UC San Diego
UC San Diego Electronic Theses and Dissertations

Title
Discovering Useful Behaviour in Reinforcement Learning

Permalink
https://escholarship.org/uc/item/46x7q9br

Author
Mavalankar, Aditi Ashutosh

Publication Date
2022

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA SAN DIEGO

Discovering Useful Behaviour in Reinforcement Learning

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy

in

Computer Science

by

Aditi Ashutosh Mavalankar

Committee in charge:

Professor Lawrence Saul, Chair
Professor Sanjoy Dasgupta
Professor Tara Javidi
Professor Julian McAuley
Professor Doina Precup

2022

Copyright

Aditi Ashutosh Mavalankar, 2022

All rights reserved.

The Dissertation of Aditi Ashutosh Mavalankar is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2022

DEDICATION

To my family,

Because a loving and supportive family

is the greatest privilege one can have.

EPIGRAPH

The true sign of intelligence
is not knowledge
but imagination.

Albert Einstein

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I want to begin by expressing my immense gratitude to my advisor, Lawrence Saul, for taking me on as his PhD student during one of the most turbulent stages of my PhD. Not only did he give me full freedom to pursue my interests, but he also kept motivating me throughout my journey as an independent graduate student researcher; for that, I am extremely grateful.

I also want to thank Doina Precup, Sanjoy Dasgupta, Tara Javidi, and Julian McAuley, for serving on my committee and for their useful feedback on my work.

I came to the United States in 2016 for my Masters' degree, having greatly enjoyed my time as a summer intern in Sarah Creel's lab in the Cognitive Science department; thanks, Sarah, for this chance! Prof. Christine Alvarado, at this time, gave me the opportunity to be the graduate student mentor of the Early Research Scholars' Program for the 2016-2017 and 2017-2018 academic years. This program enabled me to develop an array of skills including mentoring students, learning basics across a breadth of research areas faster, developing trust and facilitating open communication among research groups as well as with me and their research advisors. I later also got the opportunity to collaborate with Christine, and also Gerald Soosai Raj and Adrian Salguero, on Google ExploreCSR 2020, to increase participation of students from underrepresented communities in Computer Science research. Christine has been a major reason behind my decision to pursue a PhD, and she believed in my ability to do research and lead a team long before I believed it was possible; thank you, Christine, for being such an inspiring person I have looked up to, and for always pushing me to achieve the best!

I spent two wonderful summers (2017 and 2018) at Amazon Lab126 in Sunnyvale, where I was one of the first interns to work on the very cool Amazon Astro. Thanks to my amazing colleagues and the senior leadership for making both internships such memorable experiences!

I also want to thank Sicun Gao and Garrison Cottrell for being my advisors for about a year each before we parted ways.

The only in-person conference I was able to attend during my PhD was ICML 2019, during which I met several accomplished researchers; I had the pleasure of meeting Jonathan J.

Hunt, with whom I engaged in very interesting discussions during and after the conference, and who also helped review a draft of my first paper, which I greatly appreciate. I also would like to acknowledge Devi Parikh, Yao Qin, Khimya Khetarpal, and Shane Gu for acting as my mentors at several points in my PhD and giving me useful career and research advice.

A major part of the work in this thesis was done in collaboration with researchers from DeepMind, where I had the steepest learning curve throughout my research journey. It was at DeepMind that I was able to shape my overall research direction, and where I received extremely useful guidance from some of the best researchers in the world; I will attempt to thank them here.

I am very grateful to Doina Precup for giving me the opportunity to work under her guidance as a Research Scientist intern at DeepMind twice during my PhD. Doina gave me this chance when I was just getting started in the field, and was looking for guidance and collaborations; she has continued to be a source of inspiration to me since then, and it has been an honour for me to get to work with her.

I want to express my deep gratitude to André Barreto, with whom I collaborated before, during, and after, my internships at DeepMind. I reached out to André sometime in 2020 with ideas to extend his work on the Option Keyboard, and he patiently walked me through converting my half-baked ideas into a concrete approach. Ever since that time, he has been a very supportive collaborator, mentor, and friend, and has looked out for me and promoted my work several times. Many thanks to my friend and collaborator at DeepMind, Zach Holland, for stepping in to help in any capacity when the project was in its most uncertain stage, and also for all the brainstorming sessions about environment design and rigour in experiments; Zach's support and thoughtful critique was instrumental in maintaining the momentum of the project on several occasions. Many thanks also to David Abel, for all the brainstorming sessions on approaching the problem from a theoretical perspective, for giving me very valuable career advice, and for being such a fun person to work with. Many thanks to Diana Borsa for all the brainstorming sessions on both projects we worked on together; I greatly enjoyed our collaboration and learnt a lot from her during several times when we had differing perspectives on the problem.

I also want to thank Zita Marinho, who was my closest collaborator in my second internship at DeepMind, in addition to being a very supportive friend during my job search. I have learnt a lot during my collaboration with Zita, and I hope that I can continue to work with and learn from her going forward. Many thanks also to Tom Schaul, who facilitated my collaboration with Zita, and whose valuable inputs and insights helped shape the course of our project. Also thanks to other wonderful people at DeepMind, who have acted as my mentors and given me a bunch of useful advice at different stages: Marlos C. Machado, Feryal Behbahani, Isabela Albuquerque, and Gheorghe Comanici. Many thanks to the awesome friends I made during my time in Montréal: Abbas, Alexandre, Anita, Ankit, Ivana, Kory, Mariana, Rachael, Owen, Zhitao, and the rest of the DeepMind Montréal team.

This lonely and somewhat debilitating experience of doing a PhD independently has been made tolerable, even enjoyable, because of the incredible friendships that I have come to cherish over the years. Infinite thanks to Nishant Bhaskar, who has been akin to an older brother, always there to help me out in any bad situation I found myself in, all throughout my PhD; to Vraj Shah, who has been my best friend throughout graduate school, and has always managed to lift up my spirits in my worst failures by reminding me about my abilities and achievements; to Aashka Vora, with whom I reunited after spending several years in kindergarten and school together, and with whom, due to our almost perfectly aligned interests and opinions, I have spent many enjoyable weekends hanging out, playing games, having co-reading sessions, binge-watching movies and TV shows, sharing experiences, burdens, and joys, just like we did in our school days; to Jay Sheth, who has provided unlimited entertainment throughout the last six years with his pure awesomeness and also our many deep conversations about work and life; to Nimish Srivastava, for always stepping up to help me, and anyone for that matter, during moments of crisis, and for being an incredibly fun person to be around; to Ajitesh Gupta, for being a constant friend throughout my Masters and even after, during my internships in the Bay Area, and for those reminiscences of IIIT times; to Rishikesh Ghewari, Pranay Surana, Ujjwal Gulecha for our many stress-busting game nights during my time in the Bay and also later during

Covid; to Sophia Sun, with whom I bonded most closely due to a shared sorrow during our PhD experiences, and whom I have come to know and admire much better when she was my roommate, through numerous conversations about work, our social lives, and several other things; to Zhi Wang, with whom I also bonded due to the same shared sorrow, and who always makes fun of me and I of him (I will miss that terribly); to Geelon So, whom the three of us (Sophia, Zhi, and I) assimilated, and have had several enjoyable times and stimulating conversations with in the lab and outside; to Amanda Song for gifting me her TV, which was a bigger part of me being happy during grad school than I care to admit; to Mary Anne Smart for being my conference buddy at ICML, and whom I have come to know as one of the nicest kindest people; to Jennifer Chien for constantly motivating me during my job search, and for putting in massive efforts in a direction I deeply care about, that of making computer science more inclusive and welcoming to everyone; to Saransh Gupta for being one of the nicest most helpful and positive people in the department; to Shubham Rathi for having stayed in touch with me years after IIIT, and for the deep intriguing philosophical discussions but also the fun shallow conversations; to Anurag Gupta, Saksham Aggarwal, Tanmay Sahay, Abhineet Jain, Chandan Kharbanda for the virtual Covid game nights; and to Animesh Shah, Khyati Shah, Gaurav Mahajan, Khalil Mrini, Yutong Shao, Bodhisattwa Majumder, Martha Gahl, Rex Lei, Akshat Singhal, Armita Ardeshiri, Vignesh Gokul, Yuda Song, Quan Vuong, Jean Choi, Megan Golbek, for all the fun times!

I also want to thank my relatives in California for providing familial support many times: my grand-uncle Dr. Kiran Magiawala, who bought me my first Macbook, and has always been in strong espousal of my educational endeavours; my uncle and aunt, Anil and Kavita Mavalankar, for giving me a home to stay during my internship at UC San Diego in 2015, and for subsequent enjoyable visits to their home during grad school; my father's friend Yatin Trivedi and his family for their great effort in helping me out in my job search, especially his son Mihir Trivedi, who is one of my closest friends in the US.

One crucial element of my PhD life that has helped me stay somewhat calm throughout the process is yoga. For this, I give full credit to my yoga teacher, Reena Patel, who has guided me into becoming more accepting of life and its challenges and uncertainties, and taught me that mental flexibility is the true desired goal in life, the achievement of which will expedite the attainment of goals in all aspects of life.

A PhD is, for so many of us who choose to go down this path, an intensely frustrating and challenging journey, while simultaneously being a liberating revelation. It is in these times that we fall back on those we love, trust, and ever so often regrettably take for granted, to manage us emotionally. So finally, while any words I attempt to write here will be a gross underestimation of their contribution to my life, the major credit for me being able to successfully complete my PhD goes to my family: my father Dr. Ashutosh Mavalankar, who is my role model, taught me that integrity, equanimity and humility are the most important things to have in life, and that any success that is achieved in the absence of any of these is of no consequence, and who I am privileged to have as my 24×7 personal therapist; my mother Sejal Mavalankar, for her unfailing conviction in my ability to achieve anything I want, her supreme efforts in ensuring my physical and mental well-being, and for always encouraging me to think positively regardless of the situation, and set the highest goals and standards for myself in my career and in life; my sister Amishi Mavalankar, who is the most practical sensible person I know and is also my best friend, and has welcomed me into her home for several months in the last few years with as much warmth and love as I have always had back at home in India; my brother-in-law Aakash Patel, who has given me very useful life advice during my most difficult periods in life, and who, along with my sister, has always given me a home to stay, and who also happens to be the best cook I know; my partner Abhinav Moudgil, who has managed to keep everything in control with his great sense of humour combined with his unassailable logic in the face of adversity, who has more confidence in my abilities than I have ever had, taught me to dive deep into new areas fearlessly, and as a result, got me excited about the field of machine learning, and who is my forever cheerleader.

Chapter 3 is based on the material as it appears in Beyond Tabula Rasa in Reinforcement Learning Workshop at the International Conference on Learning Representations, 2020

(Aditi Mavalankar, *Goal-conditioned Batch Reinforcement Learning for Rotation Invariant Locomotion*). The dissertation/thesis author was the primary investigator and author of this paper.

Chapter 4 is being prepared for submission (Aditi Mavalankar, André Barreto, David Abel, Diana L. Borsa, G. Zacharias Holland, Doina Precup, *Synthesizing the Option Keyboard: Option Discovery in Reinforcement Learning*). The dissertation/thesis author was the primary investigator and author of this paper.

Chapter 5 is being prepared for submission (Aditi Mavalankar*, Geelon So*, Yian Ma, Sanjoy Dasgupta, Doina Precup, *Lifelong Exploration in Infinite Graphs*). The dissertation/thesis author was the co-primary investigator and author of this paper.

VITA

2016	Bachelor of Technology (Honours) in Computer Science and Engineering International Institute of Information Technology, Hyderabad, India
2022	Master of Science in Computer Science University of California San Diego
2022	Doctor of Philosophy in Computer Science University of California San Diego

PUBLICATIONS

A. Mavalankar, A. Barreto, D. Abel, D. Borsa, G. Z. Holland, D. Precup, "Synthesizing the Option Keyboard: Option Discovery in Reinforcement Learning", *manuscript in preparation*, 2022.

A. Mavalankar*, G. So*, Y. Ma, S. Dasgupta, D. Precup, "Lifelong Exploration in Infinite Graphs", *manuscript in preparation*, 2022.

V. Gokul, **A. Mavalankar**, S. Dubnov, "The Option-Duet: Synthesizing Music using Options", *manuscript in preparation*, 2022.

A. Mavalankar, "Goal-conditioned batch reinforcement learning for rotation invariant locomotion", *International Conference on Learning Representations (ICLR) Beyond Tabula Rasa in RL Workshop*, 2020.

Y. Song, **A. Mavalankar**, W. Sun, S. Gao, "Provably Efficient Model-based Policy Adaptation", *International Conference on Machine Learning (ICML)*, 2020.

N. S. Uppara, **A. Mavalankar**, K. Vemuri, "Eye tracking in naturalistic badminton play: comparing visual gaze pattern strategy in world-rank and amateur players", *7th Workshop on Pervasive Eye Tracking and Mobile Eye-Based Interaction*, 2018.

V. Choppella, G. Ahuja, **A. Mavalankar**, "How does a program run? A visual model based on annotating abstract syntax trees", *International Conference on Learning and Teaching in Computing and Engineering*, 2016.

A. Mavalankar, T. Kelkar, V. Choppella, "Generation of Quizzes and Solutions Based on Ontologies – A Case for a Music Problem Generator", *International Conference on Technology for Education*, 2015.

A. Mavalankar, S. Dagar, K. Vemuri, "Decoding (un)known opponent's game play, a real-life badminton eye tracking study", *EuroAsianPacific Joint Conference on Cognitive Science*, 2015.

ABSTRACT OF THE DISSERTATION

Discovering Useful Behaviour in Reinforcement Learning

by

Aditi Ashutosh Mavalankar

Doctor of Philosophy in Computer Science

University of California San Diego, 2022

Professor Lawrence Saul, Chair

Applying reinforcement learning techniques to real-world problems as well as longstanding challenges has seen major successes in the last few years. In the reinforcement learning setting, an agent interacts with the environment, which gives it feedback in the form of a scalar reward signal. This reward signal may be available to the agent at every step, or it may be available after the agent has completed several subtasks in succession. Thus, it may be desirable for the agent to extract useful information from its interactions with the environment, that it can reuse to expedite learning.

In this dissertation, we will discuss three approaches for an agent to learn useful behaviour. Each of these three approaches will extract different kinds of useful information from the environment, and use it in a manner consistent with the reinforcement learning objective: maximizing the reward. The first approach discovers useful *representations* of the observation space by exploiting symmetries in the agent and environment. The second approach discovers useful *skills* that can be used in combination with each other to solve complex tasks. The third approach discovers useful *states* that can serve as good starting points for the agent to explore in very large environments. We suggest that all of these ways of discovering useful behaviour will be crucial in the development of intelligent agents.

Chapter 1

Introduction

Consider a student who wants to score well in her mathematics exam. In order to achieve that goal, there are several things that she would have to do, such as attending classes everyday to learn the subject, reading the required textbooks, solving problems to strengthen her understanding of the topics taught, maintaining a healthy lifestyle in terms of diet, sleep, and exercise, among others. Now assume she wants to do well in her physics exam. Would she be required to come up with a separate plan for this? No, since all of the things that she did in order to achieve her goal of doing well in mathematics would apply to physics as well. The course content would inevitably be different, but the skills involved would remain unchanged to a large extent. Of course, if the physics exam has a laboratory component, she would have to learn that part afresh, but a large part of her experience for preparing for the mathematics exam would be useful in preparing for her physics exam. This is an example of the class of Reinforcement Learning (RL) algorithms that we will discuss in this thesis: algorithms that enable the agent to discover different behaviours that are useful in reaching a goal, or maximizing the reward the agent gets from the world. We propose three possible ways to achieve this in the RL setting, and discuss them intuitively in the following paragraphs.

The first approach we propose caters to the scenario where preparing for the physics exam requires the student to do the same things as she did for the mathematics exam – attending classes, solving problems, maintaining good health. The only difference is in the topics studied in these two subjects. In this scenario, it would be useful for the student to get a good grasp of the more general concepts of how to study for a subject and how to acquire and refine her problem-solving abilities, regardless of the subject she is studying. The set of problems we address in the RL framework is analogous to this scenario: the agent is expected to exploit such *invariances* in the environment to generalize its performance over a number of tasks. Our proposed approach enables the agent to exhibit good generalization across tasks by learning latent representations that capture these invariances. The intuition is simple: any information that the agent can extract from the environment that helps it to solve the current task can be reused to solve another task from a larger set of tasks. Our approach condenses the high-dimensional observations seen by the agent into low-dimensional latent representations that retain task-invariant reward-maximizing properties. An interesting point to note here is that we as the algorithm designer, are incorporating an inductive bias into the learning process, which is our knowledge of the form these invariances take.

The second approach we propose is applicable to a more general scenario: the student now has several choices of things she could pursue – attending classes, solving problems, which are the same as the ones discussed earlier, but also those of learning how to dance, learning how to ride a bike, listening to music, watching television, learning how to cook food, learning how to play football, etc. However, the objective remains the same: she wants to score well in her mathematics exam. Since there are several choices at her disposal, she is free to choose the ones she believes will help her in achieving her goal. Evidently, she cannot pursue all activities listed, since that would be infeasible. The obvious two choices she would select are the ones that directly help her achieve her goal – attending mathematics classes and solving topic-related problems. However, it is possible that some of the other choices may indirectly help her reach her goal faster, or with more ease. For instance, it may be the case that dancing is her favourite hobby; since dancing is also a form of exercise, it helps her maintain good health, which makes her goal easier to achieve. Similarly, it may be the case that she is not very good at football and is afraid of being injured in the process of learning how to play, making it a bad activity for her to pursue. Thus, the problem to be solved is: what is the best set of things the student should pursue in order to achieve her goal? An important aspect of framing the problem in this particular way is that it allows for easy adaptation to changing goals. If the goal now changes to doing well on the physics exam, she need only swap out the mathematics-relevant activities with physics-relevant activities, and retain the other activities that helped her achieve her goal. The analogous RL problem is the option discovery problem: how can the agent autonomously discover options that enable it to achieve high reward in an accelerated timeframe? Here, options are analogous to the activities that the student can choose to pursue; each activity has its own associated option, and the problem for the student now is to discover which of these options are actually useful in achieving her goal. We propose an approach that allows the agent to discover which options, or combinations of options, are useful in maximizing the environment reward. An intuitive way to think of combinations of options is to consider the case when the student likes cooking and listening to music, but does not have the time to do them separately, and thus decides to listen to music while cooking. In this case, this is a useful combination of options. A bad combination could be listening to music while solving problems since it could reduce her concentration. Our proposed approach discovers a useful basis of options that the agent can use individually or in combination with each other to maximize the environment reward.

Finally, the third approach we propose has a slightly different flavour; say the student has to pick subjects that she wants to study in greater depth. In order to make this selection, she would consider studying subjects that are more rewarding to her in terms of her aptitude in them, her interest in them, or a mixture of both these factors. As a result, the student has to decide which subjects are worth pursuing for her, and discard others along the way. It is worth noting that this decision has to be made intermittently at several stages as she studies these subjects in more detail. For instance, if she chooses that physics and mathematics are most rewarding for her and chooses to explore areas in these fields, she will reach a phase where she has studied both these subjects in some detail and decides to pursue physics further. Since she has chosen physics and has several potential areas to study in physics like particle physics, nuclear physics, etc., she might not have the time to pursue specialized topics in mathematics. This is an instance of the class of problems we attempt to solve with our third approach. The analogous problem in RL is exploration – the agent has to continuously explore in the environment to find avenues that can lead to higher reward so that it does not get stuck, while exploiting its past experiences interacting with the environment. We propose solving this problem by learning a joint model of the environment dynamics and reward function to guide exploration. We represent this model in the form of a graph in which the nodes are the states visited by the agent and the edges represent the probability that the agent will explore in the corresponding part of the graph indicated by the edge. We combine the RL setting with methods in graph theory to derive a solution to this class of problems.

From the above motivating example, we now look at the question we want to answer in this thesis:

> *How can reinforcement learning agents, through their interactions with the environment, extract useful information that they can reuse to expedite learning in either a single environment or across environments?*

While this is a broad question, we take a multi-faceted perspective to this question in the thesis of this dissertation:

> *The discovery of useful behaviour may endow agents with the ability to exhibit faster learning in large complex environments, or generalization across different environments. This behaviour may manifest implicitly in the learning of useful environment states or representations thereof, or explicitly in the form of skills that are reusable and consequential in the maximization of the environment reward.*

1.1 Technical Contributions

This organization and technical contributions of this thesis are as follows:

- Chapter 2 covers the basic background on RL including the RL framework, and the various components of the environment and agent.

- In Chapter 3, we first propose a novel 2-stage setting termed goal-conditioned batch RL, which involves learning policies on more complex tasks by reusing data collected using a policy for a relatively simpler task. Second, we propose a new approach that learns useful representations of the observation space by utilizing the underlying symmetries of the environment and the agent in this modified setting.

- In Chapter 4, we propose a novel formulation of the problem of option discovery. Second, we propose an approach to solve this problem by synthesizing a compact set of options that can be used simply, or in combination with each other. Third, we devise two new environments and discuss the intuitions behind environment design in the context of our problem. Finally, we discuss a revision to our originally suggested approach in order to make it amenable to more complex environments by enhancing scalability.
- In Chapter 5, we propose a novel problem setting in the lifelong learning paradigm, where the environment is an infinite graph which cannot be covered by the agent, and has sparse rewards, which are not replenished once collected. We then propose to solve this approach by storing a model of the environment dynamics and reward function experienced by the agent in its memory that the agent can use to guide exploration in reward-relevant directions.
- Finally, we conclude this thesis in Chapter 6 with exciting future research directions in each of the individual works, as well as the unifying theme of the thesis, i.e. the discovery of useful behaviour from these discussed perspectives.

Chapter 2

Background

In the RL framework described in Sutton and Barto (1998), an agent interacts with an environment and learns to take actions that maximize the reward signal determined by the environment. The environment is modeled as a Markov Decision Process (MDP), consisting of: 1) a state space $\mathscr{S}$, 2) an action space $\mathscr{A}$, 3) the transition dynamics of the environment $P\colon \mathscr{S} \times \mathscr{S} \times \mathscr{A} \to [0,1]$, where $P(s'|s,a)$ is the probability of transitioning into state s' by taking action a in state s, 4) a reward function $\mathscr{R} : \mathscr{S} \times \mathscr{A} \times \mathscr{S} \to \mathbb{R}$ that gives the reward for this transition, and 5) a discount factor γ. The agent takes action $a \in \mathscr{A}$ in state $s \in \mathscr{S}$, and reaches state s' according to the transition dynamics of the environment given by $P(s'|s,a)$. At each step, the agent receives a reward r from the environment. In reality, the agent rarely gets the entire state information. Instead, it receives some representation of the state. The general RL framework is shown in Fig. 2.1.

Definition 2.0.1 (Episode)**.** At each time step $t = 0, ...$, the environment provides the agent with a representation of its state s_t, and the agent takes actions a_t. The environment then returns the reward for that action r_{t+1} as well as the representation of the next state s_{t+1}. This interaction between the agent and the environment gives $\{s_0, a_0, r_1, s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T, s_T\}$, which is called an *episode*. It is also referred to as a *trajectory* or a *rollout*. Here, T is the final time step of the episode.

Definition 2.0.2 (Return)**.** The return at time t, denoted by G_t, is a function of the sequence of

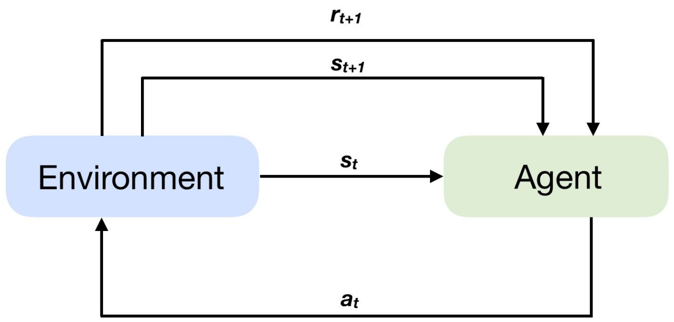


Figure 2.1. The general RL framework: At time t, an agent receives a representation of the state s_t from the environment, takes action a_t, and gets the reward for that action r_t, along with the representation of the state at the next time step s_{t+1}.

rewards an agent gets in an episode. The most common setting in RL involves calculating the expected discounted return for an episode, i.e.

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots \tag{2.1}$$

where γ is called the discount factor, and $0 \leq \gamma \leq 1$. The value of γ determines the importance the agent places on future rewards. If γ is closer to 0, it implies that the agent is optimizing for short-term rewards.

Definition 2.0.3 (Policy)**.** A policy is a function that maps the state of the environment observed by the agent to the probabilities of actions that the agent can take. The agent's goal is to learn a policy denoted by π and parameterized by θ, $\pi_\theta : \mathscr{S} \to \mathscr{A}$, $a = \pi_\theta(s)$, to maximize the expected return.

Definition 2.0.4 (Value function)**.** There are two types of value functions: state-value function and action-value function. The state-value function, denoted by $v_\pi(s)$, gives the expected return of an agent taking actions according to a policy π_θ, starting from state s. It is given by

$$V^\pi(s) = \mathbb{E}[G_t | s_t = s], \forall s \in \mathscr{S} \tag{2.2}$$

The action-value function, denoted by $q_\pi(s,a)$, gives the expected return of an agent taking action a in state s, and then following policy π. It is given by

$$Q^\pi(s,a) = \mathbb{E}[G_t | s_t = s, a_t = a], \forall s \in \mathscr{S}, a \in \mathscr{A} \tag{2.3}$$

These expressions for the state-value and action-value functions are called the Bellman equations.

There are two main categories of RL algorithms: model-based methods and model-free methods. Model-based methods estimate the transition dynamics of the environment and/or reward function of the environment, which are unknown in most scenarios, and use these for improving the policy. Model-free methods optimize the policy and/or value function directly, without attempting to learn any model of the transition dynamics or reward function. There are three classes of model-free algorithms:

1. Policy gradient algorithms: directly optimize the policy to maximize the expected return from the environment.

2. Value-based algorithms: Learn a state-value function or action-value function of the optimal policy.

3. Actor-critic algorithms: Bridge the gap between policy gradient algorithms and value-based algorithms by estimating the state-value function or action-value function of the current policy, and then improving the policy based on the estimated function.

Chapter 3

Discovering useful representations

An advantage of using deep learning techniques in the RL framework to learn policies, models, value functions, etc. is that they enable generalization across states, as opposed to the tabular RL setting, where this cannot be done in a straightforward way. As a result, deep RL methods can now be used in environments with more complex observation spaces. However, as the observation space becomes more complex, it makes the task of learning the policy harder for the agent. Furthermore, while the agent can generalize across observations that look very similar to each other, it is still incapable of generalizing across observations that do not look very similar to each other, but are semantically equivalent. We suggest that in this scenario, it may be useful to incorporate the semantic similarity between observations into the learning of the policy. Since this is information that is privy to the algorithm designer, and is not available as part of the environment, it could be construed as an inductive bias that we as the algorithm designers are incorporating into the learning of the policy or value function. While there are several types of such semantic properties that the agent or the environment could possess, we consider the property of symmetry. Now, symmetry can be a property of the agent as well as the environment. We consider one particular instantiation of symmetry: the relative positions of the agent and the goal, which is part of the environment, remain unchanged. Thus, the agent should, in principle, execute the same policy to reach the goal in all cases where this condition holds, irrespective of its absolute position in the environment, its orientation with respect to the world axes, etc. Thus, in this case, it may be useful to incorporate this type of translational or rotational *invariance* into the learning of the policy. This would enable the agent to exhibit goal-seeking behaviour in parts of the observation space that it has not experienced earlier, without having to learn the corresponding policy from scratch.

Reinforcement learning (RL) algorithms aim at learning a policy that maximizes the reward an agent gets from the environment. One class of RL algorithms attempts to learn policies that enable the agent to achieve a large set of goals. In this case, the policy is conditioned on the goal, in addition to the state. These algorithms are referred to as goal-conditioned RL algorithms (Kaelbling, 1993; Schaul et al., 2015). It is challenging to learn goal-conditioned policies because the agent has to generalize over a large set of goals. This problem becomes more evident in high-dimensional continuous control tasks like, say, goal-directed locomotion, where the agent has to learn a goal-conditioned policy that enables it to reach any arbitrary goal position. This is the problem setting we will consider for our proposed approach: goal-conditioned RL.

We propose a new approach for learning goal-conditioned policies by framing this problem in the batch RL setting. In Batch RL (Lange et al., 2012), the agent has to learn an optimal policy from a fixed set of samples collected from the environment, containing details of the state, action, and task-specific reward provided by the environment. This is in contrast to the standard RL setting in which the agent is allowed to interact with the environment while it improves its policy.

We consider the following batch RL setting: the agent is given a batch consisting of data collected while learning a policy for a simpler task. We use this batch to learn a goal-conditioned policy for the more complex task. We refer to this modified setting as *goal-conditioned batch RL*, which is a juxtaposition of these two individual settings. Our approach can be used to solve complex tasks that exhibit invariances in their state and/or goal spaces. These invariances can be exploited to learn complex tasks more quickly by using data collected while learning simpler tasks. We exploit the rotation invariance shown by the state and goal spaces to learn goal-conditioned policies for complex tasks efficiently without using a large number of environment

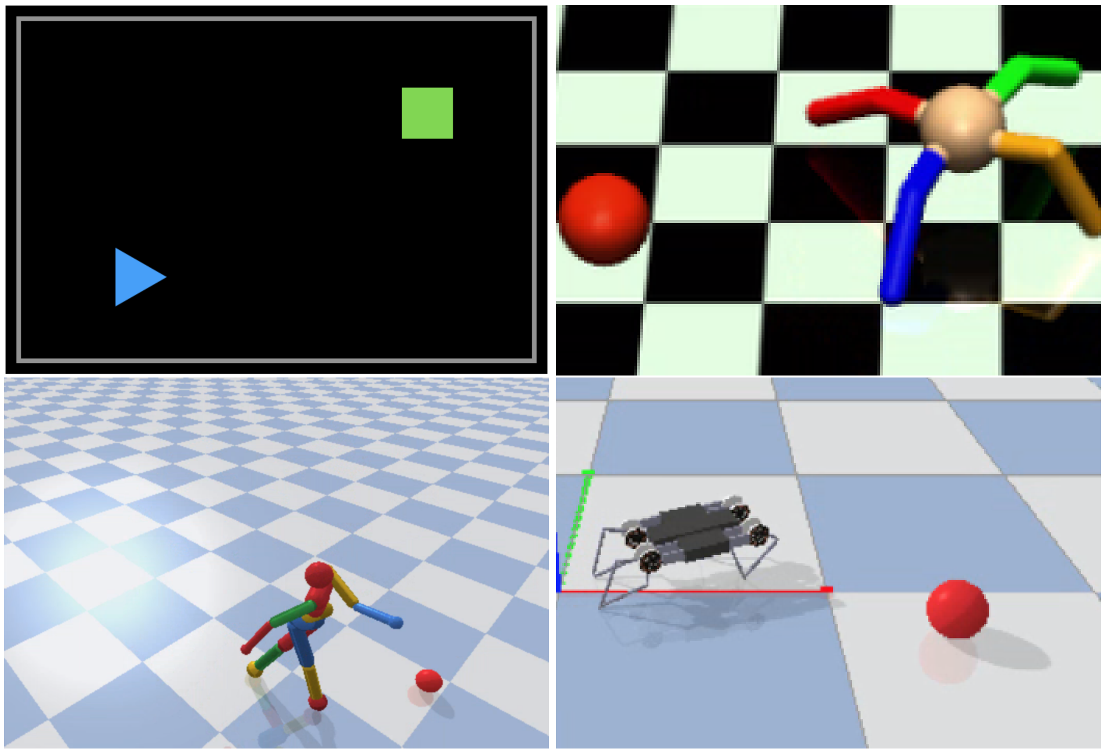


Figure 3.1. Environments: 1) GridWorld with discrete action space, 2) Ant, 3) Humanoid, 4) Minitaur with coninuous action spaces.

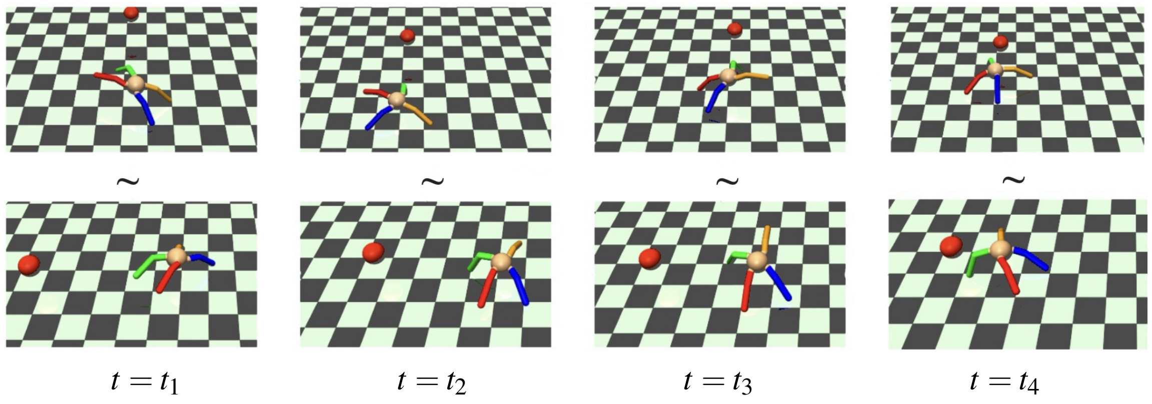


Figure 3.2. An example of *equivalent* trajectories. The first row shows intermediate steps in trajectory τ, and the second row shows the corresponding intermediate steps in $\tilde{\tau}$ ($t_1 < t_2 < t_3 < t_4$). In both these trajectories, the agent takes the same actions to reach the goal (represented by the red sphere); $\tau \sim \tilde{\tau}$

interactions. Our approach is based on these ideas, and uses two key techniques: (1) data augmentation to generate trajectories that are *equivalent* to those present in the batch (example of equivalent trajectories given in Fig. 3.2), and (2) Siamese networks to generate similar latent embeddings for equivalent state-goal pairs presented in the augmented data.

In this chapter, we will highlight the following points of our proposed setting, approach and evaluation:

1. We propose a modified batch RL setting, called goal-conditioned batch RL, to learn goal-conditioned policies,
2. We use data augmentation in conjunction with representation learning to achieve goal-directed locomotion in the goal-conditioned batch RL setting.
3. We show that our method achieves the best results on the goal-directed locomotion task in a Gridworld setting, as well as in various complex continuous control environments, compared to standard RL and goal-conditioned RL methods.

3.1 Approach

In the goal-conditioned batch RL setting that we propose, the aim is to learn a goal-conditioned policy from a fixed batch of data. This batch is collected while learning a policy for a simpler task. Thus, the agent first learns a policy for the easier task, and uses the data it collects while learning this policy to learn a goal-conditioned policy for the more complex task. This setting is shown in Fig. 3.3. There are three steps involved: 1) data collection (Section 3.1.1), 2) data augmentation (Section 3.1.2), and 3) learning equivalence between trajectories in the augmented data (Section 3.1.3).

3.1.1 Data Collection

The first step involves collecting batch data for a simpler task. We could use any standard RL algorithm to learn a policy $\pi\colon \mathscr{S} \to \mathscr{A}, \pi(s) = a$ for this simpler task. While this policy π is

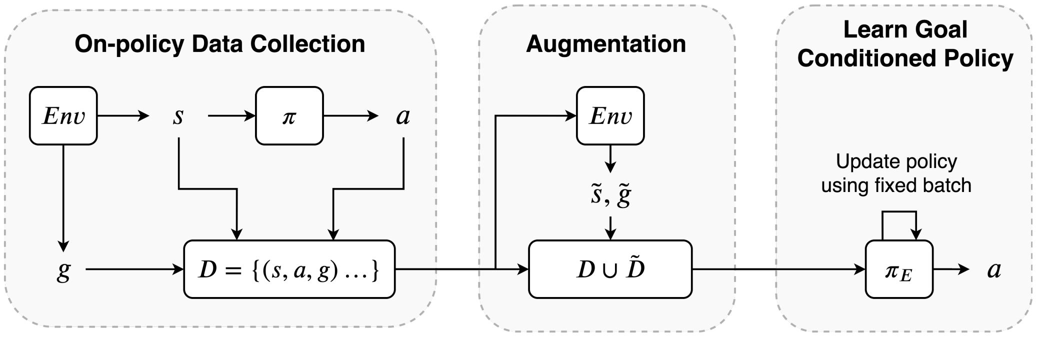

On-policy Data Collection
Env
s
π
a
g
D = {(s, a, g) ...}
Augmentation
Env
s̃, g̃
D ∪ D̃
Learn Goal Conditioned Policy
Update policy using fixed batch
π_E
a

Figure 3.3. The goal-conditioned batch RL setting. There are 3 stages: (1) Data collection (2) Data augmentation, and (3) Learning the goal-conditioned policy π_E from the batch data $\mathscr{D} \cup \tilde{\mathscr{D}}$.

