+2}, possibly terminating with probability $\beta(s_{t+2})$, and so on. Once an option terminates, the agent chooses a new option from the set of available options for the current state, defined by $\mathcal{O}_s$. It is convenient to extend $\mathcal{O}_s$ for every $s \in \mathcal{S}$ to also cover the primitive actions in the set $\mathcal{A}_s$, which can be represented as a special type of option that can be initiated anywhere, only lasts a single time step and always chooses a: $o_a = \langle \mathcal{I} : \mathcal{S},\ \pi : a\ \forall s \in \mathcal{S},\ \beta : 1\ \forall s \in \mathcal{S} \rangle$. By extending the option set, we have $|\mathcal{O}| = |\textit{regular options}| + |\textit{primitive options}|$.

We define the policy responsible for choosing a new option o from the set $\mathcal{O}_s$ as a policy over options $\mu : \mathcal{S} \times \mathcal{O} \rightarrow [0, 1]$. Given the context of μ, we need to adapt many of the concepts

from Sections 2.1.1, 2.1.2, and 2.1.4 to deal with a set of options which may elapse for several discrete time steps, as opposed to primitive actions that always complete after a single time step t. The probability distribution over actions $p(s', r|s, a)$, for instance, explains only a single, deterministic transition from s to s', given a. The options equivalent must take into account the termination condition, which can differ based on the number of elapsed steps. We generalize a state-prediction over options as:

$$p^o_{ss'} = \sum_{k=1}^{+\infty} p(s', k)\gamma^k, \ \forall s \in \mathcal{S} \ , \tag{11}$$

where $p(s', k)$ is the probability that the option terminates in s' after k steps. Similarly, we can redefine the state-value and action-value—now option-value—functions as follows:

$$\begin{aligned} V_\mu(s) &\doteq \sum_{o \in \mathcal{O}_s} \mu(s, o) \left[r^o_s + \sum_{s'} p^o_{ss'} V_\mu(s') \right] \\ Q_\mu(s, o) &\doteq r^o_s + \sum_{s'} p^o_{ss'} \sum_{o' \in \mathcal{O}_{s'}} \mu(s', o') Q_\mu(s', o') \end{aligned} \tag{12}$$

Sutton *et al.* define *intra-option* learning as techniques to estimate optimal values in the context of options (SUTTON; PRECUP; SINGH, 1999). Assuming that o is Markov, we can update $Q^*_{\mathcal{O}}(s_i, o)$ for each intermediate state s_i where o is not terminated, as continuing to follow the current option is still considered a valid experience of o in s_i. Furthermore, we can also use experience from one option to update others that follow similar courses of action, but may terminate in different time steps. By using experience with one option to update all other options, irrespective of their role in generating said experience, we can define these learning methods as *off-policy* training. In the subsequent update functions, the authors denote $U^*_{\mathcal{O}}(s, o)$ as the optimal value of a state-option pair given that o is Markov and is executed upon arrival at s, defined by $U^*_{\mathcal{O}}(s, o) = (1 - \beta(s))Q^*_{\mathcal{O}}(s, o) + \beta(s) \max_{o' \in \mathcal{O}} Q^*_{\mathcal{O}}(s, o')$. The Bellman equation for $Q^*_{\mathcal{O}}(s, o)$, given $U^*_{\mathcal{O}}(s, o)$, can be defined as:

$$Q^*_{\mathcal{O}}(s, o) = \sum_{a \in \mathcal{A}_s} \pi(s, a) \left[r^a_s + \sum_{s'} p^a_{ss'} U^*_{\mathcal{O}}(s', o) \right] \tag{13}$$

The original options framework does not touch on the subject of automatic option discovery. The authors do, however, define the concept of *subgoals*, which are manually formulated by assigning a *terminal subgoal value* $g(s)$ for each state to indicate how desirable it is for the learned option to terminate at s. Then, learning the option becomes a matter of solving the Bellman equation for $Q_g(s_t, a_t)$ with any method mentioned in either Section 2.1.4 or Section 2.1.5. Manually specifying subgoals, however, is of limited usefulness in more complex scenarios, as it may not be apparent for a human what defines a good subgoal. In the past two decades, several authors have tackled the problem of discovering options without human input. We shall

focus solely on the Option-Critic architecture (BACON; HARB; PRECUP, 2017), as it is the foundation for the works preceding ours.

2.3 The Option-Critic Architecture

Unlike previous works, the Option-Critic Architecture does not try to identify subgoals. Instead, it uses the policy gradient theorem (SUTTON et al., 1999; KONDA; TSITSIKLIS, 1999) to derive new policies, thus addressing the scalability issues that arise with finding subgoals and their respective optimal policies in more complex problems (BACON; HARB; PRECUP, 2017). One of the method's greatest improvements over previous work is that it can gradually learn the intra-option policies and the policy over options at the same time, incurring no slowdown when compared to the traditional RL learning process.

The Option-Critic (OC) model is based on a two-timescale framework, more commonly known as Actor-Critic—which is also the reference for its name. An Actor-Critic algorithm combines a *critic* relying on value function approximation to update the policy parameters of an *actor*, leading its gradient towards improvement (KONDA; TSITSIKLIS, 1999). In the context of OC, the intra-option policies, termination functions, and policy over options are part of the actor, while Q_U and $A_\mathcal{O}$ belong to the critic. $Q_U(s,o,a)$ refers to the value of executing an action a in the state-option pair $U(o,s)$—described at the end of Section 2.2. Meanwhile, $A_\mathcal{O}$ refers to the advantage function over options $A_\mathcal{O}(s,o) = Q_\mathcal{O}(s,o) - V_\mathcal{O}(s)$ (BAIRD III, 1993).

Given the parameterized nature of the actor, the authors denote $\pi_{o,\theta}$ as the intra-option policy of option o parameterized by θ, and $\beta_{o,\vartheta}$ as the termination function of o, parameterized by a different value ϑ. The gradients of the intra-option policy w.r.t. θ and terminations w.r.t. ϑ are respectively shown in Equation 14. For the detailed theorem and proofs of each gradient, see (BACON; HARB; PRECUP, 2017).

$$\begin{aligned}\frac{\partial Q_\mathcal{O}(s,o)}{\partial\theta} &= \sum_{s,o}\mu_\mathcal{O}(s,o|s_0,o_0)\sum_a\frac{\partial\pi_{o,\theta}(a|s)}{\partial\theta}Q_U(s,o,a)\\ \frac{\partial Q_\mathcal{O}(s,o)}{\partial\vartheta} &= -\sum_{s',o}\mu_\mathcal{O}(s',o|s_1,o_0)\frac{\partial\beta_{o,\vartheta}(s')}{\partial\vartheta}A_\mathcal{O}(s',o)\end{aligned} \tag{14}$$

By combining options evaluation and options improvement in the same iteration, one can use a method such as *intra-option Q-learning* (SUTTON; PRECUP; SINGH, 1999) to determine Q_U and $A_\mathcal{O}$ and use both values to update θ and ϑ, respectively. The authors applied the proposed architecture into the four-rooms scenario from (SUTTON; PRECUP; SINGH, 1999) and found that the discovered options' terminations match the ones manually encoded by the original authors, which demonstrates that the Option-Critic method can derive options for that problem just as well as a human could—which in this case was the optimal set of options.

Despite discovering a good set of options in the four rooms experiment, Option-Critic pre-

sented limitations in more complex tasks like the Arcade Learning Environment (BELLEMARE et al., 2013), where the authors found that the termination gradient tends to "shrink" options over time to the point where they terminate after a single time step. The issue was remediated by adding a small regularization term $\xi = 0.01$ to the advantage function: $A_{\mathcal{O}}(s,o) + \xi = Q_{\mathcal{O}}(s,o) - V_{\mathcal{O}}(s) + \xi$. A subsequent work from the same authors tackles this issue by using deliberation cost, which shall be presented below.

