Sub-Neural Policies: Option Discovery via Neural Decomposition

by

Mahdi Alikhasi

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

Master of Science

Department of Computing Science
University of Alberta

© Mahdi Alikhasi, 2024

Abstract

In reinforcement learning, agents solve problems through interactions with the environment. However, when faced with intricate environmental dynamics, learning can become challenging, resulting in sub-optimal policies. A potential remedy to this situation lies in the transfer of knowledge from previously solved tasks to enhance the efficiency of the agent. In this dissertation, we investigate this approach, focusing on the decomposition of neural network policies for Markov Decision Processes into reusable sub-policies, which can be "helpful" for unforeseen tasks. We consider neural networks with piecewise linear activation functions, since they can be transformed into oblique decision trees. Each sub-tree within an oblique decision tree corresponds to a sub-policy associated with the primary task. We hypothesize that some of these sub-policies can be helpful in downstream tasks. Given that the number of these sub-policies grows exponentially with the neural network's size, we select a subset of such sub-policies while minimizing the Levin Loss. We transform the selected sub-policies into temporally extended actions, or options. To validate the algorithm's ability to discover helpful options, we present empirical findings on two challenging grid-world domains, each characterized by distinct dynamics. The experimental results show that options can "occur naturally" within neural network encoding policies. Our results suggest that the process of decomposing neural network serves as a promising avenue for option discovery.

Preface

This thesis represents a collaborative work between the author, Mahdi Alikhasi, and his supervisor, Dr. Levi Lelis. The author played a role in conducting experiments, analyzing results, and crafting insightful conclusions. Throughout this research journey, the supervisor's guidance and feedback were instrumental in shaping the direction of the work, refining methodologies, and enhancing both written content and presentations. We acknowledge the contribution of Spyros Orfanos, whose codebase formed the basis for implementing neural network decomposition techniques in this thesis. We also acknowledge that this thesis is based on the paper "Unveiling Options with Neural Network Decomposition", submitted to ICLR 2024, which is currently under review.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Levi Lelis, whose unwavering guidance and support have been instrumental throughout the journey of this research. His invaluable insights and mentorship have played a pivotal role in shaping this thesis and my growth as a researcher. I also thank Dr. Marlos Machado and Dr. Matthew Taylor, both esteemed members of my thesis committee. Their feedback and comments enhanced the quality of this work.

I am profoundly thankful to my parents, whose presence and support have been a constant source of strength, especially during challenging times. To my cherished friends, scattered across the globe from Germany to the United States, your enduring friendship and encouragement have been a constant source of motivation. Your absence was felt, but your presence in spirit was deeply appreciated. A special acknowledgment goes to my girlfriend, whose unwavering emotional support sustained me during the most trying moments of this endeavor.

To all these remarkable individuals in my life, I extend my heartfelt thanks. Your love, support, and belief in me have been the cornerstone upon which this work has been built. Without you, this achievement would not have been possible.

Table of Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Problem Definition 1
1.3 Contribution 3
1.4 Thesis Statement 4
1.5 Outline 4

2 Literature Review **6**
2.1 Options 6
2.2 Transfer Learning 9
2.3 Compositional Methods 10

3 Learning Options with Neural Network Decomposition **12**
3.1 Decomposing Neural Policies into Sub-Policies 14
3.2 Synthesizing and Selecting Options 17
3.2.1 Greedy Approximation to Select Options 19

4 Experiments **23**
4.1 Baselines 23
4.2 Problem Domains 25
4.2.1 MiniGrid 26
4.2.2 ComboGrid 29
4.3 Empirical Results 32
4.4 Qualitative Results 37

5 Conclusions & Future Work **40**
5.1 Conclusions 40
5.2 Future Work 41

Bibliography **43**

Appendix A: Experiment Details **48**
A.1 Plots 48
A.2 Architecture and Parameter Search 48

Appendix B: Parameter Search **51**
B.1 MiniGrid 51
B.2 ComboGrid 52

List of Tables

B.1 Four Rooms 1 best parameters - PPO 52
B.2 Four Rooms 1 best parameters - DQN 52
B.3 Four Rooms 2 best parameters - PPO 53
B.4 Four Rooms 2 best parameters - DQN 53
B.5 Four Rooms 3 best parameters - PPO 54
B.6 Four Rooms 3 best parameters - DQN 54
B.7 ComboGrid 3x3 best parameters - PPO 55
B.8 ComboGrid 3x3 best parameters - DQN 55
B.9 ComboGrid 4x4 best parameters - PPO 56
B.10 ComboGrid 4x4 best parameters - DQN 56
B.11 ComboGrid 5x5 best parameters - PPO 57
B.12 ComboGrid 5x5 best parameters - DQN 57
B.13 ComboGrid 6x6 best parameters - PPO 58
B.14 ComboGrid 6x6 best parameters - DQN 58

List of Figures

3.1 A neural network with two inputs, two ReLU neurons in the hidden layer, and one Sigmoid neuron in the output neuron is shown on the left. All bias terms of the model are 1; for simplicity, we omit bias values. The neural tree representing the same function encoded in the network is shown on the right. The root of the tree represents the neuron A_1^2, the nodes in the second layer represent the neuron A_2^2, and the leaf nodes represent the output neuron A_1^3, where $\sigma(\cdot)$ is the Sigmoid function. The colors of the neurons match the colors of the nodes in the tree that represent them. 14

4.1 MiniGrid Simple Cross Tasks in set $\mathcal{P}$ 26

4.2 MiniGrid Four Rooms Tasks 26

4.3 ComboGrid tasks in $\mathcal{P}$. In this depiction, the agent is highlighted in red, while the goals are denoted in green, and the walls are represented in grey. 29

4.4 ComboGrid $\mathcal{P}'$ tasks 29

4.5 Performance of different methods on MiniGrid Domain 33

4.6 Performance of different methods on ComboGrid Domain, using PPO algorithm 34

4.7 Performance of different methods on ComboGrid Domain, using DQN algorithm 35

4.8 This figure displays the heatmap of visited cells in the Four Rooms 1 environment for both the Dec-Options and Vanilla-RL methods. The first row depicts the Dec-Options method's heatmap, while the second row shows the Vanilla-RL method. The first, second, and third columns show the heatmap after 1, 3200, and 12800 steps, respectively. 39

Abbreviations

CNN Convolutional Neural Networks.

DIF Diffusive Information Flow.

DQN Deep Q-Network.

EWC Elastic Weight Consolidation.

MDP Markov Decision Process.

ML Machine Learning.

PNN Progressive Neural Networks.

PPO Proximal Policy Optimization.

ReLU Rectified Linear Units.

RL Reinforcement Learning.

SMDP Semi-Markov Decision Process.

VAE Variational AutoEncoder.

Chapter 1

Introduction

1.1 Motivation

Reinforcement learning (RL) is a class of problems where an agent learns through interactions with the environment. In RL, The agent attempts to learn policies that maximize the expected cumulative reward [1]. One key feature of intelligent systems is their ability to transfer learned policies from one task to another, a concept explored in previous research [2, 3]. Effective transfer learning depends on the agent's ability to retain knowledge from one task and selectively apply it to unforeseen challenges [4]. The transferring of knowledge can be especially promising when the agent is facing a more complex task, motivating our investigation into the domain of knowledge transfer in RL.

To accomplish this objective, our work uses the concept of temporally extended actions, known as "options" [5, 6].

1.2 Problem Definition

In this work, we use the framework of Markov Decision Processes (MDPs) (S, A, p, r, γ) [7, 8] to model the environment. In this definition, S represents the set of possible states of the environment. A is the set of actions that the agent can take. Actions are the choices available to the agent in any given state. Furthermore, p is the transition function, denoted as $p(s_{t+1}|s_t, a_t)$. p models how the environment changes over time.

Given the current state s_t and the action taken a_t, it specifies the probability of transitioning to the next state s_{t+1}. The reward function, represented as r, determines the immediate reward that the agent receives when transitioning from one state to another. It defines the goal or objective of the agent, as it assigns numerical rewards to different state transitions. γ in $[0, 1]$ is the discount factor.

In a reinforcement learning problem, one attempts to find a policy π that maximizes the expected sum of discounted rewards, also known as the return. The return is defined as: $\mathbb{E}_{\pi,p}[\sum_{k=0}^{\infty} \gamma^k R_{k+t+1} | s_t]$. A policy is a probability function $\pi = Pr(a_t|s_t)$ that receives a state s and an action a and returns the probability in which a is taken in s. In this work, we use actor-critic algorithms, which aim to learn both a policy and a value function. Value function $\mathcal{V}^{\pi}(s)$ is the expected return if the agent follows the policy π and starts at state s. The policy these algorithms learn, denoted π_θ, is parameterized and encoded using neural networks.

We define a set of tasks $\mathcal{P} = \{\rho_1, \rho_2, \cdots, \rho_n\}$, which represent different MDPs for which the agent learns policies to maximize the return. After learning policies for the tasks in $\mathcal{P}$, we evaluate the agent's performance while learning policies for a different set of tasks $\mathcal{P}'$, where $\mathcal{P}' \cap \mathcal{P} = \emptyset$. In this setting, the focus is on transferring knowledge from the tasks in $\mathcal{P}$ to the tasks in $\mathcal{P}'$ using the concept of options [9].

Our work comprises two primary components. The first component involves the extraction of sub-policies from a trained neural network, while the second component focuses on transferring these sub-policies as options across different tasks. Initially, we employ deep neural networks [10] for policy learning, with a specific emphasis on the use of two-part piecewise-linear activation functions, such as Rectified Linear Units (ReLUs) [11]. The adoption of ReLU activation functions has become a trend in numerous RL algorithms in recent years [12].

For a given input value, each neuron of the network is either in one part of the linear function or in the other. For the ReLU functions, we say that each neuron is active or inactive, as it produces a zero value or a linear combination of its input

values [13–15]. The activation pattern of such a network specifies which neurons are active and which neurons are inactive [16]. For a given activation pattern, we can re-write the function the network represents as a linear function of its input. This is because each neuron represents a linear function, and the combination of linear functions is also linear. This type of network can be mapped to an oblique decision tree [17], a type of tree where each node encodes a linear function of the input. In the mapping, each node in the tree represents a neuron in the network, and each path from root to leaf in the tree represents an activation pattern of the network [15]. This mapping enables us to extract sub-policies via the decomposition of their equivalent oblique decision tree. Each sub-tree within the primary tree corresponds to a function of the input, thereby representing a sub-policy within the neural network represents.

Subsequent to the decomposition of the neural network into its sub-policies, we employ the options framework, encapsulating each sub-policy within a fixed-duration while loop to facilitate the creation of options. Notably, the number of sub-policies is exponentially proportional to the number of neurons within the deep neural network. Hence, we introduce a selection algorithm founded on the minimization of the Levin Loss [18] applied to a uniform policy, utilizing the acquired options. This uniform policy represents the random policy of the agent at the early stages. This minimization of the Levin loss enhances the probability of encountering promising states early in the training process, contingent on task similarities. Additionally, to validate our hypothesis, we focus on policies with small neural networks. The choice of small neural networks allows for the evaluation of all sub-policies, without being conflicted with the problem of exploding number of possible sub-policies.

1.3 Contribution

Our work offers several contributions.

- We demonstrate the capacity to extract options from small neural networks

using ReLU activation functions, even when such extraction was not the primary objective during training. This enables the extraction of options from "legacy" policies, under the assumptions outlined in this work.

- Our method operates without the need for human supervision or prior domain knowledge before training, autonomously learning all option-related parameters from the data, including the number of options, and the start and the termination of options. Furthermore, our approach uses simple feedforward neural networks, making it simple and reusable in different scenarios.

- Our method introduces a novel approach for extracting sub-policies through neural network decomposition, which holds potential applicability in RL, continual learning, and program synthesis. However, it is important to note that the full extent of these possibilities is beyond the scope of this investigation and warrants further exploration.

1.4 Thesis Statement

This thesis introduces a novel method for the extraction of sub-policies from feedforward neural networks, defining them as "Options". The primary objective of this research was to verify whether **neural networks encoding policies encode sub-policies that can be turned into "helpful" temporally extended actions**.

1.5 Outline

We will discuss the following chapters in the rest of this dissertation. Chapter 2 provides a review of reinforcement learning and knowledge transfer. It also provides state-of-the-art work on options. It sets the stage by introducing fundamental concepts, highlighting their relevance, and identifying research gaps. Chapter 3 delves into the core of our research. We describe the methodology, including the extraction of sub-policies using neural network decomposition, the options framework, and the

Levin Loss minimization process. Our experimental setup is detailed to prepare for empirical evaluation. Chapter 4 presents the experimental results. We explore specific grid-world domains, focusing on options discovery, and comparing them with related baselines. Chapter 5 concludes our study, summarizing key findings and contributions. We discuss implications, acknowledge limitations, and suggest future research directions. This chapter encapsulates the significance of our work and the potential it holds for the future of RL.

Chapter 2

Literature Review

2.1 Options

A particularly promising direction in the field of RL involves the application of skills or options. This approach has been a central focus of many reinforcement learning researchers in the past decade. The concept of options introduces a hierarchical structure, empowering agents to break down tasks into sub-tasks. Option addresses the challenge of handling multilevel temporal abstractions over actions, as elucidated by Sutton *et al.* [5]. In their work, they formulate the option framework in RL using the Markov decision process (MDP) [7, 8] and Semi-Markov decision processes (SMDPs). However, it should be noted that this framework relies on human expertise for option design. This limitation has spurred interest in developing methods that do not require human intervention. This literature review explores recent advancements in option discovery and their potential implications for RL.

Frans *et al.* [19] introduce a hierarchical framework [20] for option learning. Their structure consists of two levels of neural networks, a high-level one that does the general decision-making, and lower-level neural networks that interact directly with the environment for a fixed duration of c steps. These low-level neural networks, serving as options, can be then transferred across tasks. However, a limitation lies in the manual determination of the number and duration of the options, which could require domain-specific knowledge. Tessler *et al.* [21] adopt a similar approach for

transferring options across tasks. Initially, they train policies on simpler tasks, encapsulating these policies as reusable "skills". These skills are in fact wrapped policies with fixed duration loops, and are made available to agents in a call-and-return manner during new tasks along with primitive actions. This work motivated us to have a similar baseline in our experiments. In Chapter 4 we show that wrapping previously trained policies is not sufficient, since these policies can be task-specific and not suitable for unseen future tasks. Thus, by neural network decomposition, we aim to find sub-policies that are generalizable. Andreas *et al.* [22] introduce a modular learning method that leverages task sketches to capture high-level task structure and relationships. However, this approach heavily relies on human supervision for task sketch generation.