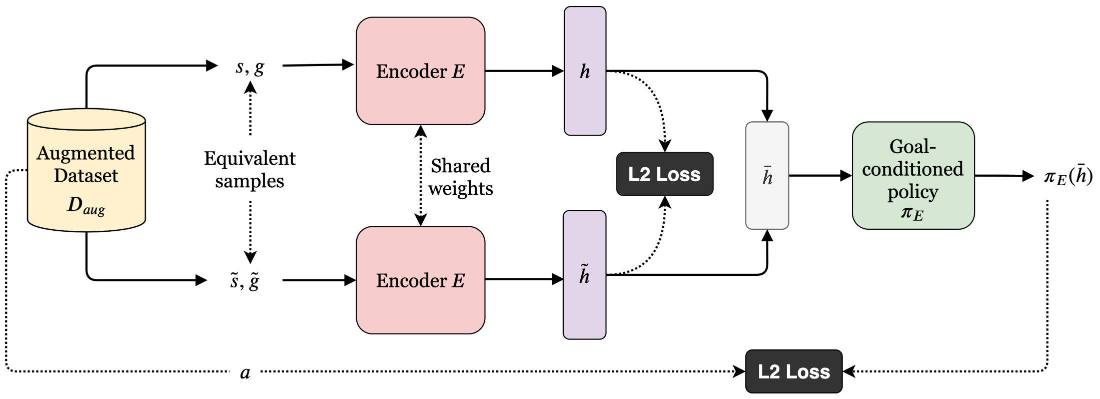


Figure 3.4. Our approach involves two components: encoder E that generates embeddings h and $\tilde{h}$ for a pair of equivalent samples (s, g) and $(\tilde{s}, \tilde{g})$ respectively using shared weights, and goal-conditioned policy with equivalence π_E that takes as input the mean of these embeddings $\bar{h}$ to output the required action. For discrete action spaces, we use cross-entropy loss, instead of L_2 loss, to update π_E.

being learnt, we collect the batch data $\mathscr{D}$, which includes the set of all transitions (s, a, g) that the agent experiences; here, s is the state, a is the action, and g is the goal. Note that π is not conditioned on the goal g; rather, g is observed while learning π. g belongs to the set of goals over which we want our eventual goal-conditioned policy to generalize, for the more complex task. For instance, for the goal-directed locomotion task in continuous control environments that we will further discuss in Section 3.2, g refers to the 3-D coordinates of the agent's position after taking action a in state s.

3.1.2 Data Augmentation

The property that we want to utilize to learn a goal-conditioned policy from fewer training samples is that this policy should be invariant to the orientation of the agent. In other words, the agent's policy for walking towards a goal at a certain position with respect to itself should be the same, regardless of the orientation of the agent. We use this property to augment the batch $\mathscr{D}$ with trajectories that are *equivalent* to those present in $\mathscr{D}$.

Definition 3.1.1 (Equivalence)**.** Two trajectories
$\tau_1 = \{s_1, a_1, g_1, s_2, a_2, g_2, ..., s_T, a_T, g_T\}$ and
$\tau_2 = \{\tilde{s_1}, \tilde{a_1}, \tilde{g_2}, \tilde{s_2}, \tilde{a_2}, \tilde{g_2}, ..., \tilde{s_T}, \tilde{a_t}, \tilde{g_t}\}$ are equivalent if for any $\theta \in [0, 2\pi)$:

- $\tilde{a_1} = a_1, \tilde{a_2} = a_2, ..., \tilde{a_t} = a_T$
- $\tilde{s_1} = rot(s_1, \theta), \tilde{s_2} = rot(s_2, \theta), ..., \tilde{s_T} = rot(s_T, \theta)$
- $\tilde{g_2} = rot(g_1, \theta), \tilde{g_2} = rot(g_2, \theta), ..., \tilde{g_t} = rot(g_T, \theta)$

where $rot(x, \theta)$ is a rotation of x about the agent by an angle θ.

We use the above definition of equivalence to augment the batch $\mathscr{D}$ with trajectories that are equivalent to those present in $\mathscr{D}$. For each trajectory τ observed in the dataset $\mathscr{D}$, we generate another trajectory $\tilde{\tau}$ that is equivalent to it.

3.1.3 Our Approach: Enforcing Equivalence

Once we have the augmented dataset, we shift the focus to learning a model that incorporates this equivalence into the training procedure. In order to do this, we learn two models: an encoder E, and a policy π_E. The encoder is defined as $E\colon \mathscr{S} \times \mathscr{G} \to \mathbb{R}^k$. It outputs $h = E(s, g)$, where h is a k-dimensional embedding produced by the encoder network. The goal-conditioned policy with equivalence π_E is defined as $\pi_E\colon \mathbb{R}^k \to \mathscr{A}, \pi_E(h) = a$, where h is the k-dimensional output of the encoder.

We want the embedding h to capture the equivalence between two trajectories τ and $\tilde{\tau}$. Thus, if $(s, a, g) \in \tau$ and $(\tilde{s}, a, \tilde{g}) \in \tilde{\tau}$, and $(s, g) \sim (\tilde{s}, \tilde{g})$, $h = \tilde{h}$, where $h = E(s, g)$ and $\tilde{h} = E(\tilde{s}, \tilde{g})$.

Algorithm 1. Enforcing equivalence

```
Require: $\begin{cases} \mathscr{D}_{aug} = \{\tau_1, \tilde{\tau}_1, \tau_2, \tilde{\tau}_2, ..., \tau_N, \tilde{\tau}_N\} \\ \text{learning rate } \eta \\ \text{encoder } E \text{ with parameters } \phi_{enc} \\ \text{goal-conditioned policy with equivalence } \pi_E \text{ with parameters } \phi_{\pi_E} \end{cases}$
 1: Initialize $\phi_{enc}$ and $\phi_{\pi_E}$ randomly
 2: repeat
 3:     for $(\tau, \tilde{\tau}) \in \mathscr{D}_{aug}$ do
 4:         $s_1, a_1, g_1, s_2, a_2, g_2, ..., s_T, a_T, g_T \leftarrow \tau$
 5:         $\tilde{s_1}, a_1, \tilde{g_1}, \tilde{s_2}, a_2, \tilde{g_2}, ..., \tilde{s_T}, a_T, \tilde{g_T} \leftarrow \tilde{\tau}$
 6:         for $t = 1$ to $T$ do
 7:             $h_t = E(s_t, g_t)$
 8:             $\tilde{h_t} = E(\tilde{s}_t, \tilde{g}_t)$
 9:             $\bar{h}_t = \frac{1}{2}(h_t + \tilde{h}_t)$
10:         $\mathscr{L} = \lambda \mathscr{L}_{enc} + (1 - \lambda)\mathscr{L}_{\pi_E}$
11:         $\phi_{enc} \leftarrow \phi_{enc} - \eta \nabla \mathscr{L}$
12:         $\phi_{\pi_E} \leftarrow \phi_{\pi_E} - \eta \nabla \mathscr{L}$
13: until $n$ iterations
```

We generate these embeddings for the equivalent samples simultaneously; thus, we use a Siamese framework (Koch et al., 2015) to learn E i.e. h and $\tilde{h}$ are computed using shared weights.

We then provide this latent embedding h as input to the policy π_E, which produces the action the agent should take to reach goal g. This architecture is shown in Fig. 4.4. During the training process, we minimize the squared distance between the latent embedding generated by (s, g) and $(\tilde{s}, \tilde{g})$, and fit the output of the policy to the action a. For continuous action space, we use the L_2 loss, and for discrete action space, we use the cross-entropy loss. The final loss is a weighted sum of the L_2 loss for the embeddings, and the policy loss. This procedure is described in Algorithm 1.

3.2 Experiments

We show the results of our proposed approach on environments with discrete and continuous action spaces (environments shown in Fig. 3.1). Our discrete environment is a Gridworld environment, where the agent is given an image of the current environment configuration and has to navigate to the goal. Our continuous control environments include the complex Ant, Humanoid and Minitaur environments (Coumans and Bai, 2016; Brockman et al., 2016; Todorov et al., 2012). On all these environments, our method outperforms standard RL, as well as goal-conditioned RL algorithms. We also perform extensive ablation experiments in the goal-conditioned batch RL setting, where the batch consists of either random samples, or on-policy samples collected by the agent. We find that data augmentation with rotated trajectories alone beats all the above baselines, and learning the encoder on augmented samples results in the best performance across all environments.

Our experiments aim to answer the following questions:

- Is our approach better than simply learning a goal-conditioned policy using any standard RL algorithm?
- How does our approach fare against existing goal-conditioned RL algorithms?
- In the goal-conditioned batch RL setting described in Section 3.1, how does the performance change when the batch consists of random samples v/s on-policy samples v/s augmented samples?
- Does enforcing equivalence over augmented samples improve performance?

In order to answer the above questions, we perform the following experiments in each environment:

1. **Standard RL:** We use a standard RL algorithm to learn a goal-conditioned policy $\pi(s, g)$ (Schulman et al., 2017; Haarnoja et al., 2018)
2. **Goal-conditioned RL:** We use Hindsight Experience Replay (HER) to learn the goal-conditioned policy. (Andrychowicz et al., 2017; Haarnoja et al., 2018; Lillicrap et al., 2016; Mnih et al., 2013)

3. **Goal-conditioned batch RL:** We perform a series of ablation experiments, each adding one important component of our approach. In each of these baselines, we learn a single model for the goal-conditioned policy using supervised learning i.e. the input to the policy is the (s, g) pair, and the output is the action a.
    (a) **Random samples:** The batch consists of samples collected by an agent following a random policy.
    (b) **On-policy samples:** The batch consists of samples collected by the agent while learning a policy for the simpler task.
    (c) **Augmented samples:** The batch consists of samples collected by the agent while learning a policy for the simpler task, as well as samples that are *equivalent* to them.

We compare our approach with the above baselines on two sets of environments: 1) Gridworld environment with discrete action space, and 2) Continuous control environments for goal-directed locomotion. We maintain the same number of environment interactions for all methods.

3.2.1 Gridworld

We first demonstrate the effectiveness of our approach on a standard Gridworld environment. In this environment, the observation contains an image of size $12 \times 12 \times 3$ (the agent and goal are associated with different colours), and the orientation of the agent (represented by a 1-hot vector of size 4). The agent can take one of 3 actions - move forward, turn left, and turn right i.e. the action space is egocentric from the agent's perspective. The agent gets a reward of 1 if it reaches the goal, and 0 otherwise. The agent and goal are reset at the beginning of each episode.

The goal-conditioned batch RL setting involves collecting data while learning a policy for a simpler task. In this case, we will use the simpler task as a Gridworld environment in which the goal is fixed. We show in Fig. 3.5 that learning a policy in a Gridworld environment

Table 3.1. Gridworld experiments: Success rate of our algorithm compared with the baselines (best approach in bold). Our approach outperforms all baselines.

Model	**Mean**	**Std Dev**
Standard RL (PPO)	40.8	4.833
Goal-conditioned RL (HER)	4.6	2.8
Goal-conditioned batch RL		
Random samples	0.6	0.8
On-policy samples	15.5	2.579
Augmented samples	49.6	6.02
Enforcing Equivalence	**50.5**	**3.748**

with a fixed target (left) takes $\sim$ 10K timesteps, making it a much easier task than learning a policy in a Gridworld environment where the goal changes every episode (right), compared using the same algorithm (PPO). We learn a policy for this simpler environment and collect data for 150K time steps. Next, we augment the data by rotating the image and orientation by an angle $\theta \in [\pi/2, \pi, 3\pi/2]$. We also augment the data by padding the image on all four sides.

We evaluate the performance of the agent based on the reward it receives from the environment. We report the mean and standard deviation of the success rate over 100 episodes each for 10 random seeds in Table 3.1. Our contributions of data augmentation and incorporating equivalence between augmented samples emerge as the best performing approaches compared to all other baselines. We also show in Fig. 3.5 (right) that an agent trained using PPO takes 4M timesteps to reach the same performance as our approach achieves in just 150K timesteps. Comparing the two plots when the same algorithm (PPO) is used, it is evident that learning a policy for a Gridworld environment with a fixed goal is much easier than that with arbitrary goals.

At test time, we use the complex Gridworld environment that generates an arbitrary goal in each episode. Each episode has a maximum of 300 steps, for each environment. An episode terminates when the agent successfully reaches the goal or runs out of time. We report the percentage of episodes in which the agent is successfully able to reach the goal. We select 10 random seeds and test the performance of the agent on 100 episodes for each random seed.

3.2.2 Continuous Control

We now perform the same experiments on complex continuous control environments: 1) a bipedal Humanoid, 2) a quadrupedal Minitaur, and 3) a quadrupedal Ant (Brockman et al., 2016; Coumans and Bai, 2016). The action space of the Ant and Minitaur is $\mathbb{R}^8$, while that of the Humanoid is $\mathbb{R}^{17}$. The observation space is the same as that provided in the default implementations of these environments. The Humanoid is a bipedal agent with a 43-D observation space and 17-D action space. The Ant is a quadrupedal agent with a 111-D observation space and 8-D action space. The Minitaur is also a quadrupedal agent with a 17-D observation space and 8-D action space.

In the goal-conditioned batch RL setting, we first learn a policy for the simpler task of locomotion in one direction. While this policy is being learnt, we collect (s,a,g) transitions, where g is the 3-D position of the agent after taking action a in state s. We collect data for 1M, 2M, and 10M timesteps for the Ant, Minitaur, and Humanoid respectively. The next step is data augmentation; here, after each episode, we rotate the agent and the goal by some angle $\theta \in [0, 2\pi]$, and repeat the actions taken in that episode. We collect the corresponding equivalent (s,g) pairs, $(\tilde{s},\tilde{g})$, and add them to the augmented dataset $\tilde{\mathscr{D}}$.

The reward function for the policy used for data collection in the goal-conditioned batch RL setting includes the control and contact costs, and a reward for moving forward to the right. The reward function for training a goal-conditioned policy using standard RL includes the control and contact costs, and a reward for moving in the direction of the goal. In the goal-conditioned RL setting, we consider two scenarios: sparse and dense rewards. In the sparse reward experiments, the agent receives a reward of 0 if it reaches within a certain distance of the goal, and -1 otherwise. In the dense reward scenario, the agent receives a weighted sum of the control and contact costs, and the distance to the goal as its reward.

The performance of the agent is determined by the closest distance to the goal it is able to achieve. We report the mean and standard deviation of this distance in Tables 3.2, 3.3, and

Table 3.2. Results of our algorithm compared with the baselines (best approach in bold) on the Humanoid environment.

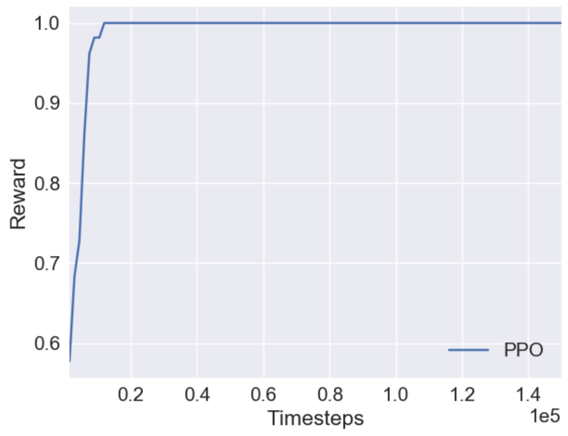
Model	Mean	Std Dev
Standard RL		
(Schulman et al., 2017; Haarnoja et al., 2018)	2.160	0.955
Goal-conditioned RL (Andrychowicz et al., 2017)		
w/ sparse reward	2.902	1.138
w/ dense reward	2.891	1.135
Goal-conditioned batch RL		
Random samples	3.159	0.579
On-policy samples	2.777	0.841
Augmented samples	1.936	0.977
Our approach: Enforcing Equivalence	**1.779**	**0.986**

Table 3.3. Results of our algorithm compared with the baselines (best approach in bold) on the Minitaur environment.

Model	Mean	Std Dev
Standard RL		
(Schulman et al., 2017; Haarnoja et al., 2018)	1.358	0.326
Goal-conditioned RL (Andrychowicz et al., 2017)		
w/ sparse reward	1.440	0.237
w/ dense reward	1.380	0.251
Goal-conditioned batch RL		
Random samples	1.598	0.282
On-policy samples	1.403	0.438
Augmented samples	1.212	0.485
Our approach: Enforcing Equivalence	**0.904**	**0.482**

Table 3.4. Results of our algorithm compared with the baselines (best approach in bold) on the Ant environment.

Model	Mean	Std Dev
Standard RL		
(Schulman et al., 2017; Haarnoja et al., 2018)	2.608	0.607
Goal-conditioned RL (Andrychowicz et al., 2017)		
w/ sparse reward	2.602	0.953
w/ dense reward	2.475	0.693
Goal-conditioned batch RL		
Random samples	3.166	0.569
On-policy samples	2.155	1.054
Augmented samples	1.721	0.978
Our approach: Enforcing Equivalence	**1.105**	**0.866**



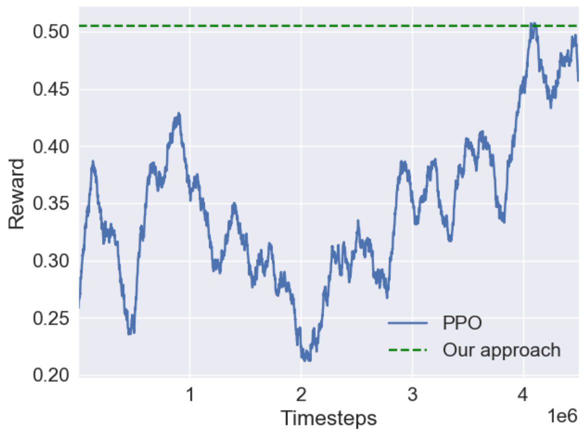


Figure 3.5. Left: Learning a policy on the easier Gridworld environment with a fixed goal using PPO during the data collection process. Right: Comparing the performance of an agent trained using PPO on the more difficult Gridworld environment with arbitrary goals with our approach, learnt using just 150K timesteps. The PPO agent takes 4M timesteps to match the performance of our agent.

3.4. We compared our approach to standard RL approaches (Schulman et al., 2017; Haarnoja et al., 2018), as well as goal-conditioned RL (Andrychowicz et al., 2017) with sparse and dense rewards. Again, our approaches using data augmentation and incorporating equivalence between samples of the augmented data outperform these baselines.

At test time, the agent is rotated by a random angle. The target is set at a distance of $\sim 2-5$ units from the agent at an angle of $[-45^\circ, 45^\circ]$ to the agent's orientation in the case of the Ant and Humanoid environments. For Minitaur, we set the target at a distance of $\sim 1.5-2.5$ units from the agent at an angle of $[-45^\circ, 45^\circ]$ to the agent's orientation. At each point, we replace the actual goal with the unit vector in the direction of the goal as the input.

Each episode consists of a maximum of 1000 steps for each environment, and the episode terminates when the agent reaches the goal, or falls down/dies. We report the closest distance from the target that the agent is able to reach, for each episode. We select 10 random seeds and test the performance of each method on 1000 episodes for each random seed. In order to ensure that the comparison between all methods is indeed fair, we set the initial configuration of the agent and the target to be the same across all methods at test time.

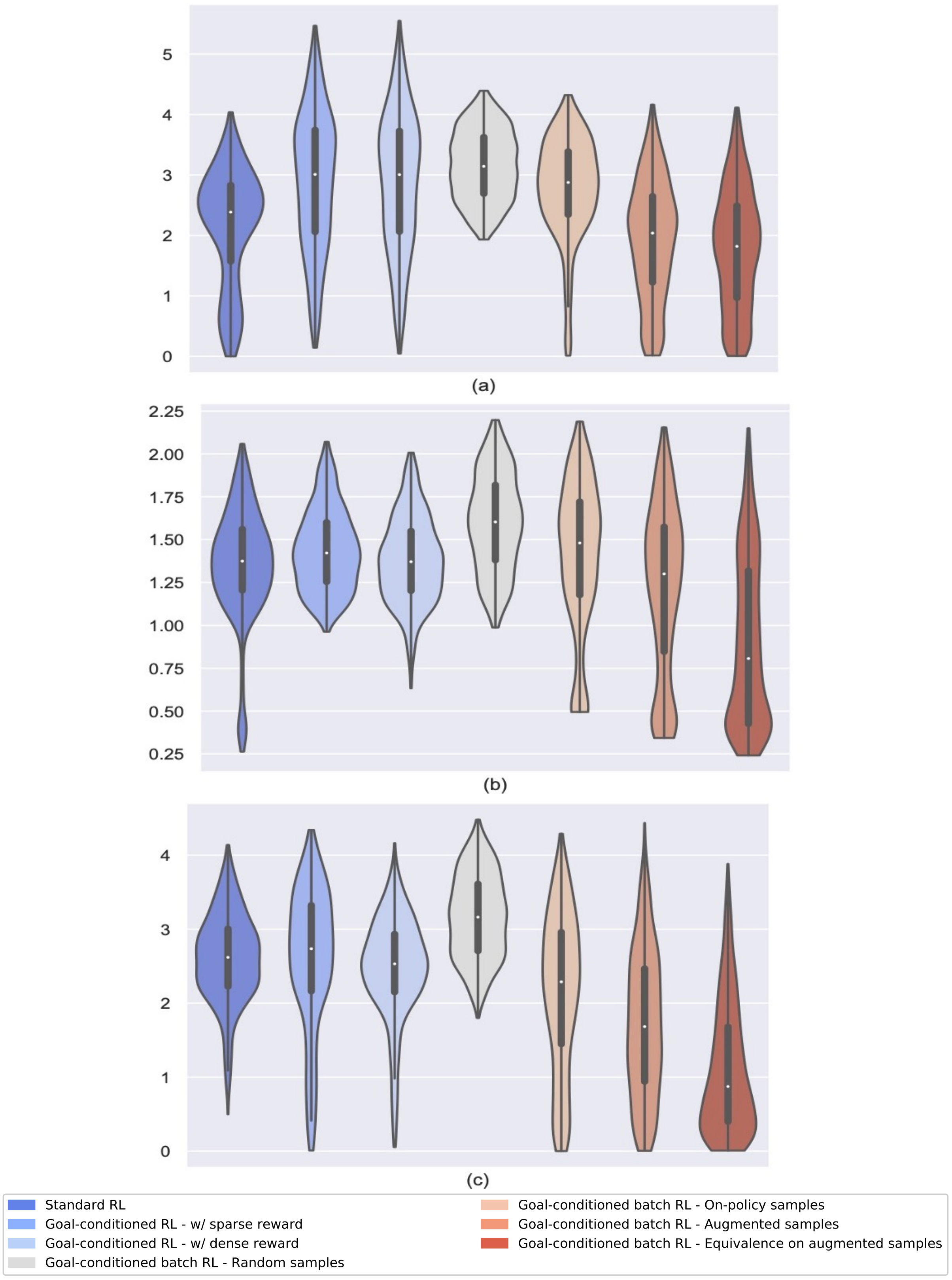


Figure 3.6. Violin plots showing the distribution of closest distance from the goal for each episode for the 3 environments: (a) Humanoid (b) Minitaur (c) Ant. We report results over 10 random seeds with 100 episodes for each random seed.

We show the distribution of distance to the goal for our approach and the baselines in the violin plots in Fig. 3.6. These plots show the improvement in performance on adding each successive component of our algorithm, as the distribution shifts to a lower mean with each addition. The peaks observed away from the mean for the baselines indicate low generalization to goals unseen during training. Qualitative examples corresponding to this are provided in Fig. 3.8 and Fig. 3.9.

We also perform an experiment to compare the qualitative performance of our approach, and the best-performing baseline using standard RL on the Humanoid environment; results are shown in Fig. 3.7.

3.2.3 Algorithms for Data Collection and Baselines

The batch data we collect for the goal-conditioned batch RL experiments consist of three kinds of data: random samples, on-policy samples, and augmented samples. We collect the same number of samples for all baselines, and also train the standard and goal-conditioned RL policies using the same number of environment interactions. Since HER is primarily targeted at the sparse-reward setting, we report results on both dense and sparse reward environments.

Gridworld. We use 150000 timesteps of environment interaction for each method. While performing data augmentation, we can simply rotate the image, and thus, do not need to interact with the environment to perform data augmentation. We discard the first 25000 samples, which consist of the agent making completely random moves. The data collection policy used for the goal-conditioned batch RL setting is PPO (Schulman et al., 2017). The standard RL baseline is also learnt using PPO, and the goal-conditioned RL baseline is learnt using Hindsight Experience Replay (HER) (Andrychowicz et al., 2017) in conjunction with Deep Q-Networks (DQN) (Mnih et al., 2013).

Ant. We use 2 million timesteps of environment interaction for each method. For the standard RL and goal-conditioned RL methods, this means training a policy for 2 million timesteps. The data collection for the goal-conditioned batch RL setting is done in the following

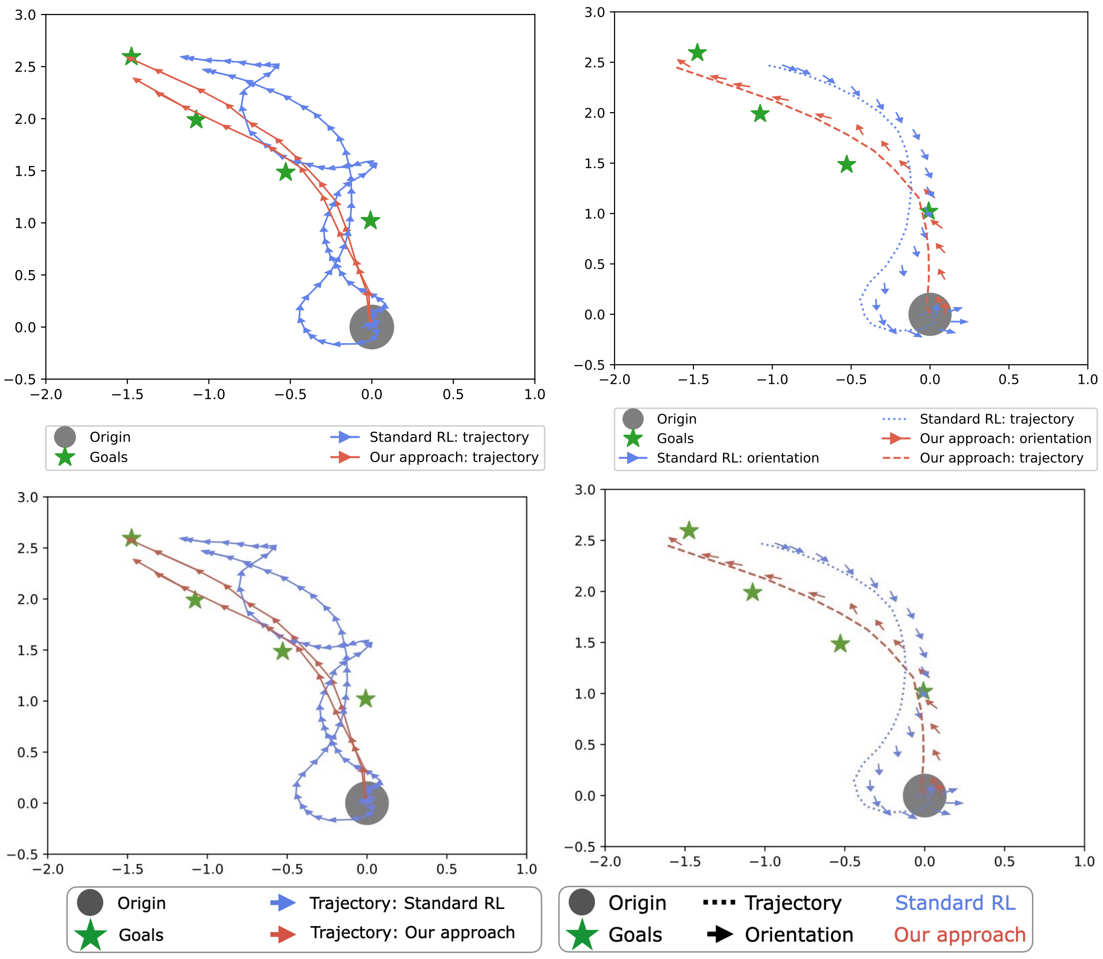


Figure 3.7. We compare the performance of the best standard RL baseline with our approach on the Humanoid. The first plot shows two successful trajectories through 4 consecutive goals. The second plot shows the orientation of the agent through one successful trajectory for both methods. The quality of the standard RL baseline is lower than that of our approach in terms of the trajectory followed, orientation of the agent, and the number of steps taken to reach the goal.

manner. For the baseline with random samples, the batch consists of 2 million timesteps of the agent taking random actions in the environment. The baseline with on-policy samples has a batch consisting of 2 million samples collected while training a standard RL algorithm for the locomotion task. The baseline with augmented samples, and our final approach that enforces equivalence, have a batch consisting of 1 million samples collected while training the standard RL algorithm, and 1 million samples that are equivalent to those collected at training time. The data collection policy is learnt using PPO (Schulman et al., 2017). The standard RL policy is also learnt using PPO, and the goal-conditioned RL baseline is learnt using HER with Deep Deterministic Policy Gradients (DDPG) (Lillicrap et al., 2016).

Minitaur. We use 4 million timesteps of environment interaction. The procedure for training the standard and goal-conditioned RL methods, as well as data collection for the goal-conditioned batch RL methods is the same as that for the Ant (discussed above). The data collection policy is learnt using PPO (Schulman et al., 2017). The standard RL baseline is also learnt using PPO, and the goal-conditioned RL baseline is learnt using HER with Soft Actor-Critic (SAC) (Haarnoja et al., 2018).

Humanoid. For the Humanoid environment, the first 5 million timesteps of training a standard RL algorithm for locomotion consisted of predominantly bad samples that led to the Humanoid falling down very early in the episode. Thus, instead of using those samples, we froze training after 5 million timesteps, and used the policy trained to collect 10 million samples. The standard and goal-conditioned RL methods were trained for 15 million timesteps, for fair evaluation. The procedure for collecting the batch data is the same as that discussed above for Ant and Minitaur. The data collection policy is learnt using SAC (Haarnoja et al., 2018). The standard RL baseline is also learnt using SAC, and the goal-conditioned RL baseline is learnt using HER with SAC.

3.2.4 Training details

In this section, we discuss the details of all the experiments performed in the standard RL, goal-conditioned RL and goal-conditioned batch RL settings. There are 3 models we learn in the goal-conditioned batch RL methods: the encoder E, the naïve goal-conditioned policy π_g, and the goal-conditioned policy with equivalence π_E. We also have the data collection policy π_{data} for batch data. The standard and goal-conditioned RL algorithms also involve learning the policies $\pi_{standard}$ and π_{goal} respectively. All these networks are MLPs with 2 hidden layers and ReLU activation.

For the standard RL and goal-conditioned RL baselines we compare our approach with, we use the hyperparameters provided in existing implementations of these algorithms (Dhariwal et al., 2017; Hill et al., 2018; Achiam, 2018). The hyperparameters involved in training all the goal-conditioned batch RL methods are: **Weighted loss parameter λ.** The loss function for our approach is a weighted sum of the encoder loss L_{enc} and the policy loss L_{π_E}. We set $\lambda = 0.9$ for the Gridworld environment, and $\lambda = 0.25$ for all continuous control environments. **Encoder output dimension k.** The output of the encoder in our approach is a k-dimensional vector. We set $k = 25$ for the Gridworld environment, $k = 10$ for the Ant and Minitaur environments, and $k = 50$ for the Humanoid environment. **Loss function.** We use L_2 loss for the continuous control environments, and cross-entropy loss for the Gridworld environment. **Optimizer.** We use the Adam optimizer (Kingma and Ba, 2015) with a learning rate 0.001 and a batch size of 512 for all goal-conditioned batch RL methods across all environments.

3.3 Additional Analysis of Results

3.3.1 Gridworld

In addition to the mean and standard deviation of the success rate of our approach compared with baselines, we report these statistics for the number of steps taken by the agent to reach the goal, for episodes that resulted in a reward of 1. These results are given in 3.5. Since

Table 3.5. Gridworld experiments: Number of steps taken by the agent to reach the goal

Model	**Mean**	**Std Dev**
Standard RL (PPO)	107.5	87.64
Goal-conditioned RL (HER)	7.57	4.92
Goal-conditioned batch RL		
Random samples	2.33	0.94
On-policy samples	10.39	4.56
Augmented samples	11.12	5.8
Enforcing Equivalence	12.79	9.12

we want the agent to reach the goal in the shortest possible amount of time, we want this metric to be low.

The three best-performing approaches, according to the results tables, are our final approach, our ablation experiment that uses augmented samples to learn a naïve goal-conditioned policy, and the PPO baseline. We see that while both approaches that we propose take a comparable number of steps to reach the goal, the PPO baseline takes a far larger number of steps. This establishes the fact that our approach is qualitatively better than the standard RL baseline.

We also see that the random baseline takes the least time on average to reach the goal. However, since the success rate of the random baseline is very low, we do not consider this value to be of importance.

We can also draw a similar conclusion for the HER baseline, which also takes a lower number of steps on average to reach the goal. Another reason for this could be the fact that HER intrinsically induces a sort of curriculum while learning, resulting in goals lying close to the agent first being achieved, and then being used to learn to reach farther goals. However, since we are using a very small number of samples from the environment for learning the goal-conditioned policy (150K steps), it is unlikely that the agent would have encountered success in reaching goals that are far from itself, resulting in lower performance. Thus, the successes we see in this baseline are due to the goal being close to the agent, leading to a lower time to success measure.

The on-policy ablation in the goal-conditioned RL setting also has similar time to success statistics as our approach and the augmented samples baseline. However, its overall performance is much lower than these methods. From this, we can deduce that while the agent has learnt a good goal-conditioned policy for the fixed goal set during the data collection process, and goals in its immediate neighbourhood, it is unable to generalize to goals lying outside its training distribution. We observe a similar result in the continuous control environments, and perform qualitative experiments to verify that it is true.

3.3.2 Continuous Control

The violin plots in Fig. 3.6 show a slight peak away from the mean for a number of baselines. This peak is a result of the agent being able to reach goals that lie within its training distribution. We conduct an additional experiment to provide qualitative proof for this. We take one instance each of a bipedal and a quadrupedal agent, the Humanoid and the Ant, and show results in Fig. 3.8 and Fig. 3.9 respectively. We fix the initial state of the agent and generate goals in a specific region. We plot the results of 10 best episodes for each method for the Humanoid and Ant environments. The standard RL baseline is sometimes successful in reaching the goal. The goal-conditioned RL algorithm (HER) with sparse or dense rewards fails early in the episode, because HER works better in sparse reward scenarios, and it takes a large number of samples to learn a good locomotion policy with only sparse rewards. In the goal-conditioned batch RL setting, the baseline trained using only on-policy samples can reach goals lying within its training distribution, on the right, but fails on other goals. Both our proposed ideas: data augmentation and enforcing equivalence, result in trajectories that reach closest to the goal in all cases.

Using random samples as the batch data in the goal-conditioned batch RL setting leads to the worst results in all environments. This happens because all agents we consider are high-dimensional continuous control agents, which cannot be expected to learn good goal-conditioned policies from random data.