2.3.1 Asynchronous Advantage Option-Critic and the Deliberation Cost Model

As previously mentioned, the Option-Critic framework suffers from a tendency to reduce options into primitive actions over time (BACON; HARB; PRECUP, 2017). The authors acknowledge this issue and propose an improved method that focuses on the concept that "good options are those which allow an agent to learn and plan faster" (HARB et al., 2018). By "planning faster", the authors mean the fact that continuing to follow the currently active option is considerably faster than terminating it and choosing a new one—in other words, having longer options is desirable not only for better temporal abstraction, but also computational performance.

Let $Z_t \doteq (S_t, O_t)$ be a random variable over state-option tuples, which represent the augmented state space when considering the flat policy over actions obtained from the policy over options μ. The value function over the augmented state space $\widetilde{V}_\theta(z) \doteq Q_\theta(s,o)$. The deliberation cost function over state-option pairs $\widetilde{D}_\theta(z) \doteq D_\theta(s,o)$ represents the expected sum of discounted costs over an immediate cost function $\tilde{c}(z,a,z') \doteq c(s,o,a,s',o')$:

$$\widetilde{D}_\theta = \mathbb{E}_\theta \left[\sum_{t=0}^{+\infty} \gamma^t \tilde{c}(Z_t, A_t, Z_{t+1}) \middle| Z_0 = z \right] \tag{15}$$

The are several ways to define a cost function that favors long options, one of them being by penalizing frequent options switches (HARB et al., 2018). The authors propose using $\beta_\theta(s',o)$, as it is the mean of a Bernoulli random variable over the two possible outcomes for termination (either switching or continuing). Thus, we can define $c_\theta(s',o) = \gamma\beta_\theta(s',o)$ and obtain the option-value function given c parameterized by θ:

$$Q^c_\theta(s,o) = \sum_a \pi_\theta(a|s,o) \left(r(s,a) + \gamma \sum_{s'} P(s'|s,a) \Big[Q^c_\theta(s',o) - \beta_\theta(s',o)(A^c_\theta(s,o) + \eta) \Big] \right), \tag{16}$$

where $\eta \in \mathbb{R}$ is a regularization coefficient which sets a baseline for how good an option ought to be—the bigger the value, the more likely an option is maintained rather then terminated.

The authors propose the Asynchronous Advantage Option-Critic (A2OC) algorithm based on Asynchronous Advantage Actor-Critic (A3C)'s structure (MNIH et al., 2016) as an improvement over the original Option-Critic's DQN-based model. Compared to Option-Critic's model

with no deliberation cost, A2OC manages to avoid single-step options, thus resulting in options with better temporal abstraction. Options now tend to terminate at intersections, which represent key decision points in the environment (HARB et al., 2018). The discovered set of options not only results in a higher overall score in each test bench, but also faster planning due to not having to pick a new option at every state transition.

2.3.2 Fast-Planning Option-Critic

As mentioned in Section 2.1.3, the more actions an agent must consider, the longer it takes for it to perform planning. The same logic applies to options, where each primitive action becomes a unitary option, joined by the fixed set of k extended options defined in training. We can define the per-iteration complexity of planning as $\sum_{s \in \mathcal{S}} \Omega(s) \times |\mathcal{S}|$, where $\Omega(s) \doteq \{o \in \mathcal{O} : s \in \mathcal{I}_o\}$. This equation represents the average number of options that can be initiated at each state (WAN; SUTTON, 2022). The authors extend option discovery to the multi-task problem, where the objective becomes solving a finite set of episodic tasks $\mathcal{N}$, all of which share the same state and action spaces $\mathcal{S}$ and $\mathcal{A}$, respectively. For each task n, the authors define a *meta-preference function* $f^n \in \mathcal{S} \times \mathcal{H} \to [0, \infty)$, which represents the agent's willingness to choose the option indexed by $h \in \mathcal{H} \doteq \{1, 2, \ldots, k + |\mathcal{A}|\}$ at every state for task n. The set of meta-preference functions for every task is denoted by $\mathcal{F} \doteq \{f^n : n \in \mathcal{N}\}$. It is important to note the usage of the option's index h in the set of options instead of the option o itself. This is done because the options themselves can change during training, while their index remain the same.

Given a set of options $\mathcal{O}$ and a set of meta-preference functions $\mathcal{F}$, we can define $\bar{q}^n_{\mathcal{O},\mathcal{F}}(s, h, a)$ as the expected cumulative reward starting from state s, choosing option o_h, and action a, and behaving in the *call-and-return* fashion—in which the agent selects an option based on a policy over options and follows its intra-option policy until termination. The option-value function can be defined as $\bar{q}^n_{\mathcal{O},\mathcal{F}}(s, h) \doteq \sum_a \pi_{\mathcal{O}}(a|s, h)\bar{q}^n_{\mathcal{O},\mathcal{F}}(s, h, a)$ and the value function as $\bar{v}^n_{\mathcal{O},\mathcal{F}}(s) \doteq \sum_h \mu^n_{\Omega(s)}(h|s)\bar{q}^n_{\mathcal{O},\mathcal{F}}(s, h)$. As we can see, the value and option-value functions are the same as in the original options framework, but with the added notation for task n and meta-preferences set $\mathcal{F}$. Similarly, the authors define $\tilde{q}^n_{\mathcal{O},\mathcal{F}}(s, h, a) \doteq -\mathbb{E}[|\Omega(S_0)| + \Omega(S_1)| + \cdots + \Omega(S_{t_{N-1}})||S_0 = s, H_0 = h, A_0 = a]$ as the negative expected cumulative number of options that the agent needs to consider before reaching a terminal state starting from state s , choosing option o_h, taking action a, and then following the policy over options $\mu^n_{\Omega(s)}$. We can define $\tilde{q}^n_{\mathcal{O},\mathcal{F}}(s, h)$ and $\tilde{v}^n_{\mathcal{O},\mathcal{F}}(s)$ in the same way as $\bar{q}^n_{\mathcal{O},\mathcal{F}}(s, h)$ and $\bar{v}^n_{\mathcal{O},\mathcal{F}}(s)$, respectively.

The objective proposed by the authors prefers a set of options such that there exists a policy choosing from them and achieving a high expected return, as well as that few options are considered at each decision point when following the policy (WAN; SUTTON, 2022). More precisely, the proposed method strives toward having few options considered at each state and a small number of options executed until termination. The authors' proposed objective can be

written as Equation 17:

$$J(\mathcal{O}, \mathcal{F}, c) \doteq \sum_n \sum_s d_0(s)\bar{v}^n_{\mathcal{O},\mathcal{F}}(s) + c \sum_n \sum_s d_0(s)\tilde{v}^n_{\mathcal{O},\mathcal{F}}(s) \ , \tag{17}$$

where $c > 0$ is a problem parameter. Furthermore, the authors also introduce the concept of interest functions as a replacement for the option's initiation set, which allow an option's initiation to be stochastic and thus more selective. This aligns with the FPOC objective of considering few options at any given state. An option using interest functions is defined as the tuple $\langle i_o, \pi_o, \beta_o \rangle$, where $i_o \in \Gamma$ is the option's interest function, $\Gamma : \{f | f : \mathcal{S} \to [0, 1]\}$. At each state, its initiation set is determined by sampling the interest functions for every option initializable in the current state: $Pr(o \in \Omega(s)) = i_o(s)$.