Bacon *et al.* [23] propose the "option-critic architecture". In their work, instead of fixed-duration options, they jointly train options and termination policies using gradient descent optimization. Nevertheless, the number of options still requires prior specification, and its impact on training efficiency is substantial. Despite this limitation, the option-critic architecture showcases promising results, serving as a suitable baseline for our comparative analysis. This comparison demonstrates our ability to leverage knowledge derived from previous tasks, compared to learning options from the task directly.

Achiam *et al.* [24] proposed an approach to option discovery using variational autoencoders (VAEs) [25]. By leveraging insights from autoencoders and ML, they encode policy trajectories into a latent space, subsequently decoding them to regenerate trajectories. Latent vectors corresponding to favorable trajectories are identified as options. This method offers a data-driven alternative to manual option design.

Options have proven to be an effective means of enhancing exploration, allowing agents to traverse a broader state space and achieve superior results. Machado *et al.* [26] and Machado *et al.* [27] explore representation learning methods for option discovery. In the former work, they employ successor state representations [28] to obtain

diffusive information flow (DIF) for option discovery. Successor state representations are the next state representations determined by the environment's dynamics and the agent's policy. In the latter work, the authors employ successor states directly to discover options. Since the successor states method uses the similarity between the next states to generalize between states, it seems a natural way to use these representations for learning options. These studies reveal that the options obtained through these algorithms lead to a reduction in the expected number of steps required to explore the state space compared to uniform random exploration. Moreover, Jinnai *et al.* [29] propose the concept of deep covering options, aiming to directly reduce the expected cover time of the state space, the expected number of steps needed to cover the state space. Their method establishes an upper bound for this cover time and demonstrates enhanced sample efficiency in experiments. Klissarov and Machado [30] is another work that focuses on learning the options that help the task through exploration. They aimed to train the options by focusing on the graph flow of the environment's dynamic and approximating the eigenfunctions related to the environment's dynamic. We selected their approach, namely DCEO, as another baseline for comparison.

Dabney *et al.* [31] is another work focusing on enhancing the exploration using temporal abstractions. In their work, they introduce a temporal abstraction of the ε-greedy algorithm. While the ε-greedy algorithm involves random exploration with a probability of ε, this method takes exploration a step further. Rather than exploring the environment one step at a time, it advocates for repeating an action for n steps, a form of temporal abstraction for exploration. The significance of this approach lies in its ability to facilitate more effective exploration of states, even those situated at a considerable distance from the greedy policy. The ε-greedy algorithm contributes to the ongoing exploration-efficiency discourse by extending the temporal aspect of exploration. It is also selected as a baseline for comparison with our approach.

Other researchers also investigated option discovery in multi-task settings [24, 32].

Igl *et al.* [32] present "Multitask Soft Option Learning", allowing learned options to adapt across tasks. They train the options in one task and transfer the learned option to the next task, but these soft options have the flexibility to evolve in response to task requirements. Nonetheless, the predefined number of options remains a prerequisite for training.

In contrast to the aforementioned works, our proposed decomposition-based method autonomously learns all option components, including the number of options, initiation, and termination, from data. Importantly, it can be used with any off-policy or on-policy RL algorithm.

2.2 Transfer Learning

Recent works have categorized the field of transfer learning and continual learning within RL into different categories and techniques [33, 34]. One of these categories involves **"Regularization Approaches"**, which entail the inclusion of a regularization term in the loss function. This strategy prevents agents from becoming excessively specialized in a single task and enables them to generalize across a spectrum of tasks. Elastic Weight Consolidation (EWC) [35] is one such approach. However, previous studies have revealed that relying solely on these regularization terms may fail when tasks have contradictory objectives or exhibit interference [36]. Interference happens when two or more tasks are incompatible for the same model [37]. An example of interference would be when for two (or more) tasks, the same state has different goals.

"Complementary Learning Systems and Memory Replay" is another category that pertains to knowledge transfer through the storage and replay of past experiences, as demonstrated in works like CLEAR [37]. However, these methods have proven challenging to adapt to on-policy algorithms, primarily suited for off-policy learning. Moreover, in cases of interference [36], past experiences can become detrimental. In our work, we lean towards the concept of transferable sub-policies, which can be seamlessly integrated with both on-policy and off-policy RL algorithms.

Weight transfer has also been explored within the realm of RL [38]. This weight transfer can manifest in various forms, including policy transfer [39], value network transfer, or even transferring entire models [40]. In the context of transfer learning, our method aligns with the group of transferring sub-policies. Unlike transferring value networks, which entail transferring low-level information, we transfer sub-policies. We hypothesize that sub-policies can generalize better compared to the transfer of the entire model, which typically proves overly specific to the task. Also, we compared our method with weight transfer methods where each model's weight is initialized with the weights learned in the previous task. Additionally, Modulating Masks [41], a transfer learning method in RL utilizing supermasking [42], is selected as another baseline example. In this approach, the authors initialize the masking for each new task with a linear combination of all previously learned masks. We chose Modulating Masks as another baseline example due to its relevance to our approach. The utilization of masking over neural networks allows the agent to access sub-networks, which can be interpreted as sub-policies.

Dynamic Architecture centers on adaptively modifying neural network structures across tasks to facilitate knowledge transfer while allowing for the acquisition of new policies. Methods such as Progressive Neural Networks [43], Dynamically Expandable Networks [44], and Progress and Compress [45] fall under this classification. These techniques leverage previously acquired features, albeit at the cost of expanding the neural network. Among these methods, we have chosen Progressive Neural Networks (PNN) as one of the promising baselines for comparing with our approaches.

2.3 Compositional Methods

Another approach toward continual learning in RL involves around **Compositional Methods** [46]. Modular approaches [47–49] aim to decompose the environment's functionality into modules and train all of these modules at once. These modules

then will serve as sub-policies or skills which can be transferred.

Goyal *et al.* [48] propose a modular approach aimed at learning policies with enhanced generalization capabilities across new environments. They introduce modules that compete with each other through information bottleneck as a part of the loss function. This bottleneck compels each module to only make decisions for a subset of states, effectively segmenting the state space into modules. π-PRL [50] is another example of modular methods. It uses differentiable languages and program synthesis instead of deep reinforcement learning. π-PRL leverages a collection of tasks to acquire sub-policies, subsequently transferring them to new tasks as a set of actions. Mendez *et al.* [49] tackle compositional methods by employing a set of modules, each equipped with its unique inputs and outputs. These modules find utility in addressing inherently submodular problems, and they are interconnected in a multi-level fashion. Each module is provided with a part of the state representation corresponding to its functionality or a feature vector derived from preceding modules.

It is essential to acknowledge that the mentioned modular approaches exhibit certain challenges, including their inherent complexity and reliance on user-specific or domain-specific information. While supplying such information may be feasible in specific scenarios, our approach seeks to provide an alternative that does not require domain knowledge.

Chapter 3

Learning Options with Neural Network Decomposition

In this chapter, we elaborate on the methodology employed in our research, which is designed to discover options using a set of tasks $\mathcal{P}$, which can be applied to another set of tasks $\mathcal{P}'$. Our method will generate policies for tasks in $\mathcal{P}$ as the first step. We hypothesize that sub-policies possess a greater potential for unseen future tasks when compared to task-specific policies. We define these sub-policies as the components of the primary policy π learned over tasks in $\mathcal{P}$ in the previous step. Thus, we use these sub-policies to create temporally extended actions and transfer them to tasks $\mathcal{P}'$. These temporally extended actions play a pivotal role in enhancing the sample efficiency of the training process, effectively expediting the learning process for tasks within the set $\mathcal{P}'$ [19, 21, 23].

To formally define these temporally extended actions, we employ the options framework [5, 6]. An option ω is characterized by a tuple comprising three components, $(I_\omega, \pi_\omega, T_\omega)$. Specifically, I_ω is a function that, given a state s_t, returns the probability of the option initiating in state s_t. Subsequently, π_ω represents the policy that the agent adheres to once the option is initiated, while T_ω serves as a function that, when provided with a state s_t, returns the probability of the option terminating in that state. In our implementation, we adopt the call-and-return execution paradigm, whereby the agent follows π_ω until the option concludes.

The key stages of our algorithm for options discovery can be summarized as follows.

1. **Training Neural Policies:** We begin by learning a set of neural policies $\{\pi_{\theta_1}, \pi_{\theta_2}, \ldots, \pi_{\theta_n}\}$ using tasks from the set $\mathcal{P}$.

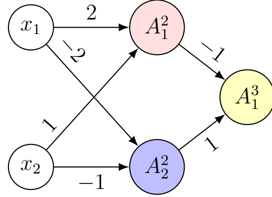
2. **Decomposition of Neural Policies:** Subsequently, each neural network encoding π_{θ_i} is decomposed into a set of sub-policies, which we collectively denote as U_i. Further details on this decomposition process can be found in Section 3.1.

3. **Options Synthesis:** From the collection of sub-policies $\cup_{i=1}^{n} U_i$, we select a subset to form a set of options, referred to as Ω. Section 3.2 explains this selection process.

4. **Expansion of Agent Action Space:** In the final step, we augment the set of actions available to the agent with the options acquired during Steps 1 to 3. Consequently, the set of available actions for tasks in $\mathcal{P}'$ becomes $A \cup \Omega$.

For Step 1, we can employ any algorithms that enable the learning of a parameterized policy π_θ, including popular techniques like policy gradient [51] and actor-critic algorithms [52]. In Step 4, the choice of an algorithm for solving MDPs is flexible, as we augment the agent's action space with the options acquired during Steps 1–3 [53]. The processes of decomposing the policies into sub-policies and defining/selecting a set of options are described in Section 3.1 (Step 2) and Section 3.2 (Step 3), respectively. As the set of options Ω is an integral part of the agent's action space for tasks in $\mathcal{P}'$, Step 3 primarily focuses on defining π_ω and T_ω for all ω within Ω, with I_ω set to encompass all states within S. We also define our options to have a fixed duration, thus our options terminate after running for some specific number of actions. We refer to both the methodology for learning options and the resultant options as the DEC-OPTIONS.

Neural Network **Neural Tree**



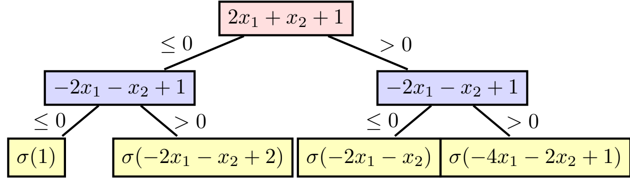


Figure 3.1: A neural network with two inputs, two ReLU neurons in the hidden layer, and one Sigmoid neuron in the output neuron is shown on the left. All bias terms of the model are 1; for simplicity, we omit bias values. The neural tree representing the same function encoded in the network is shown on the right. The root of the tree represents the neuron A_1^2, the nodes in the second layer represent the neuron A_2^2, and the leaf nodes represent the output neuron A_1^3, where $\sigma(\cdot)$ is the Sigmoid function. The colors of the neurons match the colors of the nodes in the tree that represent them.

3.1 Decomposing Neural Policies into Sub-Policies

In our study, we consider parameterized policies denoted as π_θ, which are encoded using fully connected neural networks comprising m layers $(1, \ldots, m)$. These neural networks employ piecewise activation functions, such as the ReLU functions [11]. The first layer serves as the input layer, denoted as X, while the m-th layer represents the output of the network. To illustrate, in the network diagram depicted in Figure 3.1, m is equal to 3. Each layer j is equipped with n_j neurons $(1, \ldots, n_j)$, where n_1 corresponds to the number of input values within X. The parameters connecting the layers i and $i+1$ in the network are represented as $W^i \in \mathbb{R}^{n_{i+1} \times n_i}$ and $B^i \in \mathbb{R}^{n_{i+1} \times 1}$. Within these matrices, the k-th row vector of W^i and B^i is designated as W_k^i and B_k^i, respectively, signifying the weights and bias term of the k-th neuron within the $(i+1)$-th layer. For instance, in Figure 3.1, we observe that $n_1 = 2$ and $n_2 = 2$. Additionally, we introduce $A^i \in \mathbb{R}^{n_i \times 1}$ as the values generated by the i-th layer, with $A^1 = X$ and A^m representing the model's output. The computation of a forward pass within the model entails the derivation of values for $A^i = g(Z^i)$, where $g(\cdot)$ serves as the activation function, assumed to be the ReLU function in Figure 3.1. Here, we

have $Z^i = W^{i-1} \cdot A^{i-1} + B^{i-1}$.

We introduce the concept of a "neural tree", which is represented as a binary tree (N, E), where N encompasses the nodes in the tree, and E represents the connections between these nodes. The neural tree is a representation equivalent to networks employing two-part piecewise linear activation functions, such as the ReLU. In this context, each internal node within the tree corresponds to a neuron within the layers $[2, \ldots, m-1]$ of the network (comprising all neurons except those in the input and output layer). The neurons within the output layer are represented within the leaf nodes of the tree. This mapping shares similarities with oblique decision trees, as each internal node of the neural tree defines a function $P \cdot X + v \leq 0$ concerning the input X. However, distinct from oblique decision trees, each leaf node in the neural tree signifies the computation taking place in the output layer of the network. In the case of a neural policy tailored for a Markov Decision Process (MDP) with $|A| = 2$ actions, wherein the number of output neurons is $n_m = 1$, each leaf node returns the Sigmoid value resulting from $P \cdot X + v$. In scenarios where the number of actions is $|A| > 2$, such that $n_m = |A|$, each leaf node returns a probability distribution defined by the Softmax values of $P' \cdot X + V$, with $P' \in \mathbb{R}^{n_m \times n_1}$ and $V \in \mathbb{R}^{n_1 \times 1}$. In cases involving continuous action spaces, each leaf node provides a parameterized distribution from which actions can be sampled.