Learning from a batch that *does* contain on-policy samples, i.e. samples collected while training a locomotion policy in a single direction, does improve performance over random

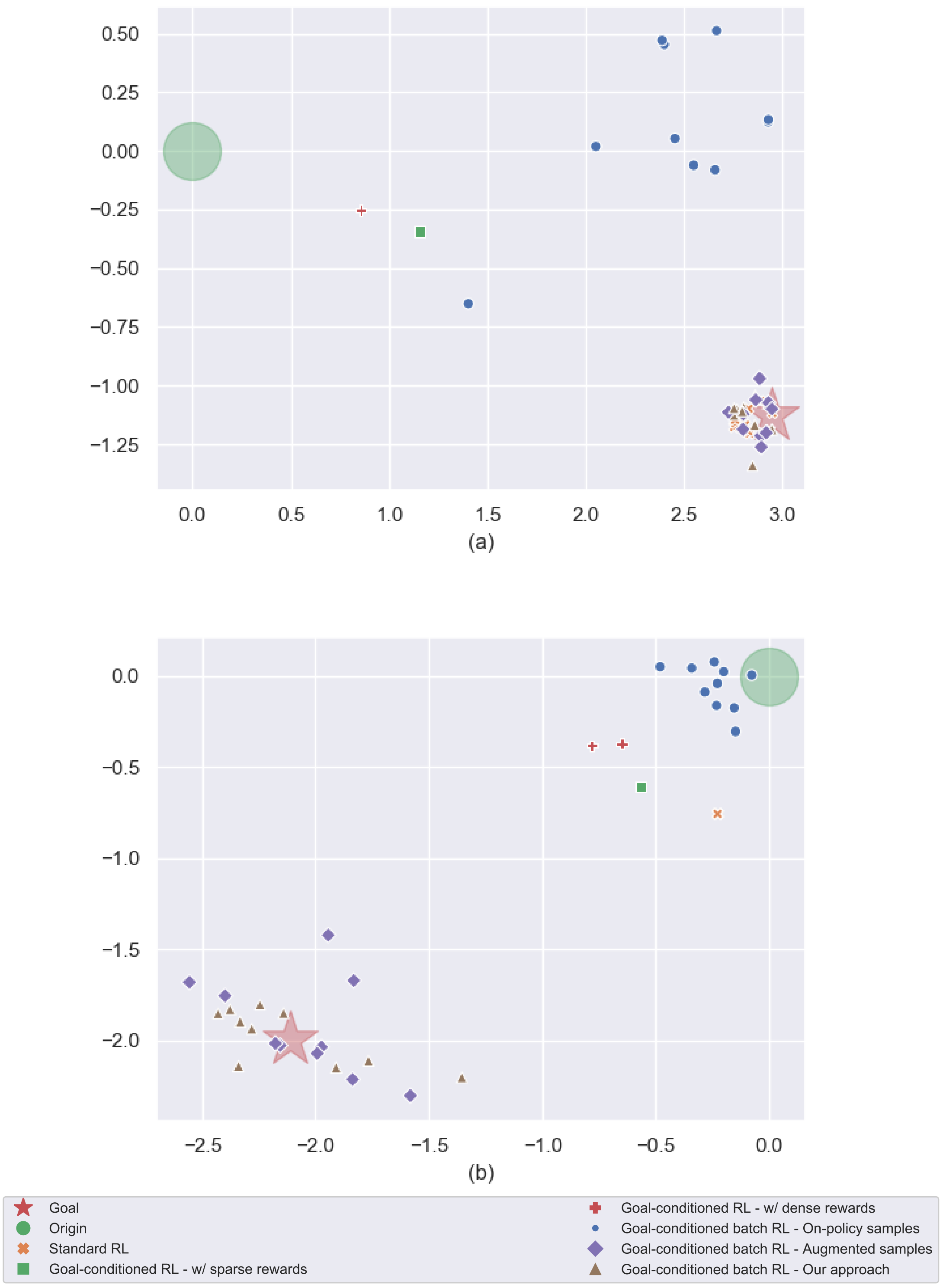


Figure 3.8. Qualitative comparison between different algorithms at test time for the Humanoid.

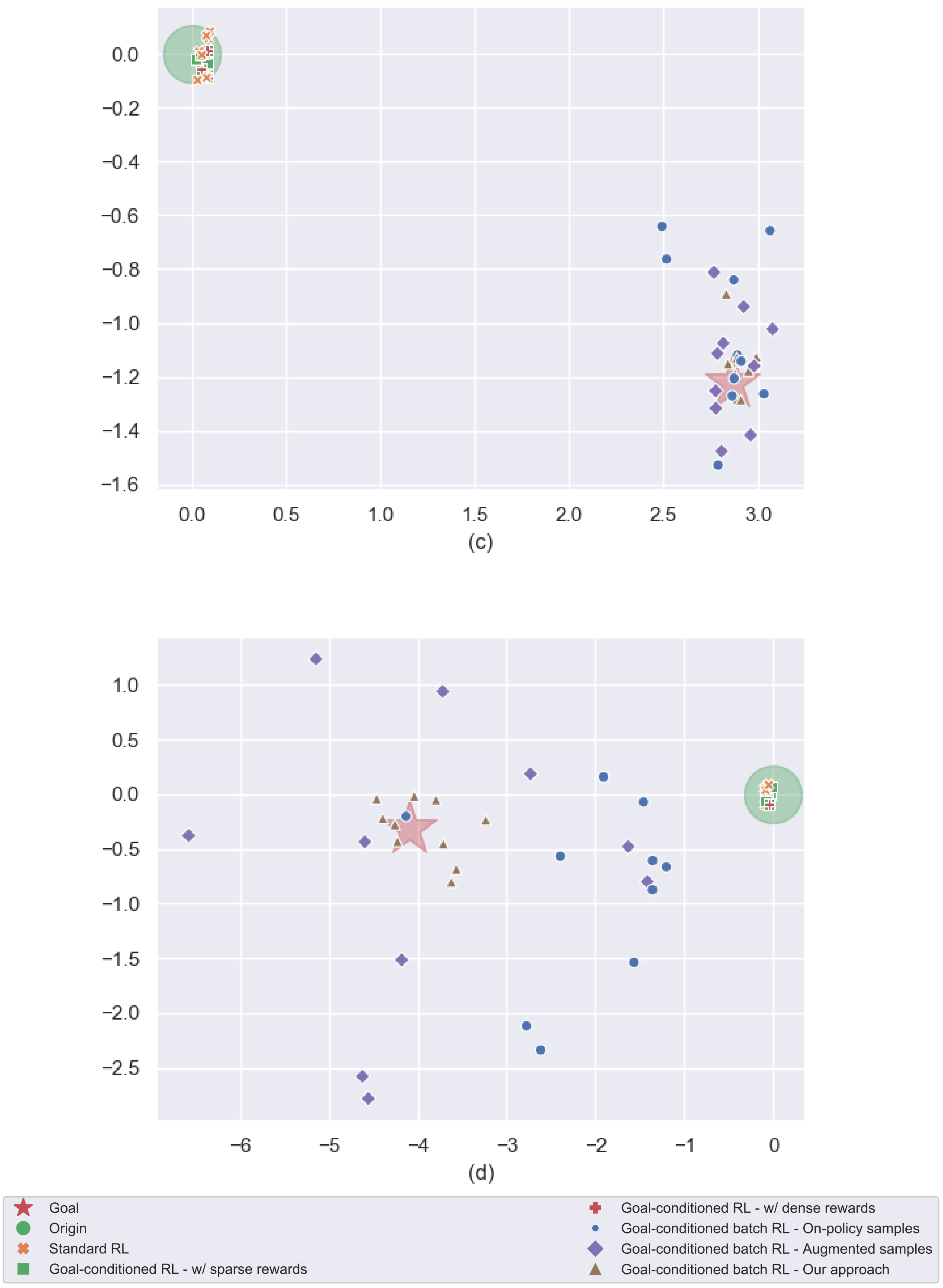


Figure 3.9. Qualitative comparison between different algorithms at test time for the Ant.

samples, but it is unfortunately unable to exceed the performance of the standard RL and goal-conditioned RL methods in all environments due to poor generalization to goals that lie out of its training distribution. However, for goals that lie within its training distribution, it exhibits smooth goal-directed locomotion, as opposed to the poor performance of goal-conditioned batch RL with random samples. This is evident from the violin plots shown in Fig. 3.6: the plot for goal-conditioned batch RL with on-policy samples has a small peak away from the mean, which consists of goals belonging to its training distribution.

Learning from augmented samples in the goal-conditioned batch RL setting improves performance over standard and goal-conditioned RL baselines, as well as over the two preliminary goal-conditioned batch RL baselines. While the gain in performance over the two simpler goal-conditioned batch RL baselines is for obvious reasons, this method works better than even standard RL and goal-conditioned RL because it consists of samples that take consistent actions to reach the goal in the case of *equivalent* state-goal configuration. In other words, this means that if we have two trajectories in the augmented dataset $\tau = \{s_1, a_1, g_1, s_2, a_2, g_2, ..., s_T, a_T, g_T\}$ and $\tilde{\tau} = \{\tilde{s_1}, \tilde{a_1}, \tilde{g_2}, \tilde{s_2}, \tilde{a_2}, \tilde{g_2}, ..., \tilde{s_T}, \tilde{a_t}, \tilde{g_t}\}$, where $(s_1, g_1) \sim (\tilde{s_1}, \tilde{g_2})$, the augmented samples collected would have $a_1 = \tilde{a_1}, a_2 = \tilde{a_2}, ..., a_T = \tilde{a_t}$, but the same policy may not be learnt by standard RL and goal-conditioned RL methods.

While augmenting the dataset with equivalent trajectories already beats all other baselines in terms of performance, enforcing equivalence between equivalent trajectories is able to achieve a far higher generalization to goals not observed in the training distribution. This is because the encoder learns the same embeddings for equivalent state-goal configurations. Thus, for any goal g that lies outside the training distribution, the agent starting from state s will be able to achieve it successfully if $(\tilde{s}, \tilde{g})$, where $(s, g) \sim (\tilde{s}, \tilde{g})$, lies in its training distribution.

An interesting observation is that standard RL works the same as or, in some cases, better than goal-conditioned RL for the goal-directed locomotion task in the sparse-reward, as well as the dense-reward settings. While this was initially surprising, we realized that since goal-directed locomotion is a difficult task, it would take a very large number of samples for the agent to reach a goal and receive a positive reward. Thus, we observe that in each case, the agent trained with dense reward (i.e. including contact and control costs in the reward function) performs better than that trained with sparse reward.

3.4 Related Work

3.4.1 Goal-conditioned RL

Goal-conditioned RL aims at learning policies or value functions that are conditioned on the state and goal spaces, so that the agent can learn to efficiently generalize to unseen goals. For instance, the value function

$$V^{\pi}(s) = \mathbb{E}^{\pi}[\sum_{i=t}^{T} \gamma^{i-t} r_{i+1} | s_t = s]$$

is transformed to

$$V^{\pi}(s,g) = \mathbb{E}^{\pi}[\sum_{i=t}^{T} \gamma^{i-t} r_{g,i+1} | s_t = s]$$

where $r_{g,t}$ is the goal-based reward at time step t and $V^{\pi}(s,g)$ is the goal-conditioned value function.

The intuition behind goal-conditioned RL provided by Schaul et al. (2015) is that the goal space may contain as much information as the state space. A specific instance provided in the paper is that if g is a goal and is described as a single state, there is a similarity between the value of nearby goal states, just like nearby states have similar values. Goal-conditioned RL is challenging because the agent is expected to generalize to (s,g) inputs that it has not encountered during the learning process. A particular advantage of using goal-conditioned reinforcement learning is that even if the final goal is unseen, the agent can achieve intermediate goals successively to reach the final goal. Another popular approach in goal-conditioned RL is Hindsight Experience Replay (Andrychowicz et al., 2017), where the agent learns the value function over *potential* goals, instead of just the final goal. This is especially advantageous in sparse-reward environments, where this relabeling of goals enables much faster learning of the policy / value function.

A closely related work to ours is by Ghosh et al. (2018), who propose learning representations that are *actionable*: the agent learns representations of the state space depicting their similarity in terms of the goal-conditioned policy that reaches them, and then uses them in different downstream tasks. The key difference is that they use goal-conditioned policies to learn the representations, while we learn a policy on a simpler task and *then* use it to achieve different goals, achieving the effect of generalizing across goals. Nair et al. (2018) also perform goal relabeling for high-dimensional visual input spaces.

3.4.2 Invariance

There have been parallel works that use symmetry to learn representations or policies faster. Mishra et al. (2019); Abdolhosseini et al. (2019) exploit symmetry in the *agent*'s action space to learn policies that preserve symmetry in the agent's gait. Lin et al. (2020) propose augmenting the experience replay buffer with additional trajectories based on the symmetries exhibited by the agent and environment; this approach is specific to DQN-like approaches that use experience replay and learn off-policy. Zhang et al. (2020) propose using bisimulation metrics to learn latent representations that capture invariance to reward-irrelevant parts of the observation.

3.4.3 Equivariance

van der Pol et al. (2020) propose a method to construct networks that are equivariant in their state-action spaces, which leads to faster convergence on a large set of environments that are fully symmetric in their state and action spaces. van der Pol et al. (2021) propose a distributed version of this approach that facilitates the sharing of group symmetries across multiple agents in the multi-agent RL setting.

3.4.4 Other approaches

Laskin et al. (2020) propose a large set of augmentation techniques, which can be used with any RL algorithm for both discrete as well as continuous control environments; these augmentation techniques could be combined with our approach to yield potentially better results.

3.5 Conclusions

We propose a modified batch RL setting, termed goal-conditioned batch RL, to solve complex tasks using data collected while learning policies for simpler tasks by exploiting invariances in the state and goal spaces of the environment. Our method combines data augmentation with a Siamese formulation in the architecture, resulting in agents being able to reach different goals much more successfully than those trained using standard RL and goal-conditioned RL methods. In addition to this, our method leads to better qualitative results in comparison with existing approaches.

3.6 Future directions

An interesting avenue for future work could be to analyze our method in an imitation learning setup. Currently, we observe that in the case of high-dimensional agents, especially Humanoid, some failures are due to the agent falling down. So, it would be interesting to see if these problems can be circumvented in the imitation learning setup. Another future direction we think is promising, is comparing the performance between maximum-entropy RL and other RL methods, during the data collection process. Since our approach uses a variant of the batch RL setting, the quality of samples in the batch has a significant impact on the performance of the goal-conditioned policy.

This chapter is based on the material as it appears in Beyond Tabula Rasa in Reinforcement Learning Workshop at the International Conference on Learning Representations, 2020 (Aditi Mavalankar, *Goal-conditioned Batch Reinforcement Learning for Rotation Invariant*

Locomotion). The dissertation/thesis author was the primary investigator and author of this paper.

Chapter 4

Discovering useful options

There are several different types of learning signals in the environment that an agent could make use of in order to maximize its reward. Many approaches in RL learn features that are relevant to the environment reward and use these features to learn the behaviour policy. However, it is conceivable that since these signals may be transient in that their presence in the environment may vary as the agent explores different parts of the observation space. In this case, the learnt features may be subject to the phenomenon of catastrophic forgetting: salient features may be forgotten when the agent is in a part of the observation space that does not contain those features. One way to manage this issue is by learning explicit policies that attain or maximize each of these individual learning signals, and execute these policies based on the part of the observation space in which the agent is present. This is a well-established technique in hierarchical RL, and one way to formalize it is through the options framework (Sutton et al., 1999; Precup, 2001), which we will discuss in depth in Section 4.1. Intuitively, an option can be thought of as a sequence of actions that the agent executes in the environment till the option is terminated. Since an option is, from this perspective, an extended action, it can be included as an action that the agent can choose to execute, in addition to the base actions provided by the environment.

One key advantage that options provide is that they enable the agent to plan and reason over longer time scales than simple one-step actions. This property of options makes them highly desirable in solving difficult RL problems like exploration and long-term planning. However, one detail is conspicuously missing: how do we design these options? Are they provided to the agent by the algorithm designer, or does the agent discover them autonomously? Most of the previous work on options assumes the former: options are provided to the agent by the algorithm designer, and the problem is then for the agent to learn how to manipulate these options. There is comparatively far less work on the latter, which is the option discovery problem.

Option discovery has been approached in several different ways in existing literature. McGovern and Barto (2001); Stolle and Precup (2002) frame option discovery as the problem of discovering useful subgoals in the form of bottleneck states: states that are frequently visited by the agent in several good trajectories. Machado et al. (2017; 2018) propose learning eigenoptions: options that are induced by intrinsic reward functions which capture diverse parts of the state space. Gregor et al. (2016) use ideas from mutual information maximization and empowerment to discover options. We discuss these and other proposed approaches for discovery in detail in Section 4.7.

In this chapter, we will approach the option discovery problem from another perspective: each option is induced by a pseudo-reward function, which is a learning signal that the agent can extract from the environment. Since there are several such signals, it may become intractable to manipulate such a large number of options. Thus, the problem of discovery is transformed into one of finding the right set of pseudo-reward functions and their corresponding options for the agent to manipulate. We will propose an approach that condenses a large number of pseudo-reward functions into a smaller set by considering linear combinations of these functions and their induced options. We will discuss the details of our proposed approach, along with its advantages and drawbacks in this chapter. We will also study the classes of environments for which our proposed approach is especially beneficial compared to other approaches.

4.1 Background

The intuition behind this particular framing of the discovery problem is that the agent has to discover the right set of learning signals from the environment that it can use in order to maximize the environment reward. We will first discuss how we define a learning signal:

Definition 4.1.1 (Cumulant)**.** A cumulant $\phi : \mathscr{S} \to \mathbb{R}^d$ is any feature that the agent can extract from its observation space. Here, d is the dimension of the feature.

Thus, we formalize a learning signal as a cumulant. Next, we define pseudo-reward functions corresponding to these cumulants:

Definition 4.1.2 (Pseudo-reward function)**.** A pseudo-reward function $c : \mathscr{S} \times \mathscr{A} \times \mathscr{S} \to \mathbb{R}$ maps each cumulant ϕ to a scalar value that indicates the pseudo-reward that the agent received in changing the value of the cumulant from $\phi(s)$ to $\phi(s')$, where s is the current state, and s' is the next state.

Note that here, we have not specified the exact form of the pseudo-reward since that can vary based on the type of cumulant. There are two types of cumulants: 1) cumulants that can be attained, and 2) cumulants that can be maximized. A simple instantiation of the pseudo-reward function in the case of cumulants that can be attained would look like:

$$c(s,a,s') = \begin{cases} 1, & \text{if } \phi(s') = \phi^* \\ 0, & \text{otherwise.} \end{cases} \tag{4.1}$$

where ϕ^* is some value that cumulant $\phi(s')$ should attain. Similarly, an instantiation of the pseudo-reward function in the case of cumulants that can be maximized could be:

$$c(s,a,s') = \begin{cases} 1, & \text{if } \phi(s') \geq \phi(s) \\ 0, & \text{otherwise} \end{cases} \tag{4.2}$$

This means that the agent would get a positive pseudo-reward if it increases or maintains the magnitude of the cumulant (norm is one way of indicating magnitude).

We framed the problem of discovery as one of finding the right set of cumulants for the agent to attain. In approaching discovery from this perspective, we alluded to an approach that is able to find a small good set of cumulants or linear combinations of cumulants in the base set. We will now visit the concepts underlying combining policies in the space of cumulants in greater detail.

4.1.1 Generalized Policy Evaluation and Generalized Policy Improvement

Definition 4.1.3 (Successor features)**.** The successor representation (Dayan, 1993) is a representation of the state in terms of its successor states to increase generalization among states that have similar successors.

If we assume that the reward can be decomposed into some feature ϕ of the space of transitions and a weight vector $\mathbf{w}$ in the following way:

$$r(s,a,s') = \phi(s,a,s')^\top \mathbf{w}$$

We can write the Q-function as (using ϕ_t instead of $\phi(s_t,a_t,s_{t+1})$ for clarity):

$$\begin{aligned} Q^\pi(s,a) &= \mathbb{E}^\pi[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... | s_t = s, a_t = a] \\ &= \mathbb{E}^\pi[\phi_{t+1}^\top \mathbf{w} + \gamma \phi_{t+2}^\top \mathbf{w} + \gamma^2 \phi_{t+3}^\top \mathbf{w} + ... | s_t = s, a_t = a] \\ &= \mathbb{E}^\pi[\sum_{i=t}^{T} \gamma^{i-t} \phi_{i+1} | s_t = s, a_t = a]^\top \mathbf{w} \\ &= \psi^\pi(s,a)^\top \mathbf{w} \end{aligned}$$

as shown by Barreto et al. (2017). Here, $\psi^\pi(s,a) = E^\pi[\sum_{i=t}^{T} \gamma^{i-t} \phi_{i+1} | s_t = s, a_t = a]$ are

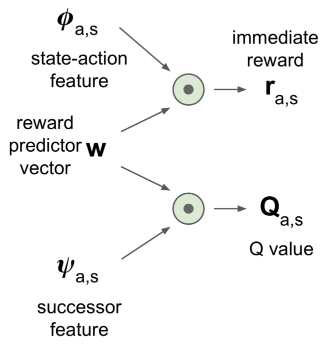


Figure 4.1. A visual explanation of the successor features ψ in terms of the features ϕ and Q-values, adapted from Zhu et al. (2017)

the successor features of (s,a) under policy π. Fig. 4.1 visually depicts the successor feature formulation.

We see that the Q-function has been disentangled into successor features, which are dependent on the environment dynamics, and the weights $\mathbf{w}$, which capture rewards. Thus, the Q-function now has two components that can be learned: $\mathbf{w}$ and ψ^{π}. However, ψ^{π} can be estimated using any RL algorithm, with ϕ as the reward in the Bellman equations. For this, we require ϕ to be either given, or learned.

For a single feature ϕ, we get its corresponding successor features ψ^{π}, where π is a policy maximizing or attaining feature ϕ. It is evident that ϕ here is a cumulant that induces a pseudo-reward function $c : \mathscr{S} \times \mathscr{A} \times \mathscr{S} \to \mathbb{R}$, which is used to learn policy π. This policy π is then used to determine the successor features ψ^{π}.

Now if we have n such features or cumulants $\{\phi_1, \phi_2, \ldots, \phi_n\}$, we will have n policies $\{\pi_1, \pi_2, \ldots, \pi_n\}$ that optimize for each of these cumulants respectively. We can get the successor features for each policy $\{\psi^{\pi_1}, \psi^{\pi_2}, \ldots, \psi^{\pi_n}\}$ and compute Q-functions $\{Q^{\pi_1}, Q^{\pi_2}, \ldots, Q^{\pi_n}\}$ respectively.

Extending this further, it is possible to obtain an evaluation of each policy π_i under each cumulant c_j i.e. $Q^{\pi_i}_{c_j}$ as shown by Barreto et al. (2019). Note that for now, we will use the terms *cumulant* and its induced *pseudo-reward function* interchangeably, but it is important to keep the difference in mind. We get the Q-values for all policies π_i under cumulant c_j as

$$Q_{c_j}^{\pi_i}(s,a) = c_j(s,a,s') + \gamma Q_{c_j}^{\pi_i}(s',a') \tag{4.3}$$

where $a' = \arg\max_b Q_{c_i}^{\pi_i}(s',b)$. Now that we have an evaluation of each policy under each cumulant, Barreto et al. (2019) propose *Generalized Policy Evaluation* that evaluates each policy $\pi_i, i \in \{1,\ldots,n\}$ under *any* linear combination of the cumulants $\{c_1,\ldots,c_n\}$.

Definition 4.1.4 (Generalized Policy Evaluation)**.** Given a set of n cumulants $\{c_1,\ldots,c_n\}$, we can evaluate a policy π under any linear combination of these cumulants given by $\mathbf{w} \in \mathbb{R}^n$ as shown in Barreto et al. (2019):

$$Q_c^{\pi} = \mathbb{E}^{\pi}\left[\sum_{k=0}^{\infty} \gamma^k \sum_{i=1}^{n} \mathbf{w}_i c_i(s_{t+k}, at+k, s_{t+k+1}) | S_t = s, A_t = a\right] = \sum_{i=1}^{n} \mathbf{w}_i Q_{c_i}^{\pi}(s,a) \tag{4.4}$$

where $c = \sum_{i=1}^{n} \mathbf{w}_i c_i$.

Thus, GPE gives us an evaluation of each policy with respect to any linear combination of the cumulants. Once the agent has evaluated each policy this way, it can greedily select the best policy in each state that maximizes this linear combination of cumulants. This is the underlying idea for Generalized Policy Improvement.

Definition 4.1.5 (Generalized Policy Improvement)**.** Given a set of policies $\{\pi_1,\pi_2,\ldots,\pi_n\}$ and the evaluation of each of these policies under the combined cumulant c, Generalized Policy Improvement (GPI) synthesizes a new policy that selects the greedy policy and its best action at each step as follows:

$$\pi_c(s) \in \arg\max_{a \in \mathscr{A}} \max_{i \in \{1,\ldots,n\}} Q_c^{\pi_i}(s,a) \tag{4.5}$$

for all $s \in \mathscr{S}$.

Here, we see that the resulting policy selects the greedy action at each step with respect to the combined cumulant; the important point to note here is that the action at each step could be the greedy action for a different policy, and thus, the new policy in this case would be different from any of the base policies, and an improvement over each of them. In the case where a single policy is better than all others, GPI reduces to executing that policy, and this provides the lower bound for GPI. Thus, in summary, GPI produces a policy that is at least as good as, and in most cases, better than, any of the individual policies.

4.1.2 The Option Keyboard

Equipped with the concepts underlying combining policies in the space of cumulants, we will now discuss the algorithm that forms the base for our proposed discovery approach. This algorithm is termed the *Option Keyboard* algorithm proposed by Barreto et al. (2019), and it utilizes the concepts of GPE and GPI to enable the agent to exhibit a combinatorial number of behaviours through composition of the base policies.

Definition 4.1.6 (Option)**.** An option consists of three components: 1) an initiation set $\mathscr{I} \subseteq \mathscr{S}$, 2) a policy $\pi : \mathscr{H} \times \mathscr{A} \rightarrow [0,1]$, and 3) a termination condition $\beta : \mathscr{S} \rightarrow [0,1]$. The initiation set $\mathscr{I}$ determines the set of states in which the option can begin execution. The policy π decides the actions the agent will take at each step until the option terminates according to the termination condition β, which gives the probability of terminating the option. Here, $\mathscr{H}$ refers to the history of states, actions, and rewards, or any function of these. Note that it is a partial history in that it consists of states, actions, and rewards experienced from the beginning of the option's execution. This is the most general definition of options; options are generally semi-Markovian. However, if we have Markov options, the policy is defined over states and actions, instead of over partial histories and actions.

The option's policy π can be learnt to maximize either the environment reward function, some pseudo-reward function, or a combination of the two. In this chapter, we will assume that an option's policy is learnt to maximize a particular pseudo-reward function induced by some cumulant.

If we consider combining options in the same way as we combined policies, we could simply use GPE and GPI on the options' policies. However, since options also contain two other parts, namely the initiation set and the termination condition, we also need to take these into consideration while performing GPE and GPI.

We consider the initiation set to be the full state space $\mathscr{S}$ for now, as is done in several works in literature.

For the termination condition β, we could formulate it in multiple ways. One way to do it is to define termination in the same way as we defined the cumulants:

$$\beta(h) = \begin{cases} 1, & \text{if cumulant has been attained} \\ 0, & \text{otherwise.} \end{cases} \tag{4.6}$$

This means that the probability of terminating the option is 0 everywhere except in cases where the option has attained the cumulant it was optimizing for.

Another way to formulate termination is by assigning a fixed probability of terminating at each step. This idea of termination was proposed by Sutton et al. (2011), who termed these Q-functions *General Value Functions*. Here, the termination probability is defined in terms of the discount factor γ as:

$$\beta(h) = 1 - \gamma \tag{4.7}$$

Note that here, the termination condition is independent of the state; the option is likely to terminate in each state with the same probability $1 - \gamma$.

Barreto et al. (2019) propose formulating the termination condition as a separate action τ that the agent can add to its action space. Thus, the action space now becomes $\mathscr{A}^+ \equiv \mathscr{A} \cup \{\tau\}$, and the termination condition can now be written as:

$$\beta(s) = \begin{cases} 1, & \text{if } \tau = \arg\max_{a \in \mathscr{A}^+} Q^{\pi}(s,a) \\ 0, & \text{otherwise} \end{cases} \tag{4.8}$$

Thus, the termination condition is learnt as a part of the option's policy and is satisfied when it yields the highest expected discounted return compared to the other actions. This is reinforced by rewarding the termination action after the cumulant has been attained by the option. We will use this formulation of the termination condition throughout the rest of the chapter.

Now that we have determined the initiation set and termination condition for each option, we can use GPE and GPI for options in the same way as we defined them for policies.

We will first learn the Q-values of each option ω_i under each cumulant c_j, $i, j \in \{1, \dots, n\}$ using temporal difference learning:

$$Q^{\omega_i}_{c_j}(h,a,s') = c_j(h,a,s') + \gamma Q^{\omega_i}_{c_j}(h',a') \tag{4.9}$$

for each transition (s,a,s'), where h is the partial history at state s, and h' is the partial history at state s'. Here, a' is the greedy action while following the option maximizing cumulant c_i:

$$a' = \arg\max_{b \in \mathscr{A}^+} Q^{\omega_i}_{c_i}(h',b) \tag{4.10}$$

Note that the greedy policy for each cumulant c_i can be extracted as in Q-learning by doing

$$\pi_i(h) \in \arg\max_{a \in \mathscr{A}^+} Q^{\omega_i}_{c_i}(h,a) \tag{4.11}$$

The approach that details the learning of these options, i.e. the Q-functions for each option ω_i under each cumulant c_j is given in Algorithm 2 (taken from Barreto et al. (2019). Here, u is a function that updates the history of the executing option.

Algorithm 2. Learning options

```
Require: $\begin{cases} C = \{c_1, c_2, ..., c_n\} \text{ : set of all } n \text{ cumulants} \\ \varepsilon_1 \text{ : probability of changing cumulant} \\ \varepsilon_2 \text{ : exploration parameter while learning options} \\ \alpha \text{ : learning rate} \\ \gamma \text{ : discount factor} \end{cases}$
1:  Select initial state $s \in S$
2:  $k \leftarrow$ Uniform($\{1, 2, ..., N\}$)
3:  repeat
4:      if Bernoulli($\varepsilon_1$) = 1 then
5:          $h \leftarrow s$
6:          $k \leftarrow$ Uniform($\{1, 2, ..., n\}$)                    ▷ pick a random $c_k$
7:      if Bernoulli($\varepsilon_2$) = 1 then $a \leftarrow$ Uniform($\mathscr{A}^+$)            ▷ explore
8:      else $a \leftarrow \text{argmax}_b \tilde{Q}^{\omega_{c_k}}_{c_k}(h, b)$                    ▷ GPI
9:      if $a \neq \tau$ then
10:         execute action $a$ and observe $s'$
11:         $h' \leftarrow u(h, a, s')$                                   ▷ update history
12:         for $i \leftarrow 1, 2, ..., n$ do
13:             $a' \leftarrow \text{argmax}_b \tilde{Q}^{\omega_{c_i}}_{c_i}(h', b)$
14:             for $j \leftarrow 1, 2, ..., n$ do
15:                 $\delta \leftarrow c_j(h, a, s') + \gamma \tilde{Q}^{\omega_{c_i}}_{c_j}(h', a') - \tilde{Q}^{\omega_{c_i}}_{c_j}(h, a)$
16:                 $\theta_{\omega_i} \leftarrow \theta_{\omega_i} + \alpha \delta \nabla_{\theta_{\omega_i}} \tilde{Q}^{\omega_{c_i}}_{c_j}(h, a)$
17:         $s \leftarrow s'$
18:     else                                                    ▷ update termination values
19:         for $i \leftarrow 1, 2, ..., n$ do
20:             for $j \leftarrow 1, 2, ..., n$ do
21:                 $\delta \leftarrow c_j(h, \tau) - \tilde{Q}^{\omega_{c_i}}_{c_j}(h, \tau)$
22:                 $\theta_{\omega_i} \leftarrow \theta_{\omega_i} + \alpha \delta \nabla_{\theta_{\omega_i}} \tilde{Q}^{\omega_{c_i}}_{c_j}(h, \tau)$
23: until stop criterion is satisfied
24: return $Q_C = \{\tilde{Q}^{\omega_{c_i}}_{c_j} \forall (i, j) \in \{1, 2, ..., n\}^2\}$
```

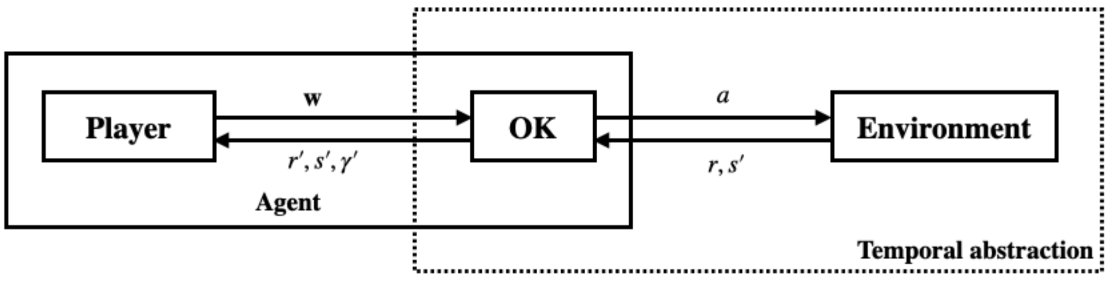


Figure 4.2. The Option Keyboard framework from Barreto et al. (2019). The agent consists of a high-level keyboard player and a low-level OK. The player issues a command **w** to the OK, which interacts with the environment at each step to execute this command.

Once these Q-functions are learnt, we can use GPE and GPI to combine these options for any linear combination **w** of the cumulants without any additional learning involved.

The obvious next question is how this weight vector **w** should be selected by the agent. For an environment reward function that can be expressed exactly as one linear combination of the cumulants, **w** can simply be estimated using any supervised learning technique, such as linear regression. However, when the reward function is non-linear, **w** could change over time, depending on the state of the environment. In this case, the agent needs to learn a separate policy $\pi' : \mathscr{S} \times \mathscr{W} \to [0,1]$ mapping each state s to a linear combination over the cumulants **w** that would yield the highest return. In other words, we can frame the problem of learning **w** as an RL problem. The agent can use any RL algorithm to learn a policy over this modified action space $\mathscr{W}$. The selected **w** then induces an option that is executed using GPE and GPI till termination. This framework is illustrated in Figure 4.2.

We will now go over the analogy used by Barreto et al. (2019) to explain the Option Keyboard algorithm shown in Figure 4.2. The algorithm is considered analogous to a piano keyboard. Each base option learnt by the agent is a *key*. The agent can play keys, or *chords*, which are the linear combinations **w** over these keys. These keys constitute the *Option Keyboard* (OK), which can play these keys and chords using GPE and GPI. The OK interacts with the environment at every time step and chooses the greedy action given by GPI. Once the key or chord is terminated, the OK returns control back to the keyboard *player*. The player then selects the next chord to be played, $\mathbf{w}'$, and passes it to the OK which repeats this same process. Note that the keys are learnt independently *before* this player learning phase and then frozen during player learning. The player uses the discounted return r' from the execution of each key to update the Q-values of the selected chord $\mathbf{w}$. The OK algorithm is as shown in Algorithm 3, and the player algorithm, for which Barreto et al. (2019) use Q-learning (but could replace it with any other RL algorithm) is detailed in Algorithm 4. Both these algorithms have been adapted from Barreto et al. (2019).

Algorithm 3. Option Keyboard

```
Require: $\begin{cases} \mathscr{Q}\text{: set of all pretrained value functions} \\ \mathbf{w}\text{ : weight vector over cumulants} \\ s\text{ : current state} \\ \gamma\text{ : discount factor} \end{cases}$
1:  $h \leftarrow s$
2:  $r' \leftarrow 0$
3:  $\gamma' \leftarrow 1$
4:  repeat
5:      $a \leftarrow \arg\max_{a'} \max_{i} \left[ \sum_j \mathbf{w}_j \mathscr{Q}_{c_j}^{\omega_i}(h, a') \right]$
6:      if $a \neq \tau$ then
7:          Take action $a$ and observe $r$, $s'$
8:          $r' \leftarrow r + \gamma' r$
9:          if $s'$ is terminal then $\gamma' \leftarrow 0$
10:         else $\gamma' \leftarrow \gamma' \gamma$
11:         $h \leftarrow u(h, a, s')$
12: until $a = \tau$ or $s'$ is terminal
13: return $s', r', \gamma'$
```

An important point here is that while in general $\mathbf{w} \in \mathbb{R}^n$, where n is the number of cumulants, learning Q_{player} over all possible configurations of $\mathbf{w}$ would require more sophisticated RL algorithms that handle continuous action spaces. Since that is not the focus of the approach, Barreto et al. (2019) consider $\mathbf{w} \in \{-1, 0, 1\}^n$, thus converting it to a discrete action space with cardinality 3^n. Thus, we will also consider $\mathscr{W}$ to be discrete.