3 RELATED WORK

As mentioned in Section 2.2, the initial work in option discovery was primarily focused on autonomously discovering subgoals. One popular approach for determining good subgoals was through the concept of *bottleneck states*, which represent regions where the agent tends to visit frequently on successful paths but not on unsuccessful ones (MCGOVERN; BARTO, 2001; MENACHE; MANNOR; SHIMKIN, 2002; STOLLE; PRECUP, 2002; ŞIMŞEK; BARTO, 2008). The idea behind learning bottleneck states as options was that, by delegating reaching important states to a separate policy, the main policy could focus on reaching the goal once it is there. The Q-Cut algorithm (MENACHE; MANNOR; SHIMKIN, 2002) saves the transition history of the MDP into a graph and applies the Max-Flow/Min-Cut algorithm on it to find such states. The authors then use experience replay to learn options whose goal is to reach one of those states. Compared to standard Q-learning, Q-cut manages to converge to an optimal result considerably faster in a segmented maze problem. The main issue with the algorithm, however, is that storing very large state spaces in a graph structure and applying computations to it may result in an excessive use of memory and CPU time, thus limiting its scalability.

The implementation proposed in (MCGOVERN; BARTO, 2001) for discovering bottleneck states, on the other hand, use multiple-instance learning (DIETTERICH; LATHROP; LOZANO-PÉREZ, 1997) to identify subgoals. As pointed by the authors, this learning method is based on the idea of "bags", where positive bags contain at least one positive instance, while negative bags consist only of negative instances. Once bottlenecks are identified by counting first visits to each state, the authors use saved experience to identify successful paths to each bottleneck state using diverse density. The more diversely dense a region is, the more positive bags and less negative bags it has which, the authors explain, defines a bottleneck region. More importantly in the context of the present work is the fact that the proposed method uses saved trajectories to form new options in training time. The algorithm maintains for each state the running average of how often it appears as a diverse density peak and, if this number goes above

some threshold, a new option is created whose policy consists of navigating towards that state. When comparing two-room gridworld navigation with primitive actions, the authors found that once a new option is created, the average number of steps to reach the goal drops considerably, which significantly accelerates the learning process.

In recent years, there have been numerous option discovery algorithms that go beyond just bottlenecks. MACHADO; BELLEMARE; BOWLING (2017) introduce the concepts of *eigenpurpose* and *eigenbehavior* for describing reward functions and policies by combining representation learning with option discovery. The authors construct proto-value functions (PVFs) from the MDP's transition matrix which represent the desire to reach some specific region in the state space. Their conclusion was that *eigenoptions*—the options discovered by their method—tend to incentivize exploration while not discovering only bottlenecks, which they argue can hinder exploration when naively added to the agent's action set. Subsequent work focus on expanding the concept of eigenoptions to stochastic environments (MACHADO et al., 2018) and deep reinforcement learning (KLISSAROV; MACHADO, 2023), respectively.

Other authors take advantage of the abstraction capabilities of options for learning skills that can be used in tasks similar to the one the option was trained on. In (KONIDARIS; BARTO, 2007), the authors propose learning options over two separate representations, one in problem-space that is Markov and particular for the current task, and other in agent-space that may not be Markov but is retained across tasks. CROONENBORGHS; DRIESSENS; BRUYNOOGHE (2008) approach skill transfer by extending the options framework to the relational setting, which allows abstractions over object identities. More recently, HAN; TSCHIATSCHEK (2022) propose finding *abstract successor options*, which represents options through their successor features. Successor features can be described as the discounted sum of features of state-action pairs encountered when starting from some state and following a given policy (BARRETO et al., 2017).

Lastly, the Option-Critic architecture presented in Section 2.3 has been used as the base for many algorithms, two of which are presented in Sections 2.3.1 and 2.3.2. Other notable variations of Option Critic are the Safe Option-Critic and the Attention Option-Critic. The Safe Option-Critic (JAIN; KHETARPAL; PRECUP, 2021) is an extension of OC that focuses on safe behavior. The authors consider a behavior as safe when it avoids regions of state-space with high uncertainty in its outcomes. By defining controllable states as those with a lower variance in TD error, the Safe Option-Critic algorithm adds controllability to the optimization objective. As a result, the proposed method managed to identify and avoid areas in the state space with high variability, which, as expected, result in more predictable returns. Meanwhile, the Attention Option-Critic (CHUNDURU; PRECUP, 2022) uses an attention mechanism for learning options that focus on different aspects of the observation space. The algorithm learns an attention mechanism with the intent of maximizing the expected cumulative return of the agent while also maximizing the distance between the attentions of options. When compared to Option-Critic, the learned options are not only more diverse, but also do not degenerate over

time, meaning that they are more evenly selected and last longer.

We base our method on the Fast Planning Option-Critic proposed in (WAN; SUTTON, 2022). The implementation details were already presented in Section 2.3.2. The proposed method manages to accomplish faster planning by reducing the number of elementary operations in option-value iterations. The proposed algorithm, however, is limited in three major ways according to the authors. The first limitation is the fact that the algorithm treats only the tabular setting, which limits the complexity of problems that can be tackled. The second limitation is the fixed set of tasks, which the authors argue should be discovered by the agent. The last major limitation is the reliance on a human-specified number of options, which we tackle in our approach. Although some of the early work in option discovery supported a dynamic number of options, recent implementations rely on a user-defined number, which can hinder their quality if set too low and hurt planning speeds if set too high. To the best of our knowledge, this is the first work that focuses on implementing a dynamic option scaling algorithm to option-critic algorithms.

4 CREATING OPTIONS DYNAMICALLY

After a background on Reinforcement Learning with Options, as well as an overview of the literature on the subject, we now present our method for dynamically increasing the number of options in training time. Our method was implemented on top of the Fast Planning Option-Critic algorithm from (WAN; SUTTON, 2022) explained in Section 2.3.2, though it can easily be adapted to other option-critic variations. The dynamic option creation algorithm is integrated into the training phase, where every episode elapsed in training is saved into a replay buffer, as explained in Section 4.1. After a pre-determined number of episodes, the algorithm interrupts the learning process and runs a separate evaluation process for some amount of episodes, adding the episodic return variance for each option into an array. Once the evaluation completes, the algorithm resumes regular training.

The algorithm repeats this process until the variances array is full, whose size is determined by an external parameter. Once it is full, the algorithm will use this array to determine whether any of the options are struggling with learning some specific behavior. The process of evaluating the option's performance is explained in Section 4.2, and consists of finding the slope of the integral for the values in the variances array, which represent the option's uncertainty. If the slope obtained by this process is above some user-defined threshold, the algorithm will initialize a new option and apply experience replay to it by using the replay buffer stored in training, as explained in Section 4.3. Once the new option is fully initialized, the algorithm resumes regular training, repeating the entire process again. This entire procedure is visually represented in Figure 1.