The definition of parameters for both internal and leaf nodes serves as a foundation for inference, which initiates at the root of the tree. The process involves determining whether $P \cdot X + v \leq 0$ is true or false, subsequently guiding the traversal towards the left or right child based on the outcome. This iterative process is continued until a leaf node is reached, at which point the leaf node computation is performed.

The choice of a two-part piecewise linear function $g(\cdot)$, like the ReLU, results in a neuron's output value being determined by one of the two linear functions composing g. For instance, in the case of a neuron employing a ReLU function, expressed as $g(z) = \max(0, z)$, the neuron's output is either 0 or z. This binary behavior

leads to the notion of an "activation pattern", which is an ordered set of binary values that denotes whether a given node within the network is active or inactive for a particular input X [13–15]. Every path within a neural tree corresponds to an activation pattern, excluding those that pertain to the output layer. For example, if the condition $P \cdot X + v \leq 0$ is met at the root of the tree, the left sub-tree represents a scenario in which the first neuron of the network is inactive, while the right sub-tree signifies the scenario where the first neuron is active. The selection of a path within the neural tree implies that each neuron represents a linear function of the input X. This property enables the combination of multiple linear functions to result in another linear function. This characteristic allows us to define the function of the output layer in each tree as a linear function of the input X.

Example 1 *In our illustrative example depicted in Figure 3.1, we explore a neural network representing a neural policy with two actions, determined by the Sigmoid value of the output neuron. The accompanying right-hand side diagram in Figure 3.1 shows the neural tree that represents the neural network. The tree accounts for all activation patterns in the neural network. For example, if both neurons in the hidden layer are inactive, then* $A_1^2 = A_2^2 = 0$ *and the output of the network is* $\sigma(0 \cdot -1 + 0 \cdot 1 + 1) = \sigma(1)$*; this activation pattern is represented by the left branch of the tree. If the first neuron (from top to bottom) in the hidden layer is active and the second is inactive, the neural network produces the output* $\sigma(-1 \cdot (2x_1 + x_2 + 1) + 1 \cdot 0 + 1)) = \sigma(-2x_1 - x_2)$*; this activation pattern is given by following the right and then left branch from the root.*

Given that each node within the neural tree represents a function of the input, each sub-tree within the neural tree represents a sub-policy of the policy encoded by the network. A neural network comprising d neurons in a single hidden layer creates a neural tree with height $d+1$. Such a tree can be decomposed into $2^{d+1}-1$ sub-policies, with one sub-policy corresponding to each node in the tree. It is worth noting that the order in which neurons are represented along the paths of the neural tree can lead

to different sub-policies. In our ongoing example, the sub-policies obtained when A_1^2 serves as the root of the tree differ from the sub-policies derived when A_2^2 is the root. The total count of distinct sub-policies for a network featuring a single hidden layer with d neurons can be calculated as $\sum_{i=0}^{d} \binom{d}{i} \cdot 2^i$. In our example, this computation results in $1+4+4=9$. To provide some perspective, the value of 1 for $i=0$ signifies the sub-policy identical to the original policy. The value of 4 for $i=1$ denotes the number of sub-policies stemming from trees rooted at the children of the tree's root. In this context, the root of the tree can represent two different neurons, each with two children, yielding a total of 4 unique sub-policies.

To be able to evaluate and consider all the possible sub-policies resulting from neural decomposition, we limit our work to only small neural policies with one hidden layer. This enables us to evaluate our hypothesis that these sub-policies can be used for discovering useful options without adding more complexity to the problem.

3.2 Synthesizing and Selecting Options

Let $\{\pi_{\theta_1}, \pi_{\theta_2}, \cdots, \pi_{\theta_n}\}$ represent the collection of policies that the agent learns for various tasks within $\mathcal{P}$. Each of these policies is capable of generating its own set of sub-policies, denoted as U_i, using the neural network decomposition methodology described earlier. The unification of these sub-policies across all tasks yields the set $U = \{U_1, U_2, \cdots, U_n\}$. Additionally, consider a sequence of state-action pairs $\{(s_1, a_1), (s_2, a_2), \cdots, (s_k, a_k)\}$, observed under policy π, with an initial state distribution μ for a task ρ. Here, s_1 is sampled from μ, and for each state s_t in the sequence, the subsequent state s_{t+1} is sampled from $p(\cdot|s_t, a_t)$, where $a_t = \arg\max_a \pi(s_t, a)$. Since the options obtained from Dec-Options act greedily, we used the $\arg\max$ operator over π to match the options' behavior in the selection process. Plus, the $\arg\max$ helps us to reduce the noises in the selection step. If ρ is episodic, s_{k+1} constitutes a terminal state, while in non-episodic scenarios, k sets the maximum horizon for the sequence. Let $\mathcal{T}_i$ denote a set of such sequences generated under π_{θ_i}

and the respective ρ_i's initial state distribution μ. The overall collection of sequences is represented as $\mathcal{T} = \{\mathcal{T}_1, \mathcal{T}_2, \cdots, \mathcal{T}_n\}$.

Since the sub-policies in U are not options and are one-step sub-policies, we use a while-loop for each sub-policy within U that iterates z times, transforming it into a temporally extended action. Once initiated, the resulting option, denoted as ω, executes for z steps before terminating. The value z associated with ω is denoted as ω_z. Within each iteration of the while-loop, the agent takes an action determined by $\arg\max_a \pi(s, a)$, where π corresponds to the sub-policy and s represents the agent's current state. The use of the $\arg\max$ operator ensures deterministic behavior within the loop. Given the length of the longest sequence in $\mathcal{T}_i$ as $k_{\max}$, we consider options with $z = 1, \cdots, k_{\max}$ for each sub-policy in U_i. The resulting set, denoted as $\Omega = \{\Omega_1, \Omega_2, \cdots, \Omega_n\}$, encompasses all while-loop options derived from U. Each Ω_i features $k_{\max} \cdot |U_i|$ options for ρ_i, with one option corresponding to each value of z.

The objective is to identify a subset of "helpful" options from the set Ω. The measure of a set's helpfulness is determined based on the Levin loss [18]. This metric quantifies the expected number of environmental steps (i.e., calls to the function p) required for an agent to reach a target state under a given policy. It assumes a deterministic p and a fixed initial state. The sole source of variability arises from the policy itself. The Levin loss for a sequence $\mathcal{T}_i$ and policy π_i is defined as Equation (3.1).

$$\mathcal{L}(\mathcal{T}_i, \pi) = \frac{|\mathcal{T}_i|}{\prod_{(s,a)\in\mathcal{T}_i} \pi(s, a)} \tag{3.1}$$

The factor $1/\prod_{(s,a)\in\mathcal{T}_i} \pi(s, a)$ represents the expected number of sequences that an agent must sample using policy π to observe $\mathcal{T}_i$. Importantly, the length of the sequence required to observe $\mathcal{T}_i$ is considered known and fixed as $|\mathcal{T}_i|$. Consequently, the agent performs exactly $|\mathcal{T}_i|$ steps within every sampled sequence.

Consider π_u as the uniform policy for an MDP, meaning it assigns an equal probability to all available actions within a given state. Furthermore, let π_u^{Ω} represent the uniform policy after augmenting the MDP actions with the options in Ω. The

inclusion of options has two effects: it can either increase or decrease the Levin loss. On one hand, the probability of selecting each action decreases, affecting not only the actions in the target sequence but all actions. On the other hand, the reduction in the number of decisions required by the agent can lead to fewer multiplications in the denominator of the loss. The primary task is to choose a subset of options from the set Ω, generated by decomposed policies, that minimizes the Levin loss.

$$\arg\min_{\Omega' \subseteq \Omega_T} \sum_{\mathcal{T}_i \in \mathcal{T}_V} \mathcal{L}(\mathcal{T}_i, \pi_u^{\Omega'}) \,. \tag{3.2}$$

To enhance the chances of selecting options that generalize well, it is essential to divide the set of tasks $\mathcal{P}$ into separate training and validation sets. Options derived from policies that involve while loops with a duration matching the sequence length for a specific task are unlikely to generalize to other tasks, as they are highly task-specific. In Equation (3.2), Ω_T denotes the set of options extracted from the policies learned for tasks in the training set, and $\mathcal{T}_V$ represents the sequences produced by applying policies learned for tasks in the validation set. Uniform policies are a sensible choice for our formulation as they serve as reasonable approximations of neural policies during the initial stages of training when network weights are randomly initialized. Minimizing the Levin loss leads to a reduction in the expected number of sequences that an agent needs to sample in order to observe high reward values. While the subset selection problem in Equation (3.2) is known to be NP-hard in general [54], so we opt for a greedy approximation to address it.

3.2.1 Greedy Approximation to Select Options

Algorithm 1 shows the pseudocode for the greedy approximation we use to solve Equation (3.2). The greedy algorithm for approximating a solution to Equation (3.2) begins with initializing Ω' as an empty set. In each iteration, the algorithm identifies and adds to Ω' the option that results in the most significant reduction in Levin loss. This iterative process continues until the inclusion of another option no longer leads

Algorithm 1 GREEDY-SELECTION

Require: Sequences of state-action pairs $\mathcal{T}_V$ and set of options Ω_T
Ensure: A subset Ω' of options Ω_T
1: $\Omega' \leftarrow \emptyset$
2: **while** True **do**
3: $\quad \omega \leftarrow \arg\min_{\omega \in \Omega_T} \sum_{\mathcal{T}_i \in \mathcal{T}_V} \mathcal{L}(\mathcal{T}_i, \pi_u^{\Omega' \cup \{\omega\}})$
4: $\quad$ **if** $\mathcal{L}(\mathcal{T}_i, \pi_u^{\Omega' \cup \{\omega\}}) \geq \mathcal{L}(\mathcal{T}_i, \pi_u^{\Omega'})$ **then**
5: $\quad\quad$ **return** Ω'
6: $\quad \Omega' \leftarrow \Omega' \cup \{\omega\}$

to a reduction in the loss. At this point, the algorithm terminates and returns the subset Ω'.

Due to the call-and-return model inherent in this approach, a dynamic programming procedure becomes essential to efficiently compute the values of $\mathcal{L}$ while selecting Ω'. This is because it is not evident which action or option the agent will use in each state within a sequence, making it challenging to minimize the Levin loss. To illustrate this, consider an option ω that provides the correct action for ω_z states within a sequence initiated at s_1. In contrast, ω' may not return a_1 for s_1 but is suitable for the sequence of ω'_z states starting from s_2. If, in this context, $\omega_z < \omega'_z$, and if employing ω in s_1 prevented the use of ω' in s_2 because ω was still executing in s_2, it is conceivable that selecting a_1 from the action space A for s_1 and subsequently initiating ω' in s_2 could minimize the Levin loss.

Algorithm 2 COMPUTE-LOSS-OPTIMIZED

Require: Sequence $\mathcal{T}$, probability $p_{u,\Omega}$, options Ω
Ensure: $\mathcal{L}(\mathcal{T}, \pi_u^{\Omega})$
1: $M[i] \leftarrow i$ for $i = 0, 1, \cdots, |\mathcal{T}| - 1$ # *initialize table as if only primitive actions are used*
2: **for** $j = 0$ to $|\mathcal{T}|$ **do**
3: $\quad$ **for** ω in Ω **do**
4: $\quad\quad$ **if** ω is applicable in s_j **then**
5: $\quad\quad\quad M[j + \omega_z] \leftarrow \min(M[j + \omega_z], M[j] + 1)$ # *ω is used in s_j for ω_z steps*
6: $\quad$ **if** $j > 0$ **then**
7: $\quad\quad M[j] \leftarrow \min(M[j-1] + 1, M[j])$
8: return $|\mathcal{T}| \cdot (p_{u,\Omega})^{-M[|\mathcal{T}|-1]}$

Algorithm 2 demonstrates the computation of $\mathcal{L}$. This procedure takes as input a sequence $\mathcal{T}$, the probability $p_{u,\Omega}$ (calculated as $\frac{1}{|A|+|\Omega|}$) representing the likelihood of selecting any action from the augmented action space (including the options in Ω) under a uniform policy. Additionally, the procedure receives the set of options Ω, the set of options that are used alongside actions to generate the sequence $\mathcal{T}$. The algorithm's objective is to compute $\mathcal{L}(\mathcal{T}, \pi_u^{\Omega})$. This algorithm uses the dynamic programming approach to calculate the Levin loss. Algorithm 2 employs a table M of size $|\mathcal{T}|$ that is initially populated with values numbered from 0 to $|\mathcal{T}|-1$. Within this table, each entry j shows the minimum number of actions (or combination of actions and options) needed to be called to reach state s_j.

Line 5 addresses the situation in which an option can be utilized. An option ω is considered for use in (s_j, a_j) if and only if $a_{j+i} = \arg\max_{a \in A} \pi_\omega(s_{j+i}, a)$ for $i = 0, \cdots, \omega_z$. This condition implies that the actions returned by ω align with the ω_z actions within the sequence that starts at a_j. When an option is employed at s_j, the agent can jump from s_j to $s_j + w_z$ in one step. Thus, we update the entry of $j + w_z$ (and any other entry after $j + w_z$) with respect to the current value of entry j in our table.

Once the calculation of every entry in the M is completed, we use the values of this table to compute the Levin loss. Since entry $|\mathcal{T}| - 1$ stores the minimum number of steps needed to reach the end of the sequence $|\mathcal{T}|$, this value is used for the computation of the Levin loss (as observed in Line 8). The utilization of the table M ensures that the loss value for any combination of actions and options is not recomputed multiple times, resulting in an efficient algorithm with time complexity of $O(|\Omega| \cdot |\mathcal{T}|)$ and the memory complexity of $O(|\mathcal{T}|)$.