Algorithm 4. Player

```
Require: { OK: option keyboard
         { $Q_\Omega$ : value functions for pretrained options
         { $\mathscr{W}$ : set of all possible combinations over options
         { $n$ : number of training steps
         { $\alpha, \varepsilon, \gamma$ : hyperparameters
1: Create $Q_{player}(s, \mathbf{w})$ parameterized by $\theta_{Q_{player}}$
2: Select initial state $s \in S$
3: while number of steps $\leq n$ do
4:     if Bernoulli($\varepsilon$) $= 1$ then $\mathbf{w} \leftarrow$ Uniform($\mathscr{W}$)
5:     else $\mathbf{w} \leftarrow \text{argmax}_{\mathbf{w}' \in \mathscr{W}} Q_{player}(s, \mathbf{w}')$
6:     $s', r', \gamma' \leftarrow OK(s, \mathbf{w}, Q_\Omega, \gamma)$
7:     $\delta \leftarrow r' + \gamma' \max_{\mathbf{w}' \in \mathscr{W}} Q_{player}(s', \mathbf{w}') - Q_{player}(s, \mathbf{w})$
8:     $\theta_{Q_{player}} \leftarrow \theta_{Q_{player}} + \alpha \delta \nabla_\theta Q(s, \mathbf{w})$
9:     if $s'$ is terminal then reset $s$
10:    else $s \leftarrow s'$
```

4.2 Approach

The key takeaway from the OK algorithm is that for any given set of cumulants $\{c_1, \ldots, c_n\}$, the agent is now equipped to synthesize an exponential number of new behaviours from a base set of keys. However, there are two questions that still remain unanswered:

1. How does the agent obtain these base cumulants?

2. Given that the keyboard player can synthesize an exponential number of behaviours, it is faced with the modified decision problem: what is the best action out of 3^n possible actions? This occurs because we transformed the problem of selecting the best action from the environment's action space $\mathscr{A}$ to the best action from the entire space of possible chords $\mathscr{W}$.

If the number of cumulants n is small, learning a player over all possible chords in $\mathscr{W}$ is straightforward, but as we increase n, the space of chords increases exponentially, resulting in a setting where $|\mathscr{A}| \ll |\mathscr{W}|$, thus making it potentially as hard as or a harder decision problem for the agent.

We propose an approach for discovery that answers both of these questions by answering the following question:

Can the agent discover a good keyboard that is smaller than the full keyboard and maximizes the environment reward?

The idea here is that if the number of cumulants n is large, the player would have to learn a policy over 3^n possible actions, i.e. a linear increase in the number of cumulants results in an exponential increase in the action space of the player. Thus, our approach proposes to maintain a fixed keyboard size $N \ll n$, so that the player now has to learn a policy over 3^N actions instead of 3^n. The question now becomes one of determining which cumulants, or combinations of cumulants, should constitute the keyboard of size N in order to achieve the maximum possible environment reward.

In order to understand the intuition behind our proposed approach, let us revert to the keyboard analogy described earlier. The aim is to find the best N keys to compose the keyboard; the mechanism underlying the playing of this keyboard is the same as described in Algorithm 4. While each key in the keyboard could maximize a single base cumulant, we hypothesize that compressing chords into keys might be more beneficial especially in scenarios where the desired keyboard size is less than the number of base cumulants required to achieve good or optimal performance. Thus, each key in the keyboard could be either a key, for which we already have a behaviour from the pretraining phase, or a chord, for which we can synthesize new behaviour using GPE and GPI.

Based on this intuition, we can view the problem of discovery as a new level of temporal abstraction added to the OK framework in Figure 4.2. In this case, the player receives a new keyboard configuration intermittently, for which it learns how to play chords in order to maximize the return. The reconstitution of the keyboard is a separate problem from that of learning the player and learning the options, each of which happens separately. An illustration of this discovery framework has been shown in Figure 4.3. Here, the OK framework is part of the larger

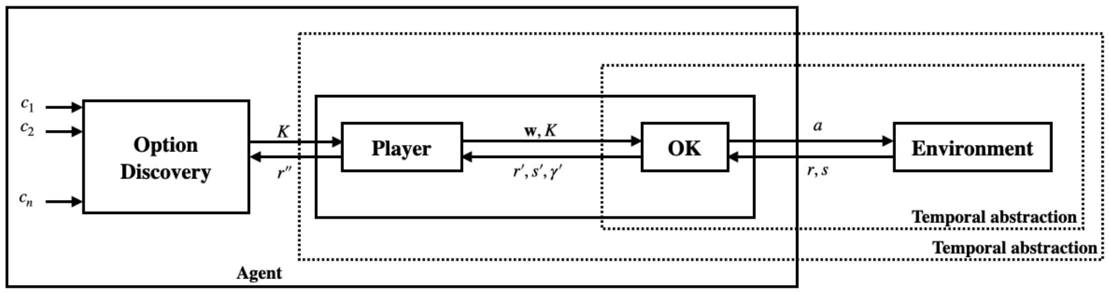


Figure 4.3. Our proposed Option Discovery framework: there is an additional level of temporal abstraction, the discovery module. This module synthesizes a keyboard and passes it to the player, which is then learnt for a fixed interval, following which it returns the observed rewards to the discovery module.

discovery framework, which adds an additional module: the discovery module. The discovery module periodically synthesizes a new keyboard K from the set of cumulants $\{c_1, c_2, \dots, c_n\}$, for which the player learns a policy to maximize the environment return using the OK algorithm, as described previously. It uses the rewards experienced by the player while learning a policy for K as feedback for synthesizing the next keyboard K'. So, for instance, if a certain chord (c_1, c_5) was considered a good chord by the keyboard player, this chord, or the underlying cumulants c_1 and c_5 would be favoured in consequent configurations of the keyboard. Now, equipped with an understanding of this framework, we will discuss our proposed approach for discovery.

Our proposed approach belongs to the generate-and-test class of algorithms: the agent periodically generates a keyboard consisting of keys and/or chords, and learns a player for a fixed time period. During this time, it also collects additional information about chords, like which chords were played most often, which chords led to the highest reward, and so on. Note that here we have used the term chords, but these also include keys, since keys can be considered special instances of chords. At specific intervals, the keyboard is reconstituted, and a new player is learnt over the improved keyboard. This improved keyboard is synthesized in the following manner: the *best M* chords from the previous keyboard are retained by transforming them into keys in the new keyboard. The remaining $N-M$ slots in the keyboard are filled by selecting the

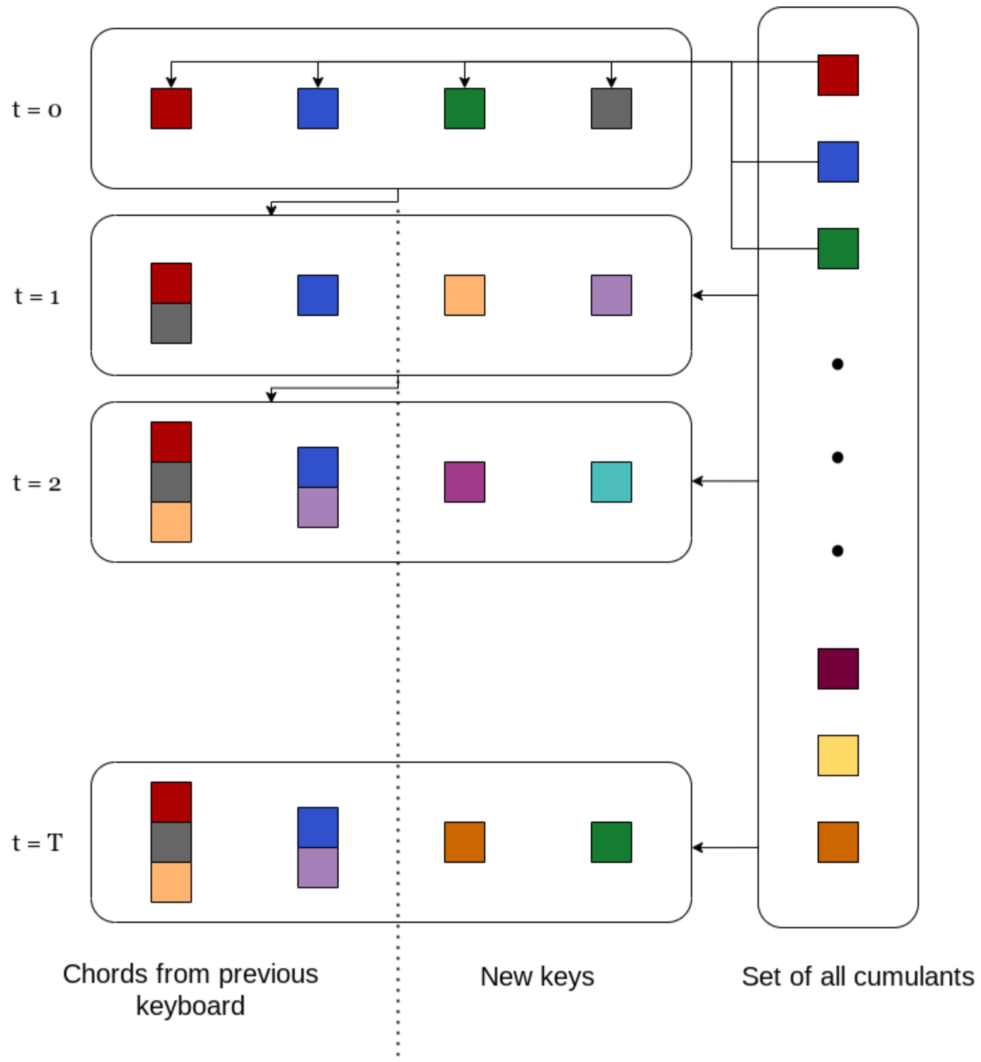


Figure 4.4. Our proposed approach for option discovery: at fixed intervals, the agent synthesizes the keyboard by selecting the best chords from the previous OK and the best keys from the base set of keys using UCB. The time is at the scale of discovery, not actual time steps. A desirable outcome would include useful chords being retained across keyboards.

Algorithm 5. OK Synthesis

```
Require: {
    select_keys : function used to select keys from the base set
    $N$ : desired size of the OK
    $M$ : number of chords to be transformed into keys in the OK
    $T$ : number of steps before the OK is reconfigured
    $C$ : base set of keys
    $\alpha$ : learning rate for bandit
1: $Q_{bandit} \leftarrow [0,0,...,0]$    ▷ Bandit Q-values; $Q_{bandit} \in \mathbb{R}^{|C|}$
2: $N_{bandit} \leftarrow [0,0,...,0]$    ▷ Visitation frequencies; $N_{bandit} \in \mathbb{Z}^{|C|}$
3: $t_{bandit} \leftarrow 0$    ▷ Timesteps elapsed at the highest level of abstraction i.e. discovery
4: $C' \leftarrow$ select_keys$(Q_{bandit}, N_{bandit}, t_{bandit}, N)$    ▷ Current set of keys in the OK $C'$
5: $t_{bandit} \leftarrow t_{bandit} + 1$
6: loop
7:     for $T$ steps do
8:         Learn OK player with $C'$ and keep statistics on use of chords $\mathbf{w}$
9:     for each chord $\mathbf{w}'$ do    ▷ update $Q_{bandit}$
10:        $r \leftarrow$ reward_stats$[\mathbf{w}']$    ▷ statistic relevant to reward collected by $\mathbf{w}$
11:        for each $c$ in $\mathbf{w}'$ do
12:            $Q_{bandit}[c] \leftarrow Q_{bandit}[c] + \alpha(r - Q_{bandit}[c])$
13:    for each $c$ in $C'$ do    ▷ update $N_{bandit}$
14:        $N_{bandit}[c] \leftarrow N_{bandit}[c] + 1$
15:    $C'' \leftarrow \phi$
16:    for $i \leftarrow 1$ to $M$ do    ▷ Transform $M$ best chords into keys
17:        Let $\mathbf{w}'$ be the chord with the $i$-th best per-step reward
18:        $C'' \leftarrow C'' \cup \{\Sigma_i \mathbf{w}'_\mathbf{i} c_i\}$    ▷ Add best chords to next OK
19:    $C' \leftarrow C'' \cup$ select_keys$(Q_{bandit}, N_{bandit}, t_{bandit}, N - M)$    ▷ Add $N - M$ keys to OK
20:    $t_{bandit} \leftarrow t_{bandit} + 1$
```

Algorithm 6. Select keys

```
Require: {
    $Q_{bandit}$ : Q-values for each arm i.e. each cumulant
    $N_{bandit}$ : Visitation frequency for each arm
    $t_{bandit}$ : Number of time steps that have elapsed
    $\varepsilon$ : Exploration parameter
    $N$ : Number of cumulants to be generated
1: for each cumulant $c$ do
2:     $U[c] \leftarrow Q_{bandit}[c] + \varepsilon\sqrt{\frac{\ln t_{bandit}}{N_{bandit}[c]}}$
3: Sort $U$ in decreasing order of values
4: return top $N$ cumulants
```

top keys from the initial set of keys. These keys are evaluated in the same way as chords from the previous keyboard are evaluated – in terms of their utility. This utility, as mentioned before, could be the average or discounted reward collected while executing the chord/key, the frequency at which keys are selected, among others. However, as opposed to chords, which are transient in that the chords that the keyboard player can play change with each keyboard reconfiguration, the evaluation of keys is static. This is done by defining a bandit with each key as an arm of the bandit. The bandit stores the expected utility of each key, and while configuring a new keyboard, keys are selected by the agent factoring in these estimated utilities of keys. This selection can be done using any method; we use the Upper Confidence Bound (UCB) algorithm to select keys.

$$A_t = \arg\max_a \left[Q_t(a) + c\sqrt{\frac{\log t}{N_t(a)}} \right] \tag{4.12}$$

Here, $Q_t(a)$ gives the utility of each key (each key is an arm a of the bandit), c controls how exploratory the selection of keys should be, t is the number of time steps that have elapsed, and $N_t(a)$ gives the number of times each key a has been selected. An illustrative diagram of our proposed approach is shown in Figure 4.4. We can see that the keyboard is synthesized intermittently by retaining some chords from the previous keyboard and selecting the remaining keys based on their UCB values. Our exact approach for synthesizing the keyboard is detailed in Algorithm 5, and the selection of keys using UCB is detailed in Algorithm 6.

4.3 Environment: The Scavenger

We first use the Scavenger environment from Barreto et al. (2019) as a testbed for our proposed approach. The Scavenger environment is a Gridworld in which there are two types of resources, or nutrients, and three food items that contain these in different proportions. A visual illustration is given in Figure 4.5. The three food items contain the two resources in the following proportions: 1) red: (1, 0), 2) blue: (0, 1), and 3) purple: (1, 1). The task is to maintain these resources within certain limits in order to achieve high reward. The observation space of

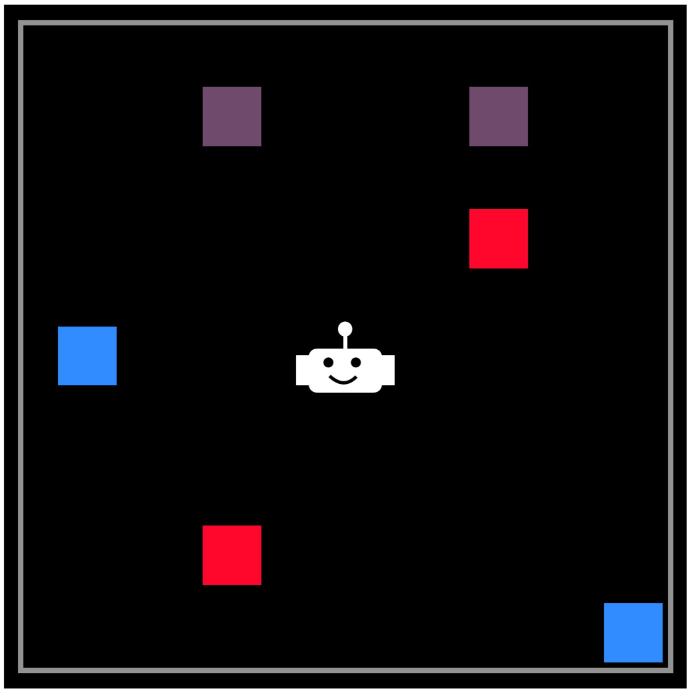


Figure 4.5. The Scavenger environment from Barreto et al. (2019). The agent has to manage resources in certain proportions to achieve high rewards.

the environment contains the grid configuration and the quantity of nutrients of each type that the agent has collected. The environment is egocentric to the agent; thus, the grid is toroidal and the agent's position is fixed at the centre of the grid. The action space consists of four discrete actions: moving one position up, down, left, or right in the grid. The environment dynamics have the following properties: 1) at each step, the agent's collection of nutrients decays by a fixed amount, 2) when the agent lands at a position containing a food item, the constituent nutrients are added to its collection i.e. the food item is "consumed", and 3) when a food item is consumed, it is respawned at a random position in the grid. The reward function is piecewise linear in the quantity of nutrients collected by the agent. If the agent's collected quantities of nutrients 1 and 2 are denoted by x_1 and x_2 respectively, we define the *desirability* of a resource in Table 4.1.

The desirability d in Table 4.1 indicates the reward the agent gets on consuming a food item containing these nutrients, as a function of its existing quantities of these nutrients (x_1, x_2). For example, if the agent has in store $(2, 3)$, and collects a red food item, which only has nutrient

Table 4.1. Reward function for the Scavenger environment. d indicates the desirability of a nutrient.

$$\mathrm{d(x_1)} = \begin{cases} +1, & \text{if } x_1 \leq 10 \\ -1, & \text{if } x_1 > 10 \end{cases} \quad \mathrm{d(x_2)} = \begin{cases} -1, & \text{if } x_2 < 5 \\ +5, & \text{if } 5 \leq x_2 \leq 25 \\ -1, & \text{if } x_2 > 25 \end{cases}$$

1, it gets a reward of $+1$, since it contains < 10 units of nutrient 1. However, if it collects a purple food item, which has both nutrients 1 and 2, it gets $+1$ for nutrient 1 and -1 for nutrient 2, yielding a net reward 0. This type of reward function leads to a complex resource management problem, requiring the agent to plan over different types of food items it can collect, and also learn policies that enable it to move towards those particular food items. This particular property of the environment makes it a good testbed for approaches involving temporal abstraction, since it seems intuitively that learning behaviour policies for the agent to move towards different types of food items would enable the high-level player to plan over longer horizons.

4.4 Experiments

In this section, we will evaluate our proposed approach on the Scavenger environment, and compare it with the OK. This first set of experiments will focus on understanding whether or not our approach is able to discover good cumulants from a given large set of cumulants. Thus, we define a larger set of cumulants of the following form: the agent has to collect n_1 units of nutrient 1, followed by n_2 units of nutrient 2. The entire list of cumulants of this form is given in Table 4.2. Per our approach, the first phase involves learning options maximizing these cumulants. Thus, the agent first learns the Q-functions $\{Q_{c_1}^{\omega_1}, Q_{c_2}^{\omega_1}, \ldots, Q_{c_{10}}^{\omega_1}, Q_{c_1}^{\omega_2}, \ldots, Q_{c_{10}}^{\omega_2}, \ldots, Q_{c_1}^{\omega_{10}}, \ldots, Q_{c_{10}}^{\omega_{10}}\}$ as described in Algorithm 2. The termination action is rewarded for each option when it has finished collecting n_1 food items containing nutrient 1, followed by n_2 food items containing nutrient 2. The function that determines how useful a chord is, i.e. the utility function used to update the Q-values for the bandit, is the frequency of a chord being played by the keyboard player. If a certain chord is played more often than others, it might be more useful, and thus, should be

Table 4.2. Cumulants for the Scavenger environment. We define cumulants in a general way allowing them to induce a mix of Markov and semi-Markov options.

	n_1	n_2
1	1	0
2	0	1
3	0	0
4	1	1
5	5	0
6	0	5
7	10	0
8	0	10
9	1	2
10	2	1

retained in future keyboards.

The results of our proposed approach for discovery compared with the oracle OK are shown in Figure 4.6. The oracle OK is the OK that contains the keys that correspond to collecting each individual type of resource i.e. cumulants 1 and 2 in Table 4.2. The keyboard is synthesized every 1 million steps, and has size 2. We use the frequency of chords selected by the player as the utility function to determine the "best" chord for the subsequent keyboard. The results indicate that the agent performs badly in the first $\sim$ 7 million steps, following which it reaches similar episodic rewards to the oracle OK.

We now further investigate why this is the case with the qualitative results in Figure 4.7. Every 1 million steps, the keyboard is retained in such a way that the best key or chord from the previous keyboard is retained as a key in the next keyboard, and the remaining slot in the keyboard is filled by the best key selected from the base set of keys. The latter is indicated in yellow in the plot. For the retained chords, if a key is present in the chord with weight $+1$, it is indicated in blue, and if it is present in the chord with weight -1, it is indicated in red (remember that we considered the possible set of combinations $\mathbf{w} \in \{-1, 0, 1\}^{|K|}$, where $|K|$ is the keyboard size, in this case, 2.

Initially, since none of the keys have been explored, 2 keys are selected from the base

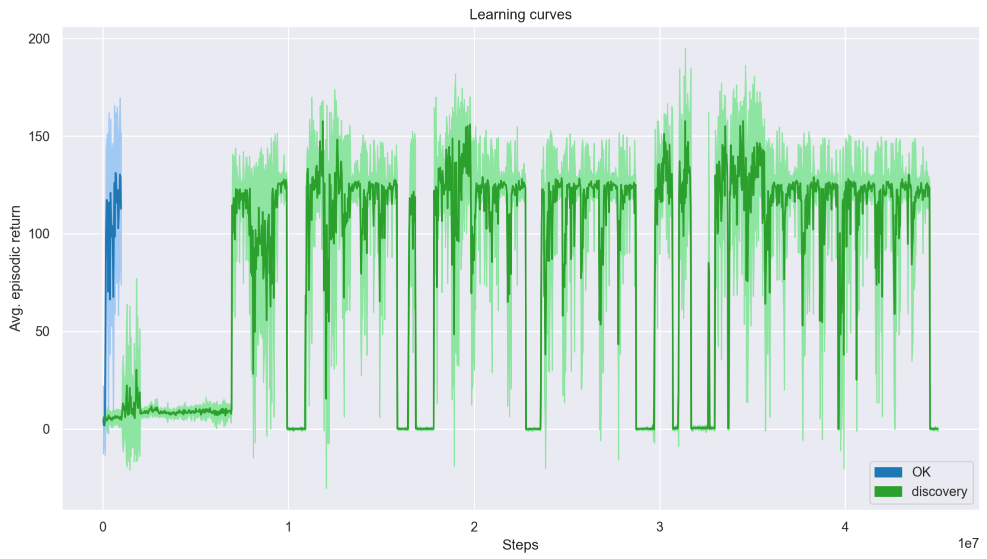


Figure 4.6. Initial results on the Scavenger: learning curves for our approach v/s the oracle OK

set of keys using UCB to constitute the keyboard. The process of retaining the best chord and selecting one key continues henceforth. An interesting point to note is that the size of the retained chord increases from 3M-7M steps; even though the keyboard size is 2, the chord size exceeds the keyboard size. This is an important benefit of our approach: *it allows multiple useful behaviours to be compressed into a single behaviour that can be executed without any additional learning required.* However, we do not see any performance gains till 7M steps, which is the first time that the key corresponding to cumulant c_2 is part of the keyboard. This is the key that corresponds to selecting a single food item containing nutrient 2.

If we revisit the reward function from Table 4.1, we see that if the agent incurs a negative or no reward by collecting food items with nutrient 2 a few times, it can get much higher positive rewards subsequently when it collects these food items, i.e. +5 each time it collects food items with nutrient 2. This makes the second key particularly desirable. This is reflected in the learning curve after 7M steps; the reward increases to ~ 125, which is close to that of the oracle OK. It is worth noting that even though the agent does experience what we is intuitively the “oracle” OK at 8M steps, it selects the same key twice in the keyboard. The reason for this is that this

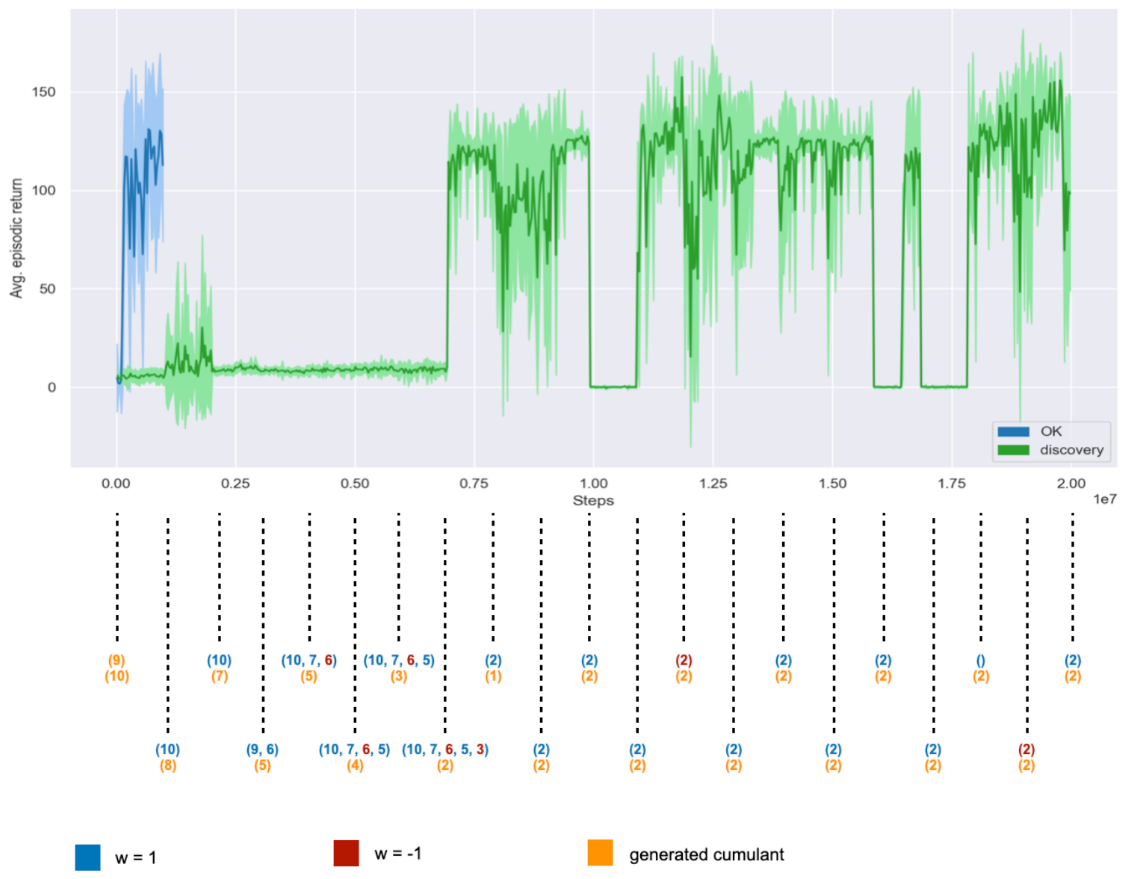


Figure 4.7. Initial results on the Scavenger: analyzing the keyboard synthesis over time and the corresponding learning curve.

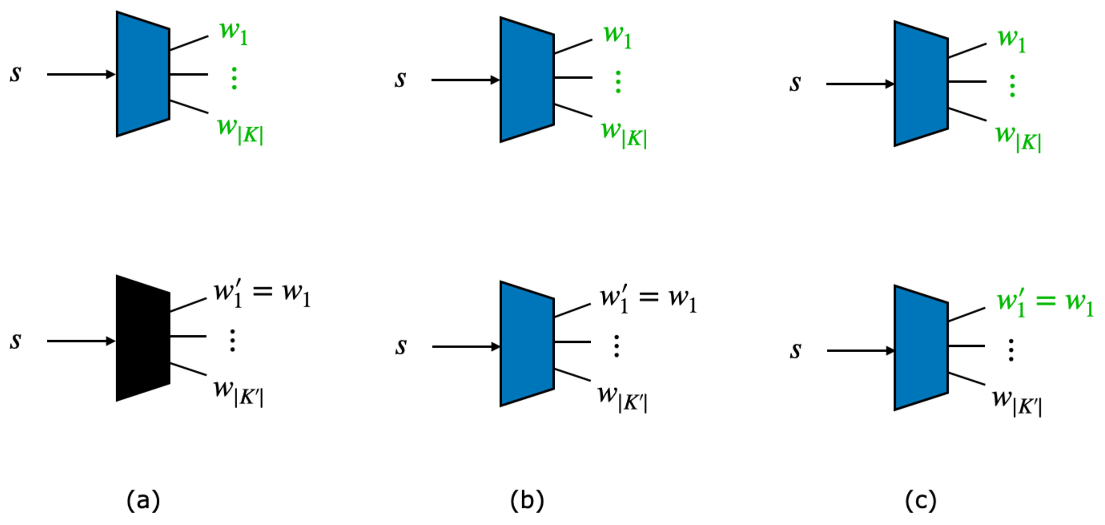


Figure 4.8. Changing player across consequent keyboards K and K': (a) Reset the whole network, (b) Reset the final layer and retain the torso, (c) Reset only those parameters of the final layer that have changed compared to the previous keyboard. The parameters that are reset are shown in black; the blue refers to the torso for keyboard K, which is retained in (b) and (c), and the green refers to the final layer parameters for keyboard K, some of which are retained in (c).

is the most naïve implementation of our proposed approach; we do not even impose any basic constraints on the keyboard such that the keys have to be distinct. Since the second key is clearly dominant when it comes to the reward, it is selected. However, since this is a simple test of our approach, the more important takeaway is that the Q-values of each arm in the bandit i.e. each key indicate that the second key is the best key, followed by the first, which is consistent with the oracle OK.

It is also worth noting that the performance of the agent sees several dips intermittently. These dips correspond to the times when the keyboard is reconfigured. All the Q-functions, for the options, as well as the player, are learnt using online Q-learning with deep neural networks as function approximators. The options are learnt during the pretraining stage and then frozen. The player learns a Q-function over the set of all chords as its action space. For a fixed keyboard, learning a player is straightforward. However, in our approach, the player has to constantly change as the keyboard changes. Thus, there are three choices when it comes to learning the player, as illustrated in Figure 4.8: 1) reset all the parameters of the player's neural network, 2)

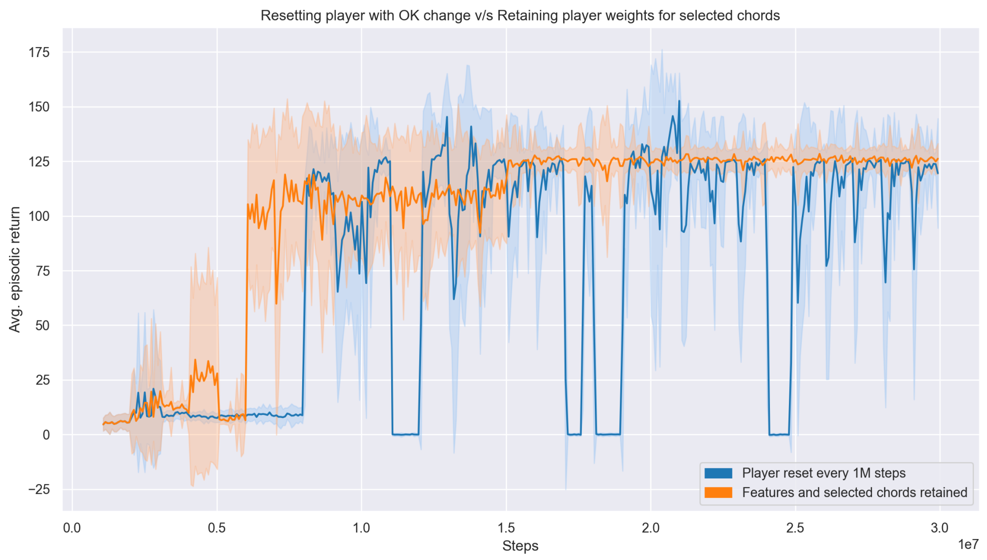


Figure 4.9. Results on the Scavenger: retaining torso and shared final layer parameters across keyboards v/s resetting parameters every time a new keyboard is synthesized. Retaining the parameters for retained chords results in the smooth orange learning curve.

retain the torso parameters and reset the final layer parameters, or 3) retain the torso parameters, as well as the final layer parameters that correspond to chords that are retained across keyboards. The intuition behind retaining the torso is that the torso captures important features of the observation, which may enable the player to learn the Q-values faster. The intuition behind retaining the final layer parameters is the same: if the previous player has learnt the Q-values for a certain chord which also occurs in the current keyboard, resetting the final layer parameters will require the new player to relearn these Q-values, which can be avoided by retaining these weights. We use this idea to transfer weights across players in the Scavenger environment and compare it with the previous version resetting the player in Figure 4.9. This simple idea results in the smooth orange curve that retains performance across keyboards compared to the blue curve which has dips in performance as the keyboard is reconfigured.

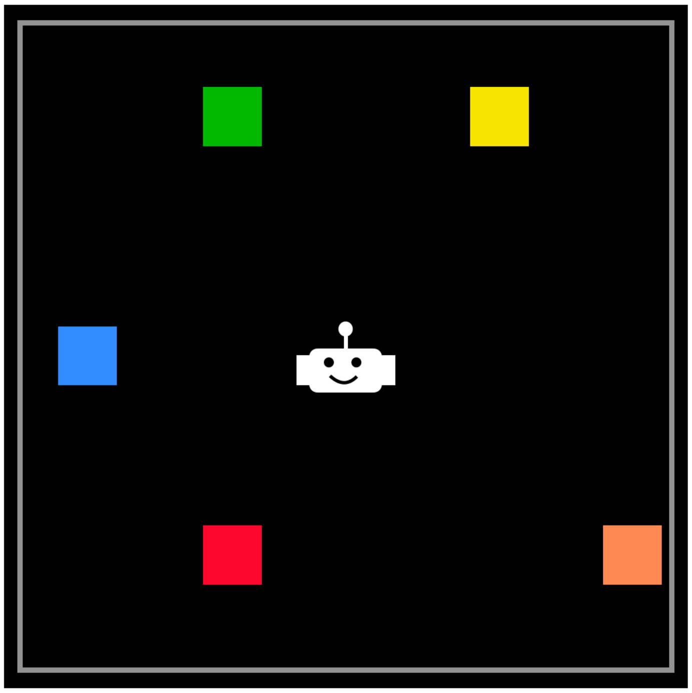


Figure 4.10. The modified Scavenger environment with 5 different types of nutrients.

4.5 Towards more complex environments

While our approach shows the intended results in the Scavenger domain, its true purpose lies in discovering a good small keyboard given a larger set of cumulants, not all of which may be useful. Our previous experiment considered a large set of cumulants designed as the attainment of different amounts of 2 types of nutrients; in this section, we propose environments in which the larger set of cumulants can be more naturally defined as the attainment of different *types* of objects, as opposed to attaining them in different proportions. This allows us to focus on environments in which there can be several different types of learning signals, out of which a few useful ones can be used to compose the keyboard, which is the main objective of our approach.

4.5.1 A modified Scavenger environment

We propose modifying the original Scavenger environment to include more types of food items. Our proposed environment includes 5 food items, each of which contains a different

Table 4.3. Reward function for the modified Scavenger environment with 5 different types of nutrients.

$$\mathrm{d}(\mathrm{x}_1) = \begin{cases} +1, & \text{if } x_1 \leq 10 \\ -1, & \text{if } x_1 > 10 \end{cases} \quad \mathrm{d}(\mathrm{x}_2) = \begin{cases} -1, & \text{if } x_2 < 5 \\ +5, & \text{if } 5 \leq x_2 \leq 25 \\ -1, & \text{if } x_2 > 25 \end{cases}$$

$$\mathrm{d}(\mathrm{x}_3) = +1 \quad \mathrm{d}(\mathrm{x}_4) = 0 \quad \mathrm{d}(\mathrm{x}_5) = 0$$

type of nutrient. The environment dynamics and action space remain the same as the original Scavenger environment. The modified reward function is shown in Table 4.3. The first two types of nutrients have the same reward associated with them as earlier. The third resource is always useful i.e. the agent gets a constant reward $+1$ every time it consumes the food item containing nutrient 3. The remaining two food items are "distractors"; the agent can consume them but they are inconsequential in terms of the environment reward.