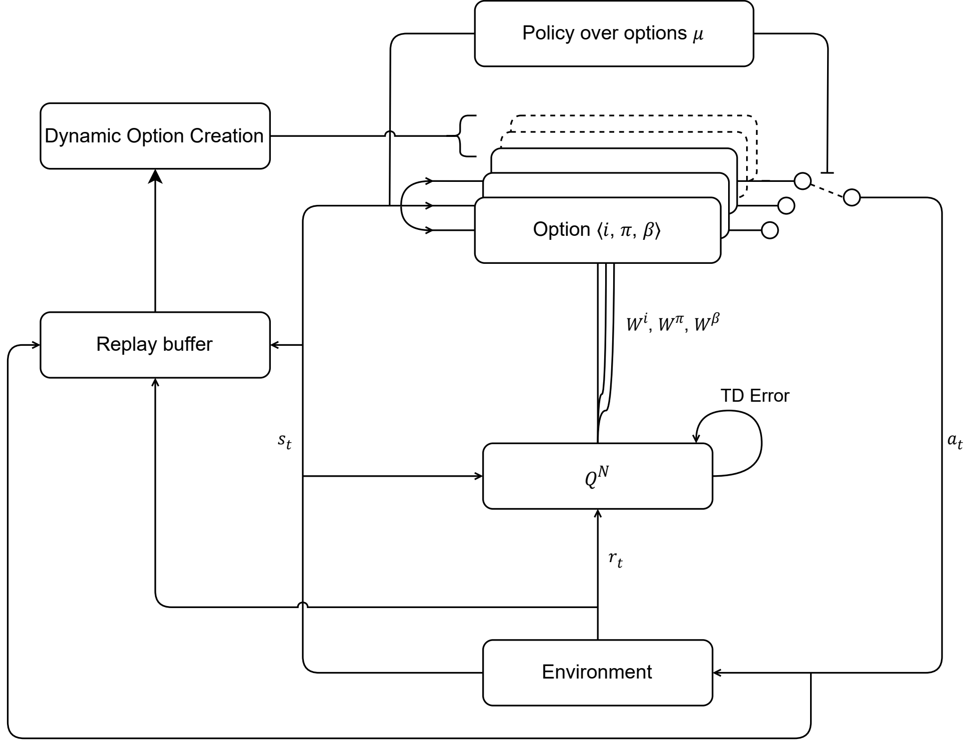

Figure 1 – Overview of the agent's learning process with dynamic option creation.

4.1 Saving experience as a replay buffer

As explained in Section 2.1.2, an MDP consists of a tuple containing a set of states $\mathcal{S}$, a set of actions $\mathcal{A}$, a reward function r, the state transition distribution p, and a discount factor γ. When an agent takes action a in state s, p determines the probability of transitioning to future states with some expected reward. When under a greedy policy π_ε, the agent will always transition to the state S_{t+1} at time t where the action-value function $q_{\pi_\varepsilon}(S_t, A_t)$ is highest, receiving a reward R_t in the process. We denote this entire process as a step, which represents the smallest period of time in an MDP. When we save a step, we are storing the agent's *experience* in that time period. We can later use this saved experience—usually as a set containing many steps—to train a different agent. This procedure is called *experience replay*, and the set containing the stored experiences is typically called a *replay buffer*.

Part of our goal is to create new options that specialize on what previous ones failed to learn. We apply experience replay to the newly created option with the intent of making sure it catches up with the previous options before resuming regular training. Let us define the replay buffer B as an array that stores the history of past episodes. Each element in the buffer represents a full episode, comprised of two items: 1. A list of tuples $\langle S, A, r, \bar{\rho}, S', \perp, tid \rangle$ representing the states in the elapsed episode, where $\bar{\rho}$ is the mean importance sampling ratio of all adjustable options, $\perp = 1$ if the step is terminal and 0 otherwise, and tid represents the episode's task id (specific for multi-task methods such as FPOC), and 2. The total episodic return $G_{1:t}$, starting from the first step in the episode, $t = 1$, until termination, $t = \tau$ if $\perp_\tau = 1$. The replay buffer is

formalized in Equation 18:

$$\begin{aligned} E &\doteq \langle S, A, r, \bar{\rho}, S', \bot, tid \rangle \\ B &\doteq \left\{ \{\{E_1, E_2, E_3, \dots, E_t\}, G_{1:t}\}^p \;\middle|\; p \in \mathcal{P} \right\}, \end{aligned} \tag{18}$$

where $p \in \mathcal{P}$ is an episode containing one or more experience steps E, and $G_p \doteq \sum_{t \in p} r_t$ is the sum of rewards in each episode p.

4.2 Evaluating an option's learning process

Every n episodes, the training is interrupted and an evaluation process is executed for m episodes. The evaluation process is executed in the same environment as the one used in training, but with a separate task sampled randomly from the set of possible tasks. This is important for us to be sure that the learned options can generalize across multiple tasks, and not just the current one. Let $\varsigma_{\mathcal{O}} \doteq \{\sigma^2_{R_t} \mid O_t = o,\ t \in \tau\}\ \forall o \in \mathcal{O}$ hold the variances of the rewards for when option o was selected in every evaluation episode $t \in \tau$, where τ represents the maximum number of evaluations before considering creating a new option. In other words, ς_o receives a new value every time the evaluation process runs, containing the variance of the rewards accumulated when o was selected in each of the m episodes. After τ evaluations, the algorithm will use these variances to decide whether a new option is necessary, and clear ς_o afterwards for every $o \in \mathcal{O}$. Algorithm 1 represents the logic for appending a new value to ς_o.

Algorithm 1: Logic for appending a new value into ς_o a.

```
   Input: Number of evaluation episodes m > 0
1  Initialize ς_O ∈ ℝ^{τ×|O|}, R^acc ∈ ℝ^{|O|} ← {∅}
2  Initialize i ← 0
3  while i < m do
4  |  Execute evaluation episode P_i^eval
5  |  foreach (s, a, o, r, s') ∈ P_i^eval do
6  |  |  R_o^acc ← R_o^acc ∪ {r}
7  |  end
8  |  i ← i + 1
9  end
10 foreach o ∈ O do
11 |  ς_o ← ς_o ∪ {variance(R_o^acc)}
12 end
```

When working with primitive actions, the variance of the value estimate for some action a can be seen as its uncertainty, which tends to decrease the more a is selected. In the early stages of training, it is expected that the return when selecting a will have a high variance, but, as training progresses, the variance in a should decrease as the policy converges. If the uncertainty

does not decrease or takes too long to do so, it means that the agent is not learning properly. The same logic can be applied to options, where the variance in returns when selecting option o should decrease over time as the policy over options μ and o's intra-option policy π_o are adjusted, but, just as primitive actions, this is not always the case. When the environment is too complex, a higher number of options can make a noticeable difference in the quality of learned policies, as having more options means the ability to learn more specific scenarios as separate intra-option policies, thus simplifying μ. Figure 2 shows the expected behavior of an option's variance throughout training.

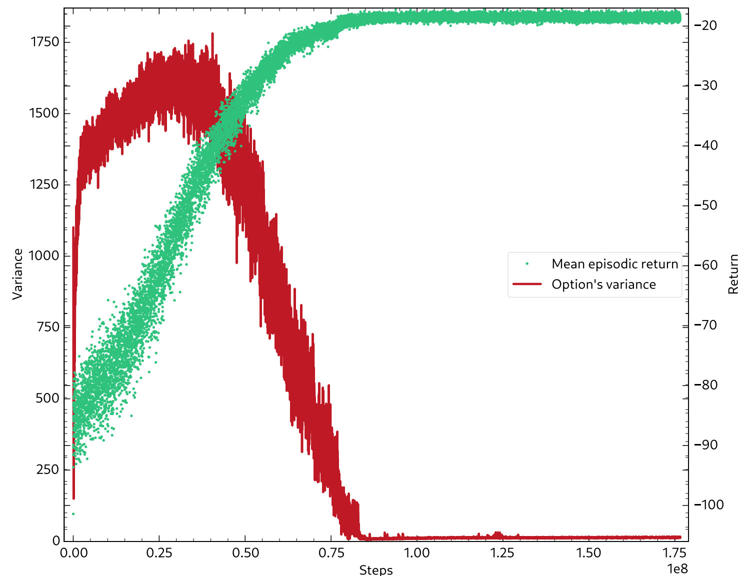

Figure 2 – Option's variance (red) compared to the agent's mean episodic return (green).