Example 2 *Let $\mathcal{T} = s_0, s_1, s_2, s_3, s_4, s_5$ be a sequence of states and $\Omega = \{\omega_1, \omega_2\}$ be a set of options. ω_1 can start in s_0 and it terminates in s_2; ω_2 can start in s_1 and it terminates in s_4. Next, we show how the table M in Algorithm 2 changes after every*

iteration of the for-loop in line 3. The value of 3 *for* $M[5]$ *indicates that state* s_5 *can*

Iterations	M
initialization	0, 1, 2, 3, 4, 5
0	0, 1, 1, 3, 4, 5
1	0, 1, 1, 3, 2, 5
2	0, 1, 1, 3, 2, 5
3	0, 1, 1, 2, 2, 5
4	0, 1, 1, 2, 2, 5
5	0, 1, 1, 2, 2, 3

be reached with three actions: a primitive action from s_0 *to* s_1, ω_2 *from* s_1 *to* s_4, *and another primitive action from* s_4 *to* s_5. *If* $p_{u,\Omega} = 0.25$, *then the optimal Levin loss value returned in line 8 of Algorithm 2 for* $\mathcal{T}$ *and* Ω *is* $\frac{6}{0.25^3} = 384$.

Chapter 4

Experiments

In this chapter, we explain the experimental methodology and outcomes of our study to assess our hypothesis that options can be extracted through the decomposition of neural network policies. In our experimental setup, we have two sets of tasks: $\mathcal{P}$ and $\mathcal{P}'$. Initially, we train agents on the set of tasks $\mathcal{P}$. Subsequently, our aim is to transfer "helpful" options from $\mathcal{P}$ to $\mathcal{P}'$, thereby expediting the training process of agents operating on tasks within $\mathcal{P}'$. While all tasks in both sets, $\mathcal{P}$ and $\mathcal{P}'$, have the same observation and output structures, they may have distinct objectives, reward systems, or dynamics.

It is important to emphasize that the Dec-Options approach is not confined to a specific reinforcement learning algorithm. Through this chapter, we demonstrate its efficacy with the Deep Q-Network (DQN) algorithm [12], in conjunction with Proximal Policy Optimization (PPO) [55]. Additional details about agent architectures, hyperparameter settings, and the libraries employed will be presented in their respective sections.

4.1 Baselines

We have chosen a set of baselines to provide insights into the effectiveness of our approach. The first baselines, which we refer to as **Vanilla-RL**, involve training the DQN and PPO algorithms using the original actions. Our hypothesis posits that our

Dec-Options framework is more sample-efficient than these "vanilla" reinforcement learning baselines.

Building upon our literature review, we have also identified a series of baselines closely related to our work. These baselines serve as reference points for assessing the efficacy of our Dec-Options approach. Two transfer learning algorithms, namely **Transfer-PPO** and **Modulating-Mask**, have been selected as baselines. In **Transfer-PPO**, we train on one task and initialize the model weights, including both the policy network and the value network, to be identical to those of the previously trained task at the outset of the next task. Modulating-Mask is inspired by the work of Ben-Iwhiwhu *et al.* [41], which bears a close relation to our approach, particularly in terms of utilizing sub-policies and sub-networks.

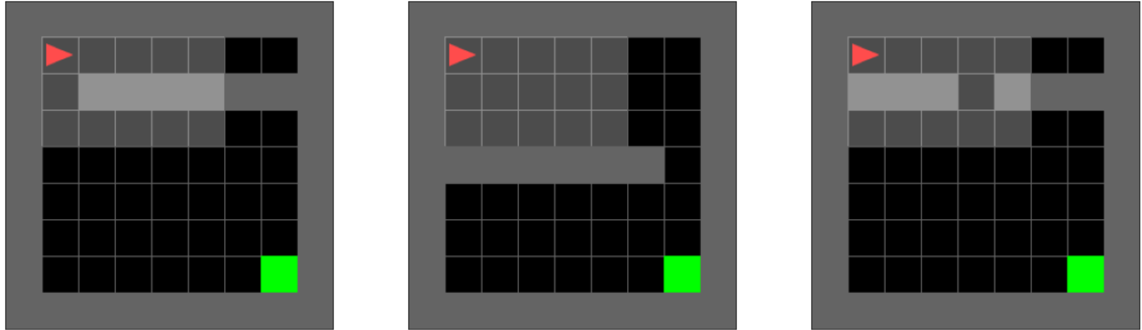
We also use the **PNN** baseline, which pertains to methods that modify the structure of neural networks. To showcase the effectiveness of transferring skills from previous tasks, we have created two additional baselines: **Neural-Augmented** and **Dec-Options-Whole**. In Neural-Augmented, we augment the action space of the target task with neural policies from previous tasks in $\mathcal{P}$. Comparing this approach with our Dec-Options framework highlights the impact of sub-policies and temporal abstraction. Dec-Options-Whole is a variant of Dec-Options where we incorporate the policy networks from previous tasks in $\mathcal{P}$ without neural decomposition. This comparison underscores the significance of neural decomposition in utilizing sub-policies.

To highlight the effectiveness of our algorithm's exploration, we consider **ez-greedy** [31] baseline. Lastly, to emphasize the influence of knowledge transfer through options, we considered **Option-Critic** and **DCEO** as representative algorithms for directly learning options within tasks $\mathcal{P}'$. Both **Option-Critic** and **DCEO** do not leverage the knowledge of policies learned for tasks in $\mathcal{P}$ and discover options by interacting with the target task directly, while our Dec-Options learn the options from previous tasks and evaluate them in the downstream problems.

Open-Loop is another baseline we experimented with. In this setting, we generate deterministic loops of all different sizes and action combinations. In this baseline, we consider having all possible options from length 1 to length K, in which K is determined by upper-bound computation. This baseline then for each length of p will generate all possible combinations of actions. Considering that the agent has M actions, then there will be M^p different options with size p. For example, if the agent has 3 actions of $\{0, 1, 2\}$, and we want to generate all options of length 2, we will end up with the set of options $\{\{0, 0\}, \{0, 1\}, \{0, 2\}, \{1, 0\}, \{1, 1\}, \{1, 2\}, \{2, 0\}, \{2, 1\}, \{2, 2\}\}$. These options' policies do not depend on the observation of the agent, hence giving them the name of **Open-Loop**. Then, we run our option selection method described earlier to select the open loop options that minimize the Levin Loss. The option selection algorithm is the same between our approach and the **Open-Loop** baseline. The only difference between these two approaches is the neural-network decomposition in our approach, and having the exhaustive set of all possible deterministic options in **Open-Loop** approach. Thus, the goal of comparing our method with **Open-Loop** is to show that the options that the neural network decomposition generates play an important role. The sub-policies acquired through decomposition would have information about the dynamics and the state of the environment that the agent can take advantage of.

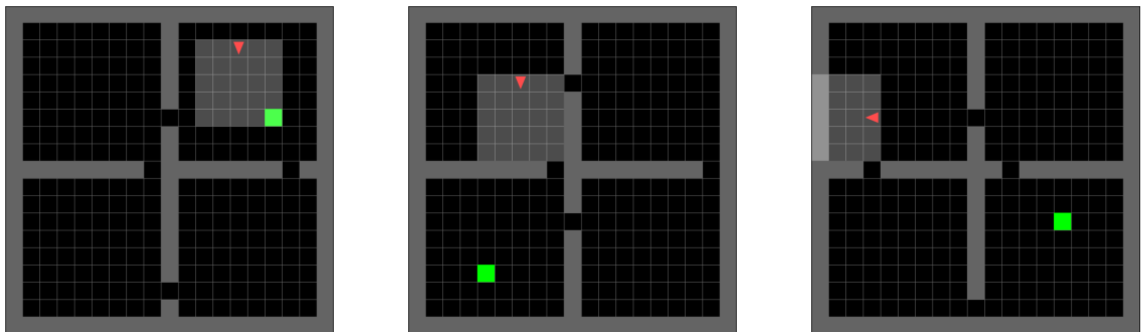
4.2 Problem Domains

We use two distinct hard exploration domains. In these domains, we created a set of simpler tasks designated as $\mathcal{P}$ and more challenging tasks grouped under $\mathcal{P}'$. While all tasks within both $\mathcal{P}$ and $\mathcal{P}'$ share certain similarities, they also exhibit unique characteristics that differentiate them. This arrangement is particularly conducive to the transfer of knowledge between tasks, a key aspect of our study. The two domains utilized in our experiments are MiniGrid [56] and ComboGrid. In the forthcoming sections, we will provide an introduction to each of these domains, outlining their key



(a) Simple Crossing 1 (b) Simple Crossing 2 (c) Simple Crossing 3

Figure 4.1: MiniGrid Simple Cross Tasks in set $\mathcal{P}$



(a) Four Rooms 1 (b) Four Rooms 2 (c) Four Rooms 3

Figure 4.2: MiniGrid Four Rooms Tasks

attributes and relevance to our research.

4.2.1 MiniGrid

Our first domain is based on the implementation of MiniGrid, as provided by Chevalier-Boisvert *et al.* [56]. This library encompasses a variety of tasks, each characterized by distinct objectives that challenge various aspects of RL. The significance of MiniGrid in our experiments lies in the fact that all these environments offer partial observations of the states, thereby evaluating the performance of our method in scenarios involving partial observable, and sparse rewards. In the MiniGrid domain, the agent's observation is limited to a small window of the environment in front of itself, as determined by the parameter "view size". Within MiniGrid, we selected a set of simpler tasks to comprise $\mathcal{P}$. Specifically, we chose the Simple Crossing tasks,

illustrated in Figure 4.1. These tasks are characterized by three different variants of "MiniGrid-SimpleCrossingS9N1-v0", all of which operate within a 9×9 grid and feature only one wall. For tasks within $\mathcal{P}'$, we sought more challenging environments within MiniGrid. Our selection consisted of three different variants of the Four Rooms environment, as depicted in Figure 4.2. In the Four Rooms environment, the agent navigates through a grid world divided into four distinct rooms, each interconnected, as shown in Figure 4.2. The primary objective here is for the agent to explore and reach its goal efficiently. The grid size for these tasks is 19×19. We selected three different variations of the Four Rooms such that they have different difficulties. In the first variant, the agent and the goal are located within the same room. In the second, they are in neighboring rooms, and in the final task, they are positioned in two non-neighboring rooms. This delineation of tasks within the MiniGrid domain serves as a foundational component of our experimentation, allowing us to explore the transfer of knowledge and options between simpler and more complex environments.

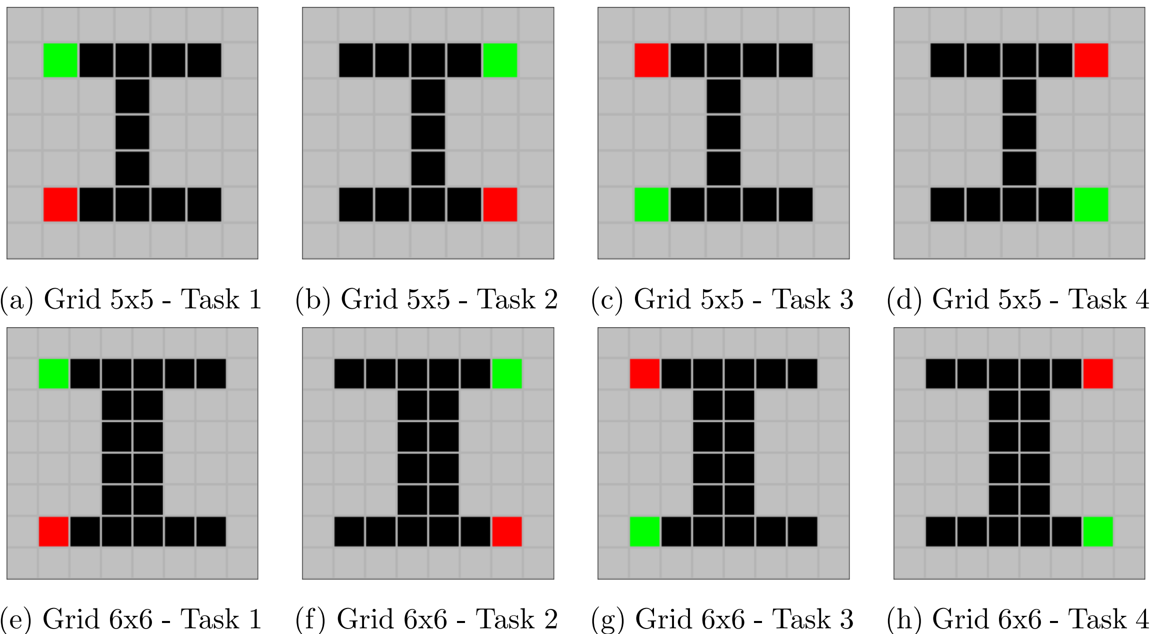
Observation and Action Space In our selected MiniGrid domains, the achievement of goals is streamlined to only require the use of three distinct actions out of the original set of MiniGrid actions. These primitive actions include:

- Turning Left (represented by 0 for the agent)
- Turning Right (represented by 1 for the agent)
- Moving Forward (represented by 2 for the agent)

Because the domain used only needs these 3 actions, we simplified the action space to only the 3 mentioned actions. Also, the observations provided to the agent in MiniGrid were simplified. For our experiments, we modified the default view size of 7×7 in MiniGrid to a reduced view size of 5×5 in front of the agent. Changing the view size of the MiniGrid domain has been done in other works before [48]. This smaller view size would help to have a smaller input allowing us to use a smaller

neural network. At each time step, the agent receives an observation comprising a modified version of the original MiniGrid's observation. This view grid is represented using a one-hot-encoding scheme, with distinct encoding for different objects within the 5×5 view grid. These objects include walls, empty floor spaces, and the goal object. Additionally, similar to the original MiniGrid setup, the agent is presented with information regarding its facing direction. This direction can take on one of four values, ranging from 0 to 3, encoded in one-hot encoding, each corresponding to one of the four cardinal directions. This directional information operates akin to a compass, aiding the agent in navigation and exploration tasks.