In this case, like before, the agent learns options for consuming each of these respective food items. The OK with all keys has a total of $3^5 = 243$ possible chords it can play; thus, the player's output space is this large set of 243 chords. On the other hand, discovering a keyboard of size 2 leads to the agent having to learn a player over $3^2 = 9$ possible chords, which is a much smaller space. We compare our approach for discovery on a keyboard of size 2, with 1 chord from the previous keyboard and 1 key selected from the base set, with this full OK baseline, i.e. the OK that has access to the full set of keys, and also with a fixed OK of size 2 that has randomly selected keys. Furthermore, we compare with a variant of our approach, which does not retain *any* chords from the previous keyboard; it only selects keys from the base set and updates them through the statistics collected by the player. The results of all of these baselines, as well as our approach that selects both keys and chords, in Figure 4.11. We indicate the optimal reward for this environment with a green dashed line – this is the episodic reward that the full OK baseline eventually achieves.

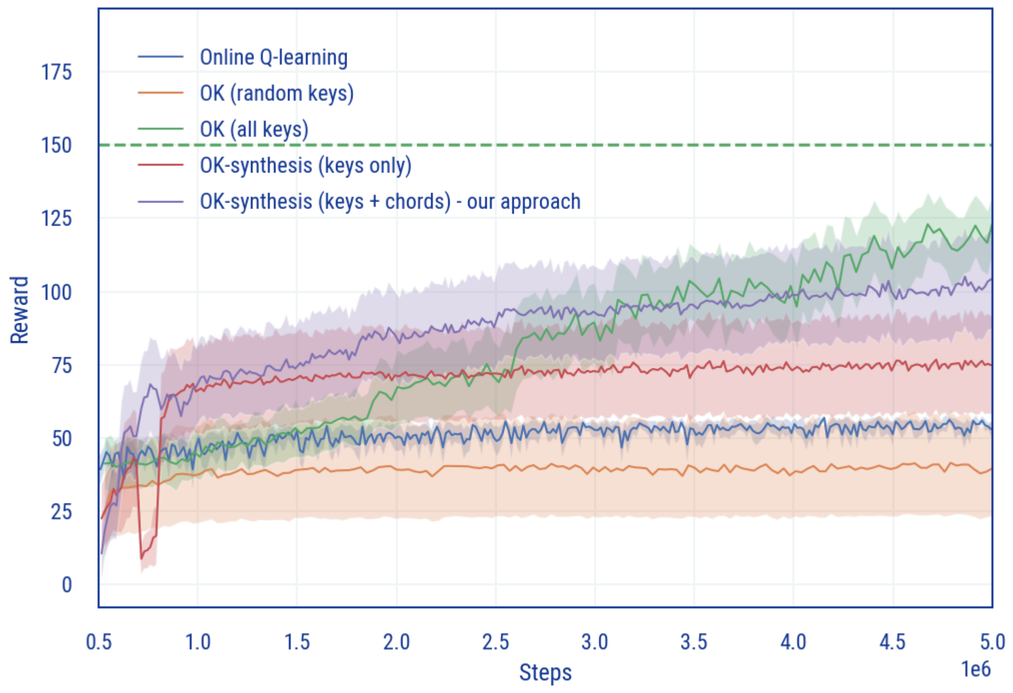


Figure 4.11. Results of our approach (purple) on the modified Scavenger environment on 30 random seeds compared to 1) online Q-learning (blue), 2) a fixed Option Keyboard with a random set of keys (orange), 3) Option Keyboard with the full set of keys (green), and 4) our approach with *no* chords from previous keyboard retained (red). Our approach reaches a good solution faster than all other approaches, but is unable to reach the optimal performance that the full OK can reach. However, in the case where the number of cumulants is large, reaching a good solution faster would be more desirable than using the full OK.

4.5.2 Analysis of results on modified Scavenger

We will begin by analyzing the learning curves shown in Figure 4.11. Note that the X-axis begins at 0.5M steps; the reason for this is that the first 0.5M steps are used by all OK-based methods for learning the options. The results shown indicate the results of all methods on 30 different random seeds with mean and 95% confidence interval plotted; these results are calculated by collecting data for 10 episodes for each random seed at fixed intervals of 15000 time steps. In the OK-synthesis based methods, we use the *reward* of each chord to determine its utility, i.e. chords that achieve higher environment rewards are more desirable. Furthermore, we also impose an obvious constraint on the keyboard composition, the same key cannot be repeated in the keyboard i.e. the degenerate solution that we saw in the initial version of the Scavenger where the same key occupies both slots of the keyboard would not be permissible now.

Online Q-learning: In this baseline, the agent is a flat agent i.e. it learns a policy over all actions directly maximizing the environment reward, without using any form of temporal abstraction. In the absence of temporal abstraction, it becomes difficult for the agent to reason over larger time scales, especially in this type of environment, where in order to achieve higher rewards, the agent has to incur negative rewards for a short period of time. Thus, it gives a suboptimal solution that avoids the food item containing nutrient 2, and only collects the food item containing nutrient 3, which always gives a $+1$ reward, and occasionally collects the food item containing nutrient 1, which gives a $+1$ reward provided its existing quantity is less than 10 units. Thus, it does not reach close to the optimal reward of ~ 150 in the first 5M time steps. However, it is worth noting that Q-learning is guaranteed to converge to the optimal reward in the limit (without function approximation), so we would expect this baseline to reach the optimal reward given sufficient time.

OK (random keys): This is a fixed-OK baseline; the agent is given a fixed set of random keys, and has to learn a player over these keys. Out of the 5 keys in the base set of keys, only 3 are useful: the keys that correspond to collecting nutrients 1, 2, and 3. Thus, any keyboard that has these keys will be able to achieve non-zero rewards on average. However, this baseline is the worst, as expected, since the keyboard is fixed and the player may not be able to perform well if the underlying keys are not rewarding.

OK (all keys): This is a fixed-OK baseline as well; however, here, the agent is given the *entire* set of base keys and has to learn a player over all these keys. Thus, the output space of the player is 243 combinations, as discussed earlier. Since the keyboard size is now 5, the agent can effectively combine *all* keys and is not constrained by the smaller keyboard size 2. Thus, the performance of this player in the limit will be higher than the other OK-based methods with a fixed keyboard size 2, since it can now learn to play chords for *all three* useful keys. This is reflected in the learning curve for this baseline as well; learning is slower initially due to the much larger output space of the player, but the agent does eventually reach optimal performance.

OK-synthesis (keys only): This is an OK-synthesis based method i.e. it follows the same approach as we proposed, but none of the best chords from the previous keyboard are retained. Thus, this keyboard can only contain keys. The reason we have included this baseline is that this is one way of determining whether or not chords are actually useful. If the best chords from the previous keyboard are not retained, the top 2 arms of the bandit comprise the keyboard. We can see in Figure 4.11 that using chords is indeed useful; this baseline does not show the same performance as our approach that retains the best chord from the previous keyboard.

OK-synthesis (keys + chords): This is an OK-synthesis based method, with the best chord from the previous keyboard retained and the remaining key selected from the base set to compose the keyboard. We can see that this method shows faster learning than the full OK baseline, and achieves performance closer to optimal than the keys-only baseline for our proposed OK-synthesis method. The reason for the former is that the player can learn much faster over the smaller output space of size 9, than the OK baseline with all keys that has an output space of size 243. In order to understand the reason for the latter, let us now take a look at the progression of the keyboard composition over time, shown in Figure 4.12.

4.5.3 Qualitative Analysis

We depict the change in the keyboard as part of the discovery process in the left column of Figure 4.12. The 5 rows in the figure correspond to five different runs of our discovery approach. The keyboard is of size 2 and is updated every $100,000$ steps. Each key is represented as a vertical bar; chords are collections of these vertical bars stacked together. Any key that is present with weight $+1$ lies above the X-axis and any key that is present with weight -1 lies below the X-axis in a single chord. Thus, for example, in the first row, the keyboard at $100,000$ steps has the following composition: $((c_1,c_3),(c_2))$ with respective weights $((1,1),(1))$. Similarly, in the second row at $150,000$ steps, the keyboard is composed of $((c_3,c_1),(c_2))$ with respective weights $((1,-1),(1))$.

We will now study the learning curves for each of these five runs in the context of their respective discovered cumulants. The learning curves are in the right column of Figure 4.12, and their corresponding discovered cumulants are in the left column.

Run 1: The initial composition of the keyboard is $((c_1),(c_3))$ with weights $((1),(1))$. After $100,000$ steps, the best chord determined by the player is (c_1,c_3) with weight $(1,1)$, and so that is included as a *key* in the next keyboard. The key selected by UCB from the base set of keys is (c_2). Now, after the player is learnt over this new keyboard, the most useful key in terms of reward is c_2, and so this key is selected as part of the next keyboard. As a result, the chord that was previously determined as useful, (c_1,c_3) with weight $(1,1)$, is forgotten, since a more rewarding key is encountered. This type of forgetting is catastrophic in the case where that chord is part of the optimal keyboard, because it means the agent will have to experience that particular chord again to determine that it is useful. We will address this type of catastrophic forgetting in the next section. We see that this run converges to the keyboard $((c_2),(c_3))$ with weights $((1),(1))$, and these are the two best keys, so the reward achieved is close to optimal, which is also reflected in its corresponding learning curve.

Run 2: In this run, we observe that the size of the retained chord grows across keyboards

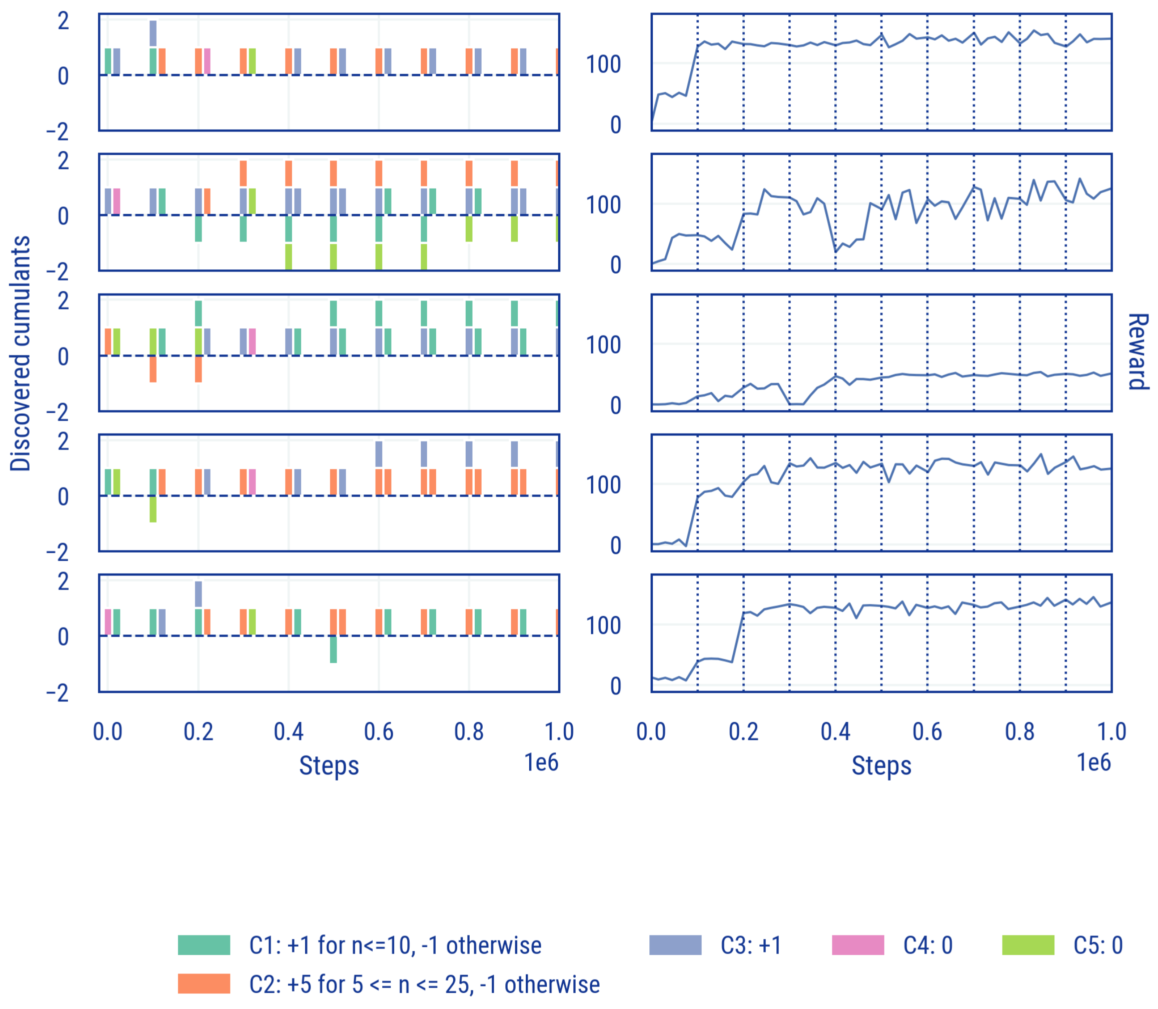


Figure 4.12. Qualitative results of our approach on the modified Scavenger environment: we visualize the synthesized keyboards over time with their corresponding learning curves for 5 different runs.

and then the keyboard converges to $((c_2, c_3, c_5), (c_3))$ with weights $((1, 1, -1), (1))$. Since the best key c_2 is part of the chord retained, the corresponding learning curve shows close to optimal performance. This curve is not as smooth as that of the first run, because the size of the chord determines the behaviour exhibited by the chord using GPE and GPI. As the size of the chord grows, the importance of a single key in a chord reduces, and that is reflected in the chord achieving that particular cumulant. Another interesting point to note is that even though c_5 is inconsequential in terms of the reward function, having behaviour that avoids it is not particularly harmful to the agent in any way, which is why its presence in the chord is not significant.

Run 3: Here, we see that the agent converges to the keyboard with keys $((c_1, c_3), (c_1))$ and corresponding weights $((1, 1), (1))$. The best key c_2 is missing in the keyboard, and as a result, the performance achieved is suboptimal, and is the same as that of the online Q-learning baseline. In order to understand why this is the case, we look at the keyboard where c_2 is experienced for the first time, at 0 steps. We see that the reward observed for this configuration of the player is very low. This is because the agent would be required to play the key c_2 for a few times and incur negative reward before achieving much higher rewards. However, based on the initialization of the player's neural network, it is possible that it requires more time for the agent to explore than the interval for which the keyboard player is learnt. Thus, in this case, 100,000 steps are not enough for the agent to discover that c_2 is a good key, which is why in the second configuration of the keyboard the chord retained is (c_5, c_2) with weights $(1, -1)$, i.e. it is considered more beneficial to *avoid* that nutrient. This trend continues in future keyboards, and eventually the chord containing c_2 is replaced with (c_1, c_3) with weight $(1, 1)$. This is a drawback of using such an approach. Note that since we are using UCB to constitute part of the keyboard c_2 would be selected multiple times in the limit, but practically it is heavily dependent on the exploration parameter of the bandit as well as the magnitude of Q-values learnt by the bandit.

Run 4: This run is similar to the first run; the agent converges to the keyboard $((c_2, c_3), (c_2))$ with weights $((1, 1), (1))$.

Run 5: Here, we see another instance of catastrophic forgetting of good chords like in the second run; the agent does discover the *optimal* keyboard of size 2 at 200,000 steps, but since c_2 is the dominant key, it forgets the second-best chord (c_1, c_3). While this is not a major problem empirically in this particular environment because there is one dominant cumulant, it could have worse consequences in environments where the set of cumulants is much larger and discovering good chords is essential to discovering a good keyboard.

4.5.4 Discussion

Summarizing the approach and results discussed above, it is clear that discovering a small set of cumulants to compose the keyboard from a larger set of cumulants leads to faster adaptation to complex reward functions compared to an agent that learns over the entire combinatorial space of an OK at full capacity. As the space of cumulants grows, learning over the entire space of chords becomes intractable. As a result, the motivation for our approach becomes stronger.

However, there is a fundamental flaw in our proposed approach: the agent is unable to retain good chords if better keys are encountered. This essentially means that if a bad player is learnt for one configuration of the keyboard, all the knowledge from previous keyboards is erased, rendering our approach impractical. This could also happen if the agent is in a part of the observation space in which a particular chord cannot be played since its corresponding cumulants are absent. However, that does not make that chord a bad chord; our approach is unable to distinguish between these scenarios and ends up discarding the chord. This intuition, however, is not tested in any way in the Scavenger environment since it is a fully observable environment and all cumulants are always present. This scenario would be encountered in a partially observable setting where it is possible that not all cumulants are present in a single observation. We use Jelly Bean World (Platanios et al., 2020) as the testbed for this intuition.

4.5.5 Jelly Bean World

Jelly Bean World (Platanios et al., 2020) is a partially observable non-episodic environment that serves as a testbed for never-ending learning. It is a procedurally generated infinite

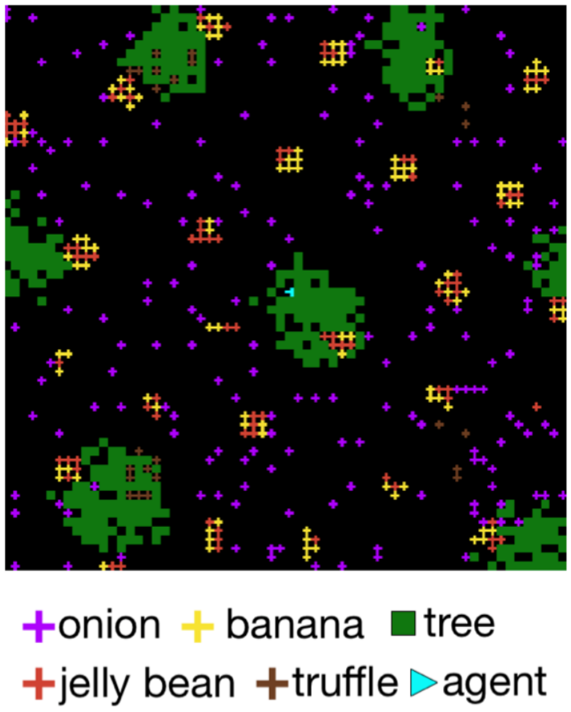


Figure 4.13. The Jelly Bean World environment.

gridworld environment, with stationary and non-stationary reward function and a large number of objects with different properties. A snapshot of the environment is shown in Figure 4.13.

The agent gets an egocentric observation and can take one of three actions – move forward, rotate left, or rotate right. The environment consists of a number of objects – onions, bananas, jellybeans, and truffles, which can be consumed by the agent, and trees, which are immovable objects and act as a source of obstruction for the agent. The objects are placed in such a way that jellybeans and onions occur together in clusters, onions are spread randomly all around, trees are clustered together to form forests, and truffles are rare and occur mostly near forests. The reward function can be stationary or non-stationary as described in Platanios et al. (2020). The stationary reward function is of the following form:

$$r(s,a) = \begin{cases} 1 & \text{if jellybean is collected} \\ -1 & \text{if onion is collected} \\ 0 & \text{otherwise} \end{cases} \tag{4.13}$$

Once an agent consumes a food item, that item is not respawned in the environment. As a result, the agent has to continuously keep exploring the environment to locate jellybeans and collect them in order to increase the accumulated reward. Note that the observation space optionally also has a *scent* that is a 3-dimensional continuous vector indicating the combined scent at a certain cell in the grid. This scent is a combination of the scents of all objects in the agent's vicinity. However, we disregard this part of the observation space and only focus on the visual part of the observation space i.e. the part of the grid within the agent's visual field. We only consider the stationary reward function for now.

4.5.6 Defining cumulants for Jelly Bean World

Since Jelly Bean World is an environment rich in resources and signals that the agent can extract, defining cumulants is easier than the simpler Scavenger environment. It is a partially observable environment, which means that it is possible that not all cumulants are present in a single observation. Thus, we define both maximization and attainment cumulants with the following purpose – the attainment cumulants induce options that enable the agent to collect an object that is in its visual field, and the maximization cumulants induce options that enable the agent to explore the environment till the respective cumulants are in the agent's visual field. Since the action space is egocentric to the agent, we also define an additional cumulant that induces an option that rewards the agent for moving. The entire set of cumulants is listed in Table 4.4. Note that since trees cannot be collected, we have omitted that particular cumulant. However, even if we were to include it, it would be akin to a random option since all possible options would be equal in terms of the pseudo-reward they achieve under this cumulant i.e. 0.

Table 4.4. Cumulants for the Jelly Bean World environment

	Object	Collect	Maximize	Move
1	Jellybean	✓		
2	-			✓
3	Onion	✓		
4	Banana	✓		
5	Truffle	✓		
6	Jellybean		✓	
7	Onion		✓	
8	Banana		✓	
9	Tree		✓	
10	Truffle		✓	

4.5.7 Analysis of results on Jelly Bean World

We show some results of testing our approach on the Jelly Bean World environment for the stationary reward function. Here, we attempt to discover a keyboard of size 4, which gives us a better sense of the chords and keys that are considered useful by the agent over different configurations of the keyboard.

Figure 4.14 shows the results of our approach on Jelly Bean World with the stationary reward function. We will analyze 5 runs as before, with a focus on the behaviour displayed by the agent while executing each chord. The keyboard is of size 4, and the best 2 chords from a keyboard are retained as keys in the subsequent keyboard. Here again, we use the reward accumulated by a chord as a measure of its usefulness. All 5 runs shown in Figure 4.14 start with the same keyboard configuration, and the keys sampled using UCB from the base set of keys are selected in the same order. The keyboard is reconfigured every 0.5M steps. One common observation across all 5 runs is that the cumulative reward for the first two keyboard configurations trends downward i.e. the agent collects more onions than jellybeans. However, this rate increases at 1M steps, when the keys that collect onions and bananas are part of the keyboard for the first time. We will analyze the keyboards for all runs from this point henceforth.

Run 1: At 1.5M steps, we see that the most useful chord is the key that collects bananas,

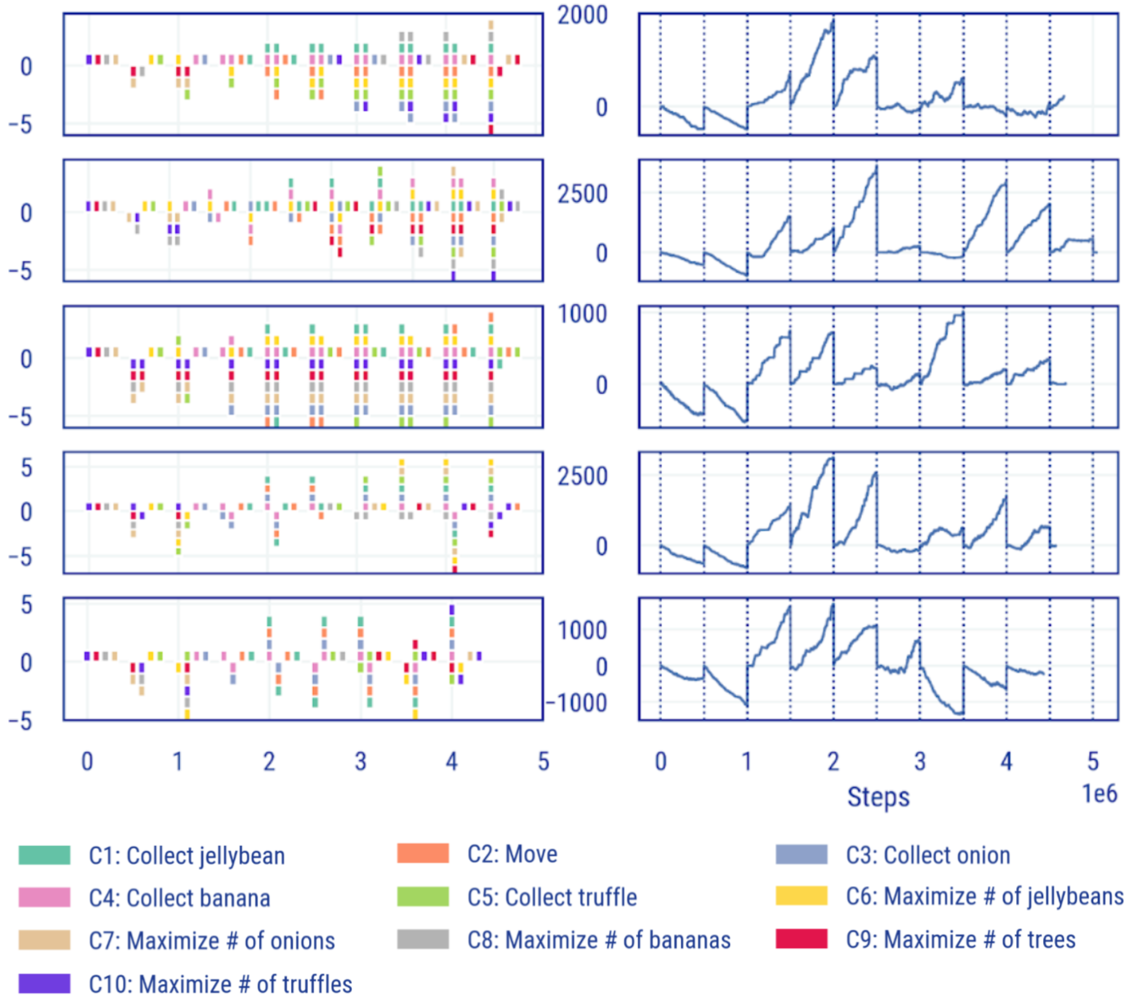


Figure 4.14. Results of our approach on the Jelly Bean World environment with a stationary reward function: we visualize the keyboard synthesis along with the corresponding learning curves for 5 random seeds.

and the second most useful key is the one that collects bananas while avoiding truffles, and minimizing the number of jellybeans. The reason that collecting bananas is considered good behaviour is that bananas are clustered with jellybeans, and as a result, if the agent collects bananas it is much more likely to collect jellybeans; this behaviour is obtained as a by-product of collecting bananas, but our approach is able to capture the advantage of this behaviour and include it in the keyboard since it is useful in the context of the environment reward. Another interesting aspect of the keyboard is that the second best chord has the *maximize # of jellybeans* cumulant with weight -1. On observing the behaviour policy for this chord, we observed that this led to behaviour that minimized the number of jellybeans by *collecting* them. While the *maximize* cumulants were designed with the view that they would enable the agent to explore in the environment till it encounters an object of a particular type, minimizing it leads to the behaviour where the agent deliberately *collects* the object to minimize the number of instances of that object present in its visual field. We see that this keyboard achieves the maximum cumulative reward compared to all previous and subsequent keyboards in this run. If we look at the next keyboard, we see that the best chord contains the keys that collect bananas and jellybeans, while avoiding moving. This would collect bananas and jellybeans that are within immediate reach of the agent; however, if they require more steps, the agent will not move towards them because the Q-value for not moving would be higher than the sum of collecting bananas and jellybeans. It is easy to see why this is the case – the *move* cumulant induces an option that is nearly Markovian, in most situations, the agent would terminate after taking the *move forward* action, and if it was obstructed by a tree, it would have to rotate and then move forward. However, on average, this cumulant is much *easier* to attain compared to those that involve collection of objects; as a result, it tends to dominate when present in a chord with these other cumulants. The second most useful chord is a superset of this chord in that it includes all the keys in the most useful chord along with two additional keys. The cumulative performance drops compared to the previous keyboard, even though the two keys selected by UCB are the same as the previous keyboard – *collect jellybean* and *move*. The reason for this is that now the chords are not useful as those in the previous keyboard. As we go further, we see that the cumulative performance continually degrades, and the keys sampled using UCB are also not those associated with high reward. While this could be solved by setting a lower exploration parameter for UCB, it also indicates that the *good* keys such as collecting jellybeans and bananas, and avoiding onions, have not accumulated enough reward in the keyboards that they were a part of, to ensure that they are selected by the agent using UCB. This shows that the quality of the chords being retained determines the overall quality of the present and future keyboards. We also see that the size of the chords being retained keeps growing. This is undesirable; when the size of a chord grows and we play the chord using GPE and GPI, any approximation errors in the underlying Q-values would be magnified and this would reflect in the behaviour of the synthesized option, which would be random.

Run 2: Here, too, we observe that when the keyboard contains small chords and keys, performance is the best. However, one interesting anomaly is at 3.5M and 4M steps; the keyboard contains large chords and still the performance is high. If we look at the chords, they include all the behaviours that we might consider useful – *collect jellybean, collect banana, maximize jellybeans, avoid onion (-1 for collecting onions)* along with some other behaviours that are benign in the context of the environment reward, such as *collect truffle, minimize number of trees*.

Run 3: We again observe similar trends as the previous runs; however, one important observation is that the two retained chords across keyboards are nearly identical. Furthermore, the performance of the keyboard relies more on the base keys than these retained chords, which is clear in the learning curve. We see that when the key that collects jellybeans is present in the keyboard, the performance is higher. Thus, in this particular run, chords do not seem to play a big role in terms of the environment reward.

Run 4: Here we see that from 1.5M to 3M steps, the keyboard consists of small chords, all of which are most influential in terms of the environment reward, and this is also reflected in the corresponding learning curve that shows the cumulative reward for each keyboard. At 1.5M steps, chord that is determined to be the most useful chord behaves to *collect bananas* *and avoid onions*. Note that this can be considered a surrogate function of the true environment reward, which is *collect jellybeans and avoid onions*. The property of the environment that jellybeans and bananas co-occur is captured in this chord, making it a useful chord. Furthermore, the most important key *collect jellybeans* is present in this keyboard as well, allowing it to accumulate high reward. If we look at the next keyboard at 2M steps, we see that the chord *collect jellybean, collect onion, collect banana, move* is present in the keyboard with weight 1 and -1 as two separate chords. This is a redundancy because playing the same chord with weight 1 and -1 would give us the same effect instead of having two separate chords. This begs the question of choosing a better heuristic to evaluate the keyboard in terms of metrics other than its performance on the environment reward, such as diversity-based metrics. Such metrics for evaluation would also mitigate the issue seen earlier in run 3, that of chords being nearly identical and thus, redundant.

Run 5: Here, the same case of redundancy is seen as in run 4. However, we also see a far more drastic effect of catastrophic forgetting of good chords; the agent's cumulative reward from 3M to 3.5M steps is negative, since it forgets the good chords learnt earlier that included the cumulants most relevant to the reward function.

One common observation across runs is that the *collect jellybean* key is no longer selected after a few keyboards since it is not considered useful by the agent. On observing the agent's behaviour in the environment, in all cases it transpired that the agent was stuck in a part of the observation space where it had consumed all food items in its visual field and in order to consume more food items it would have to explore. As a result, along with the *collect jellybean* key, the good chord $\{$*collect banana, avoid onion*$\}$ learnt by the keyboard in earlier iterations was also forgotten by the agent. This is an additional issue faced by the agent in partially observable environments as we hypothesized in the discussion for the Scavenger environment.

4.5.8 Discussion

The Jelly Bean World allowed us as the algorithm designers to define a much larger set of cumulants and combine maximization cumulants with attainment cumulants. This is especially useful since the environment is partially observable, and as a result, the maximization cumulants induce options that enable the agent to explore in a direction that might lead to the respective cumulants. For instance, one salient behaviour observed in the *maximize truffle* option was that the agent consistently moved away from empty spaces towards trees and forests since truffles are mostly found in trees. These types of properties of the environment dynamics are captured in the base options, and the higher-level discovery module makes use of these options to best maximize the environment reward by approximating the environment reward as a linear combination of the keys that achieve high reward. As a result, *collect banana, avoid onion* was a common chord learnt since it is a surrogate of the true reward function.

While Jelly Bean World actually allows us to test our approach at a larger scale, it also highlights the drawbacks of our approach, namely: 1) catastrophic forgetting of good chords discovered in earlier keyboards (as with the Scavenger), 2) large-sized chords that make the behaviour of the synthesized option using GPI close to random, especially if the error in approximating the base Q-values is high, and 3) redundancy in the keyboard in the form of chords that are nearly identical or canonical equivalents of each other.

These insights, combined with the earlier insights from experiments on the modified Scavenger environment, necessitate the development of a new approach within this paradigm of discovery that tackles some or all of these issues encountered in Scavenger and Jelly Bean World.

4.6 Towards A Better Approach

The three main issues encountered with the original approach are the following:

1. Chords that are deemed *useful* by the agent can be forgotten if there is a single bad

keyboard player at an intermediate stage, or if that chord is not considered useful by just one keyboard. Thus, an important characteristic of the new approach should be its ability to remember previous good chords, so that it can try them again in the future without having to learn them from scratch.

2. Chords can get arbitrarily large using our approach, even if the keyboard size is much smaller. This is, in general, a strength of the proposed discovery framework: multiple useful keys can be compressed into a single key, thus allowing the keyboard to store a variety of different keys that may be required to achieve higher environment rewards. However, as the size of the chord grows, the number of keys over which GPE and GPI is performed also grows. This may lead to the agent choosing the greedy action for a different option at each step, especially if the Q-values for the underlying keys are very close in value to each other. This may lead to an adverse effect in that the resultant policy shows random behaviour. This was observed in both previous environments and is a common shortcoming of this type of zero-shot composition of value functions that may not be optimal. So, it is reasonable to expect that in the new approach, the size of chords is not arbitrarily large, thus ensuring that chords are indeed useful.

3. There were several types of redundancies observed earlier from identical keys to nearly identical chords. While the former can be handled by imposing a simple constraint on the keyboard requiring all keys and chords to be unique, the latter is much harder to achieve. We explored a heuristic function that is a weighted sum of the environment reward accumulated by the agent while executing the chord and the *diversity* it adds to the existing keyboard. This diversity can be a simple metric like the Jaccard similarity between the candidate chord and the existing chords/keys in the keyboard. However, this adds an additional hyperparameter to the approach: the coefficient of the diversity-based metric, which makes it less desirable. Thus, we would ideally want the new approach to organically discover a diverse keyboard.

One way to solve the issue of catastrophic forgetting of chords would be to add each *good* chord as an arm of the bandit. This means that the bandit grows over time as new chords are added to and discarded from different keyboards. While this would in principle eliminate the problem at hand, it also introduces a new problem – each new chord that is added would be explored by UCB since its visitation frequency is low. In the worst case, this leads to a situation where the size of the bandit is the cardinality of the space of all possible chords $\mathscr{W}$ i.e. 3^n, where n is the number of keys. Here, the number of arms is exponential in the number of keys, and as a result, the key advantage of the discover approach, which is faster learning by synthesizing a small keyboard, is lost. We will consider a more general version of this approach, which includes *all* possible chords as arms of the bandit instead of adding them dynamically.

The most obvious way to mitigate this is to systematically prune the space of possible chords $\mathscr{W}$. In order to do this, we first discuss how an option synthesized using GPE and GPI behaves: say there are two cumulants c_1 and c_2 and the chord (c_1, c_2) is played with weights $(1, 1)$. Now, there are several possible behaviours that the agent could exhibit, based on the discount factor and the observation:

- If the discount factor is low, i.e. the option policies optimize for a shorter horizon, the synthesized option attains the cumulant that is closest to it.
- If the discount factor is high, and both cumulants can be attained simultaneously, the agent will execute the corresponding policy. If the two cumulants cannot be attained at the same time, the synthesized option will attain them sequentially.

While the above intuitions may seem tangential to the problems discussed previously, we will now demonstrate how we can make use of them to solve a number of problems with the earlier approach.

We attempt to prune the space of possible chords by imposing the following constraint:

A chord is valid only if its constituent cumulants are simultaneously attainable.

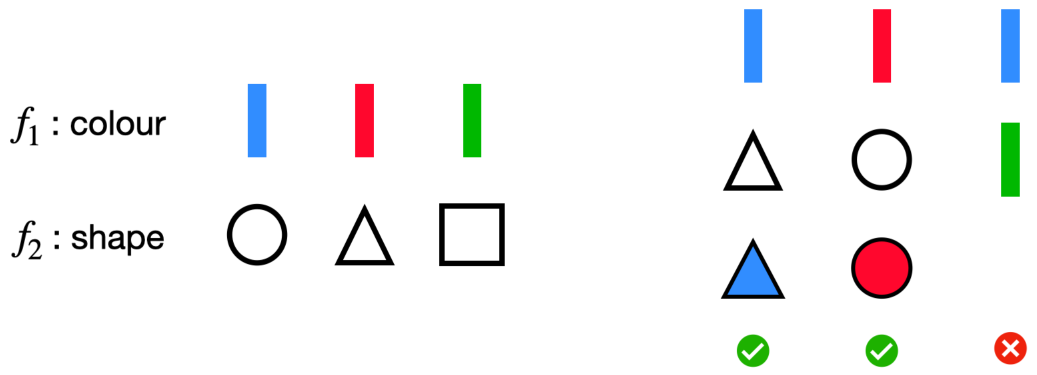


Figure 4.15. Example of valid chords in the case where we have 2 classes of features: *colour* and *shape*, each of which can assume 3 values.