As we can see, the variance starts high, increases slightly in the early stages of training and, as the episodic returns get less disperse and approach convergence, the variance starts to quickly decline, approaching zero. This means the policies have reached—or are very close to reaching—their maximum return. The issue with working directly with the variances is that, as we can see in Figure 2, they tend to be very noisy, thus making it hard to obtain any useful performance indicative in training time. Instead, we propose using the integral of the variance, which results in an ever-increasing line denoting the area between the variance and the x axis. When the variance increases, the integral's slope gets steeper and, when it decreases, the slope also decreases. This allows us to have a much clearer indicator of the variance's performance, as shown in Figure 3.

We can determine the integral's slope in a given window by first applying a linear regression to the points inside it, which will give us a linear function in the form of $y = ax + b$. We can then derivate the result relative to x and obtain a, which represents the slope of our regression. When applying this logic to a small enough window, the loss in precision is marginal, as shown in Figure 4.

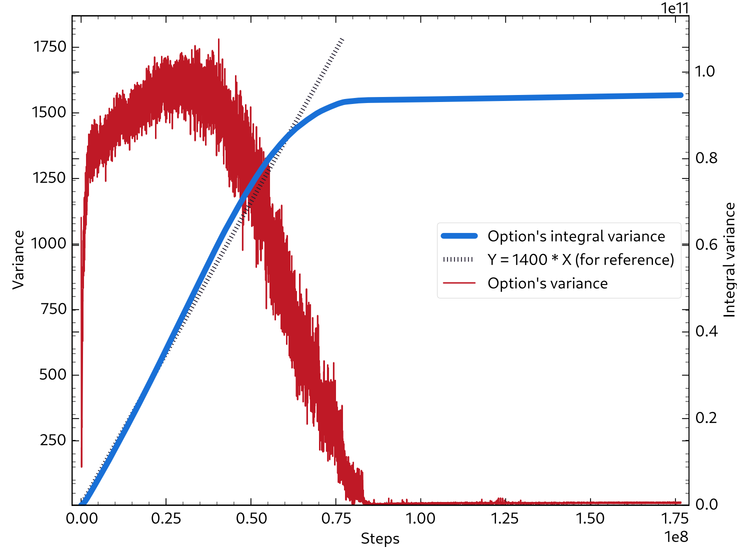

Figure 3 – Option's variance (red) and its integral (blue), compared to a linear slope (gray).

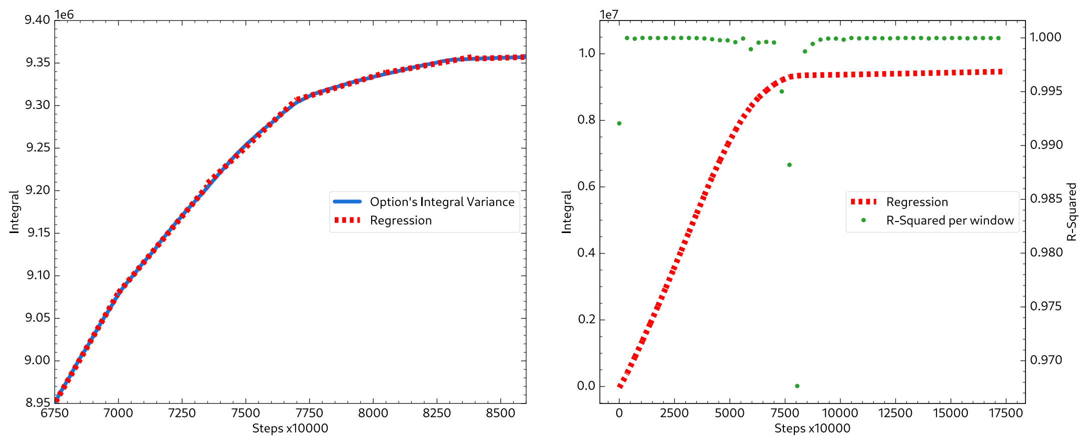

Figure 4 – Left: Option's integral variance (blue) and its linear regression (red) with a window size of 350 points. Right: Linear regression (red) and the R^2 for each window (green).

Note that we already define $\varsigma_{\mathcal{O}}$ as a list containing variances for every option $o \in \mathcal{O}$. Hence, we denote our uncertainty factor ε_o as:

$$\varepsilon_o \doteq e^{-\frac{\phi}{f}} \left[\frac{d}{dx} \; lsreg\left(\int_0^\tau \varsigma_o\right)\right] , \tag{19}$$

where $lsreg(x)$ is the linear regression over x. The discount factor $e^{-\frac{\phi}{f}}$ is used to prevent options from being created too frequently, where e is the Euler's constant, f is the number of times the algorithm has calculated the uncertainty factor but did not create a new option, and ϕ is the discount's strength. Figure 5 shows the relation between the discount value over f and different values of ϕ. A lower value in the y axis means a stronger discount, as it multiplies the rest of the equation. As we can see, increasing the value of ϕ will make the algorithm less likely to create several options in a small time period. We can adjust ϕ to give the algorithm

more time to adapt to the new option.

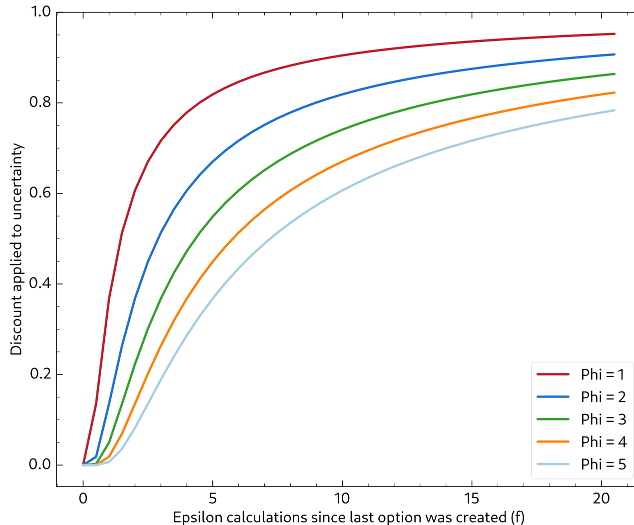

Figure 5 – Discount strength in relation to ϕ and f.

Every time $\varsigma_{\mathcal{O}}$ reaches a size of τ, our algorithm calculates ε_o for every $o \in \mathcal{O}$. Let L be the slope threshold defined by the user. If $\varepsilon_o \geq L$, it means the variance is increasing more than the desired amount and a new option is created, as described in Section 4.3. Otherwise, this procedure is repeated after τ iterations of the evaluation process.

4.3 Creating a new option that complements its predecessors

Once the algorithm has decided that creating a new option will benefit the learning process, it must initialize the new option in a way that complements the existing ones—that is, the new option must focus on where the previous ones struggle. Recall that the replay buffer B stores the total accumulated reward G_p for every episode $p \in \mathcal{P}$. By defining the average episodic return $\overline{G_{\mathcal{P}}} \doteq \frac{\sum_{p \in \mathcal{P}} G_p}{|\mathcal{P}|}$, we can determine which episodes performed above and below average, and use only the ones that performed badly to train the new option. The process of updating the new option's parameters given the replay buffer's episodes is the same as the original FPOC method, with the only difference being the value for the importance sampling ratio ρ and learning rate. While the FPOC training process uses a fixed, user provided step size α when updating Q, W^{π}, W^{β}, and W^{i}, we propose using a variable step size that gives higher importance for updates in poorly performing episodes. We call the new step size parameter *quality step discount*, denoted by Equation 20:

$$d \doteq \Upsilon \cdot \frac{1}{sigmoid\left(\frac{\overline{G_{\mathcal{P}}} - G_p}{G_{\mathcal{P}}}\right)} \quad , \tag{20}$$

where $\Upsilon \rightarrow \mathbb{R}$ is a discount factor, typically in the scale of 10^{-3}. The quality step discount d replaces the fixed learning rate only in option creation, where the algorithm applies the replay

buffer history to the new option, leaving the other options untouched. Once the replay buffer is exhausted, the algorithm returns to the regular training process, clearing the elements in $\varsigma_\mathcal{O}$ and setting f to 0. Algorithm 2 describes the entire option creation process.