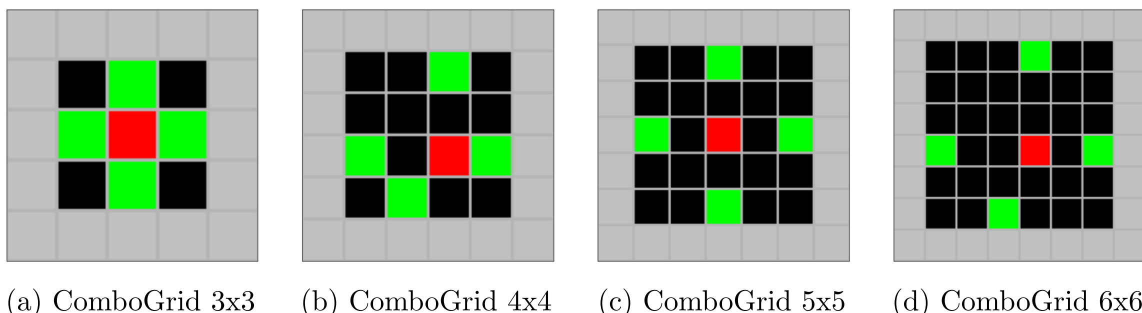
Rewards and Episodes In our experimental setup, we have defined reward structures and episode lengths that vary between the tasks in $\mathcal{P}$ and those in $\mathcal{P}'$. For the tasks within $\mathcal{P}$, the agent receives a reward of -1 for each step it takes, with the exception of reaching the goal, which yields a reward of 0. This reward structure incentivizes the agent to reach the goal as quickly as possible while penalizing excessive exploration or deviations from the optimal path. To ensure adequate exploration in these simpler tasks, the maximum episode length for tasks in $\mathcal{P}$ has been set at 1000 steps. In contrast, the tasks within $\mathcal{P}'$ are designed with sparse rewards. Here, the agent receives a reward of 0 for each step except when it successfully reaches the goal, in which case it obtains a reward of 1. This sparse reward setting encourages the agent to focus on the task of reaching the goal efficiently, as rewards are only granted upon successful goal attainment. In all tasks, episodes may terminate prematurely if the agent reaches the maximum episode length before achieving the goal. For tasks in $\mathcal{P}$, this maximum episode length is set to 1000 steps. Conversely, tasks in $\mathcal{P}'$ have a maximum episode length of $19 * 19 = 361$ steps. The starting position of the agent and the location of the goal are deterministic and fixed across episodes within a specific task. This deterministic setup ensures the hardness of the task remains constant across episodes of a task, providing a reliable and consistent environment



(a) Grid 5x5 - Task 1 (b) Grid 5x5 - Task 2 (c) Grid 5x5 - Task 3 (d) Grid 5x5 - Task 4

(e) Grid 6x6 - Task 1 (f) Grid 6x6 - Task 2 (g) Grid 6x6 - Task 3 (h) Grid 6x6 - Task 4

Figure 4.3: ComboGrid tasks in $\mathcal{P}$. In this depiction, the agent is highlighted in red, while the goals are denoted in green, and the walls are represented in grey.



(a) ComboGrid 3x3 (b) ComboGrid 4x4 (c) ComboGrid 5x5 (d) ComboGrid 6x6

Figure 4.4: ComboGrid $\mathcal{P}'$ tasks

for the agent to learn and explore.

4.2.2 ComboGrid

Our second experimental domain, known as ComboGrid, offers a fully observable environment where temporally extended actions can help deal with the problem's dynamics. This environment serves as an ideal platform for validating our hypothesis and assessing our method's performance. ComboGrid is a grid world environment in which the agent's objective is to collect goals. However, the agent's navigation is governed by sequences of actions referred to as Combos, as opposed to individual

actions. These Combos correspond to movements in the cardinal directions, with four distinct combinations defined within the environment's dynamics. To execute a successful move, the agent must perform the correct sequence of actions, aligning with one of the defined Combos. Failure to execute the correct combination results in the agent remaining in the same cell, while the history of past actions resets. These combo actions are described as follows.

- Moving **Down**: 0, 2, 2, 1
- Moving **Up**: 0, 0, 1, 1
- Moving **Right**: 1, 2, 1, 0
- Moving **Left**: 1, 0, 2, 2

For our experiments in ComboGrid, we designed four distinct variations, each characterized by a different grid size. The selected grid sizes are 3×3, 4×4, 5×5, and 6×6. This range of grid sizes enables us to evaluate our hypothesis and baselines across a spectrum of complexity. Beginning with simpler scenarios in smaller grids, we subsequently assess the robustness of our algorithm and compare it to other baselines in larger and more challenging grids. Within the ComboGrid environment, we identified four tasks for inclusion in set $\mathcal{P}$. These tasks differ in terms of the initial position of the agent and the placement of the goals, as illustrated in Figure 4.3. These differences introduce tasks that can potentially interfere with each other. One example of such interference can be seen in Task 1 and Task 3 depicted in Figure 4.3 where in Task 1 the combo leading to going up in the cells in the middle of the grid can be a good policy, but in Task 3 the combo leading to going down is in favor. Following training in the four tasks within the set $\mathcal{P}$, we transferred the learned DEC-OPTIONS to a single task within the set $\mathcal{P}'$. These specific tasks are visually represented in Figure 4.4, for different grid sizes. In these tasks, the agent is placed in the center, with the goals located on the four sides.

Observation and Action Space Throughout our experiments in ComboGrid, consistency was maintained in both the action and observation spaces, regardless of the grid size. This uniformity encompassed all tasks within sets $\mathcal{P}$ and $\mathcal{P}'$. Agents operated with a consistent set of primitive actions, represented by integers 0, 1, and 2. These actions were utilized to create combos following the specifications outlined in 4.2.2. The combos are the same across all the domains in ComboGrid. At each time step, the agent receives a comprehensive view of the entire grid, presented in the form of a one-hot encoding representation. This representation encompassed not only the agent's position but also the positions of the goals within the grid. Consequently, the dimensionality of the observation space was directly proportional to the size of the grid, resulting in a representation of size $grid_size * grid_size * 2$. This representation provided the agent with a holistic understanding of the environment's state, enabling effective decision-making. Additionally, the agent received the sequence of past actions applied in the current cell as part of its input. This action history, integrated into the agent's observations, allowed for a temporal understanding of the agent's past actions, ensuring that the problem is Markovian. This consistent action and observation space framework ensured a standardized environment across tasks, enabling a thorough and meaningful evaluation of our method and baselines within the ComboGrid domain.

Rewards and Episodes Within our ComboGrid experiments, we have defined distinct reward structures and episode lengths for tasks in sets $\mathcal{P}$ and $\mathcal{P}'$, tailoring these parameters to the specific objectives and challenges posed by each set of tasks. For tasks within the set $\mathcal{P}$, the reward structure is designed such that the agent receives a reward of -1 for each step taken in the environment, with the exception of reaching the goal, which yields a reward of 0. This reward scheme encourages the agent to reach the goal as quickly as possible. The maximum episode length is set at $grid_size * grid_size * 80$ steps. Conversely, tasks within set $\mathcal{P}'$ present a

distinct objective. Here, the primary objective of the agent is to collect a total of 4 goals distributed across the grid. The reward structure is sparse, with the agent receiving a reward of 0 for each step taken in the environment, except when collecting goals. Upon reaching each goal, the agent receives a reward of 10. Collecting all 4 goals accumulates a total reward of 40. The maximum episode length for tasks in $\mathcal{P}'$ is set at $grid_size * grid_size * 16$ steps. In both sets of tasks, episodes may terminate if the agent reaches the maximum allowable steps within an episode. This termination mechanism ensures that episodes do not run indefinitely, allowing for controlled experimentation.

4.3 Empirical Results

For training across the task set $\mathcal{P}$, we used actor-critic algorithms [52], specifically PPO, and implemented them using the Stable-baselines framework [57]. The policy network was configured as a small feed-forward neural network with ReLU activation functions. This design choice ensures that we can comprehensively explore the space of sub-policies, allowing us to evaluate our hypothesis that these sub-policies can be used for discovering "helpful" options. Since the value network is not used in Dec-Options, we left the size of the value network unconstrained.

In Figure 4.5, we present the results obtained from our experiments conducted within the MiniGrid Four Rooms domain. These results showcase the performance of our Dec-Options method alongside selected baseline algorithms. The figures are organized into two sections, as follows: **1)** The first three rows of the figure show the performance of agents trained using the PPO algorithm. **2)** The last row of the figure shows the performance of agents trained using the DQN algorithm.

In all of these plots, we display the average return achieved by the agents, based on data from 24 independent agents for each baseline. Additionally, the plots include a 95% confidence interval computed over these runs. As detailed in Section 4.2.1, it is important to note that the maximum achievable reward for an agent in each episode

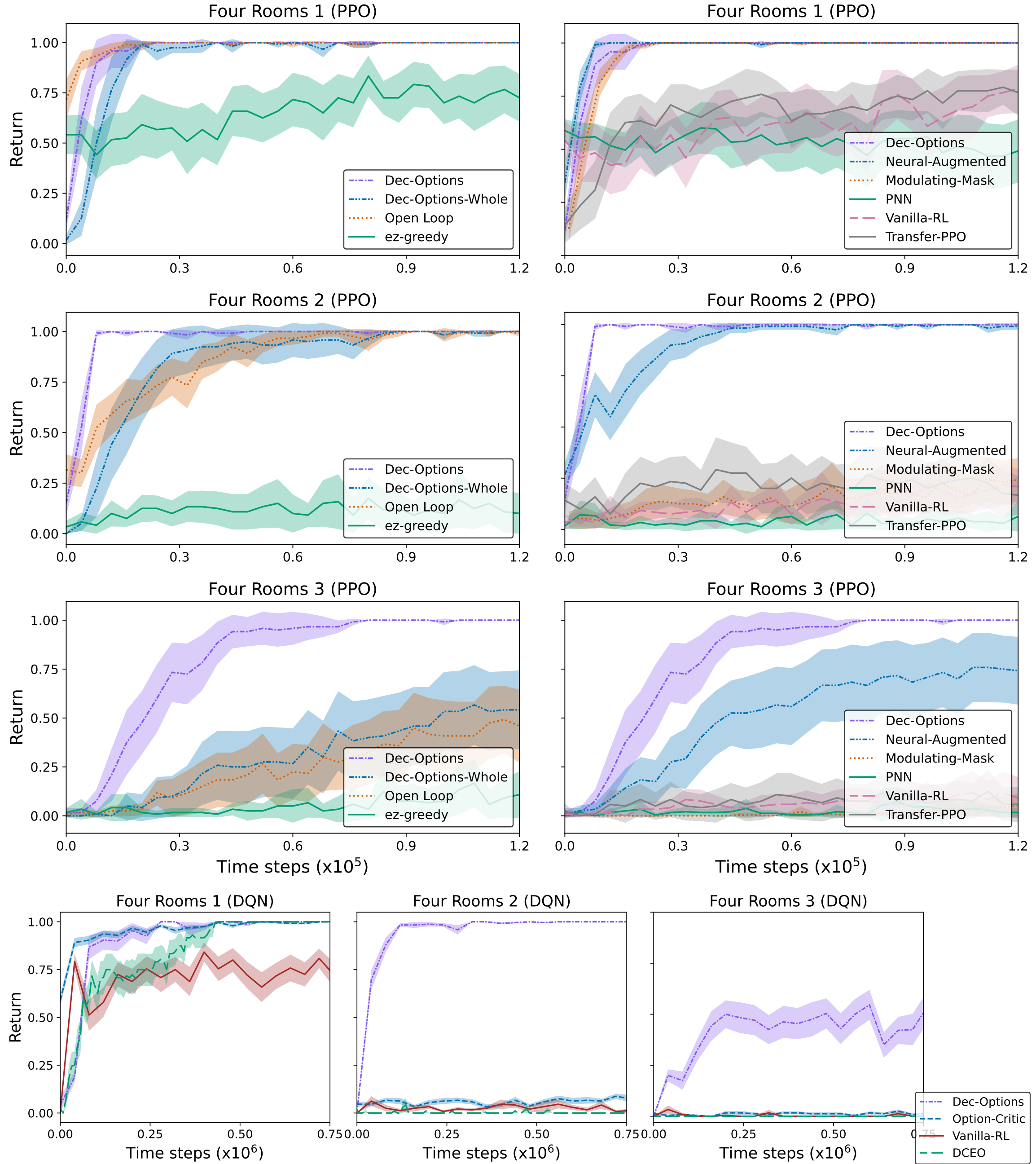


Figure 4.5: Performance of different methods on MiniGrid Domain

is 1, attained upon reaching the terminal goal. Within the Four Rooms domain, which includes three different variations as described in Section 4.2.1, several observations can be made. The Vanilla-RL baseline faces difficulties in learning effective policies, even in the case of the simplest variation, Four Rooms 1. These challenges stem from the environment's dynamics and the impact of partial observability, where