We give a simple intuitive example of this in Figure 4.15. Assume there are different objects in the environment that have two properties: colour and shape. There are three types of colours – red, blue, and green, and three types of shapes – circle, triangle, and square. The objects in the environment cannot have multiple colours or shapes. Now, the example in Figure 4.15 shows that $(blue, triangle)$ and $(red, circle)$ are valid chords since these cumulants can be attained simultaneously by the agent. However, $(blue, green)$ is not a valid chord since an object cannot be blue and green, making the constituent cumulants not simultaneously achievable.

Note that if the chord $(blue, green)$ is played by the Option Keyboard, the agent will attain *one* of these cumulants at a time, and then terminate. As a result, we get the effect of a logical OR when this chord is played. However, if the chord $(blue, triangle)$ is played by the Option Keyboard, contingent on the discount factor, the agent will attempt to perform a logical AND i.e. it will try to attain both of these cumulants simultaneously since that will lead to the highest discounted return and will be reflected in the Q-values computed using GPE.

So, it is clear that the aim is to only include chords that perform a logical AND of the underlying keys. Why would that be helpful?

1. Chords that induce options which perform a logical OR of the underlying cumulants can

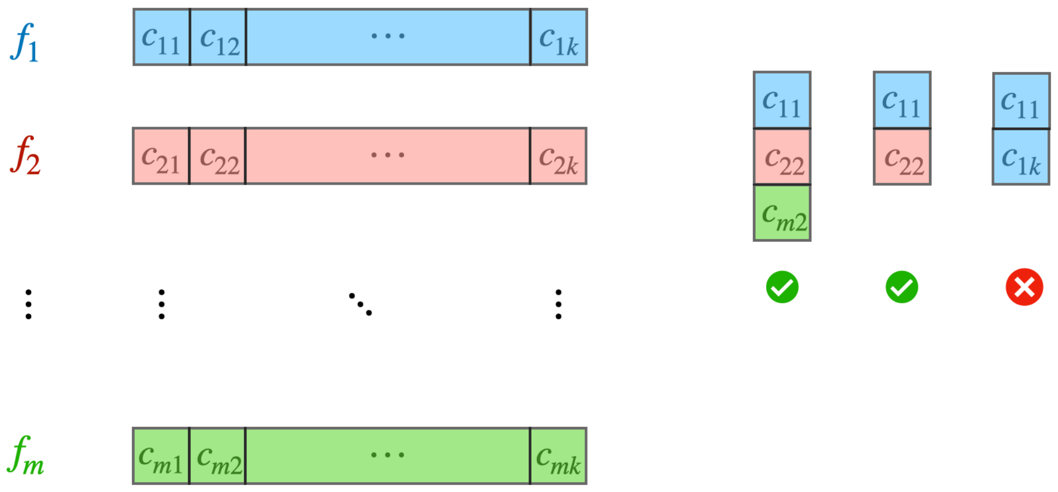

f_1
c_{11} c_{12} $\cdots$ c_{1k}
f_2
c_{21} c_{22} $\cdots$ c_{2k}
f_m
c_{m1} c_{m2} $\cdots$ c_{mk}
c_{11} c_{22} c_{m2}
c_{11} c_{22}
c_{11} c_{1k}

Figure 4.16. General rule for construction of chords: no two cumulants from the same feature class can coexist in a chord.

be obtained as a sequential composition of their underlying keys. As a result, the behaviour exhibited would not give any added advantage compared to simply using the constituent keys.

2. Chords that induce options which perform a logical AND of the underlying cumulants cannot be obtained as a sequential composition of their underlying keys. This is because a sequential composition of the underlying keys *blue* and *triangle* would only perform a logical AND if the closest blue object to the agent happens to be a triangle, and vice versa. Thus, in expectation, the desired effect of simultaneously achieving both cumulants cannot be obtained by a simple sequential execution of the underlying keys, and thus, including this chord gives an added advantage compared to simply using the constituent keys.

A more general visualization of this example is given in Figure 4.16. Here, we have *classes* of features that are *disentangled*. We draw inspiration for this intuition from Grimm et al. (2019), who propose an approach for learning successor features for *maximization* cumulants. We extend their intuitions to *attainment* cumulants. The agent extracts disentangled features from the observation space. Each of these disentangled features can assume several values. While these values are in general, continuous, we assume discrete features for the sake of simplicity. Thus, for each feature, the agent can take one of k values, and there are m features. Thus, we have a total of $n = m \times k$ cumulants, each of which induces a key.

Now, we look at how chords are constructed in this general setting. Figure 4.16 shows that chords which compose keys induced by cumulants from *different* features are valid. On the other hand, composing two keys from the same feature class is not valid, since those cumulants cannot be simultaneously attained. Note that we have omitted the weight -1 for conceptual clarity. From this point onward, we will talk about binary chords i.e. $\mathscr{W} \equiv \{0,1\}^n$, but the approach can be extrapolated to larger spaces for chords as well.

What is the advantage of this particular assumption on the cumulants? The first advantage is that each chord can now be treated as an *atomic unit* that induces an option which cannot be obtained as a sequential execution of any of its components. It can alternatively be thought of as a basis vector in the space of behaviours induced by linear combinations of the cumulants. The second advantage is that the space of chords reduces; if we consider the naïve way of synthesizing chords, we get $\mathscr{W} \equiv \{0,1\}^n$, which was the case earlier. Thus, the number of chords that the OK can play is 2^n in this case. Now, if we operate under the assumption of disentangled cumulants defined above, we see that the number of chords that the OK can play changes to $(k+1)^m$. This is because a chord can now contain at most one cumulant from each feature class, leading to $k+1$ choices per feature, and since there are m features, the total number of combinations possible is $(k+1)^m$.

Let us now analyze the edge cases for this approach for constructing chords. If $k = 1$, i.e. each feature can only assume a single value, we arrive at a space of $2^m = 2^n$ chords, since $n = m \times k$. This means that all possible combinations of cumulants are valid. Intuitively, it implies that there is a single object in the environment that assumes these values for the different disentangled features, making all possible chords valid. In this case, we do not see any reduction in the space of chords. However, this environment in itself is a trivial environment since all keys and chords are equivalent in terms of the environment reward. Since there is a single object in the environment, each key and chord will try to attain that object, and thus, they are all equally useful. Thus, this is not an interesting environment to study in the context of discovery.

On the other end of the spectrum, if $m = 1$, i.e. there is only one feature that can take k different values, we arrive at a space of $k = n$ chords, since $n = m \times k$. This means that *no* combination other than the trivial one-hot combinations or keys, are valid. This means, at an intuitive level, that there are n objects in the environment, each of which possesses a single feature, and all these objects assume a different value for that feature. This is the same as the Scavenger environment described in Section 4.5.1. Here, the space of chords is the same as the space of keys, since no non-trivial combination is valid. Thus, this is also not an interesting setting to study in the context of analyzing how useful *chords* are for discovery.

In all other cases, we observe a reduction in the space of chords compared to the previous naïve version. If we consider the example in Figure 4.15, we see that $m = 2$ and $k = 3$, because we have 2 features – *colour* and *shape*, each of which assumes 3 values – $\{$*blue, red, green*$\}$ and $\{$*circle, triangle, square*$\}$ respectively. Without imposing the additional constraint on the construction of chords, we have $2^n = 2^{m \times k} = 2^6 = 64$ chords. After imposing the constraint on chords, we arrive at $(k+1)^m = 4^2 = 16$ chords, giving us a $4\times$ reduction in the space of chords.

A schematic diagram of applying this constraint on the construction of chords to our proposed discovery approach is shown in Figure 4.17. This diagram extends the example shown in Figure 4.15. The arms of the bandit include all valid chords, which solves the problem of catastrophic forgetting of good chords encountered earlier. The assumption allows the first reduction in the space of chords. However, this is not the only place where we see the benefits of making the assumption of disentangled cumulants. We see an additional benefit of doing so at the level of the player: instead of learning a player over $2^{|K|}$ chords, where $|K|$ is the size of the keyboard, the number of chords now depends on the keyboard configuration. We see in Figure 4.17 that for the two keyboard configurations shown, the output size of the player is variable but smaller than the original fixed size 16. Thus, learning a policy for the player also becomes faster due to the reduced output space.

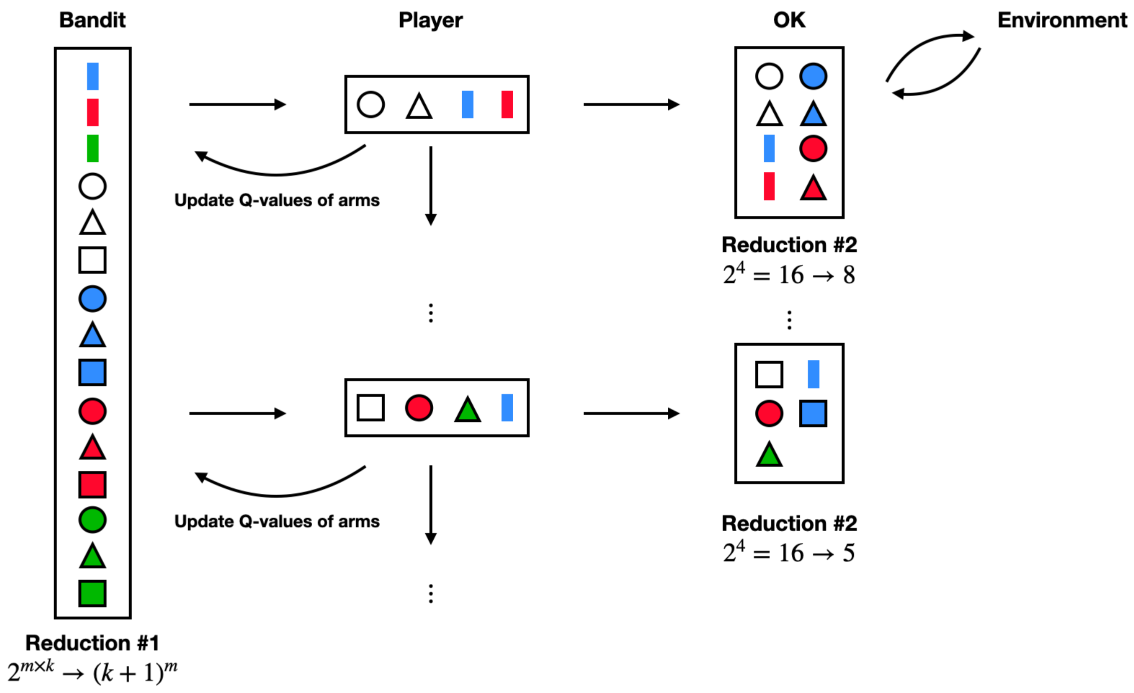


Figure 4.17. Our modified approach for option discovery: 1) the OK can only play valid chords instead of all possible chords, and 2) the bandit has *all* valid chords as arms, instead of only the base set of keys.

Note that we have eliminated the part of the approach that retained keys from the previous keyboard, and substituted it with a bandit over all valid chords. This allows us to remember previous *good* chords, and has the added advantage of removing the hyperparameter M that determines how many chords from the previous keyboard should be retained in the new keyboard. Thus, the agent intermittently composes the *entire* keyboard by selecting chords from the bandit using UCB.

Since the number of chords is now much larger, we propose constituting the first few keyboards with only keys. This helps because when a keyboard consists of only keys, it can play more chords than when it consists of chords, since the larger the size of the chord, the more unlikely it is for the underlying cumulants to be simultaneously achievable. This means that we get an initial estimate of the bandit arms faster. However, even if we do this, there are certain chords that will not be played in these first few keyboards since their underlying keys were parts of different keyboards. To eliminate this issue, we propose a workaround by initializing the values of these chords through bootstrapping: the utility of these chords that are not experienced is calculated as the average of the utilities of their constituent keys.

The modified approach including the learning of the player and the synthesis of the keyboard are described in Algorithms 7 and 8 respectively.

Now that the basic approach is in place with the constraint on construction of chords, we have just one remaining part to handle – transfer of player parameters across keyboards. In the earlier approach, since we were retaining chords from the previous keyboard, retaining their corresponding weights and resetting the other weights was straightforward. However, as the agent now synthesizes the entire keyboard from the bandit over chords, it is possible that the entire keyboard is different from the previous keyboard, and no transfer of final layer parameters is possible.

While this is a well-known issue in continual learning literature, with several approaches proposed to mitigate it (Kirkpatrick et al., 2017), we motivate our method for retaining parameters in the following manner: while the player can in principle use any RL method to learn a policy

Algorithm 7. Player

```
Require: { $K$ : current keyboard configuration
           $C$ : set of all valid chords
           $Q_\Omega$ : value functions for pretrained options
           $\mathscr{W}$ : set of all possible combinations of size$|K|$
           $n$ : number of training steps
           $\alpha, \varepsilon, \gamma$ : hyperparameters
           compose_chord : function that converts a combination to a chord
           u : utility function that determines the usefulness of a chord
 1: Create $Q_{player}(s, \mathbf{w})$ parameterized by $\theta_{Q_{player}}$
 2: $\mathscr{W}' \leftarrow \{\mathbf{w}' \in \mathscr{W}$ if compose_chord$(K, \mathbf{w}') \in C\}$        ▷ Player can only choose valid
    combinations
 3: Select initial state $s \in S$
 4: while number of steps $\leq n$ do
 5:     if Bernoulli$(\varepsilon) = 1$ then $\mathbf{w} \leftarrow$ Uniform$(\mathscr{W}')$
 6:     else $\mathbf{w} \leftarrow \text{argmax}_{\mathbf{w}' \in \mathscr{W}'} Q_{player}(s, \mathbf{w}')$
 7:     $s', r', \gamma' \leftarrow OK(s, \mathbf{w}, Q_\Omega, \gamma)$
 8:     $\delta \leftarrow r' + \gamma' \max_{\mathbf{w}' \in \mathscr{W}'} Q_{player}(s', \mathbf{w}') - Q_{player}(s, \mathbf{w})$
 9:     $\theta_{Q_{player}} \leftarrow \theta_{Q_{player}} + \alpha\delta\nabla_\theta Q(s, \mathbf{w})$
10:     if $s'$ is terminal then reset $s$
11:     else $s \leftarrow s'$
12: Create vector $U$ denoting utility of each chord
13: for each $\mathbf{w} \in \mathscr{W}'$ do
14:     $c \leftarrow$ compose_chord$(K, \mathbf{w})$
15:     $U[c] \leftarrow$ u$(c)$
    return $U$
```

Algorithm 8. OK Synthesis with Disentangled Cumulants

```
Require: {  select_keys : function used to select keys/chords from the bandit
            $N$ : number of keys in the OK
            $K$ : number of steps before the OK is reconfigured
            $C$ : base set of keys
            $\alpha$ : learning rate for bandit
 1: $Q_{bandit} \leftarrow [0,0,...,0]$                    ▷ Bandit Q-values; $Q_{bandit} \in \mathbb{R}^{|C|}$
 2: $N_{bandit} \leftarrow [0,0,...,0]$                    ▷ Visitation frequencies; $N_{bandit} \in \mathbb{Z}^{|C|}$
 3: $t_{bandit} \leftarrow 0$        ▷ Timesteps elapsed at the highest level of abstraction i.e. discovery
 4: $C' \leftarrow$ select_keys$(Q_{bandit}, N_{bandit}, N)$          ▷ Current set of keys in the OK $C'$
 5: $t_{bandit} \leftarrow t_{bandit} + 1$
 6: loop
 7:     for $K$ steps do
 8:         Learn OK player with $C'$ and keep statistics on use of chords $\mathbf{w}$
 9:     for each chord $\mathbf{w}'$ do                                    ▷ update $Q_{bandit}$
10:         $r \leftarrow$ Q_stats$[\mathbf{w}']$
11:         $Q_{bandit}[\mathbf{w}'] \leftarrow Q_{bandit}[\mathbf{w}'] + \alpha(r - Q_{bandit}[\mathbf{w}'])$
12:         $N_{bandit}[\mathbf{w}'] \leftarrow N_{bandit}[\mathbf{w}'] + 1$
13:     $C' \leftarrow$ select_keys$(Q_{bandit}, N_{bandit}, N)$                 ▷ Synthesize the next OK
14:     $t_{bandit} \leftarrow t_{bandit} + 1$
```

over chords, we use a Q-learning based method to learn the player's policy. This means that the output of the neural network for the player is the Q-values of the chords that can be played by the player. We propose saving the final layer parameters for all the chords. The player's torso is maintained when there is a change in the keyboard, as before. The final layer parameters are initialized to the saved values.

There are two issues we foresee with this method of transferring weights across keyboards:

1. The player's torso is continually updated with each keyboard, and thus, the final layer parameters for chords that were tried in the past may be inconsistent with the updated torso.

2. The Q-values estimated for chords in a keyboard model the expected discounted return in each state of the chords *in the context of the keyboard configuration*. In other words,

this means that the Q-value of a particular chord in a state might be different in different keyboards even if the torso is identical. For two keyboards K_1 and K_2, if we compare the Q-values of a chord $\mathbf{w}$ in state s,

$$Q_1(s,\mathbf{w}) = r'(s,\mathbf{w}) + \gamma' \max_{\mathbf{w}' \in \mathscr{W}_{K_1}} (s', \mathbf{w}') \tag{4.14}$$

$$Q_2(s,\mathbf{w}) = r'(s,\mathbf{w}) + \gamma' \max_{\mathbf{w}' \in \mathscr{W}_{K_2}} (s', \mathbf{w}') \tag{4.15}$$

where s' is the state after executing chord $\mathbf{w}'$. Note that while the first term i.e. the reward obtained from the OK may be equal, the second term involves a max over all chords present in the keyboard. Thus, if $K_1 \neq K_2$, it is possible that $\max_{\mathbf{w}' \in \mathscr{W}_{K_1}} (s', \mathbf{w}') \neq \max_{\mathbf{w}' \in \mathscr{W}_{K_2}} (s', \mathbf{w}')$, leading to different Q-values for the same chord $\mathbf{w}$ in state s.

However, we hypothesize that this way of transferring weights provides a good *initialization* to the player, since these initial estimates of the Q-values, albeit noisy, may be closer to the true Q-values than a random initialization.

Now we proceed to experiments to test our proposed approach. Since the previous approaches are not amenable to the type of disentangled cumulants we described, we propose two new environments that are extensions of existing environments.

4.6.1 The Block Stacking Environment.

This environment is an extension of the commonly used environment in RL involving stacking blocks in the right order. While this environment has several possible instantiations, such as those in the robotics () and planning () settings, we tweak it in such a way that it is a good testbed for our discovery approach with disentangled cumulants. There are 2 categories of features – *shape* and *colour*, each of which can assume 3 values – {*circle, square, triangle*}, and {*red, blue, green*}. Note that these are the same cumulants as those shown in Figure 4.15. Furthermore, there is a stacking location where some blocks have to be stacked in a particular

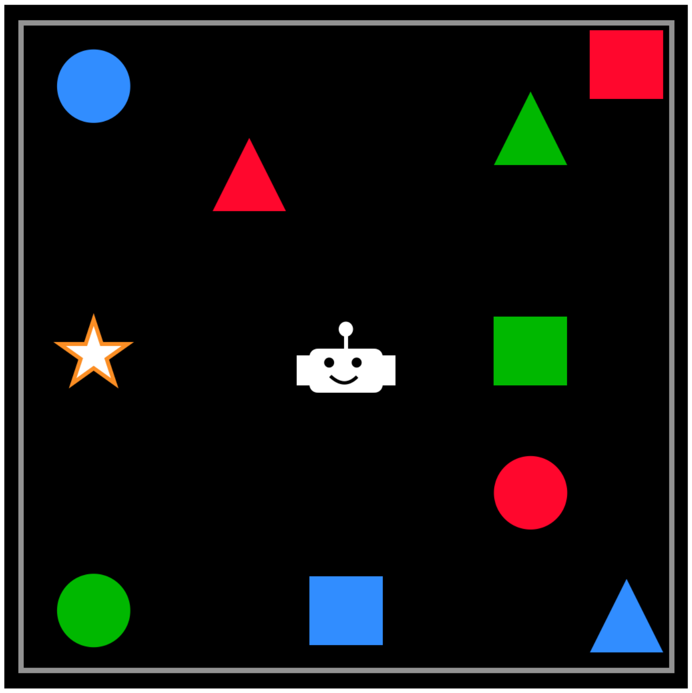


Figure 4.18. The Block Stacking Environment

order. The observation that the agent receives is egocentric, and the gridworld is toroidal, like the Scavenger environment. The environment is shown in Figure 4.18. The stacking location is represented by a star, all other objects are the blocks, and the agent is at the centre of the grid. The action space consists of the four actions corresponding to moving in each direction – up, down, left, and right. If the agent lands at a position that contains a block, it automatically picks up the block. If it was already holding a block, it swaps it with the new block. The observation consists of the grid as well as the block that the agent is currently holding. If the agent lands at the stacking location, it automatically drops the block it is holding. That block is not re-spawned in the grid, since it is considered stacked. The reward function is of the following form:

$$r(s,a) = \begin{cases} +1 & \text{for each desirable property in the collected block} \\ -1 & \text{if the wrong block is stacked} \\ +100 & \text{if the correct block is stacked} \\ 0 & \text{otherwise} \end{cases} \tag{4.16}$$

The environment is episodic with each episode of length 100. The blocks have to be stacked in the order: *red circle* $\rightarrow$ *blue circle* $\rightarrow$ *blue triangle*. The agent gets small intermediate rewards on collecting blocks that possess desirable properties, such as red blocks or circular blocks; however, it gets a much larger reward each time the correct block is stacked.

4.6.2 Results

Here, we present the results of our approach and baselines on the Stacking environment. The agent has to stack blocks at the centre, which is the stacking location, in the order *red circle* $\rightarrow$ *blue circle* $\rightarrow$ *blue triangle*. As mentioned earlier, stacking the correct block gives a large positive reward, but if the agent stacks the wrong block, it incurs a penalty.

The cumulants for this environment can be defined in a straightforward manner: each cumulant corresponds to collecting a block that assumes a certain value for an attribute. This gives rise to 6 such cumulants, each of which rewards the agent for collecting a block with the desired value for that attribute. In addition to this, we also define a cumulant that rewards the agent for carrying the block that it is currently holding to the centre.

We first plot the results of the Q-learning baseline and the full OK baseline in Figure 4.19. We see here that the full OK baseline outperforms the flat Q-learning agent. This is to be expected since the agent needs to reason over longer horizons to achieve higher reward, which can be achieved using temporal abstraction. Qualitatively, we observe that the Q-learning agent completely avoids the stacking location and alternates between collecting two blocks that have desirable properties. For instance, the properties *red* and *triangle* are desirable based on

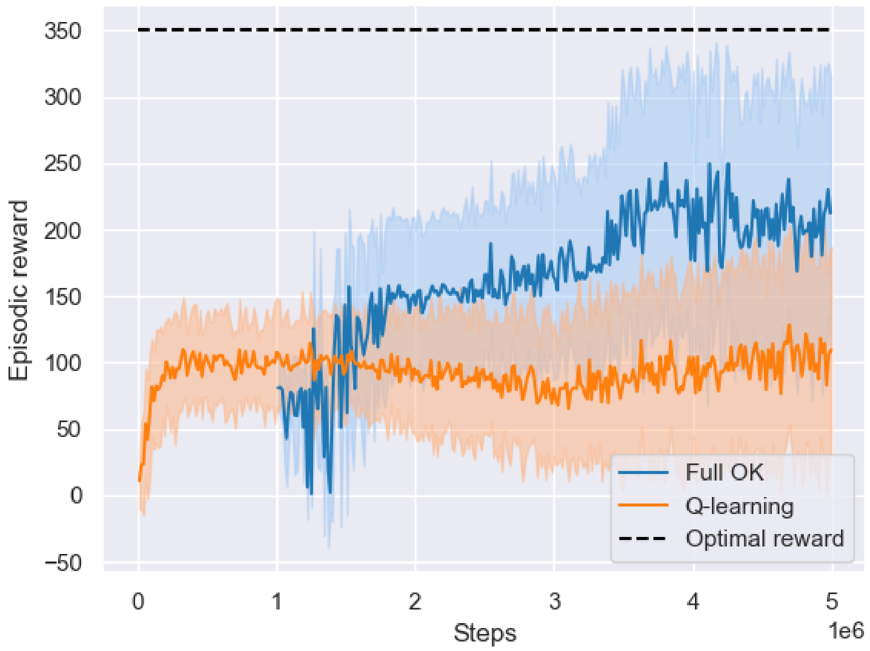


Figure 4.19. Results of Q-learning and the full OK on the Block Stacking environment: the Q-learning agent is a flat agent, and the full OK agent has no capacity limits on the size of the keyboard, i.e. it learns a player over the entire base set of options.

the reward function, so the agent might alternate between collecting a *red circle* and a *green triangle*, say and achieve high, albeit suboptimal, reward. This is the trend that we see in the corresponding learning curve for Q-learning in Figure 4.19. The agent settles at a lower reward of ~ 100, and exhibits the above described type of behaviour.

In contrast, the OK agent that utilizes the full base set of options is able to achieve higher reward, since it is able to figure out the *right* first block to stack. Following that, the agent has to figure out the next good block to stack. Note that in the first 5 million steps, the reward lies between 200 and 250, which means that the agent correctly stacks in most cases, the first block or the first two blocks correctly. In several cases, the agent correctly stacks the first block and then alternates between the collection of two *good* blocks, leading to a reward of ~ 200, whereas in other cases, it correctly stacks *two* blocks, and then alternates between the collection of two good blocks. However, its ability to stack the right block is reflected in its performance in Figure 4.19, where the blue curve corresponding to it is higher than the orange one corresponding to the Q-learning baseline. In the first 0.5M steps, the reward is negative; this is easily explained by the agent stacking the incorrect blocks and incurring negative reward.

Now, we look at the results of applying our approach to the Block Stacking environment by synthesizing a keyboard of size 4. The qualitative plots for 5 runs is shown in Figure 4.20. The agent reconfigures the keyboard every 200,000 steps in these experiments. Since there are 7 cumulants, and the keyboard is of size 4, the first two keyboard configurations will explore each key, and in the second keyboard, will have the key determined to be the best key by UCB. We will analyze each of these runs separately:

Run 1: The learning curve on the right is smooth and shows performance similar to the Q-learning baseline. If we look at the keyboards synthesized over time, it is evident that the *deposit at centre* cumulant is missing, and this is the cumulant that will enable the agent to stack the correct blocks and achieve high rewards. The most frequently seen chords in the keyboards for this run are *(collect red, collect circle), (collect blue), (collect blue, collect triangle)*, and *(collect red, collect triangle)*. These make sense; the agent gets rewarded when any collected

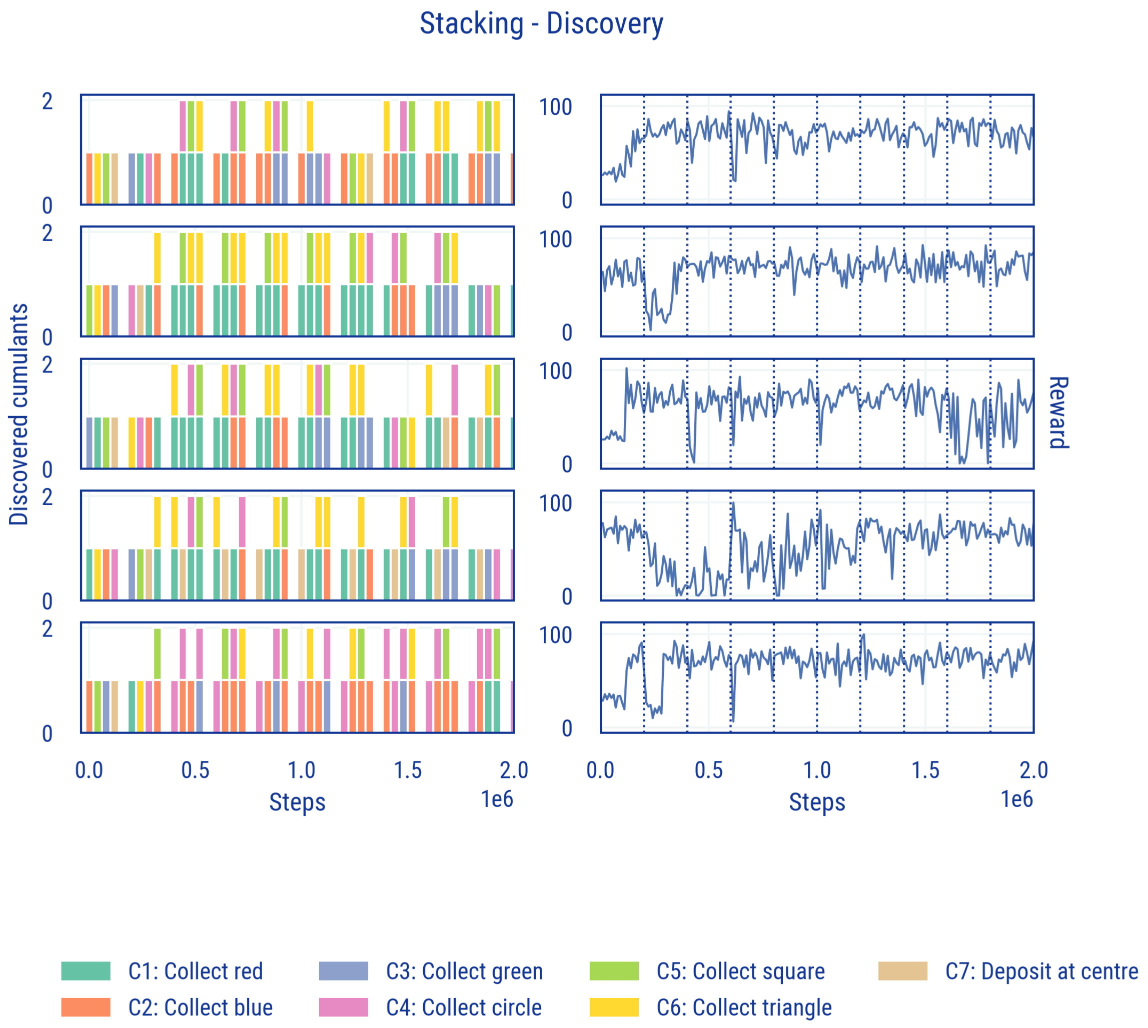


Figure 4.20. Qualitative results of our approach on the Block Stacking environment for a keyboard of size 4.

block has a desirable property, and each of these chords will collect blocks that have at least one of these properties. An interesting point to note here is that the chord *(collect red, collect triangle)* is a good chord, because the agent will get a reward of $+2$ on collecting it, but it can be considered a deceptive reward, since actually stacking the red triangle at the centre will lead to the agent incurring a penalty.

Run 2: The trends in the learning curve and the keyboard synthesis are similar to those in run 1.

Run 3: We see an interesting trend around 1.6M steps here; the cumulant *deposit at centre* is part of the keyboard. However, instead of leading to better performance, we see a sharp decline, followed by high-variance episodic returns. In order to see why this is the case, we look at the corresponding keyboards. At 1.6M steps, the agent can potentially deposit these blocks at the centre: *(red), (red, triangle)*, or *(blue circle)*. It is possible for the agent to correctly stack the first *two* blocks by first choosing the *(collect red)* chord followed by *(deposit at centre)* $\rightarrow$ *(blue, circle)* $\rightarrow$ *(deposit at centre)*. However, there is a caveat here; this will only work if the agent collects the red *circle* for the first option. Since there are three different types of red blocks, the agent in expectation has equal probability of collecting each one. Thus, if the agent collects any block other than the red circle and deposits it, it will incur a negative reward. In addition to this, even though the agent has discovered that the blue circle is part of the environment reward and must be stacked, this knowledge is inconsequential if it is unable to stack the first block correctly, since the blue triangle should be stacked *second*.

Run 4: This run is different from the others in that the Q-value for the *deposit at centre* cumulant is determined by the agent to be high enough for it to be included in all keyboards. However, this is not reflected in the corresponding learning curve. We see that the reason for this is that the keyboards have the chords *(red)* and *(red, triangle)*, but not *(red, circle)*. Thus, we see the same effect as described in run 3 above.

Run 5: The trends here are similar to those in runs 1 and 2.

One common observation across runs is that the agent is able to at least do as well as the

Q-learning baseline since it is able to figure out good chords to constitute the keyboard that lead to good reward, if not the optimal reward. However, there are certain hyperparameters that need to be set accordingly for this method to show its true advantages; for instance, in the experiment discussed above, we set the keyboard update interval to 200,000 steps; however, it is conceivable that, given a sufficient amount of time to learn the player for a certain keyboard, the agent may be able to discover that in spite of giving penalties for stacking the wrong blocks, the *deposit at centre* chord is capable of giving much higher rewards if it is played by the keyboard player correctly.

4.6.3 Additional Analysis

Here, we show the results of an additional experiment where we synthesize a smaller keyboard of size 2. We show the qualitative results of this experiment in Figure 4.21. The results here are similar to those of the Q-learning baseline, and the synthesized keyboards also reflect the same trends as some of the runs in Figure 4.20. However, it is important to note here that for a keyboard of size 2, this is close to the optimal performance it can reach. The reason for this is that the best keyboard that can be synthesized is *(red, circle), (deposit at centre)*, which would lead to a reward of $\sim 100-150$, since the chords for the second and third blocks would be excluded on account of the capacity limit of the keyboard.