Algorithm 2: Dynamic option creation algorithm.

Input: Replay buffer size z, option creation threshold L, option adjust rate n, option evaluation episodes m, quality discount factor Υ, maximum training steps t_{max}, option consider rate τ, option consider discount strength ϕ

1 Initialize $t,\ n_{elapsed},\ p,\ tid,\ f \leftarrow 0$
2 Initialize $\perp \leftarrow 1$
3 Initialize $\varsigma_\mathcal{O} \in \mathbb{R}^{\tau \times |\mathcal{O}|}, B \in \mathbb{R}^{z \times 2} \leftarrow \{\varnothing\}$
4 **while** $t < t_{max}$ **do**
5   **if** $\perp = 1$ **then**
6         $tid \leftarrow$ Task ID sampled from set of possible tasks
7         $n_{elapsed} \leftarrow n_{elapsed} + 1$
8         $p \leftarrow p + 1$
9   **end**
10   **if** $n_{elapsed} = n$ **then**
11   Run evaluation for m episodes, store variances in $\varsigma_\mathcal{O}$
12   **if** $|\varsigma_\mathcal{O}| = \tau$ **then**
13   **foreach** $o \in \mathcal{O}$ **do**
14                 $\varepsilon_o \leftarrow e^{-\frac{\phi}{f}} \left[\frac{d}{dx}\ lsreg\left(\int_0^\tau \varsigma_o\right)\right]$
15   **if** $\varepsilon_o > L$ **then**
16   Initialize new column in Q, W^π, W^β, W^i with 0
17   **for** $p = 1, 2, \ldots, z$ **do**
18   **foreach** $E_p = \langle S_p, A_p, r_p, \bar{\rho}_p, \perp_p, S'_p, tid_p\rangle \in B_{p,0}$ **do**
19   **if** $G_p < \overline{G_\mathcal{P}}$ **then**
20                                 $d \leftarrow \Upsilon \cdot \frac{1}{sigmoid\left(\frac{\overline{G_\mathcal{P}} - G_p}{\overline{G_\mathcal{P}}}\right)}$
21   Run option train step w.r.t. $d,\ E_p$
22   **end**
23   **end**
24   **end**
25                     $f \leftarrow 0$
26   Jump to line 32
27   **else**
28                     $f \leftarrow f + 1$
29   **end**
30   **end**
31   **end**
32         $n_{elapsed} \leftarrow 0$
33   **end**
34     $S_t, A_t, r_t, \bar{\rho}_t, \perp, S'_t \leftarrow$ Run regular training step for t
35     $B_p \leftarrow \{B_{p,0} \cup \{\langle S_t, A_t, r_t, \bar{\rho}_t, \perp, S'_t, tid\rangle\},\ B_{p,1} + r_t\}$
36     $t \leftarrow t + 1$
37 **end**

---

5 RESULTS

In this section, we test the method proposed in Section 4 in different scenarios, comparing its learning progress to the one from the FPOC algorithm, used as baseline. Our goal with these experiments is to confirm that our approach is able to efficiently create new options whenever necessary, while yielding similar or higher accumulated reward than if the number of options was set manually. We also look at the intra-option policies learned by the dynamically created options, and compare their behavior with the policies learned by fixed options.

5.1 Testing methodology

For testing the hypotheses mentioned above, we compare our dynamic option creation method with the base FPOC algorithm in the four rooms environment. Although simple, we chose this environment as it is the most common for comparing tabular option discovery methods. The four rooms environment, as the name implies, is composed of four interconnected rooms in a square grid, where each room has a 1×1 gap connecting it to its neighbor. When an episode starts, the agent and a goal are randomly placed in the environment. When the agent reaches the goal state, the episode terminates, and a new episode is randomly generated. The agent has four primitive actions available: up, $down$, $left$, $right$. For every action taken that does not terminate the episode, the agent receives a reward of -1. The agent receives a reward of 0 when it reaches the goal state. Figure 6 shows a visual representation of the environment used in testing.

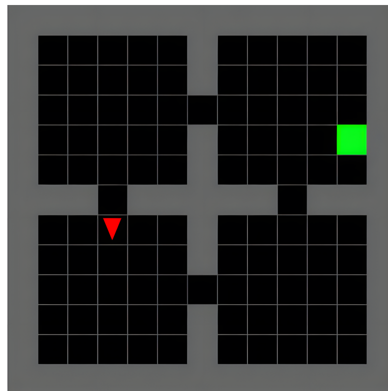

Figure 6 – Visual representation of the four rooms environment. The agent is represented by the red arrow, and the goal state by the green square.

All experiments consisted of 150,000,000 training steps, which proved to be more than enough for all methods to reach their maximum return. After the training procedure was completed, we calculated the moving average of the mean accumulated reward for every 10,000 episodes, using a window size of 100 data points. We use the moving average as a way to reduce the variability in mean episodic returns, thus helping better visualize the learning behavior.

We consider a method as superior in terms of maximum accumulated reward if it consistently averages higher reward after reaching its maximum return, where the results tend to stagnate until the end of the training procedure.

In addition to the traditional four rooms environment, we have also compared both algorithms on a modified version of the same problem, where the agent receives a reward of -10 every time it traverses a corridor. As we show in Section 5.2, this slight modification in the environment has a noticeable impact in learning performance, given that the policy now also needs to consider the penalty of going through doors. Lastly, we also experimented with adding a difference in room sizes on top of the modified environment. In this third experiment, the vertical wall that divides the bottom rooms has been shifted to the right by one block, which adds extra complexity by having the rooms not being symmetrical.

5.2 Numerical results

Figure 7 shows the mean episodic return for each method in the traditional four rooms environment. As we can see, our algorithm performed very similar to FPOC, as it did not create any new options. Given the simple nature of the problem, both methods are able to completely learn it with just one option, combined with the four primitive options from the agent's action set. Changing the number of options in the FPOC experiments barely had an effect on the final accumulated reward, where the difference in reward among results can be mostly explained by the algorithm's inherent stochasticity.

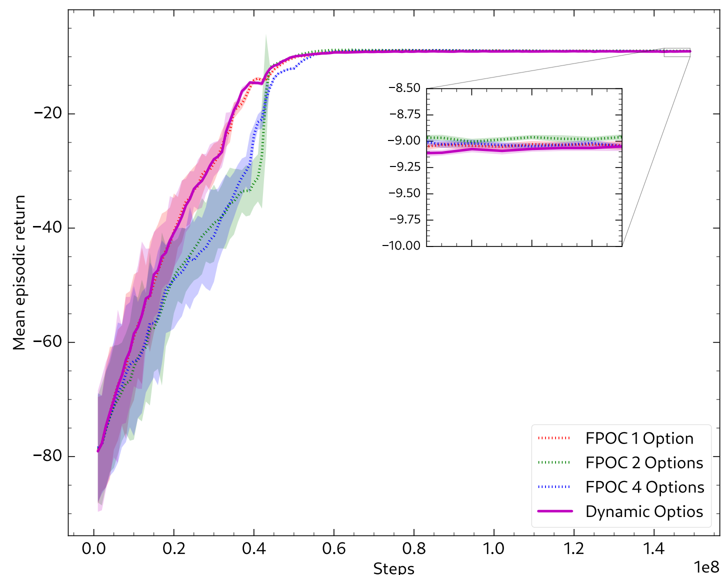

Figure 7 – Dynamic option creation vs. FPOC in four rooms environment.