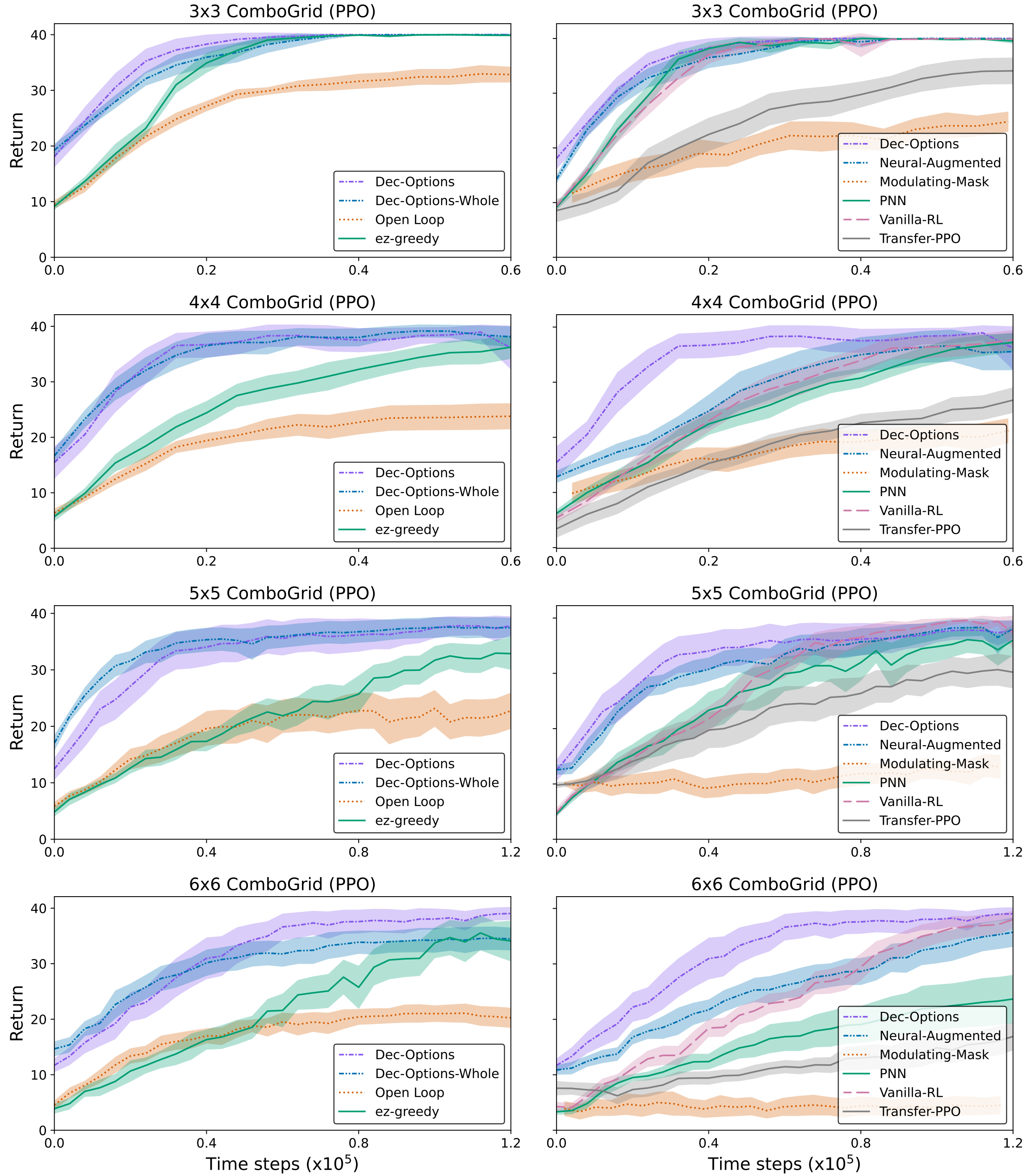


Figure 4.6: Performance of different methods on ComboGrid Domain, using PPO algorithm

agents cannot simply move toward a direction, but it must first turn to the correct direction and then move. Transfer-PPO and PNN exhibit similar performance, which can be attributed to the interference caused by task differences [36]. Since in this scenario repeating the actions (like moving forward) for multiple steps can lead to

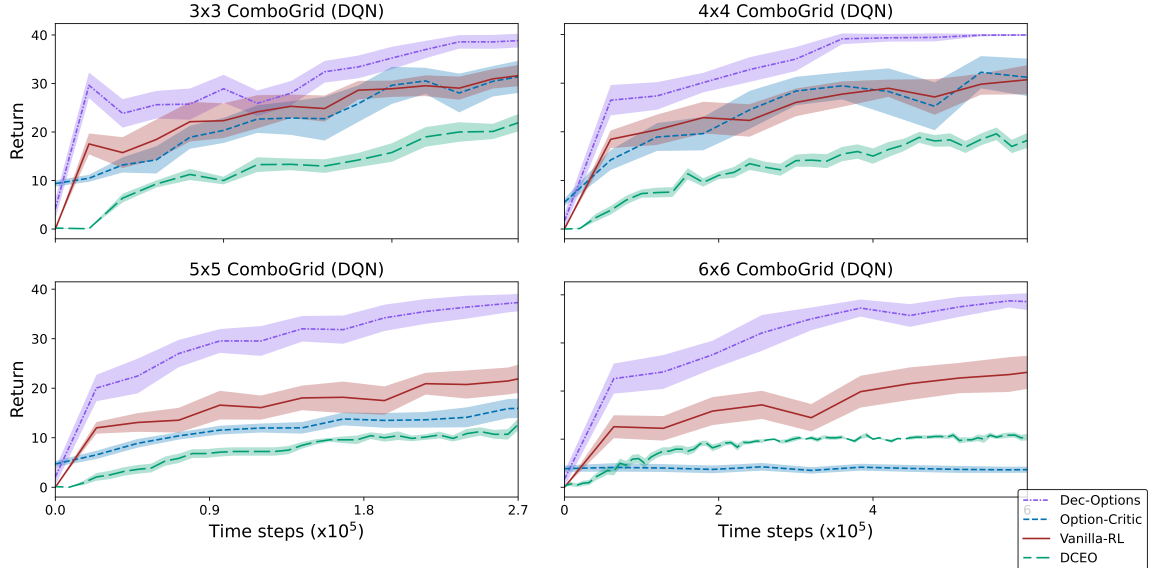


Figure 4.7: Performance of different methods on ComboGrid Domain, using DQN algorithm

better exploration and potentially better policy, we expected the ϵz-greedy baseline to perform well. The ϵz-greedy showed to have a better exploration strategy compared to the simple ϵ-greedy algorithm [31]. However, this approach did not present much more than Vanilla-RL. One reason here could be that this approach only repeats a randomly selected action for a number of steps and only in the exploration strategy, which is triggered only at random with probability ε. Also, note that doing actions like rotating in the MiniGrid environment multiple times will result in the agent rotating around itself, and might not provide much new information to the exploration strategy.

In contrast, Dec-Options, Dec-Options-Whole, Neural-Augmented, and Modulating-Mask, Open-Loop perform well in the Four Rooms 1 scenario, showcasing their effectiveness in overcoming challenges presented by the environment. Moving to harder scenarios such as Four Rooms 2, most baselines encounter difficulties, with successful convergence observed primarily in transfer learning-based methods. Dec-Options consistently performs well, even in these more challenging scenarios, underscoring its robustness. As tasks become increasingly demanding, such as in the case of Four

Rooms 3, Dec-Options stands out as the only method capable of achieving optimal solutions.

Comparing the Dec-Options with the Open-Loop baseline also suggests that the observation feedback gives more flexibility to our discovered options, making them more useful for unseen future tasks.

In the context of DQN agents, Option-Critic, DCEO, and Dec-Options surpass Vanilla-RL, demonstrating their utility in enhancing learning performance. Furthermore, an example trajectory from the Four Rooms 3 scenario in the Section 4.4 highlights the effectiveness of our approach. The sequence of actions taken by the trained agent is substantially reduced from a trajectory of size 39 to just 10 actions. This reduction in complexity demonstrates the significant advantage of our discovered options. This sequence is shown in the 4.4.

In Figure 4.6 and Figure 4.7, we present the results obtained from our experiments conducted within the ComboGrid domain. Similar to Figure 4.5, we organize the results into two sections, showcasing the performance of agents trained with the PPO algorithm in Figure 4.6 and agents trained with the DQN algorithm in Figure 4.7. Notably, in this domain, the maximum achievable return for an agent in an episode is set at 40, as described in Section 4.2.2. Our experiments within the ComboGrid domain encompassed various grid sizes, starting from 3×3 and progressively scaling up to grids of 6×6. As depicted in Figure 4.6 and Figure 4.7, in the smallest case of ComboGrid, most baselines exhibit similar performance. However, as the size of the grid increases, the complexity and challenges presented by the environment intensify. It is important to note that a naive repetition of actions is not an effective policy in this context, as the combos within the environment comprise different actions. Repeating a single action multiple times does not lead to meaningful exploration of the state space, which is evident from the performance of the ϵz-greedy baseline, closely resembling that of Vanilla-RL.

It is evident from the figures that as we transition to larger grid sizes, notably

6×6, Dec-Options outperforms other methods, learning more rapidly. The results suggest that Dec-Options can be robust and sample-efficient when confronted with different levels of environmental complexity. This observation holds for experiments conducted with the DQN algorithm as well. Similarly to our findings in MiniGrid, the performance gap between Dec-Options and other methods widens as the grid size increases, showcasing the effectiveness of our approach across varying degrees of complexity.

While we assess Dec-Options alongside other algorithms designed for learning options, it is crucial to recognize that Dec-Options addresses a fundamentally distinct problem compared to the other methods under consideration. Specifically, DCEO focuses on exploration during the process of learning options for a specific task. In contrast, our approach involves learning options across a set of tasks and subsequently assessing their utility in solving subsequent problems.

4.4 Qualitative Results

Dec-Options learned long options for MiniGrid. The sequence below shows the actions from the initial state to goal (left to right) of the Dec-Options agent in the Four Room 3 environment. This trajectory is sampled from one of the trained agents. Here, 0, 1, and 2 mean ‘turn right’, ‘turn left’, and ‘move forward’, respectively. The curly brackets show what is covered by one of the options learned. The episode finishes after the agent takes four actions from the option. The option reduces the number of agent decisions from 39 to only 10.

$$0, \underbrace{2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 2, 2}_{\text{Option 1}}, 1, \underbrace{0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0}_{\text{Option 1}}, 2, 2, 2, 2, 2, \underbrace{0, 2, 2, 2}_{\text{Option 1}}$$

Similarly, we show sub-sequences of an episode of the Dec-Options agents in the ComboGrid.

ComboGrid 3×3 (full trajectory):

$$\underbrace{\overbrace{0,2,2,1}^{\text{Down}}}_{\text{Option 3}},\underbrace{\overbrace{0,0,1,1}^{\text{Up}}}_{\text{Option 2}},\underbrace{\overbrace{0,0,1,1}^{\text{Up}}}_{\text{Option 2}},\underbrace{\overbrace{1,2,1,0}^{\text{Right}}}_{\text{Option 2}},\overbrace{0,\ \underbrace{2,2,1}_{\text{Option 4}}}^{\text{Down}},\underbrace{\overbrace{1,0,2,2}^{\text{Left}}}_{\text{Option 2}},\overbrace{1,\ \underbrace{0,2,2}_{\text{Option 4}}}^{\text{Left}}$$

ComboGrid 4×4:

$$\underbrace{\overbrace{0,2,2,1}^{\text{Down}}}_{\text{Option 2}},\underbrace{\overbrace{1,2,1,0}^{\text{Right}}}_{\text{Option 2}},\overbrace{\underbrace{0,0}_{\text{Option 1}}\ ,\ \underbrace{1,1}_{\text{Option 1}}}^{\text{Up}},\underbrace{\overbrace{1,0,2,2}^{\text{Left}}}_{\text{Option 2}},\underbrace{\overbrace{0,2,2,1}^{\text{Down}}}_{\text{Option 2}},\underbrace{\overbrace{0,2,2,1}^{\text{Down}}}_{\text{Option 2}},\underbrace{\overbrace{0,2,2,1}^{\text{Down}}}_{\text{Option 2}},\overbrace{1,\ \underbrace{0,2}_{\text{Option 1}}\ ,2}^{\text{Left}}$$

ComboGrid 5×5:

$$\overbrace{0,0,1,1}^{\text{Up}},\overbrace{0,0,1,\ 1}^{\text{Up}},\overbrace{1\ ,\ 2,1,0}^{\text{Right}},\overbrace{1,2,1\ ,\ 0}^{\text{Right}},\overbrace{0\ ,\ 2,2\ ,\ 1}^{\text{Down}},\overbrace{0\ ,\ 2,2\ ,1}^{\text{Down}}$$

$$\underbrace{0,0,1,1,0,0,1}_{\text{Option 2}},\ \underbrace{1,1}_{\text{Option 1}}\ ,\underbrace{2,1,0}_{\text{Option 5}},\underbrace{1,2,1}_{\text{Option 5}},\ \underbrace{0,0}_{\text{Option 4}}\ ,\ \underbrace{2,2}_{\text{Option 4}}\ ,\ \underbrace{1,0}_{\text{Option 4}}\ ,\ \underbrace{2,2}_{\text{Option 3}}\ ,1$$

ComboGrid 6×6:

$$\underbrace{\overbrace{1,0,2,2}^{\text{Left}},\overbrace{1,0,2,2}^{\text{Left}}}_{\text{Option 2}},\overbrace{\underbrace{0,0}_{\text{Option 3}}\ ,\ \underbrace{1,1}_{\text{Option 1}}}^{\text{Up}},\overbrace{\underbrace{0,0}_{\text{Option 3}}\ ,\ \underbrace{1,1}_{\text{Option 1}}}^{\text{Up}},\overbrace{\underbrace{0,0}_{\text{Option 3}}\ ,\ \underbrace{1,1}_{\text{Option 1}}}^{\text{Up}},\overbrace{\underbrace{1,2,1}_{\text{Option 4}}\ ,0}^{\text{Right}}$$

It can be shown that in the grid of size 3×3, the learned options almost match exactly the dynamics of the problem, except for Option 4 which only partially executes the sequence of actions that allow the agent to move left.

For Combogrid of size 4×4, option 2 learns sequences of actions that move the agent to another cell on the grid (e.g., "Down" and "Right"). Option 1 is shorter than Option 2 and it applies in many situations. For example, calling Option 1 twice can move the agent up, or even finish a left move to then move up.

In the larger grids of size 5×5 and 6×6, Dec-Options learns longer and more complicated options. For instance, for 5×5, Option 2 can almost complete the sequence of actions to go up twice, while for 6×6, Option 2 can perform the sequence of actions to go left twice.