Another interesting aspect of this approach is that it relies heavily on the base set of options being optimal or close to optimal. If the Q-values for these options have errors, these errors compound when we take into account the player learning and the discovery module. We show evidence of this insight in Figure 4.22. In principle, if we have the ideal keyboard, the agent should be able to quickly learn a policy that gives optimal performance. It should reach this optimal performance faster than the discovery approach, as well as the full OK baseline, since the action space of the player will be smaller, and the best chords are given to the agent. However, we see that for two keyboards that are conceivably ideal, this trend is not displayed. In fact, in the orange learning curve, which corresponds to the ideal keyboard of size 4, the performance is

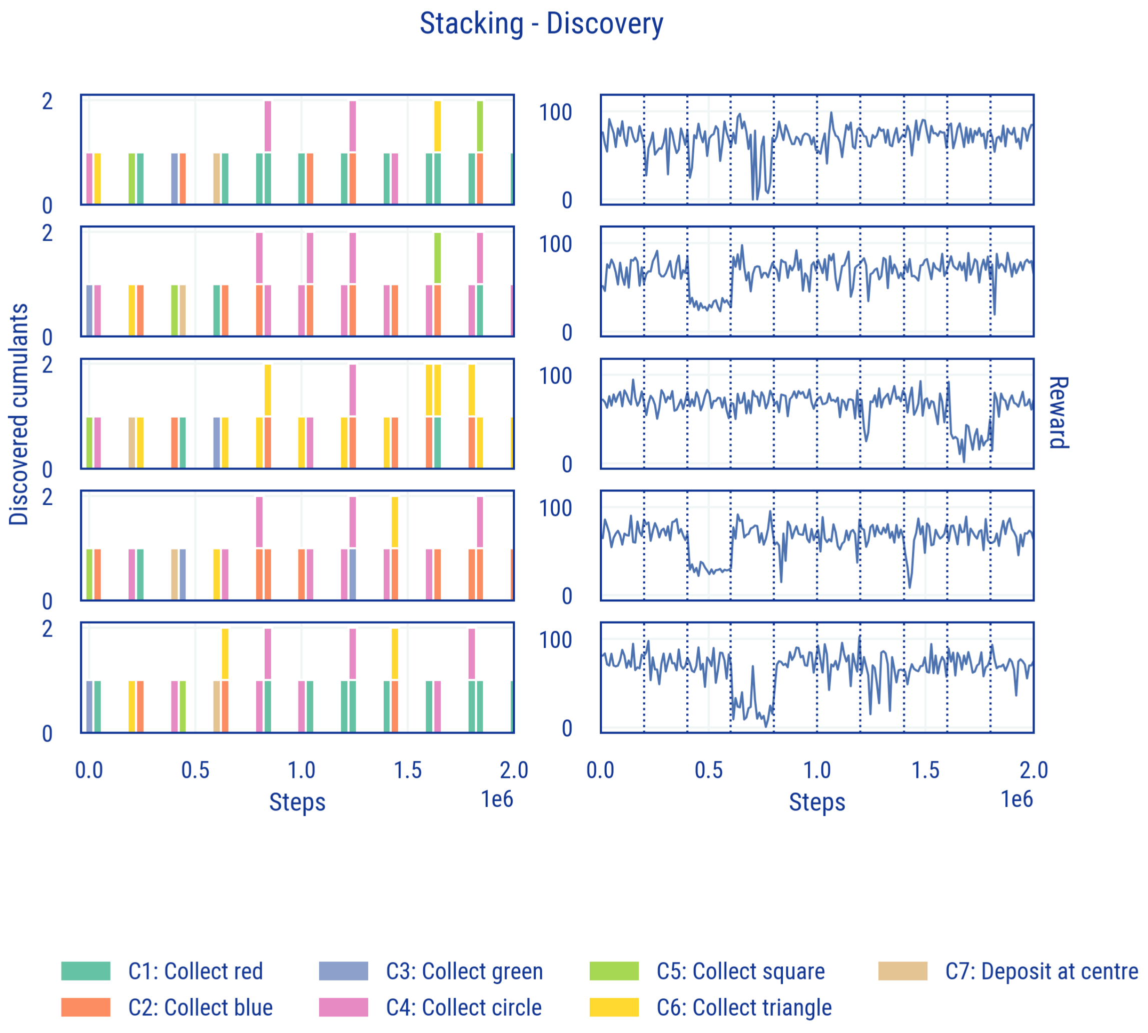


Figure 4.21. Qualitative results of our approach on the Block Stacking environment for a keyboard of size 4

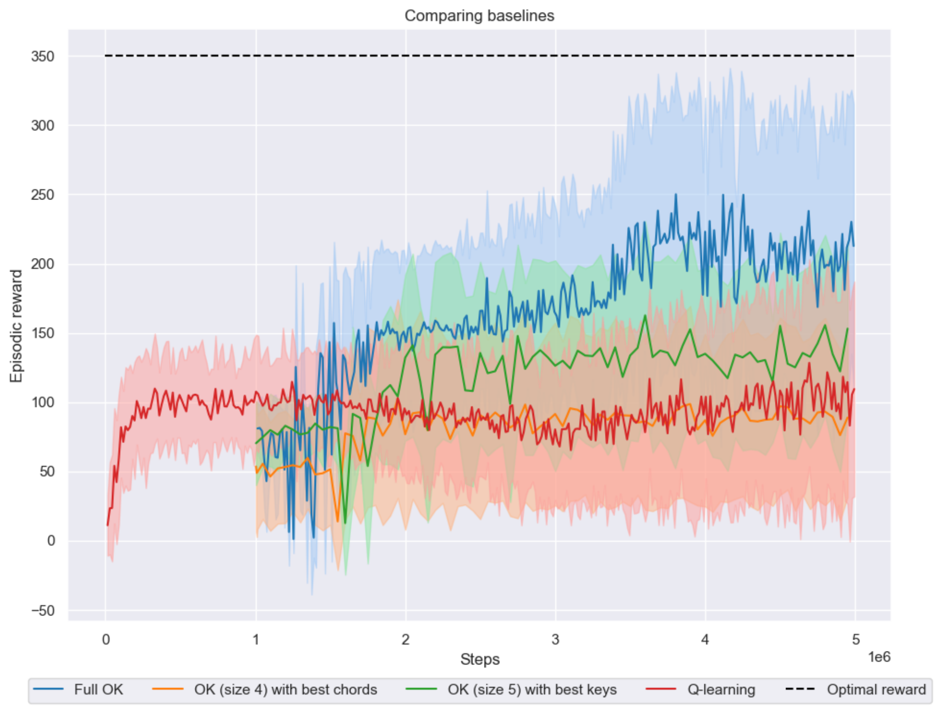


Figure 4.22. Learning curves for baselines compared with two *optimal* keyboards. The orange curve corresponds to a keyboard with fixed configuration *(red, circle), (blue, circle), (blue, triangle), (deposit at centre)*. The green curve corresponds to a keyboard with fixed configuration *(red), (blue), (circle), (triangle), (deposit at centre)*. Both of these could be reasonably considered by a human to be the *ideal* keyboards of size 4 and 5 respectively.

similar to that of the Q-learning baseline. The reason for this is that it is not always necessary that the option synthesized for a chord attains the correct block. Sometimes, because of incorrectly estimated Q-values of the base options, we observe that instead of collecting the right block, the agent oscillates between two states, resulting in a meaningless option synthesized for a chord. As a result of this, even if the underlying cumulants should in principle induce optimal performance, this is not observed empirically.

4.6.4 Discussion

Here, we showed the results of our approach on the Stacking environment, and the evolution of the keyboard over time. There are some insights that we can draw from the results and from the observed behaviour of the agent:

1. The proposed approach gives an easily interpretable solution since the cumulants that comprise the keyboard are available to us at each time, and we can observe the behaviour of the agent based on the chord that is being played by the keyboard player.

2. It is also scalable in that we have reduced the space of potential chords and are trying out keyboards that are smaller than the full OK.

3. The base options heavily impact the performance of the agent. Thus, any error in estimating the Q-values of the base options will percolate to the keyboard player, and eventually to the discovery module.

4. We have essentially reduced the problem of discovering the ideal keyboard to one of discovering the best chords. Thus, instead of selecting one keyboard from a set of size $\binom{2^n}{k}$, we are selecting the top k chords from a set of size 2^n. While this helps with scalability, it also means that since not all keyboards will be experienced, the agent may be unable to get to the optimal performance in a reasonable amount of time.

4.7 Related Work

4.7.1 Option Discovery

Option discovery has been approached in prior work from a diverse set of perspectives, ranging from learning options along with the high-level controller in an end-to-end manner (Bacon et al., 2017; Klissarov et al., 2017; Tiwari and Thomas, 2019), to learning intrinsic reward functions or eigenpurposes that induce eigenoptions (Machado et al., 2017; 2018; Liu et al., 2017), to option discovery specifically tailored to the lifelong learning setting (Tessler et al., 2017). Earlier work on option discovery framed the problem as one of discovering useful subgoals in the form of bottleneck states i.e. states that occur more frequently in successful trajectories or states that connect useful rewarding scenarios (McGovern and Barto, 2001; Stolle and Precup, 2002; Şimşek and Barto, 2004; Şimşek et al., 2005). There also exist several works in literature that cater to option discovery in continuous high-dimensional domains. Some common directions are discovery of motor primitives (Heess et al., 2016; Frans et al., 2017) that can be used by a higher-level control module, as well as skill chaining (Konidaris and Barto, 2009; Bagaria and Konidaris, 2019).

4.7.2 Unsupervised Skill Discovery

There is also a large body of work on unsupervised skill discovery i.e. learning skills without using the environment reward function by using mutual information maximization and other means of maximizing diversity in the discovered set of skills (Gregor et al., 2016; Warde-Farley et al., 2018; Eysenbach et al., 2018; Sharma et al., 2019; Baumli et al., 2021).

Our work directly builds on that of Barreto et al. (2019) who propose an approach to combine learnt skills without any additional learning. Similar to prior works in skill discovery (Sharma et al., 2019; Eysenbach et al., 2018), the skills are first learnt in a pretraining stage, following which they are used in a downstream task to maximize the environment reward. These learnt skills, which we formalize as options, are combined using GPE and GPI used previously in Barreto et al. (2017; 2018; 2019; 2020) to synthesize new more complex skills in a zero-shot fashion. Our work proposes an approach to use these complex skills synthesized using GPE and GPI to constitute the OK and solve the discovery problem.

4.8 Conclusion

In this chapter, we proposed an approach that attempts to solve the option discovery problem by selecting features from the observation space that are relevant to the reward function. We used the Option Keyboard algorithm as the underlying approach to combine options induced by these features in a zero-shot manner. Our method has the added advantage of providing an interpretable solutions to the discovery problem in that the synthesized keyboard can easily be analyzed intuitively since the constituents of the keyboard, the chords, are expressed in terms of these features.

4.9 Future directions

Since our approach relies on the optimality of the base set of options, it would be interesting to analyze the performance of our approach with optimal options and suboptimal options with varying levels of suboptimality. Along the same lines, it would be very beneficial to have an algorithm that allows for the base options to be updated online, instead of having the two-stage method that we use. Furthermore, if we truly want to tackle the discovery problem without any bells and whistles, the features that are maximized or attained to induce options must be autonomously discovered by the agent instead of being prespecified by the algorithm designer.

This chapter is being prepared for submission (Aditi Mavalankar, André Barreto, David Abel, Diana L. Borsa, G. Zacharias Holland, Doina Precup, *Synthesizing the Option Keyboard: Option Discovery in Reinforcement Learning*). The dissertation/thesis author was the primary investigator and author of this paper.

Chapter 5

Discovering useful states for exploration

A group of leprechauns is under attack by humans, and they have to run away from the rainbow with their pots of gold. They all run off in different directions away from the base of the rainbow where the humans will inevitably go searching for them. However, each leprechaun's pot has a hole in it that keeps getting larger; as a result, the leprechauns lose gold coins along the way.

In this setting, the agent is a human chasing the leprechauns. Since the world is infinitely large, from the perspective of the agent, it is not possible for the agent to visit all possible places in the world. Thus, it must make use of these gold coins dropped by the leprechauns as cues for potential directions the leprechaun/s might have gone in.

What does the human agent know?

- The leprechauns possess gold coins.
- The leprechauns will not part with their coins willingly, so if the agent does see any coins, they will have been dropped purely by accident. Note that this makes the implicit assumption that the leprechauns are unaware that they are losing gold coins; if they were aware of this, they could potentially use this fact to deceive the humans.
- The number of gold coins is indicative of either the number of leprechauns that crossed a particular part of the world, or of the distance travelled by the leprechauns, or both.

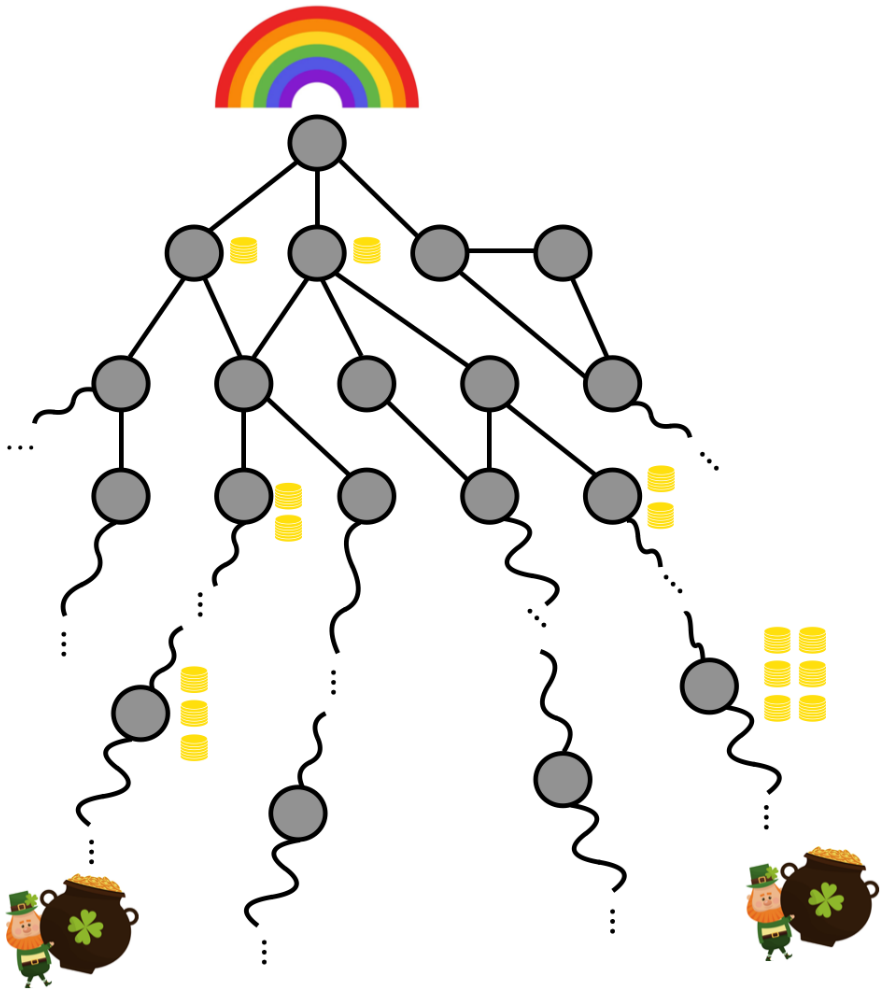


Figure 5.1. An illustration of the problem: the leprechauns scatter away, leaving gold coins behind and the human agent has to chase them using these gold coins as signals.

What does the human agent not know?

- How many leprechauns there were originally.
- How many gold coins each leprechaun had. Note that this is for the finite case where leprechauns have a finite number of gold coins.

Once the human collects the gold coin/s at a particular place, they are not replenished, i.e. the next time the agent lands at that spot, it does not see any gold coins.

We make two assumptions for the human in addition to the points listed above:

- The human cannot visit every place in the world, as stated earlier.

- The human has infinite memory, i.e. it remembers every place it has been to and the number of gold coins it collected there.

The task is simple, the human agent has to search for leprechauns by following the gold coins left by them. The more gold coins it encounters, the more likely it is to be on the path traveled by one or more of the leprechauns. Furthermore, the higher the quantity of gold coins at a certain spot, the better it is for the human agent since it indicates that either multiple leprechauns crossed that spot, or that it is on the exact path followed by a single leprechaun and has gone deeper than it would trivially.

5.1 Formalizing the problem in the RL framework

Here, we will map the above described problem to an RL problem and draw an analogy between the human, world, and assumptions discussed above to their RL counterparts.

- *Agent*: The human chasing the leprechauns is the RL agent.
- *Environment*: The world in this case is the environment; we consider the environment to be a graph that is infinite.
- *State*: In this case, we use the terms *state* and *observation* interchangeably; the state is a particular position in the world, or a node in the graph.
- *Action*: The action is an edge in the graph, or a transition that the agent makes when it goes from one position to another. To simplify this, we could also think of an action being any adjacent node to the current node that the agent could travel to.
- *Reward*: The number of gold coins at any spot or node is the reward that the agent receives on collecting them.

Assumptions on the environment

- Since the gold coins are not replenished, the reward function is non-stationary in that the reward at a certain node vanishes after the first time that node is visited by the agent. This is an assumption on the environment.
- The leprechauns slowly lose gold coins, which means the environment reward is sparse.
- Since the leprechauns are unaware of the holes in their pots, they lose coins accidentally. This means that the reward function is *not deceptive*; the presence of gold coins, in this case, non-zero rewards, does indicate that that state is indeed desirable.
- The state space is infinite, and cannot be covered by the agent.

Assumptions on the agent

- The agent has infinite memory, and so, in principle, can store every single node and edge that it has experienced in the graph, along with the corresponding rewards obtained.
- The human is unaware of how many leprechauns were present; as a result, there is no *optimal policy* that it can learn which will enable it to chase *all* the leprechauns. This may be construed as an assumption on the environment, but we include it in the agent's assumption since it deals with the behaviour of the agent in an environment where there is no policy that can enable the agent to capture *all* the leprechauns.

This problem has a flavour of lifelong learning in several ways. First, the assumption that the environment is infinitely large and the agent is much smaller than the environment is characteristic of the problem of lifelong learning. Second, because of this reason, there may not exist the notion of an *optimal policy*; instead, it makes more sense to evaluate the performance of an agent in terms of the overall reward it accumulates over time, the rate at which it accumulates this reward, the number of new states it has visited from the beginning of its life, etc. In this chapter, we will propose an approach to solve this problem in a manner that incorporates the benefits of breadth-first search (BFS), which collects *all* rewards systematically, but for this

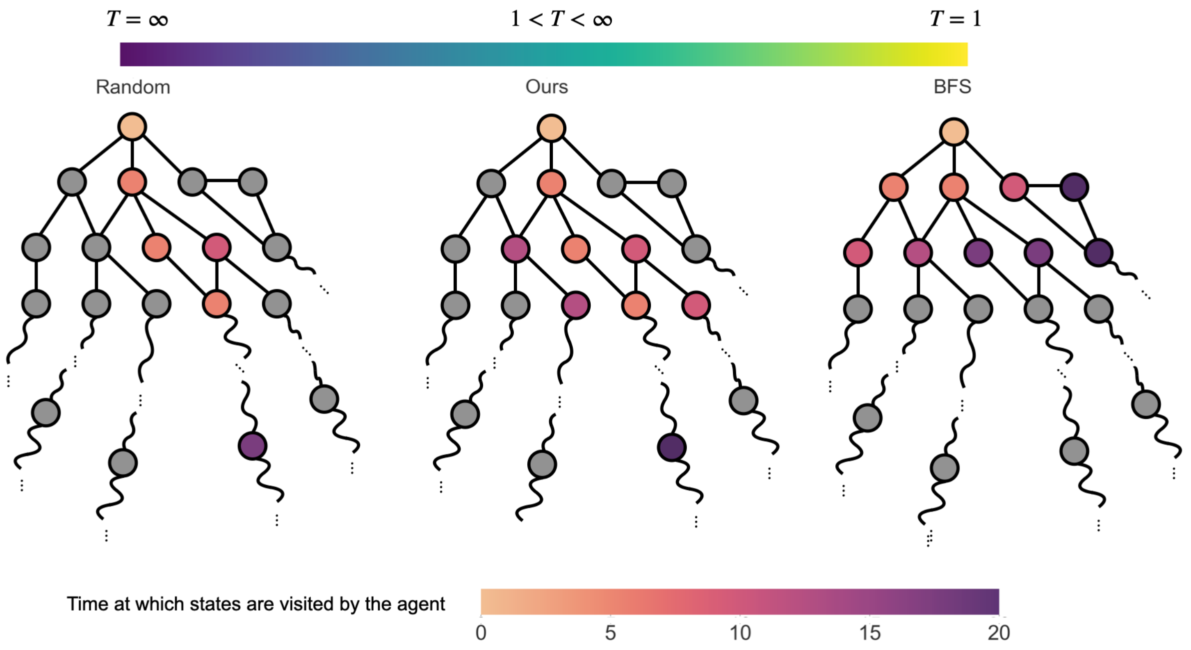


Figure 5.2. Illustration of our approach in the spectrum of graph algorithms. The colours of the nodes indicate the time at which they are visited in each respective approach. The grey nodes indicate the nodes that are not visited during the first 20 steps as indicated by the colour bar at the bottom.

reason is very slow in exploring the state space, and random search, which explores more states than BFS, but is reward-agnostic. Thus, our approach can be considered to be somewhere in the middle of the spectrum ranging from random-search to BFS.

5.2 Lifelong Learning

Here, we will position our problem in the realm of lifelong learning and establish the scope of our proposed approach. We derive the requirements of a lifelong learning agent from Schaul et al. (2018):

1. The environment can be much larger than the agent's capacity and so, the agent may visit only a tiny fraction of states – our proposed approach is motivated by this exact idea.
2. The observations can be much richer than the agent's capacity to store them in memory – we assume for now that the agent has infinite memory so this requirement is not satisfied.

3. The reward signal may provide little information – this is partially true with our approach; the reward signal is sparse but the agent can still extract useful information from it.

4. The environment's dynamics and reward functions may change over time – we assume for now that the environment dynamics and reward function remain unchanged over time.

5. The agent is never reset and has a single very long interaction with the environment – this is true for our proposed approach, we do not reset the agent.

6. The agent can irreversibly change its possible futures – this is an indirect consequence of the type of problem setting we consider, and orthogonal to our proposed approach.

5.3 Approach

Our proposed approach attempts to solve the problem by alternating between an exploration phase and a decision phase, as shown in Figure 5.3. In the exploration phase, the agent explores the environment and stores all states (nodes), transitions (edges) and rewards in its model $\mathscr{G}_{agent}$. This exploration strategy could be any policy π_{agent}; for the sake of simplicity, let us assume that the agent explores randomly, i.e. π_{agent} is a random policy. Thus, the agent's model while it is exploring is a subgraph of the true environment. Once this exploration phase is over, the agent enters the decision phase. We can assume for now that the duration of the exploration phase is fixed; however, this can be replaced with any heuristic function or learning algorithm to determine when the agent should stop exploring. In the decision phase, the agent chooses a state in its model to which it backtracks so that it can explore from this state. This idea is similar to that proposed in the Go-Explore set of algorithms by Ecoffet et al. (2021); we discuss key differences between our approach and theirs in Section 5.7. The major difference is in how these candidate states for backtracking are selected. We propose to do this by considering the boundary of the agent's model $\mathscr{G}_{agent}$. The states that lie on the outer boundary of $\mathscr{G}_{agent}$, i.e. the states that are connected to nodes in $\mathscr{G}_{agent}$, are possible candidates for exploration. The agent could uniformly sample any of these nodes from the outer boundary and explore from then on. However, it might be useful to consider the reward the agent can expect at each of these nodes while making this decision.

We propose the following to estimate the reward at each boundary node: all boundary nodes are treated as *absorbing states*; the agent adds these nodes and the edges leading to them to its model $\mathscr{G}_{agent}$ temporarily, but these edges are directed. Let us call this temporary agent model $\tilde{\mathscr{G}}_{agent}$. If A is the adjacency matrix for $\mathscr{G}_{agent}$, assume $\tilde{A}$ is the adjacency matrix for $\tilde{\mathscr{G}}_{agent}$. Note that $\tilde{A}$ will *not* be symmetric, since the agent considers boundary nodes to be absorbing states. Now, since the edge weights do not hold significance in our setting, the adjacency matrices are all binary. We normalize the adjacency matrix $\tilde{A}$ along its rows and then compute $\lim_{n\to\infty}\tilde{A}^n$. This can be obtained easily by an eigendecomposition of $\tilde{A}$. An interesting point is that in the limit as n tends to infinity, the entire mass of the adjacency matrix is concentrated at the absorbing states. This allows us to compute the effective reward at each of these absorbing states by simply performing $r_{diffused} = r_{\tilde{\mathscr{G}}_{agent}} \times \lim_{n\to\infty}\tilde{A}^n$. Here, $r_{\tilde{\mathscr{G}}_{agent}}$ is the reward vector for $\tilde{\mathscr{G}}_{agent}$. One important thing to note here is that the agent does not know the frequency at which it obtains a reward. As a result, we consider *all* random walks between all pairs of nodes in $\tilde{\mathscr{G}}_{agent}$. If the agent were aware of the frequency n at which the leprechauns drop the gold coins, we could instead compute $\sum_{i=1}^{n}\tilde{A}^n$.

Once the agent has the diffused rewards $r_{diffused}$, it can sample a boundary node for exploration in any way; specific methods include greedy sampling, random sampling, or softer methods like softmax. After selecting the boundary node which is the best candidate for exploration, the agent takes the shortest path to this node *with respect to its model* $\mathscr{G}_{agent}$. It is important to note here that the shortest path on the agent's model may not be the true shortest path; this could happen when the agent has not experienced a transition (edge) that directly connects two nodes that exist in $\mathscr{G}_{agent}$. Since the boundary nodes are not actually experienced by the agent, we go back to using $\mathscr{G}_{agent}$ instead of $\tilde{\mathscr{G}}_{agent}$. This ends the decision phase, and the agent again starts exploring. This process repeats in a cyclic fashion. This entire approach is

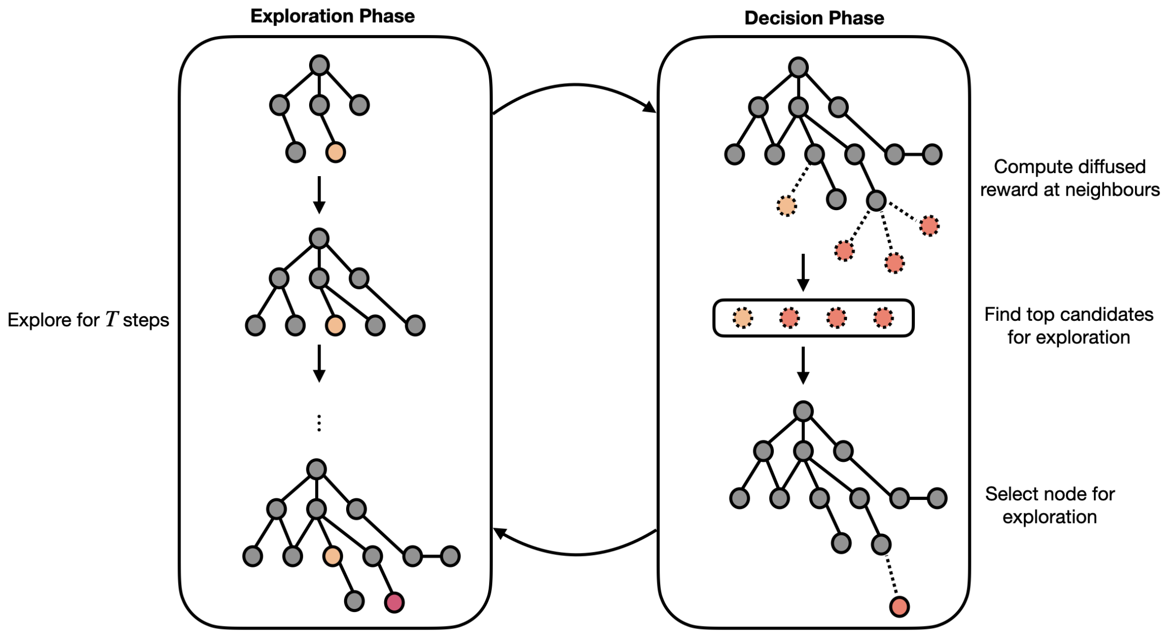

Exploration Phase
Decision Phase
Explore for T steps
Compute diffused reward at neighbours
Find top candidates for exploration
Select node for exploration

Figure 5.3. Our approach: the agent alternates between an exploration phase and a decision phase.

detailed in Algorithm 9, and the method to compute diffused reward is detailed in Algorithm 10.

5.4 Experiments

In this section, we present empirical results comparing our approach with reasonable baselines. We also discuss the environment setting, and evaluate our approach by considering different configurations of the environment. We will consider two methods of evaluating our approach. Finally, we will provide an analysis of the impact of making different design decisions in the algorithm in terms of the evaluation criteria.

5.4.1 Baselines

We test our approach on some large graphs and compare them with the following baselines:

1. **Random:** This is a naïve random exploration method, that selects at each step a neighbour from a uniform distribution over neighbours. Thus, it is reward-agnostic.

Algorithm 9. Lifelong Exploration in Infinite Graphs

Require:

`get_boundary_nodes`	compute the outer boundary of a subgraph
`get_boundary_edges`	get edges along the boundary of a subgraph
`compute_diffused_reward`	diffuse actual reward to the outer boundary
`shortest_path`	compute the shortest path between two nodes
π_{agent}	exploration policy of the agent
E	environment
T	exploration time
f_{select}	candidate selection strategy
s_0	initial state

```
1: $\mathscr{G}_{agent} \leftarrow \{$nodes: $\{s_0\}$, edges: $\{\}\}$        ▷ Agent's model of the environment dynamics
2: $r_{\mathscr{G}_{agent}} \leftarrow \{s_0 : 0\}$        ▷ Agent's model of the environment reward
3: $t \leftarrow 0$
4: loop
5:     for $T$ steps do        ▷ Exploration phase
6:         $s_{t+1} \sim \pi_{agent}(E, s_t)$
7:         Take action $s_{t+1}$ in env $E$ and observe reward $r_{t+1}$
8:         if node $s_{t+1} \notin \mathscr{G}_{agent}$ then        ▷ Add state and reward to agent's model
9:             $\mathscr{G}_{agent} \leftarrow \mathscr{G}_{agent} \cup \{s_{t+1}\}$
10:            $r_{\mathscr{G}_{agent}} \leftarrow r_{\mathscr{G}_{agent}} \cup \{s_{t+1} : r_{t+1}\}$
11:        if edge $(s_t, s_{t+1}) \notin \mathscr{G}_{agent}$ then        ▷ Add edge to agent's model
12:            Add edge $(s_t, s_{t+1})$ to $\mathscr{G}_{agent}$
13:        $t \leftarrow t + 1$
14:    $\mathscr{B} \leftarrow$ get_boundary_nodes($\mathscr{G}_{agent}$)        ▷ Decision phase
15:    $\mathscr{E} \leftarrow$ get_boundary_edges($\mathscr{G}_{agent}$)
16:    $r_{diffused} \leftarrow$ compute_diffused_reward($\mathscr{G}_{agent}, \mathscr{B}, \mathscr{E}$)
17:    $s_g \leftarrow f_{select}(\mathscr{B}, r_{diffused})$        ▷ Select best boundary node for exploration
18:    $p \leftarrow$ shortest_path($\mathscr{G}_{agent}, s_t, s_g$)
19:    repeat        ▷ Take shortest path to best boundary node
20:        $s_{t+1} \leftarrow p$.next()
21:        Take action $s_{t+1}$ in env $E$
22:        $t \leftarrow t + 1$
23:    until $s_t = s_g$
```

Algorithm 10. Computing the diffused reward

```
Require: { $\mathscr{G}_{agent}$   agent's model of the environment dynamics
         { $r_{\mathscr{G}_{agent}}$   agent's model of the environment reward
         { $\mathscr{B}$   nodes at the outer boundary of the agent's model
         { $\mathscr{E}$   edges at the boundary of the agent's model
 1: $\tilde{\mathscr{G}}_{agent} \leftarrow \mathscr{G}_{agent} \cup \{$nodes: $\mathscr{B}$, edges:$\mathscr{E}\}$   ▷ Add boundary nodes and edges to agent's model
 2: $r_{\tilde{\mathscr{G}}_{agent}} \leftarrow r_{\mathscr{G}_{agent}}$   ▷ Augment reward with zeros for newly added nodes
 3: for each node $b \in \mathscr{B}$ do
 4:     $r_{\tilde{\mathscr{G}}_{agent}} \leftarrow r_{\tilde{\mathscr{G}}_{agent}} \cup \{b : 0\}$
 5: for each node $b \in \mathscr{B}$ do
 6:     for each edge $(b, s') \in \mathscr{E}$ do
 7:         Remove $(b, s')$ from $\tilde{\mathscr{G}}_{agent}$   ▷ Make boundary nodes absorbing states
 8: $\tilde{A} \leftarrow$ adjacency_matrix($\tilde{\mathscr{G}}_{agent}$)
 9: $\tilde{A} \leftarrow$ row_normalize($\tilde{A}$)
10: $\tilde{A}_\infty \leftarrow \lim_{n\to\infty} \tilde{A}^n$
11: $r_{diffused} \leftarrow r_{\tilde{\mathscr{G}}_{agent}} \times \tilde{A}_\infty$
12: return $r_{diffused}$
```

2. **BFS:** This is a classical algorithm for searching in graphs: Breadth First Search (BFS). In this method, the agent explores nodes at each depth d from the initial node before proceeding to the nodes at depth $d+1$. This method is also reward-agnostic.

3. **UCB:** In this method, we use the Upper Confidence Bound (UCB) method for selecting the node to be visited next at each step. This method is *not* reward-agnostic, since the value of each neighbour incorporates the reward obtained, i.e. $Q(s') \leftarrow r(s') + c\sqrt{\frac{\log N}{n_{s'}}}$, where $r(s')$ is the reward observed at the candidate next node (initialized to 0 if the node is unexplored), c is the exploration coefficient, N is the visitation count of the parent i.e. the current node, and $n_{s'}$ is the visitation count of the candidate node s'. Thus, this approach continuously explores till it reaches a point where all neighbours have been explored, and then acts greedily based on the reward observed at each neighbour. Due to its strong exploratory component, it is conceivably a strong baseline with which we can compare our proposed approach.

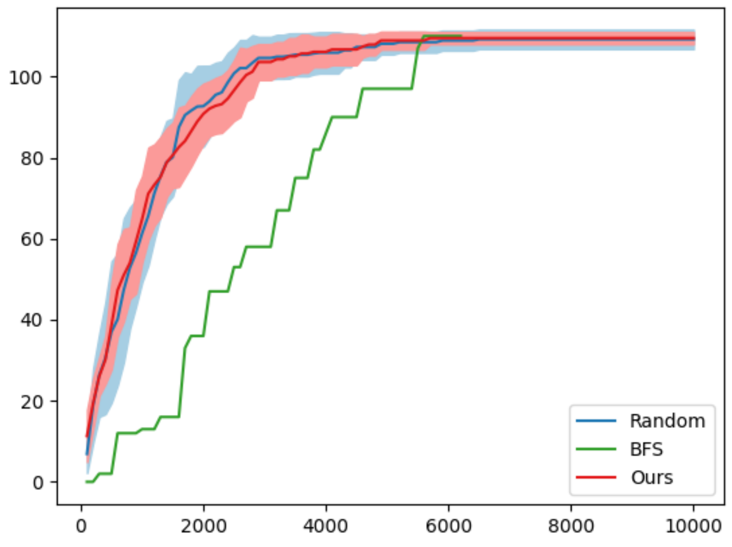


Figure 5.4. Results of our approach and baselines on a random graph: the X-axis indicates the number of steps in the environment and the Y-axis indicates the total reward accumulated by the agent.

5.4.2 Environment

In the setting we have described, the environment is an infinite graph. We use a large graph as a proxy for an infinite graph.

Random graphs

We first start with an Erdős-Rényi graph, or a random graph with a few thousand nodes as the underlying graph for the environment. We experimented with random connected graphs of size 1000, 5000, and 10000 with different degrees. The reward was generated by simulating a variable number of particles that traversed the graph in a random fashion, and left an increasing number of non-zero rewards intermittently. The result of this experiment is shown in Figure 5.4. Here, we see that the cumulative reward of our approach is similar to that of the random baseline. In hindsight, this was to be expected; for a highly connected random graph, the diameter is quite small, because of which the agent can achieve high reward through a random policy easily. Thus, in order to simulate a more reasonable graph for lifelong exploration, we consider a random tree.

Random trees

We consider a random tree of size 5000 for all our experiments. The reward function is determined by simulating a certain number of particles that at each step move *away* from the origin or the root node, and leave non-zero increasing rewards intermittently. For the purpose of our experiments, we fix this interval to be 5, i.e. at each depth $5d$ in the tree, the agent can get rewards. We report results for different numbers of these particles, in addition to results on different configurations of our agents.

5.4.3 Evaluation Criteria

We propose using the following two ways to evaluate our approach and the baselines:

1. **Cumulative Reward:** As is evident, we measure progress in terms of the total reward accumulated by the agent during its lifetime.
2. **State Visitation:** We also report the number of distinct states visited by the agent over its lifetime as a way of measuring its exploratory capability.

5.5 Results

For our approach, the following design decisions are to be made:

- Length of the exploratory phase: We consider three values in our experiments: $T = 10$, $T = 50$, and $T = 100$.
- Exploration strategy: We consider two possible strategies for the agent to use during the exploration phase: random and novelty-seeking. Random is self-explanatory; the novelty-seeking strategy on the other hand, prioritizes neighbours that are not present in the agent's model. If all neighbours have been experienced, it selects one uniformly at random.

- Candidate selection strategy: We consider three possible strategies for selecting the next best candidate for exploration: top-*k*, greedy, and softmax. All of these functions are applied to the diffused reward computed at the candidate nodes. For all our experiments, when we use the top-*k* strategy, we use $k = 3$.

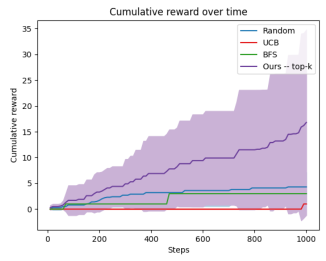
We will show results of our approach by tuning each of these knobs, and later analyze the impact of each of these design decisions.

For all experiments, the results shown indicate the mean and standard deviation across 10 different runs on the same graph.