As mentioned in Section 5.1, we also modified the four rooms environment to give a higher penalty for when the agent goes through hallways in an attempt to increase the problem complexity. Figure 8 shows the mean episodic returns for FPOC and our method in this version of

the environment. This time, the algorithm triggered the creation of a new option, which caused a sudden drop in mean episodic return values. However, the algorithm was able to quickly recover, converging around 25,000,000 steps after FPOC. Furthermore, once the dynamic options algorithm reached its peak, its mean returns were slightly higher than the ones found in the fixed options algorithms.

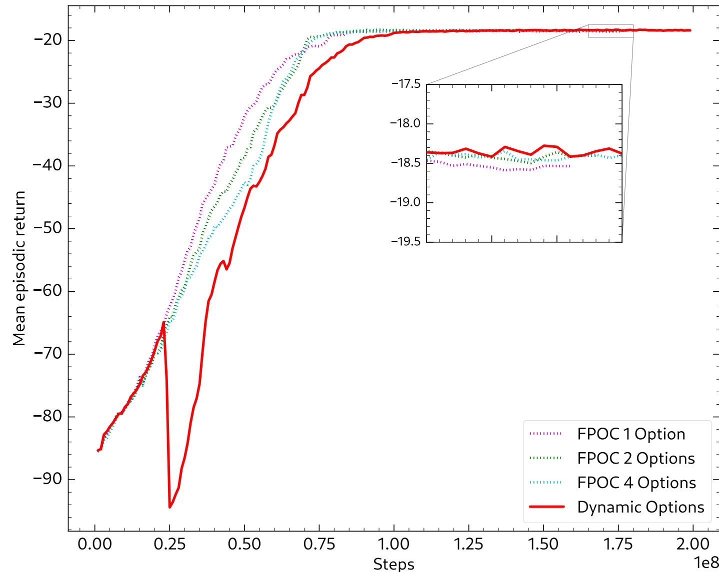

Figure 8 – Dynamic option creation vs. FPOC in modified four rooms environment.

The third and last experiment builds upon the previous one, adding extra complexity to the learning task by changing the size of the two bottom rooms. As we can see in Figure 9, this modification had the most impact on results, where we can now see that having more options makes a noticeable difference in mean episodic returns. Just like in the previous test, our algorithm created a single option, but this time did not achieve the highest reward, staying between the results obtained by FPOC with 1 and 2 options.

In Section 5.3, we look into the policies learned by this last experiment, as well as its initiation and termination functions.

5.3 Learned policies

Given the sub-optimal performance in the third experiment, we use its results as the base for investigating how our method derived new policies from experience replay, shown in Figure 10. Given that the new option (presented on the right) was created after 30,000,000 steps and initialized with experience from the existing option, its behavior, although different, resembles the parent option in some aspects. For instance, both policies present a circular pattern, which is especially evident in the first option's termination function β. Another similarity can be found in the walls next to the bottom right room, where neither option fully explored the area, which could explain the lower reward in this experiment. There are, on the other hand, several cases

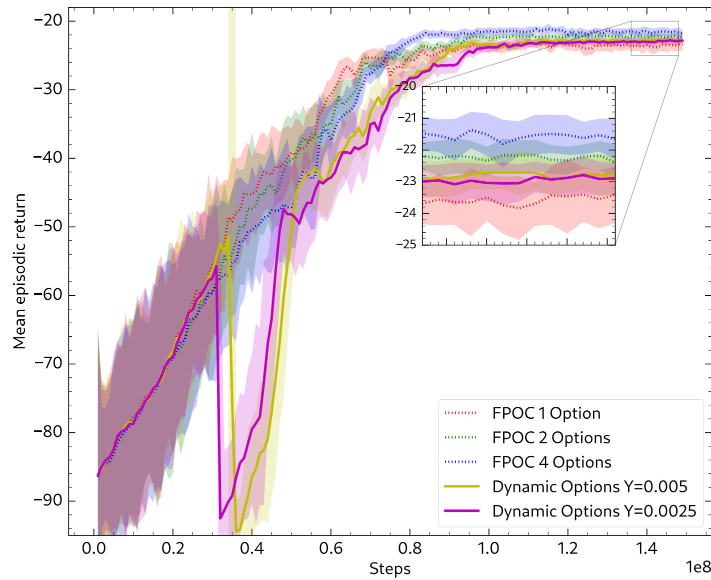

Figure 9 – Dynamic option creation vs. FPOC in modified four rooms environment, where Y represents the Υ parameter in Equation 20.

where both options complement each other. For example, while the termination function in the first option tends to prefer ending execution closer to hallways, the second option ends in the middle of each room, as well as some walls. The policy over options μ also presents a mutually exclusive behavior, where the second option tends to be selected primarily at the top left room, as opposed to the first option, whose preference is at the top right.

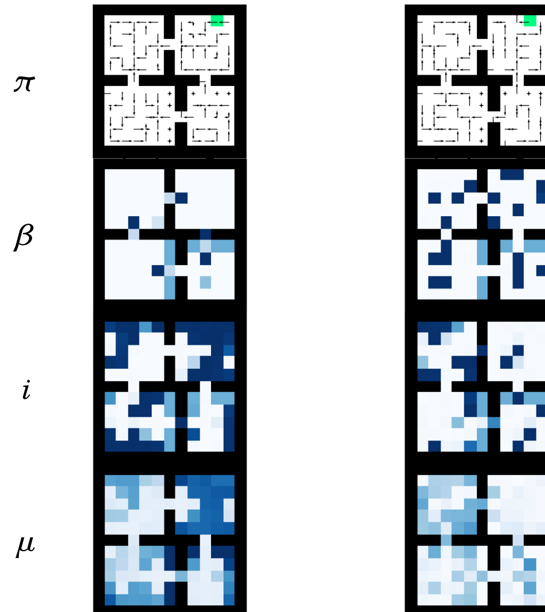

Figure 10 – From top to bottom: intra-option policies, termination functions, interest functions, and policy over options for each adjustable option.

Some of the undesired behavior present in the learned options can be attributed to our approach to experience replay. Although our approach towards selecting which episodes to use

when initializing options, as well as the quality step discount in Equation 20, helped achieve better results, more robust experience replay methods can be found in literature. Although outside of the scope of this work, future experiments should focus on implementing more robust experience replay techniques such as the ones in (FANG et al., 2019; ANDRYCHOWICZ et al., 2017; SCHAUL et al., 2015).

6 CONCLUSION

In this work, we proposed a novel method for dynamically creating options in training time in option-critic algorithms. Our approach uses the variance of an option's returns as an indication of uncertainty, which tends to decrease as the intra-option policy converges to its optimality. When compared to the Fast-Planning Option-Critic in (WAN; SUTTON, 2022) used as the base for our implementation, our approach tends to learn options only as necessary without a significant increase in training steps needed for convergence. We consider learning options dynamically as a vastly superior approach, given that complex problems heavily benefit from more options, but estimating the best number for an environment is not possible most of the time. Dynamic option creation has the potential to reduce re-training efforts by guessing the correct number of options in the first try, thus making the option-critic architecture more robust.