Heatmap Analysis We employed heatmaps to visually depict the distribution of visited cells within the MiniGrid environment during the training process of tasks

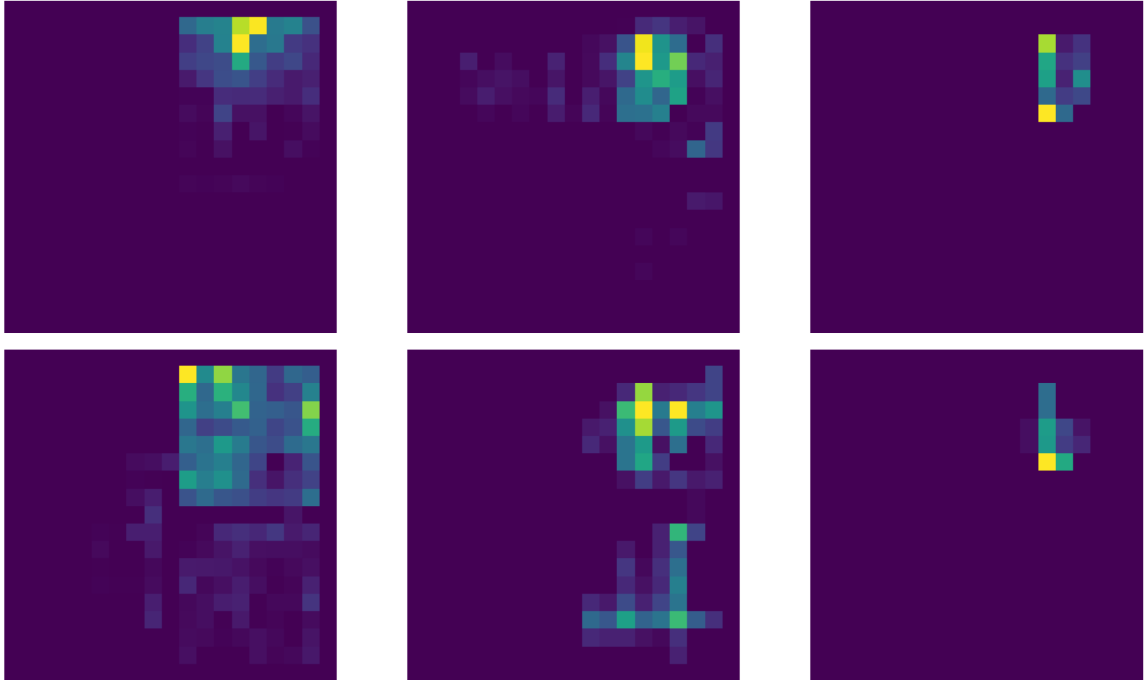


Figure 4.8: This figure displays the heatmap of visited cells in the Four Rooms 1 environment for both the Dec-Options and Vanilla-RL methods. The first row depicts the Dec-Options method's heatmap, while the second row shows the Vanilla-RL method. The first, second, and third columns show the heatmap after 1, 3200, and 12800 steps, respectively.

in $\mathcal{P}$. These heatmaps offer insights into the states that the agent explores at various training stages. A comparison of the heatmaps generated by the Dec-Options method with those produced by Vanilla-RL, as shown in Figure 4.8, reveals contrasting exploration patterns and their influence on task completion. Our findings reveal that an agent equipped with Dec-Options explores the space more deliberately, leading to a quicker discovery of the goal state. In contrast, traditional methods like Vanilla-RL tend to distribute their visits uniformly across all cells in the early stages and visit each of those states many times. For example, in the first two columns of Figure 4.8, the Dec-Options agent mostly explores the cells in the room where the goal is located. While the Vanilla-RL agent spends some explorations to explore other rooms.

Chapter 5

Conclusions & Future Work

5.1 Conclusions

In this research, we introduce a novel method called Dec-Options which serves as a means to discover options by extracting partial policies from neural networks. Our work showcases the practical utility of the sub-policies, particularly in the context of option discovery. Through experimentation and analysis, we conclude that sub-policies that can be turned into options are present in neural policies, revealing the feasibility of extracting them directly from the network architecture. This offers a perspective on the potential for neural networks to have options encoded in themselves.

Given that the number of options we can extract from neural networks grows exponentially with the number of neurons in the network, we employ a greedy procedure to select a subset of these options. This selection process is guided by the minimization of the Levin loss, a critical metric in our methodology. Minimizing the Levin loss not only optimizes the selection of sub-policies but also augments the likelihood that the agent will execute sequences of actions leading to high-reward states, based on past task experiences.

To validate the effectiveness of our approach, we conducted experiments on grid-world problems that demand intricate exploration. The outcomes of these experiments provide support for our hypothesis: options extracted from neural networks

encoding policies for a set of tasks can accelerate the learning process in similar but different tasks. These results underscore the practical value of our approach, illuminating its potential to expedite learning and problem-solving in complex scenarios.

5.2 Future Work

In terms of future research, our method presents opportunities for exploration in several directions. One avenue involves scaling the approach to larger neural networks. This investigation could provide valuable insights into the method's performance when dealing with larger models, contributing to a better understanding of its generalizability. An answer to the scalability of the method, and an interesting future work lies in delving into combinatorial search techniques within the space of sub-policies. By exploring strategies for refined and targeted searches in this space, we can potentially enhance the efficiency and effectiveness of our approach, leading to improved learning and knowledge transfer.

Extending the application of our method to more intricate neural network architectures, particularly convolutional neural networks (CNNs), and Long Short-Term Memory (LSTM), is another promising direction. Evaluating its performance in the context of CNNs could offer insights into its adaptability across diverse network structures and its applicability to different types of tasks.

Additionally, there is a compelling opportunity to conduct an analysis of the semantics embedded within the extracted sub-policies. Understanding the learned representations and behaviors of these sub-policies can provide deeper insights into the decision-making processes of the neural network, contributing to a more interpretable understanding of the knowledge encoded within the network.

Furthermore, the neural decomposition method introduced in our work serves as a general approach for extracting information from neural networks. Investigating the broader applicability and potential extensions of this approach, for example in program synthesis, could be a compelling avenue.

In summary, future research could explore these avenues to further advance the understanding of neural network behaviors, enhance the applicability and scalability of our method, and application of our method in other research areas.

Bibliography

[1] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[2] D. André and S. Markovitch, "Learning representations for the transfer of learning," in *Proceedings of the International Joint Conference on Artificial Intelligence*, vol. 5, 2005, pp. 415–420.

[3] M. E. Taylor and P. Stone, "Transfer learning for reinforcement learning domains: A survey," *Journal of Machine Learning Research*, vol. 10, pp. 1633–1685, 2009.

[4] D. L. Silver, Q. Yang, and L. Li, "Lifelong machine learning systems: Beyond learning algorithms," in *2013 AAAI spring symposium series*, 2013.

[5] R. S. Sutton, D. Precup, and S. Singh, "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning," *Artificial intelligence*, vol. 112, no. 1-2, pp. 181–211, 1999.

[6] D. Precup, *Temporal abstraction in reinforcement learning*. University of Massachusetts Amherst, 2000.

[7] R. Bellman, "A markovian decision process," *Journal of mathematics and mechanics*, pp. 679–684, 1957.

[8] M. L. Puterman, *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[9] G. Konidaris and A. Barto, "Building portable options: Skill transfer in reinforcement learning," in *International Joint Conference on Artificial Intelligence*, 2007, pp. 895–900.

[10] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[11] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the International Conference on International Conference on Machine Learning*, 2010, pp. 807–814.

[12] V. Mnih *et al.*, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[13] G. Montúfar, R. Pascanu, K. Cho, and Y. Bengio, "On the number of linear regions of deep neural networks," in *Proceedings of the International Conference on Neural Information Processing Systems*, Cambridge, MA, USA: MIT Press, 2014, 2924–2932.

[14] L. Zhang, G. Naitzat, and L.-H. Lim, "Tropical geometry of deep neural networks," in *Proceedings of the International Conference on Machine Learning*, J. Dy and A. Krause, Eds., ser. Proceedings of Machine Learning Research, vol. 80, PMLR, 2018, pp. 5824–5832.

[15] G.-H. Lee and T. S. Jaakkola, "Oblique decision trees from derivatives of relu networks," in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=Bke8UR4FPB.

[16] M. Raghu, B. Poole, J. Kleinberg, S. Ganguli, and J. S. Dickstein, "On the expressive power of deep neural networks," in *Proceedings of the International Conference on Machine Learning*, 2017, pp. 2847–2854.

[17] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and regression trees*. CRC press, 1986.

[18] L. Orseau, L. H. S. Lelis, T. Lattimore, and T. Weber, "Single-agent policy tree search with guarantees," in *Proceedings of the International Conference on Neural Information Processing Systems*, Curran Associates Inc., 2018, 3205–3215.

[19] K. Frans, J. Ho, X. Chen, P. Abbeel, and J. Schulman, "Meta learning shared hierarchies," *arXiv preprint arXiv:1710.09767*, 2017.

[20] R. Parr and S. Russell, "Reinforcement learning with hierarchies of machines," *Advances in neural information processing systems*, vol. 10, 1997.

[21] C. Tessler, S. Givony, T. Zahavy, D. Mankowitz, and S. Mannor, "A deep hierarchical approach to lifelong learning in minecraft," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 31, 2017.

[22] J. Andreas, D. Klein, and S. Levine, "Modular multitask reinforcement learning with policy sketches," in *International conference on machine learning*, PMLR, 2017, pp. 166–175.

[23] P.-L. Bacon, J. Harb, and D. Precup, "The option-critic architecture," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 31, 2017.

[24] J. Achiam, H. Edwards, D. Amodei, and P. Abbeel, "Variational option discovery algorithms," *arXiv preprint arXiv:1807.10299*, 2018.

[25] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *arXiv preprint arXiv:1312.6114*, 2013.

[26] M. C. Machado, C. Rosenbaum, X. Guo, M. Liu, G. Tesauro, and M. Campbell, "Eigenoption discovery through the deep successor representation," in *International Conference on Learning Representations (ICLR)*, 2018.

[27] M. C. Machado, A. Barreto, D. Precup, and M. Bowling, "Temporal abstraction in reinforcement learning with the successor representation," *Journal of Machine Learning Research*, vol. 24, no. 80, pp. 1–69, 2023.

[28] P. Dayan, "Improving generalization for temporal difference learning: The successor representation," *Neural computation*, vol. 5, no. 4, pp. 613–624, 1993.

[29] Y. Jinnai, J. W. Park, M. C. Machado, and G. Konidaris, "Exploration in reinforcement learning with deep covering options," in *International Conference on Learning Representations (ICLR)*, 2020.

[30] M. Klissarov and M. C. Machado, "Deep laplacian-based options for temporally-extended exploration," in *Proceedings of the International Conference on Machine Learning*, JMLR.org, 2023.

[31] W. Dabney, G. Ostrovski, and A. Barreto, "Temporally-extended {\epsilon}-greedy exploration," *arXiv preprint arXiv:2006.01782*, 2020.

[32] M. Igl *et al.*, "Multitask soft option learning," in *Conference on Uncertainty in Artificial Intelligence*, PMLR, 2020, pp. 969–978.

[33] G. I. Parisi, R. Kemker, J. L. Part, C. Kanan, and S. Wermter, "Continual lifelong learning with neural networks: A review," *Neural networks*, vol. 113, pp. 54–71, 2019.

[34] S. Powers, E. Xing, E. Kolve, R. Mottaghi, and A. Gupta, "Cora: Benchmarks, baselines, and metrics as a platform for continual reinforcement learning agents," in *Conference on Lifelong Learning Agents*, PMLR, 2022, pp. 705–743.

[35] J. Kirkpatrick *et al.*, "Overcoming catastrophic forgetting in neural networks," *Proceedings of the national academy of sciences*, vol. 114, no. 13, pp. 3521–3526, 2017.

[36] S. Kessler, J. Parker-Holder, P. Ball, S. Zohren, and S. J. Roberts, "Same state, different task: Continual reinforcement learning without interference," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 36, 2022, pp. 7143–7151.

[37] D. Rolnick, A. Ahuja, J. Schwarz, T. Lillicrap, and G. Wayne, "Experience replay for continual learning," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[38] S. Narvekar, B. Peng, M. Leonetti, J. Sinapov, M. E. Taylor, and P. Stone, "Curriculum learning for reinforcement learning domains: A framework and survey," *The Journal of Machine Learning Research*, vol. 21, no. 1, pp. 7382–7431, 2020.

[39] A. Clegg, W. Yu, Z. Erickson, J. Tan, C. K. Liu, and G. Turk, "Learning to navigate cloth using haptics," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2017, pp. 2799–2805.

[40] K. Shao, Y. Zhu, and D. Zhao, "Starcraft micromanagement with reinforcement learning and curriculum transfer learning," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 3, no. 1, pp. 73–84, 2018.

[41] E. Ben-Iwhiwhu, S. Nath, P. K. Pilly, S. Kolouri, and A. Soltoggio, "Lifelong reinforcement learning with modulating masks," *arXiv preprint arXiv:2212.11110*, 2022.

[42] M. Wortsman *et al.*, "Supermasks in superposition," *Advances in Neural Information Processing Systems*, vol. 33, pp. 15 173–15 184, 2020.

[43] A. A. Rusu *et al.*, "Progressive neural networks," *arXiv preprint arXiv:1606.04671*, 2016.

[44] J. Yoon, E. Yang, J. Lee, and S. J. Hwang, "Lifelong learning with dynamically expandable networks," *arXiv preprint arXiv:1708.01547*, 2017.

[45] J. Schwarz *et al.*, "Progress & compress: A scalable framework for continual learning," in *International conference on machine learning*, PMLR, 2018, pp. 4528–4537.

[46] K. Khetarpal, M. Riemer, I. Rish, and D. Precup, "Towards continual reinforcement learning: A review and perspectives; 2020," *URL https://arxiv.org/abs/2012*, vol. 13490, 2012.

[47] L. Kirsch, J. Kunze, and D. Barber, "Modular networks: Learning to decompose neural computation," *Advances in neural information processing systems*, vol. 31, 2018.

[48] A. Goyal, S. Sodhani, J. Binas, X. B. Peng, S. Levine, and Y. Bengio, "Reinforcement learning with competitive ensembles of information-constrained primitives," *arXiv preprint arXiv:1906.10667*, 2019.

[49] J. A. Mendez, H. van Seijen, and E. Eaton, "Modular lifelong reinforcement learning via neural composition," *arXiv preprint arXiv:2207.00429*, 2022.

[50] W. Qiu and H. Zhu, "Programmatic reinforcement learning without oracles," in *International Conference on Learning Representations*, 2021.

[51] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, vol. 8, no. 3-4, pp. 229–256, 1992.