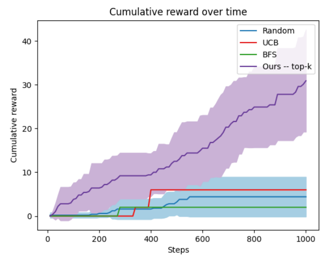
5.5.1 Varying the number of reward-generating particles

Here, we look at the result of varying the number of particles that deposit rewards at the nodes. Note that this is an *environment*-level ablation. The environment dynamics and reward functions are different for each experiment, but consistent across baselines. The agent configuration remains the same throughout these experiments: the length of the exploratory phase is 10 steps, the exploration strategy used is novelty-seeking, and the strategy used to select next candidates is the top-*k* strategy.

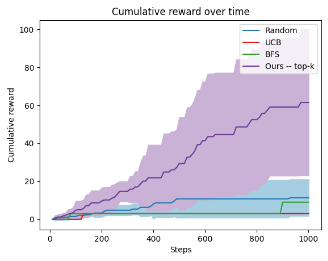
The results in Figure 5.5 indicate that our approach on average accumulates higher reward than any of the baselines. The high variance is on account of the fact that different agents may discover useful states at different points of time during their initial exploratory phases. The results of the random and BFS baselines are as expected due to their reward-agnostic nature. Furthermore, although the UCB baseline is not reward-agnostic, it still gives a preference to previously unexplored neighbours over explored greedy neighbours. As a result, its reward also increases much more slowly. If we look at the state visitation over time, UCB is as good as or better than our approach. This is also expected since our approach has a decision stage that is exploitative in terms of the reward and involves taking steps towards promising states through backtracking on a graph consisting of already experienced states.



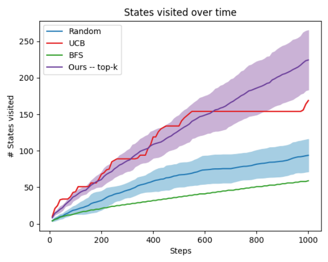
(a) $n = 1$



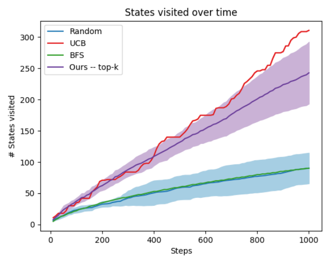
(b) $n = 2$



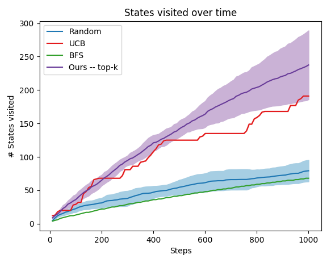
(c) $n = 3$



(d) $n = 1$



(e) $n = 2$



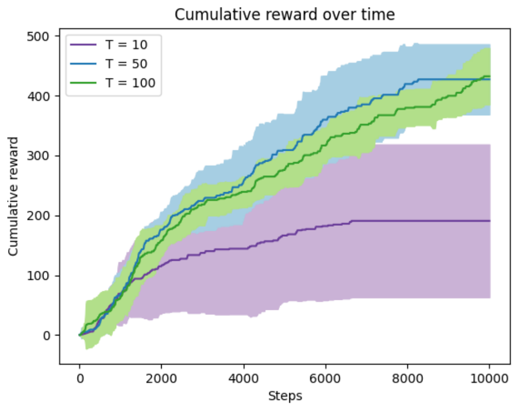
(f) $n = 3$

Figure 5.5. Results of our approach and baselines for a varying number of leprechauns. The top row shows the reward accumulated by each agent over time, and the bottom row shows the number of distinct states visited by each agent over time.

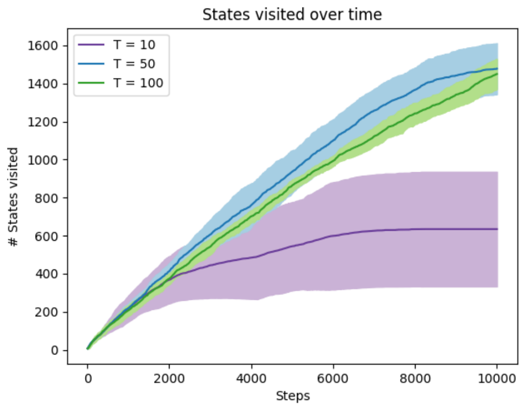
5.5.2 Varying the length of the exploratory phase

In this ablation experiment, we compare the performance of our agent on different durations of the exploration phase. We try $T = 10$, $T = 50$, and $T = 100$. The exploration strategy is novelty-seeking, and the candidate selection strategy is top-k.

The results in Figure 5.6 clearly indicate that alternating between the decision phase and the exploratory phase very frequently causes the agent to stop exploring, which is the case for $T = 10$. Here, we see that along with the stagnated cumulative reward, the state visitation for this experiment also stagnates after a point, indicating that the agent is not capable of adding good candidates for exploration to its model due to the short duration of the exploratory phase. As we allow the agent to explore for a longer time, we see that it keeps accumulating rewards by consistently exploring new states, which can observed in the trends for $T = 50$ and $T = 100$ in both plots in Figure 5.6.

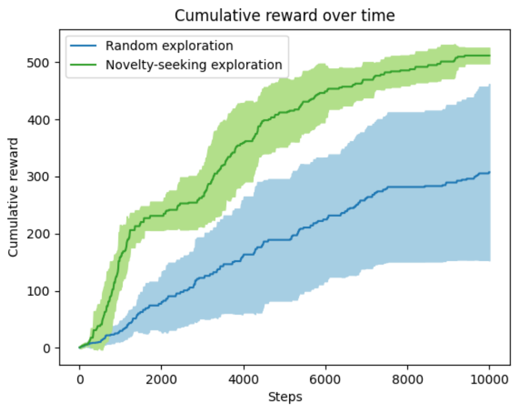


(a) Cumulative reward

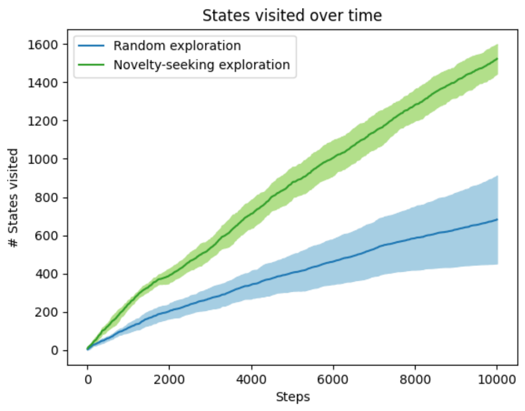


(b) State visitation

Figure 5.6. Ablation experiment for our approach comparing agents that have different lengths of their exploratory phases.

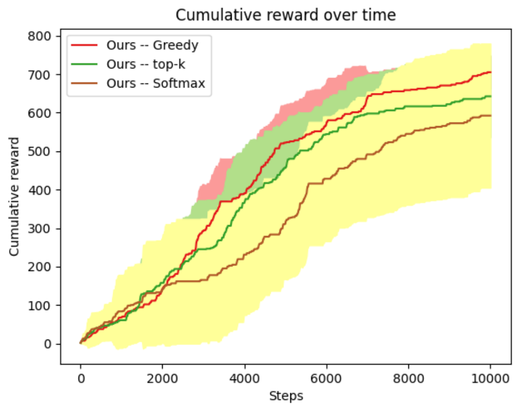


(a) Cumulative reward

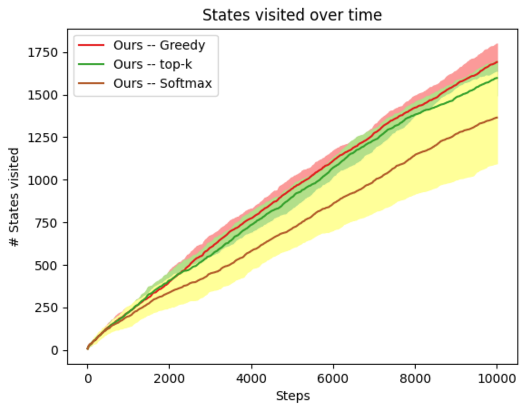


(b) State visitation

Figure 5.7. Ablation experiment for our approach comparing agents that have different exploration strategies.



(a) Cumulative reward



(b) State visitation

Figure 5.8. Ablation experiment for our approach comparing agents that have different candidate selection strategies.

5.5.3 Varying the exploration strategy

In this ablation experiment, we compare the performance of our agent on different exploration strategies. We try the random strategy and the novelty-seeking strategy.

From the plots in Figure 5.7, it is very clear that the choice of exploration strategy has a significant impact on the performance of our agent. Choosing a random strategy for exploration gives worse results because the agent is equally likely to go to a neighbour it has already seen before as it is to go to a neighbour that is unexplored. Thus, even when we have a decision phase in which the agent backtracks to a good state from which it can explore, the agent is unable to do very well if the model of the environment that it has is limited due to a bad exploration strategy.

5.5.4 Varying the candidate selection strategy

In this ablation experiment, we compare the performance of our agent on different candidate selection strategies. We try top-*k*, greedy, and softmax, as described earlier.

The plots in Figure 5.8 indicate that the choice of a candidate selection strategy based on the diffused reward does not have a major impact on performance in terms of the cumulative reward, or state visitation. It seems that the softmax strategy is slightly worse; this might be the case because it assigns a non-zero probability to neutral candidates as opposed to the greedy strategy which always selects the candidate node for exploration with the best diffused reward, and the top-k strategy, which selects one out of the top k candidates.

5.6 Qualitative Analysis

In this section, we present qualitative results for our approach and the baselines on a much smaller graph with 250 nodes, and 2 reward-generating particles. The full environment is shown in Figure 5.9. As we can see, the reward function is very sparse, with only a few nodes having non-zero reward. The agent starts at the origin (marked with a star).

We now show qualitative results for the states and rewards experienced by each agent over time in Table 5.1. These figures show the states and rewards visited by the agent over time in each of the baselines and our approach.

We first look at the subgraphs experienced by the respective agents at time $t = 10$. We see that the random and BFS baselines experience a smaller number of states than UCB and our approach. This is to be expected for the reasons discussed previously; BFS visits all nodes at a certain depth from the root before moving to the next level, while for the random agent, it is as likely to visit a previously experienced neighbour as an unexplored one. UCB and our approach visit a similar number of states. Note that none of the approaches experiences any reward in the first 10 steps. This is reasonable behaviour; since the reward function is sparse, the initial reward is harder to stumble upon.

Next, we look at the results at time $t = 20$. The BFS results are as expected, the agent visits fewer states than the others. The random agent experiences a slightly higher number of states. However, the main interesting trend we see is in the UCB agent. We would expect the UCB agent to visit more states than or a similar number of states as our agent; however, we see that the UCB agent does not visit *any* new states in time $10 < t \leq 20$. For this, we take another look at the original graph in Figure 5.9. We see that the path along which the agent initially went

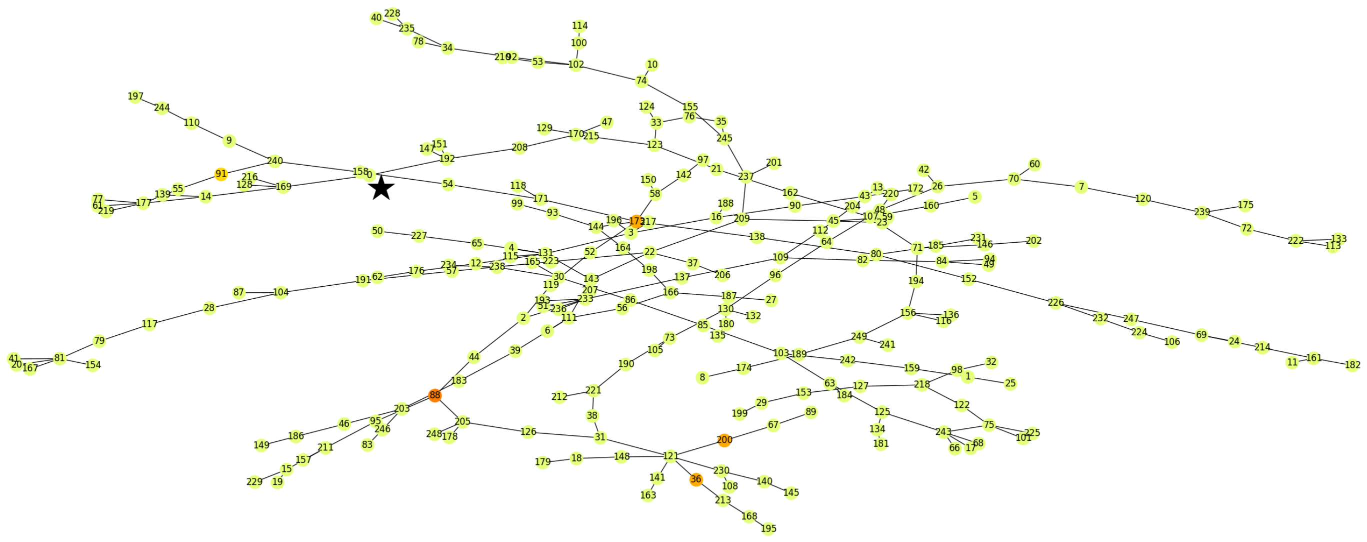
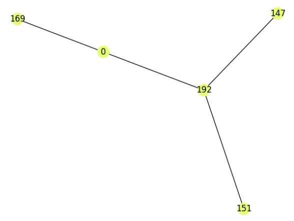
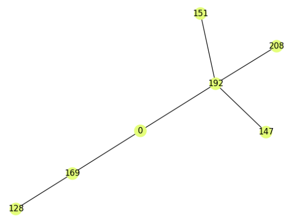
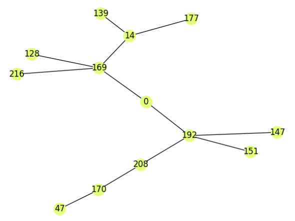
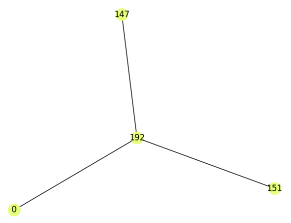
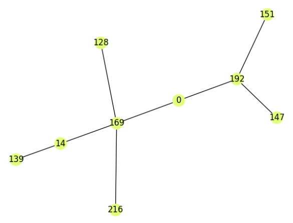
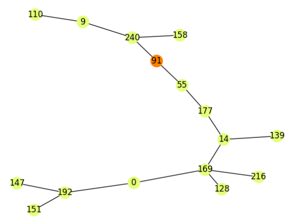
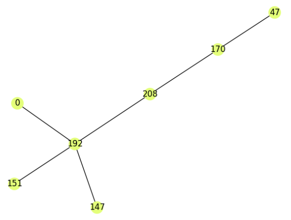
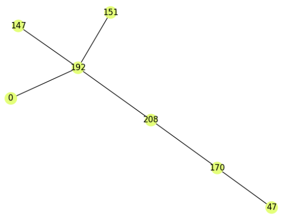
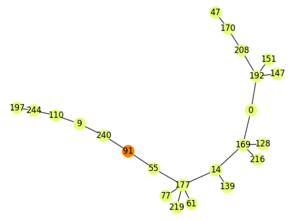
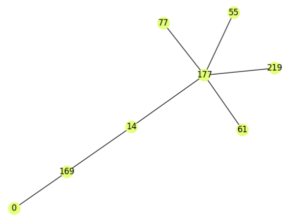
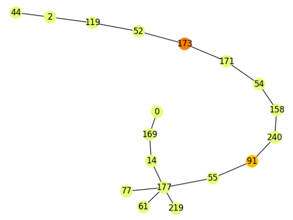
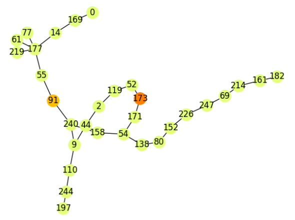


Figure 5.9. An illustrative example of a smaller graph with 250 nodes. The colours of the nodes indicate the reward at these nodes. Most rewards are light yellow, indicating no reward. Only a few nodes have non-zero rewards indicated in chrome, mustard, or orange colours. The starting point for each agent, the origin, is indicated with a star.

Table 5.1. Qualitative results showing the states and rewards experienced by each agent (baselines + our approach) at 10, 20, and 50 steps respectively.

	$t = 10$	$t = 20$	$t = 50$
BFS			
Random			
UCB			
Ours			

exploring did not contain any rewards, just as our agent's initial exploratory path did not contain any rewards. However, our agent was able to successfully backtrack to an already experienced state and start exploration afresh, while the UCB agent was unable to escape that path because that required a sequence of states to picked uniformly at random. It may be conceivable that our agent got *lucky* in that its initial exploratory path would eventually yield rewards faster than the initial exploratory path selected by the UCB agent. However, we see that even at $t = 50$, when the UCB agent *does* stumble upon a good path, it still prioritizes experiencing all possible neighbours of a node, thus delaying its accumulation of reward. This is evident in the diagram, where we see that several neighbours of nodes 177, 14, 169, and 192 are experienced, as opposed to our approach, which, at $t = 20$ and $t = 50$, discovers more rewarding states exhibiting a chain-like structure in its visitation.

5.7 Related Work

5.7.1 Curiosity-based exploration

Houthooft et al. (2016) propose using variational inference to encourage exploration; similar to our approach, they maintain a model of the environment dynamics, however, they modify the reward function to include an intrinsic reward computed as the information gain about the agent's belief of the environment dynamics. Thus, this method falls into the *curiosity-based* exploration class of approaches, which is quite different in flavour from our proposed approach and was originally proposed by Schmidhuber (1991). Another method that incorporates an intrinsic reward based on maximizing the agent's curiosity about the environment dynamics is the work by Pathak et al. (2017). Here, the intrinsic reward signal is used as a proxy for a dense reward signal in environments where the reward signal is extremely sparse or unavailable.

5.7.2 Count-based exploration

Count-based exploration techniques also incorporate an intrinsic reward in the RL objective; however, this intrinsic reward is now based on the visitation count of each state and encourages exploration. Bellemare et al. (2016) generalize tabular count-based exploration to non-tabular data by using density models to obtain pseudo-counts for state visitation. Ostrovski et al. (2017) extend this by improving the quality of the density models. Tang et al. (2017), on the other hand, achieve this by mapping states to a hash table and then using the counts of each discrete hash code in the hash table as the intrinsic reward. Machado et al. (2020) on the other hand, use the successor representation to perform count-based exploration.

5.7.3 Lifelong exploration

Garcia and Thomas (2019) address the aspect of lifelong learning that deals with changing MDPs, and formulate searching for an optimal exploration strategy as an RL problem, and use Model-Agnostic Meta-Learning (Finn et al., 2017) to learn this meta-policy for exploration. Fu et al. (2021) tackle the same aspect of lifelong learning, but instead propose learning using a Bayesian method for exploration.

5.7.4 Other approaches

Perhaps closest to our proposed approach is the Go-Explore set of algorithms (Ecoffet et al., 2021). However, there are a few key differences between their problem setting and ours. First, their approach is motivated from an empirical perspective, and the key idea is that the agent by returning to promising states for exploration, can explore all states, or a reasonably large part of the state space. On the other hand, we explicitly make the assumption consistent with lifelong learning that the agent cannot possibly experience all states, and will visit a fraction of the entire state space. Second, their approach leverages the ability of the agent to reset the simulator to a set state so that it can begin exploring thenceforth. We do not make this assumption; in our approach, in order to go to a previously determined promising state, the agent must make the corresponding transitions to states given by the shortest path algorithm. Thus, instead of resetting to a promising state, the agent actually takes steps in the environment to reach that state. Note that while our approach in its current form cannot handle non-stationary environment dynamics, it is capable of doing so because we do not make the explicit assumption of always being able to reset to a particular state if that state is not reachable under the new dynamics.

5.8 Conclusions

In this chapter, we proposed an approach that can continually discover states that are useful for exploration by taking into consideration the rewards obtained at those states. We proposed an approach that diffuses the already experienced rewards into candidate states for exploration to determine the best candidate. Our approach alternates between an exploration phase, where the agent seeks novelty, and a decision phase, where the agent selects a direction for exploration greedily. Our results indicate that our approach outperforms reward-agnostic graph-based methods like random exploration and breadth-first search, as well as a reward-maximizing UCB-like method on relatively large graphs. We also provide a detailed ablation study of the impact of several important design decisions on our proposed approach, and also evaluate all baselines and our approach for different levels of sparsity of the environment reward function.

5.9 Future Directions

One immediate future direction we are keen to pursue is to test the exploration capacity of our algorithm in a different setting: consider an environment in which rewards *can* be replenished. Thus, an agent would get a fixed reward each time it lands at a certain state. Now, in this setting, we would let our exploratory approach run for a finite number of steps after which we would evaluate each agent based on the subgraph that it has learnt; what is the maximum reward that an agent can achieve? This would require more thought as the most trivial solution would involve an agent converging to a single edge attached to the maximally-rewarding node discovered. We present more directions for future work in Chapter 6.

This chapter is being prepared for submission (Aditi Mavalankar*, Geelon So*, Yian Ma, Sanjoy Dasgupta, Doina Precup, *Lifelong Exploration in Infinite Graphs*). The dissertation/thesis author was the co-primary investigator and author of this paper.

Chapter 6

Conclusions

In this thesis, we explored three ways in which RL agents can discover useful behaviour to maximize the environment reward:

1. By discovering useful representations of the observation space: this is done by exploiting invariances that exist in the environment or the agent and incorporating them into the learning of representations that can generalize across tasks. Our work shows that learning such representations offers a quantitative advantage by allowing generalization to other tasks by removing the need to learn a new policy each time the task is changed. Furthermore, it also shows qualitative advantage in the continuous control tasks discussed by maintaining similar gait across tasks, compared to the vanilla policy that learns a new gait each time the task is changed.

2. By discovering useful options: this is done by determining which options, or combinations of options, are useful in maximizing the environment reward. We use the Option Keyboard, which enables zero-shot composition of options, as the underlying method and discover options that constitute this keyboard. Our work shows that the agent discovers options that capture the essence of the true reward function, and in addition to that, if there is extra space on the keyboard, also discovers options whose underlying objectives are surrogates to the true environment reward.

3. By discovering useful states for exploration: this is done by maintaining a model of

the environment dynamics and the reward function, and periodically choosing the best candidate node for exploration by estimating the reward at each node on the outer boundary of the agent's model of the environment. This enables the agent to explore freely, which is a benefit offered by random exploration, while also being cognizant of the reward function by backtracking to good candidates at regular intervals.

6.1 Future Directions

We will now look at interesting future directions for this thesis, and also discuss some aspects of the approaches proposed in the individual chapters that can form interesting directions for future research.

6.1.1 Non-stationary environments

One interesting direction is to evaluate and possibly extend these approaches in non-stationary environments. Environment non-stationarity can arise from non-stationarity in any or several parts of the environment. We will analyze each of these potential causes of non-stationarity and discuss which issues arise in our approaches that would require further research to work in each of these settings.

Non-stationarity in the observation space

If we consider an environment in which the observation space is non-stationary, the representations that the agent extracts, which incorporate the property of invariance we discussed in Chapter 3, also have to be updated according to the change in observations. Thus, we might need to substitute the offline setting for learning representations with a dynamic equivalent that can update these representations adaptively.

The approach we discussed for option discovery in Chapter 4 would also need to be tweaked in the case of non-stationarity in the observation space. This change would be most cumbersome in the learning of the base options, since the options themselves would perform badly when the observations lie out of the distribution on which they were trained. In this case, this non-stationarity impacts the base options and thus, even if the discovery module and player should in principle give the best possible reward, the poor performance of the base options on the changed observations impacts the overall reward.

Non-stationarity in the reward function

If the reward function changes in a manner that retains the symmetries that were used to learn the invariant representations, our approach still holds. However, if the symmetries that are applicable to generalizing to the new task change, or new symmetries need to be added, our approach will need to be modified to have an online component that can update the learnt representations.

In a setting where the environment function is non-stationary, the notion of the *usefulness* of options also changes; now, it becomes more desirable to have part of the option keyboard that retains useful options across reward functions, and the other part containing reward-relevant options, which can be updated more frequently as the reward changes.

In the infinite graph setting, the agent's model of the environment reward should also change over time as the reward function changes. However, since the agent is unaware of these changes, it would have to experience those nodes and edges again, which would become more expensive as the size of its model grows.

Non-stationarity in the environment dynamics

The impact of non-stationarity in the environment dynamics would be similar to that of a non-stationary observation space on the learning of invariant representations. The Q-values would also change, necessitating the re-learning of the base options in the case of our proposed discovery approach. For the exploration approach discussed in Chapter 5, similar to the above discussed change in the agent's model of the reward, the agent's model of the environment dynamics also has to be updated. However, since that requires the agent to experience many visited states again, it is undesirable.

6.1.2 Option Discovery

The discovery approach proposed in Chapter 4 has two stages that are executed sequentially: 1) a pretraining stage, where the base options are learnt, and 2) the discovery stage where the keyboard player iterates over several configurations of the keyboard and attempts to discover the best configuration to maximize the environment reward. Now, it is conceivable that the pretraining may not lead to perfect options in several cases: 1) non-stationarity in the observation space (as discussed above), 2) partial observability, 3) insufficient time to learn optimal options. As a result, it becomes imperative to learn or update options online. Furthermore, especially in the case of partial observability, the ability to add new options also becomes necessary.

Adding new options to the base set also adds new chords to the bandit, resulting in an ever-growing bandit as new options are discovered. While our assumptions ensure that in most practical settings, the number of chords will not be exponential in the number of base options, it is still very large. Thus one interesting future direction would be to design a mechanism for chords to be added to and evicted from the bandit.

6.1.3 Exploring in Infinite Graphs

One key assumption in our setting is that the environment reward is sparse but not deceptive. In the context of the motivating example we have provided in Chapter 5, this means that the leprechauns running away from the humans are unaware that their pots have holes in them and they are leaving traces behind in the form of gold coins. However, we could think of a more complex setting where the leprechauns *are* aware that they are leaving behind traces in the form of these gold coins, and could use them to potentially mislead the humans. This would correspond to a *deceptive* reward function, where initially high rewards may be very suboptimal in the long run. This would require us to select candidates for exploration in a different way in the decision phase, than choosing some variation of a greedy manner as is the case currently.

Bibliography

F. Abdolhosseini, H. Y. Ling, Z. Xie, X. B. Peng, and M. van de Panne. On learning symmetric locomotion. In *Motion, Interaction and Games*, pages 1–10. 2019.

J. Achiam. Spinning Up in Deep Reinforcement Learning. 2018.

M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, O. P. Abbeel, and W. Zaremba. Hindsight experience replay. In *Advances in Neural Information Processing Systems*, pages 5048–5058, 2017.

P.-L. Bacon, J. Harb, and D. Precup. The option-critic architecture. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

A. Bagaria and G. Konidaris. Option discovery using deep skill chaining. In *International Conference on Learning Representations*, 2019.

A. Barreto, W. Dabney, R. Munos, J. J. Hunt, T. Schaul, H. van Hasselt, and D. Silver. Successor features for transfer in reinforcement learning. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 4058–4068, 2017.

A. Barreto, D. Borsa, J. Quan, T. Schaul, D. Silver, M. Hessel, D. Mankowitz, A. Zidek, and R. Munos. Transfer in deep reinforcement learning using successor features and generalised policy improvement. In *International Conference on Machine Learning*, pages 501–510. PMLR, 2018.

A. Barreto, D. Borsa, S. Hou, G. Comanici, E. Aygün, P. Hamel, D. Toyama, J. hunt, S. Mourad, D. Silver, and D. Precup. The option keyboard: Combining skills in reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

A. Barreto, S. Hou, D. Borsa, D. Silver, and D. Precup. Fast reinforcement learning with generalized policy updates. *Proceedings of the National Academy of Sciences*, 117(48): 30079–30087, 2020.

K. Baumli, D. Warde-Farley, S. Hansen, and V. Mnih. Relative variational intrinsic control. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 6732–6740, 2021.

M. Bellemare, S. Srinivasan, G. Ostrovski, T. Schaul, D. Saxton, and R. Munos. Unifying count-based exploration and intrinsic motivation. *Advances in neural information processing systems*, 29, 2016.

G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016.

E. Coumans and Y. Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning. *GitHub repository*, 2016.

P. Dayan. Improving generalization for temporal difference learning: The successor representation. *Neural Computation*, 5(4):613–624, 1993.

P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, and P. Zhokhov. Openai baselines. https://github.com/openai/baselines, 2017.

A. Ecoffet, J. Huizinga, J. Lehman, K. O. Stanley, and J. Clune. First return, then explore. *Nature*, 590(7847):580–586, 2021.

B. Eysenbach, A. Gupta, J. Ibarz, and S. Levine. Diversity is all you need: Learning skills without a reward function. *arXiv preprint arXiv:1802.06070*, 2018.

C. Finn, P. Abbeel, and S. Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International conference on machine learning*, pages 1126–1135. PMLR, 2017.

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

H. Fu, S. Yu, M. Littman, and G. Konidaris. Bayesian exploration for lifelong reinforcement learning. 2021.

F. Garcia and P. S. Thomas. A meta-mdp approach to exploration for lifelong reinforcement learning. *Advances in Neural Information Processing Systems*, 32, 2019.

D. Ghosh, A. Gupta, and S. Levine. Learning actionable representations with goal-conditioned policies. *arXiv preprint arXiv:1811.07819*, 2018.

K. Gregor, D. J. Rezende, and D. Wierstra. Variational intrinsic control. *arXiv preprint arXiv:1611.07507*, 2016.

C. Grimm, I. Higgins, A. Barreto, D. Teplyashin, M. Wulfmeier, T. Hertweck, R. Hadsell, and S. Singh. Disentangled cumulants help successor representations transfer to new tasks. *arXiv* *preprint arXiv:1911.10866*, 2019.

T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *35th International Conference on Machine Learning*. PMLR, 10–15 Jul 2018.

N. Heess, G. Wayne, Y. Tassa, T. Lillicrap, M. Riedmiller, and D. Silver. Learning and transfer of modulated locomotor controllers. *arXiv preprint arXiv:1610.05182*, 2016.

A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu. Stable baselines. https://github.com/hill-a/stable-baselines, 2018.

R. Houthooft, X. Chen, Y. Duan, J. Schulman, F. De Turck, and P. Abbeel. Vime: Variational information maximizing exploration. *Advances in neural information processing systems*, 29, 2016.

L. P. Kaelbling. Learning to achieve goals. In *Proc. of IJCAI-93*, pages 1094–1098, 1993.

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015*, 2015.

J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, D. Hassabis, C. Clopath, D. Kumaran, and R. Hadsell. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, 114(13):3521–3526, 2017. doi: 10.1073/pnas.1611835114. URL https://www.pnas.org/doi/abs/10.1073/pnas.1611835114.

M. Klissarov, P.-L. Bacon, J. Harb, and D. Precup. Learnings options end-to-end for continuous action tasks. *arXiv preprint arXiv:1712.00004*, 2017.

G. Koch, R. Zemel, and R. Salakhutdinov. Siamese neural networks for one-shot image recognition. In *ICML deep learning workshop*, volume 2, 2015.

G. Konidaris and A. Barto. Skill discovery in continuous reinforcement learning domains using skill chaining. *Advances in neural information processing systems*, 22:1015–1023, 2009.

S. Lange, T. Gabel, and M. Riedmiller. Batch reinforcement learning. In *Reinforcement learning*, pages 45–73. Springer, 2012.

M. Laskin, K. Lee, A. Stooke, L. Pinto, P. Abbeel, and A. Srinivas. Reinforcement learning with augmented data. *arXiv preprint arXiv:2004.14990*, 2020.

T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra.

Continuous control with deep reinforcement learning. *ICLR 2016*, 2016.

Y. Lin, J. Huang, M. Zimmer, Y. Guan, J. Rojas, and P. Weng. Invariant transform experience replay: Data augmentation for deep reinforcement learning. *IEEE Robotics and Automation Letters*, 5(4):6615–6622, 2020.

M. Liu, M. C. Machado, G. Tesauro, and M. Campbell. The eigenoption-critic framework. *arXiv preprint arXiv:1712.04065*, 2017.

M. C. Machado, M. G. Bellemare, and M. Bowling. A laplacian framework for option discovery in reinforcement learning. In *International Conference on Machine Learning*, pages 2295–2304. PMLR, 2017.

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*, 2018.

M. C. Machado, M. G. Bellemare, and M. Bowling. Count-based exploration with the successor representation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 5125–5133, 2020.

A. McGovern and A. G. Barto. Automatic discovery of subgoals in reinforcement learning using diverse density. In *Proceedings of the Eighteenth International Conference on Machine Learning*, pages 361–368, 2001.

S. Mishra, A. Abdolmaleki, A. Guez, P. Trochim, and D. Precup. Augmenting learning using symmetry in a biologically-inspired domain. *arXiv preprint arXiv:1910.00528*, 2019.

V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

A. V. Nair, V. Pong, M. Dalal, S. Bahl, S. Lin, and S. Levine. Visual reinforcement learning with imagined goals. *Advances in Neural Information Processing Systems*, 31:9191–9200, 2018.

G. Ostrovski, M. G. Bellemare, A. Oord, and R. Munos. Count-based exploration with neural density models. In *International conference on machine learning*, pages 2721–2730. PMLR, 2017.

D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell. Curiosity-driven exploration by self-supervised prediction. In *ICML*, 2017.

E. A. Platanios, A. Saparov, and T. Mitchell. Jelly bean world: A testbed for never-ending learning. *arXiv preprint arXiv:2002.06306*, 2020.

D. Precup. Temporal abstraction in reinforcement learning. 2001.

T. Schaul, D. Horgan, K. Gregor, and D. Silver. Universal value function approximators. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pages 1312–1320. JMLR.org, 2015.

T. Schaul, H. van Hasselt, J. Modayil, M. White, A. White, P.-L. Bacon, J. Harb, S. Mourad, M. Bellemare, and D. Precup. The barbados 2018 list of open issues in continual learning, 2018. URL https://arxiv.org/abs/1811.07004.

J. Schmidhuber. A possibility for implementing curiosity and boredom in model-building neural controllers. In *Proceedings of the First International Conference on Simulation of Adaptive Behavior on From Animals to Animats*, page 222–227, Cambridge, MA, USA, 1991. MIT Press. ISBN 0262631385.

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

A. Sharma, S. Gu, S. Levine, V. Kumar, and K. Hausman. Dynamics-aware unsupervised discovery of skills. *arXiv preprint arXiv:1907.01657*, 2019.

Ö. Şimşek and A. G. Barto. Using relative novelty to identify useful temporal abstractions in reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*, page 95, 2004.

Ö. Şimşek, A. P. Wolfe, and A. G. Barto. Identifying useful subgoals in reinforcement learning by local graph partitioning. In *Proceedings of the 22nd international conference on Machine learning*, pages 816–823, 2005.

M. Stolle and D. Precup. Learning options in reinforcement learning. In *International Symposium on abstraction, reformulation, and approximation*, pages 212–223. Springer, 2002.

R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 1998.

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

R. S. Sutton, J. Modayil, M. Delp, T. Degris, P. M. Pilarski, A. White, and D. Precup. Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction. In *The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 761–768, 2011.

H. Tang, R. Houthooft, D. Foote, A. Stooke, O. Xi Chen, Y. Duan, J. Schulman, F. DeTurck, and P. Abbeel. # exploration: A study of count-based exploration for deep reinforcement learning.

Advances in neural information processing systems, 30, 2017.

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*, volume 31, 2017.

S. Tiwari and P. S. Thomas. Natural option critic. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 5175–5182, 2019.

E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. In *IROS 2012*, pages 5026–5033. IEEE, 2012.

E. van der Pol, D. Worrall, H. van Hoof, F. Oliehoek, and M. Welling. Mdp homomorphic networks: Group symmetries in reinforcement learning. *Advances in Neural Information Processing Systems*, 33, 2020.

E. van der Pol, H. van Hoof, F. A. Oliehoek, and M. Welling. Multi-agent mdp homomorphic networks. *arXiv preprint arXiv:2110.04495*, 2021.

D. Warde-Farley, T. Van de Wiele, T. Kulkarni, C. Ionescu, S. Hansen, and V. Mnih. Unsupervised control through non-parametric discriminative rewards. *arXiv preprint arXiv:1811.11359*, 2018.

A. Zhang, R. McAllister, R. Calandra, Y. Gal, and S. Levine. Learning invariant representations for reinforcement learning without reconstruction, 2020. URL https://arxiv.org/abs/2006.10742.

Y. Zhu, D. Gordon, E. Kolve, D. Fox, L. Fei-Fei, A. Gupta, R. Mottaghi, and A. Farhadi. Visual semantic planning using deep successor representations. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 483–492, 2017.