Despite the benefits mentioned above, our method has limitations that are not handled in the scope of the current work. The most noteworthy shortcoming is the reliance on a human-provided uncertainty threshold L, which can severely hinder learning if set incorrectly, requiring multiple runs to be properly adjusted. This limitation contradicts with the added benefit mentioned at the end of the previous paragraph, hence future work should focus on determining L based on parameters gathered from the training process or the environment itself. Another limitation future work should consider tackling is the high variability between runs with the same parameters. This is an expected consequence of using the variance of episodic returns when considering a new option, given that the variance in a small window of episodes can vary enough between experiments to go over the threshold. Finding ways of reducing the impact of the variance's variability in ε_o could also be the scope of future work. As explained in Section 5.3, future work should also consider using more robust approaches to experience replay to better initialize options. Lastly, the present work did not test the proposed method in more complex environments, where we believe our approach could bring the greatest benefit as these types of scenarios typically require more options to perform well. This limitation comes from the tabular nature of the base algorithm, which often limits its application to small gridworlds. Future work could adapt our dynamic option creation method to extend option-critic algorithms based on function approximation, or adapt FPOC to support this approach.

REFERENCES

ANDRYCHOWICZ, M.; WOLSKI, F.; RAY, A.; SCHNEIDER, J.; FONG, R.; WELINDER, P.; MCGREW, B.; TOBIN, J.; PIETER ABBEEL, O.; ZAREMBA, W. Hindsight Experience Replay. In: ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS, 2017. **Annals...** Curran Associates: Inc., 2017. v. 30.

BACON, P.-L.; HARB, J.; PRECUP, D. The Option-Critic Architecture. **Proceedings of the AAAI Conference on Artificial Intelligence**, [S.l.], v. 31, n. 1, Feb 2017.

BAIRD III, L. C. **Advantage updating**. [S.l.]: WRIGHT LAB WRIGHT-PATTERSON AFB OH, 1993.

BARRETO, A.; DABNEY, W.; MUNOS, R.; HUNT, J. J.; SCHAUL, T.; HASSELT, H. P. van; SILVER, D. Successor features for transfer in reinforcement learning. In: ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS, 2017. **Annals...** Curran Associates: Inc., 2017. v. 30.

BELLEMARE, M. G.; NADDAF, Y.; VENESS, J.; BOWLING, M. The arcade learning environment: an evaluation platform for general agents. **Journal of Artificial Intelligence Research**, [S.l.], v. 47, p. 253–279, 2013.

BELLMAN, R. Dynamic programming. **Princeton University Press**, [S.l.], v. 89, p. 392, 1957.

CHUNDURU, R.; PRECUP, D. Attention Option-Critic. **arXiv preprint arXiv:2201.02628**, [S.l.], Jan 2022.

CROONENBORGHS, T.; DRIESSENS, K.; BRUYNOOGHE, M. Learning Relational Options for Inductive Transfer in Relational Reinforcement Learning. In: _____. **Inductive Logic Programming**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 88–97. (Lecture Notes in Computer Science, v. 4894).

DIETTERICH, T. G.; LATHROP, R. H.; LOZANO-PÉREZ, T. Solving the multiple instance problem with axis-parallel rectangles. **Artificial intelligence**, [S.l.], v. 89, n. 1-2, p. 31–71, 1997.

FANG, M.; ZHOU, T.; DU, Y.; HAN, L.; ZHANG, Z. Curriculum-guided Hindsight Experience Replay. In: ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS, 2019. **Annals...** Curran Associates: Inc., 2019. v. 32.

HAN, D.; TSCHIATSCHEK, S. Option Transfer and SMDP Abstraction with Successor Features. In: THIRTY-FIRST INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE, 2022, Vienna, Austria. **Proceedings...** International Joint Conferences on Artificial Intelligence Organization, 2022. p. 3036–3042.

HARB, J.; BACON, P.-L.; KLISSAROV, M.; PRECUP, D. When Waiting Is Not an Option: learning options with a deliberation cost. **Proceedings of the AAAI Conference on Artificial Intelligence**, [S.l.], v. 32, n. 1, Apr 2018.

JAIN, A.; KHETARPAL, K.; PRECUP, D. Safe Option-Critic: learning safety in the option-critic architecture. **The Knowledge Engineering Review**, [S.l.], v. 36, p. e4, 2021.

KLISSAROV, M.; MACHADO, M. C. Deep Laplacian-based Options for Temporally-Extended Exploration. **arXiv preprint arXiv:2301.11181**, [S.l.], Jan 2023.

KONDA, V. R.; TSITSIKLIS, J. N. Actor-Critic Algorithms. In: ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS, 1999. **Annals...** MIT Press, 1999. v. 12.

KONIDARIS, G.; BARTO, A. Building Portable Options: skill transfer in reinforcement learning. In: IJCAI, 2007. **Annals...** [S.l.: s.n.], 2007. v. 7, p. 895–900.

MACHADO, M. C.; BELLEMARE, M. G.; BOWLING, M. A Laplacian framework for option discovery in reinforcement learning. In: OF THE 34TH INTERNATIONAL CONFERENCE ON MACHINE LEARNING, 2017. **Proceedings...** PMLR, 2017. p. 2295–2304. (Proceedings of machine learning research, v. 70).

MACHADO, M. C.; ROSENBAUM, C.; GUO, X.; LIU, M.; TESAURO, G.; CAMPBELL, M. Eigenoption Discovery through the Deep Successor Representation. **arXiv preprint arXiv:1710.11089**, [S.l.], Feb 2018.

MCGOVERN, A.; BARTO, A. G. Automatic Discovery of Subgoals in Reinforcement Learning using Diverse Density. **Computer Science Department Faculty Publication Series. 8.**, [S.l.], 2001.

MENACHE, I.; MANNOR, S.; SHIMKIN, N. Q-Cut—Dynamic Discovery of Sub-goals in Reinforcement Learning. In: MACHINE LEARNING: ECML 2002, 2002, Berlin, Heidelberg. **Annals...** Springer Berlin Heidelberg, 2002. p. 295–306. (Lecture Notes in Computer Science, v. 2430).

MNIH, V.; BADIA, A. P.; MIRZA, M.; GRAVES, A.; LILLICRAP, T.; HARLEY, T.; SILVER, D.; KAVUKCUOGLU, K. Asynchronous methods for deep reinforcement learning. In: INTERNATIONAL CONFERENCE ON MACHINE LEARNING, 2016. **Annals...** [S.l.: s.n.], 2016. p. 1928–1937.

SCHAUL, T.; QUAN, J.; ANTONOGLOU, I.; SILVER, D. Prioritized Experience Replay. **arXiv**, [S.l.], 2015.

ŞIMŞEK, Ö.; BARTO, A. Skill characterization based on betweenness. In: ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS, 2008. **Annals...** Curran Associates: Inc., 2008. v. 21.

STOLLE, M.; PRECUP, D. Learning Options in Reinforcement Learning. In: _____. **Abstraction, Reformulation, and Approximation**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002. p. 212–223. (Lecture Notes in Computer Science, v. 2371).

SUTTON, R. S.; BARTO, A. **Reinforcement learning**: an introduction. Second edition. ed. Cambridge, Massachusetts London, England: The MIT Press, 2020. (Adaptive computation and machine learning).

SUTTON, R. S.; MCALLESTER, D. A.; SINGH, S. P.; MANSOUR, Y. Policy Gradient Methods for Reinforcement Learning with Function Approximation. **Advances in neural information processing systems**, [S.l.], v. 12, 1999.

SUTTON, R. S.; PRECUP, D.; SINGH, S. Between MDPs and semi-MDPs: a framework for temporal abstraction in reinforcement learning. **Artificial Intelligence**, [S.l.], v. 112, n. 1–2, p. 181–211, Aug 1999.

WAN, Y.; SUTTON, R. S. Toward Discovering Options that Achieve Faster Planning. **arXiv**, [S.l.], n. arXiv:2205.12515, Sep 2022. arXiv:2205.12515 [cs].

WATKINS, C. J.; DAYAN, P. Q-learning. **Machine learning**, [S.l.], v. 8, p. 279–292, 1992.