[52] V. Konda and J. Tsitsiklis, "Actor-critic algorithms," in *Advances in Neural Information Processing Systems*, S. Solla, T. Leen, and K. Müller, Eds., vol. 12, MIT Press, 1999, pp. 1008–1014.

[53] T. D. Kulkarni, K. Narasimhan, A. Saeedi, and J. Tenenbaum, "Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation," in *Advances in Neural Information Processing Systems*, D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, Eds., vol. 29, Curran Associates, Inc., 2016.

[54] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[55] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[56] M. Chevalier-Boisvert *et al.*, "Minigrid & miniworld: Modular & customizable reinforcement learning environments for goal-oriented tasks," *CoRR*, vol. abs/2306.13831, 2023.

[57] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-baselines3: Reliable reinforcement learning implementations," *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021. [Online]. Available: http://jmlr.org/papers/v22/20-1364.html.

[58] S. M. LaValle, M. S. Branicky, and S. R. Lindemann, "On the relationship between classical grid search and probabilistic roadmaps," *The International Journal of Robotics Research*, vol. 23, no. 7-8, pp. 673–692, 2004.

Appendix A: Experiment Details

A.1 Plots

The figures presented in Section 4.3 were generated following a standardized methodology, ensuring consistency and robustness across all experiments. This methodology involved a well-defined procedure:

- We initiated the process with a comprehensive hyperparameter search for each baseline. This search enabled us to select the most effective hyperparameters for the respective algorithms.
- We conducted training for each baseline using 30 different seeds.
- We evaluated the performance of the seeds and identified the six seeds with the poorest results. These six seeds were subsequently discarded from further analysis.
- The remaining 24 seeds, which demonstrated more promising results, were used to calculate the mean performance. Additionally, we computed the 95% confidence intervals for each baseline.

A.2 Architecture and Parameter Search

For algorithms utilizing the Proximal Policy Optimization (PPO) framework, such as **Vanilla-RL**, **Neural-Augmented**, **Transfer-PPO**, **PNN**, **ez-greedy**, **Dec-Options-Whole**, and **Dec-Options**, we relied on the stable-baselines library [57].

This library provided the foundation for training and evaluation. For the **PNN** method, we integrated the code from (https://github.com/arcosin/Doric) and combined it with the PPO algorithm from the stable-baselines. The **Modulating-Mask** method utilized the code provided by the original authors (https://github.com/dlpbc/mask-lrl). For the **ez-greedy** algorithm, we integrated the temporally-extended ε-greedy algorithm with the PPO algorithm from the stable-baselines. We set the ε to be equal to 0.01 and the μ to be equal to 2, as they did the same in their work.

We conducted hyperparameter searches using a grid search method [58] for each algorithm, covering parameters clipping parameter, entropy coefficient, and learning rate. These parameters are reported for each domain in their respective sections. The optimal hyperparameters for each algorithm were selected based on this search.

In scenarios where the Deep Q-Network (DQN) was employed, such as **Vanilla-RL** and **Dec-Options** baselines, we used the stable-baselines framework. Similarly to PPO, we performed a parameter search, this time targeting Tau and the learning rate while keeping other parameters fixed. **Option-Critic** implementations were based on (https://github.com/lweitkamp/option-critic-pytorch), where we used the best parameters found for the learning rate and the number of options. All the parameter searches mentioned were performed using the grid search method. Also, for the codebase of **DCEO** baseline, we used the original paper's implementation (https://github.com/mklissa/dceo/). The parameter search is applied to the learning rate, the number of options, and the probability of option execution during training.

The choice of network architecture varied depending on the domain and the method used. In the MiniGrid domain, we employed feedforward networks for the policy and value networks. The policy network structure consisted of a single hidden layer comprising 6 nodes, while the value network used three hidden layers with sizes [256, 256, 256]. As tasks transitioned to $\mathcal{P}'$, we adapted the policy network structure to include more hidden layers and neurons to accommodate the increased complexity of the tasks. The selected size of deep neural network for tasks in $\mathcal{P}'$ was set to [50, 50,

50]. The **Transfer-PPO** method maintained a consistent policy network structure of size [50, 50, 50] across all tasks in both $\mathcal{P}$ and $\mathcal{P}'$. The **Modulating-Mask** method retained the neural network structure as it existed in the original implementation, which included a shared feature network with three hidden layers of sizes [200, 200, 200], followed by one hidden layer of size [200] for policy, and one hidden layer of size [200] for value networks. DQN-based methods, **Vanilla-RL** and **Dec-Options**, along with **Option-Critic** and **DCEO**, the neural network structure remained consistent with three hidden layers, each with sizes [200, 200, 200].

In the ComboGrid domain, we applied a methodology consistent with the MiniGrid domain to maintain a standardized approach. For tasks in $\mathcal{P}$, the policy network featured a single hidden layer with 6 nodes. Similar to MiniGrid, For tasks in $\mathcal{P}'$, we increased the policy network structure to include one hidden layer with 16 nodes. The **Transfer-PPO** method maintained a uniform policy network structure across all tasks in both $\mathcal{P}$ and $\mathcal{P}$. This structure consisted of one hidden layer with 16 nodes. The value network structure for tasks in both $\mathcal{P}$ and $\mathcal{P}'$ featured three hidden layers with sizes [200, 200, 200]. As for the **Modulating-Mask** method, the neural network structure is as before.

For methods that employed DQN, such as **Vanilla-RL**, **Dec-Options**, and **Option-Critic**, we adopted a consistent neural network structure. This structure included two hidden layers with sizes [32, 64].

The discounting factor for all tasks in $\mathcal{P}'$ is set to 0.99.

Appendix B: Parameter Search

In pursuit of reasonable hyperparameters, we performed a parameter search for all methods used in our experiments. We have outlined the search spaces for each method below.

B.1 MiniGrid

For methods using the PPO algorithm, we searched over the following hyperparameters:

- **Learning Rate:** 0.005, 0.001, 0.0005, 0.0001, 0.00005
- **Clipping Parameter:** 0.1, 0.15, 0.2, 0.25, 0.3
- **Entropy Coefficient:** 0.0, 0.05, 0.1, 0.15, 0.2

For methods relying on the Deep Q-Network (DQN) paradigm, our parameter search encompassed the following hyperparameter lists:

- **Learning Rate:** 0.01, 0.005, 0.001, 0.0005, 0.0001
- **Tau:** 1., 0.7, 0.4, 0.1

In the case of the Option-Critic method, we searched over the following hyperparameters:

- **Learning Rate:** 0.01, 0.005, 0.001, 0.0005, 0.0001, 0.00005
- **Number of Options** 2, 3, 4, 5, 6

As for the DCEO baseline, we searched over the following hyperparameters:

- **Learning Rate:** 0.01, 0.005, 0.001, 0.0005, 0.0001, 0.00005
- **Number of Options:** 3, 5, 10
- **Option Probability:** 0.2, 0.7, 0.9

	Clipping	**Entropy Coeff.**	**Learning Rate**
Vanilla-RL	0.15	0.05	0.0005
Neural-Augmented	0.2	0.2	0.0005
Transfer-PPO	0.15	0.15	0.0001
PNN	0.1	0.0	0.0001
Modulating-Mask	0.15	0.1	0.001
ez-greedy	0.15	0.05	0.0005
Dec-Options-Whole	0.3	0.15	0.0005
Dec-Options	0.25	0.1	0.0005
Open-Loop	0.25	0.05	0.001

Table B.1: Four Rooms 1 best parameters - PPO

	Tau/Number of Options	**Learning Rate**
Vanilla-RL	0.7	0.0001
Option-critic	2	0.0001
Dec-Options	0.7	0.0005
DCEO	5 (Probability: 0.9)	0.0005

Table B.2: Four Rooms 1 best parameters - DQN

B.2 ComboGrid

For methods utilizing the PPO algorithm, we searched over the following hyperparameter ranges:

	Clipping	Entropy Coeff.	Learning Rate
Vanilla-RL	0.1	0.2	0.0005
Neural-Augmented	0.15	0.0	0.0005
Transfer-PPO	0.1	0.05	5e-05
PNN	0.25	0.0	0.005
Modulating-Mask	0.2	0.0	0.01
ez-greedy	0.1	0.0	0.0001
Dec-Options-Whole	0.25	0.05	0.0005
Dec-Options	0.2	0.1	0.001
Open-Loop	0.15	0.0	0.0005

Table B.3: Four Rooms 2 best parameters - PPO

	Tau/Number of Options	Learning Rate
Vanilla-RL	1.0	0.0005
Option-critic	2	0.0001
Dec-Options	1.0	0.001
DCEO	10 (Probability: 0.9)	0.001

Table B.4: Four Rooms 2 best parameters - DQN

- **Learning Rate:** 0.05, 0.01, 0.005, 0.001
- **Clipping Parameter** 0.05, 0.1, 0.15, 0.2, 0.25, 0.3
- **Entropy Coefficient:** 0.0, 0.05, 0.1, 0.15, 0.2

As for the Modulating-Mask method, we explored the following hyperparameters:

- **Learning Rate:** 0.005, 0.001, 0.0005, 0.0001, 0.00005
- **Clipping Parameter** 0.1, 0.15, 0.2, 0.25
- **Entropy Coefficient:** 0.0, 0.05, 0.1, 0.15, 0.2

	Clipping	Entropy Coeff.	Learning Rate
Vanilla-RL	0.2	0.0	5e-05
Neural-Augmented	0.1	0.0	0.001
Transfer-PPO	0.1	0.05	0.0001
PNN	0.15	0.0	0.001
Modulating-Mask	0.1	0.0	0.005
ez-greedy	0.25	0.05	0.0005
Dec-Options-Whole	0.15	0.05	0.001
Dec-Options	0.2	0.1	0.001
Open-Loop	0.2	0.0	0.001

Table B.5: Four Rooms 3 best parameters - PPO

	Tau/Number of Options	Learning Rate
Vanilla-RL	0.7	0.001
Option-critic	3	0.0001
Dec-Options	0.1	0.0005
DCEO	10 (Probability: 0.9)	0.0001

Table B.6: Four Rooms 3 best parameters - DQN

For methods relying on the Deep Q-Network (DQN) paradigm, our parameter search encompassed the following hyperparameter lists:

- **Learning Rate:** 0.05, 0.01, 0.005, 0.001, 0.0005, 0.0001
- **Tau:** 1., 0.7, 0.4, 0.1

In the case of the Option-Critic method, we searched over the following hyperparameter combinations:

- **Learning Rate:** 0.01, 0.005, 0.001, 0.0005, 0.0001, 0.00005
- **Number of Options** 2, 3, 4, 5, 6

For the DCEO baseline, we explored the following hyperparameter combinations:

- **Learning Rate:** 0.01, 0.005, 0.001, 0.0005, 0.0001
- **Number of Options:** 3, 5, 10
- **Option Probability:** 0.2, 0.7, 0.9

	Clipping	Entropy Coeff.	Learning Rate
Vanilla-RL	0.15	0.1	0.01
Neural-Augmented	0.25	0.05	0.01
Transfer-PPO	0.3	0.05	0.001
PNN	0.2	0.1	0.01
Modulating-Mask	0.1	0.1	0.0005
ez-greedy	0.15	0.05	0.005
Dec-Options-Whole	0.15	0.05	0.005
Dec-Options	0.2	0.05	0.005
Open-Loop	0.1	0.0	0.005

Table B.7: ComboGrid 3x3 best parameters - PPO

	Tau/Number of Options	Learning Rate
Vanilla-RL	1.	0.001
Option-critic	2	0.001
Dec-Options	1.	0.001
DCEO	10 (Probability: 0.2)	0.0005

Table B.8: ComboGrid 3x3 best parameters - DQN

	Clipping	**Entropy Coeff.**	**Learning Rate**
Vanilla-RL	0.1	0.0	0.005
Neural-Augmented	0.3	0.0	0.005
Transfer-PPO	0.05	0.2	0.001
PNN	0.2	0.1	0.005
Modulating-Mask	0.25	0.15	0.0001
ez-greedy	0.2	0.05	0.005
Dec-Options-Whole	0.25	0.05	0.01
Dec-Options	0.25	0.0	0.005
Open-Loop	0.1	0.0	0.005

Table B.9: ComboGrid 4x4 best parameters - PPO

	Tau/Number of Options	**Learning Rate**
Vanilla-RL	0.7	0.0005
Option-critic	3	0.0005
Dec-Options	1.	0.0005
DCEO	3 (Probability: 0.2)	0.0005

Table B.10: ComboGrid 4x4 best parameters - DQN

	Clipping	**Entropy Coeff.**	**Learning Rate**
Vanilla-RL	0.25	0.1	0.005
Neural-Augmented	0.15	0.0	0.005
Transfer-PPO	0.1	0.2	0.001
PNN	0.25	0.05	0.005
Modulating-Mask	0.2	0.05	0.001
ez-greedy	0.2	0.15	0.005
Dec-Options-Whole	0.2	0.05	0.005
Dec-Options	0.2	0.05	0.005
Open-Loop	0.1	0.0	0.01

Table B.11: ComboGrid 5x5 best parameters - PPO

	Tau/Number of Options	**Learning Rate**
Vanilla-RL	0.7	0.001
Option-critic	4	0.0001
Dec-Options	1.	0.001
DCEO	5 (Probability: 0.7)	0.001

Table B.12: ComboGrid 5x5 best parameters - DQN

	Clipping	Entropy Coeff.	Learning Rate
Vanilla-RL	0.1	0.05	0.005
Neural-Augmented	0.2	0.05	0.005
Transfer-PPO	0.3	0.05	0.001
PNN	0.05	0.0	0.005
Modulating-Mask	0.15	0.0	0.005
ez-greedy	0.2	0.05	0.005
Dec-Options-Whole	0.2	0.0	0.001
Dec-Options	0.15	0.05	0.005
Open-Loop	0.15	0.0	0.005

Table B.13: ComboGrid 6x6 best parameters - PPO

	Tau/Number of Options	Learning Rate
Vanilla-RL	0.7	0.001
Option-critic	2	0.005
Dec-Options	0.7	0.001
DCEO	3 (Probability: 0.9)	0.0001

Table B.14: ComboGrid 6x6 best parameters - DQN