

UC Irvine

UC Irvine Electronic Theses and Dissertations

Title

Hierarchical Reinforcement Learning with Model-Based Planning for Finding Sparse Rewards

Permalink

<https://escholarship.org/uc/item/89j5c7j1>

Author

Bartley, Travis D.

Publication Date

2023

Copyright Information

This work is made available under the terms of a Creative Commons Attribution-NonCommercial-NoDerivatives License, available at <https://creativecommons.org/licenses/by-nc-nd/4.0/>

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA,
IRVINE

Hierarchical Reinforcement Learning with Model-Based
Planning for Finding Sparse Rewards

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Electrical and Computer Engineering

by

Travis D. Bartley

Dissertation Committee:
Assistant Professor Yasser Shoukry, Co-Chair
Professor Fadi Kurdahi, Co-Chair
Associate Professor Marco Levorato

2023

DEDICATION

To my wife, Nikita.

TABLE OF CONTENTS

	Page
LIST OF FIGURES	v
LIST OF TABLES	vii
LIST OF ALGORITHMS	viii
ACKNOWLEDGMENTS	ix
VITA	x
ABSTRACT OF THE DISSERTATION	xiii
1 Introduction	1
1.1 The Sparse Reward Problem	1
1.2 Thesis Outline	2
2 Background	3
2.1 Markov Decision Processes	5
2.2 Semi-Markov Decision Processes	6
2.3 Model-Free RL	8
2.4 Model-Based RL	10
2.4.1 Model-Based Planning	11
2.4.2 Monte Carlo Tree Search	13
2.5 Hierarchical Reinforcement Learning	16
3 Hierarchical Reinforcement Learning Agent for Path Planning	20
3.1 Introduction	20
3.1.1 Waterworld Random Maze Task	22
3.1.2 Gridworld Random Maze Task	23
3.2 Methods	24
3.2.1 Hierarchical Agent with Feature Crafted Goals	24
3.3 Results	27
3.4 Discussion	29

4	Learning Diverse Policies for Path Planning	30
4.1	Introduction	30
4.2	Hierarchical Agent with Learned Diverse Policies	31
4.3	Results	33
4.3.1	Deepmind Lab Random Maze Task	33
4.4	Discussion	36
5	Hierarchical Reinforcement Learning Agent for General Tasks	37
5.1	Introduction to OpenAI Gymnasium Atari Tasks	37
5.2	Hierarchical Agent for General Task Solving	39
5.2.1	PUCT	39
5.2.2	NN Architecture	40
5.2.3	Worker Policy Termination	40
5.3	Results	41
5.3.1	Experimental Procedure	41
5.3.2	Atari Games Performance	42
5.4	Discussion	42
6	Conclusion	46
6.1	Summary	46
6.2	Future Directions	47
	Bibliography	48
	Appendix A OpenAI Baselines for Atari Learning Environment	52

LIST OF FIGURES

	Page
2.1 The reinforcement learning paradigm. [1]	4
2.2 Hierarchical reinforcement learning example. [2]	5
2.3 The state trajectory of an MDP is made up of small, discrete-time transitions, whereas that of an SMDP comprises larger, continuous-time transitions. Options enable an MDP trajectory to be analyzed in either way. [3]	7
2.4 Monte Carlo tree search method. [4]	14
3.1 Waterworld environment.	23
3.2 Gridworld random maze task.	23
3.3 Hierarchical agent with goal-based planning.	24
3.4 Illustration Monte Carlo tree search planning process for maze solving.	26
3.5 Planning results and illustration of goals (red) from a random episode.	28
3.6 Training results for A3C agent	28
4.1 DIAYN Algorithm. [5]	32
4.2 DIAYN Algorithm: the discriminator is updated to better predict the skill, and the skill is updated to visit diverse states that make it more discriminable. [5]	32
4.3 DeepMind Lab environment.	34
4.4 Textual representation of 4 selected random mazes in DeepMind Lab navigation task. The goal is represented by "G" and walls represented by "*".	34
4.5 Worker policies trained according to DIAYN. Each color represents a different policy. The plot shows the trajectories of the agent over 10 mini-episodes of 100 frames for each different policy.	35
4.6 Training loss of worker DIAYN actor.	35
4.7 Training loss of worker DIAYN critic1.	35
4.8 Training loss of worker DIAYN critic2.	36
4.9 Training loss of worker DIAYN discriminator.	36
4.10 Maze solving performance of hierarchical agent, as measured in average time to solve maze per episode.	36
5.1 Performance of hierarchical agent on BeamRider.	42
5.2 Performance of hierarchical agent on Breakout.	43
5.3 Performance of hierarchical agent on Enduro.	43
5.4 Performance of hierarchical agent on Pong.	43
5.5 Performance of hierarchical agent on Qbert.	44

5.6	Performance of hierarchical agent on Seaquest.	44
5.7	Performance of hierarchical agent on SpaceInvaders.	44

LIST OF TABLES

	Page
5.1 Summary of the results of the hierarchical agent (HRL) on selected games from the Atari Learning Environment.	42

LIST OF ALGORITHMS

	Page
1 Pseudocode for manager function	25
2 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread	27
3 Pseudocode for Path Planning Hierarchical Agent.	31

ACKNOWLEDGMENTS

I would like to thank...

My advisors Professor Fadi Kurdahi and Assistant Professor Yasser Shoukry for their guidance. Professor Anima Anandkumar and Associate Professor Aparna Chandramowliswaran for advising me early on in my Ph.D. journey. Professor Emre Neftci and Professor Nikil Dutt for advising me through the bulk of my PhD work and serving on my Ph.D. candidacy committee- thank you for your mentorship and the knowledge you shared with me. Professor Jeff Krichmar for serving on my candidacy committee. Associate Professor Marco Levorato for providing helpful feedback and for serving on my candidacy committee and dissertation committee.

Professor Kazusuke Maenaka and all of the members of the ERATO Maenaka Human-Sensing Fusion Project, as well as Professor Shuji Tanaka and all of the members of the Tanaka Shuji MEMS Lab. Thank you for helping me start my research career and for all the things you taught me.

All the members and friends of the Neuromorphic Machine Intelligence (NMI) Lab for the fascinating discussions and debates: Roman Parise, Armaan Saini, Dan Barsever, Takashi Nagata, Georgios Detorakis, Andrew Hanson, Jack Kaiser, Massimiliano Iacono, Jinwei Xing, and Jordan Rashid.

All the members of the Cognitive Anteater Robotics Laboratory (CARL) for the many excellent presentations and discussions.

Amy Pham, who saved me more times than I can count in my academic journey.

My good friends Micah Jackson, Armond Murray, Lingge Li, Alexios Vulomekas, Evelia Salinas, and Cami Sifferlen.

The funding sources that made my Ph.D. studies possible: National Science Foundation (NSF), Defense Advanced Research Projects Agency (DARPA), and Intel Corporation.

My whole family, the Bartleys and Paranjapes, for their love and support, especially my parents, Terry and Jennifer Bartley.

VITA

Travis D. Bartley

EDUCATION

Doctor of Philosophy in Electrical and Computer Engineering University of California, Irvine	2023 <i>Irvine, California</i>
Master of Science in Electrical and Computer Engineering University of California, Irvine	2017 <i>Irvine, California</i>
Bachelor of Science in Electrical and Computer Engineering The Ohio State University	2010 <i>Columbus, Ohio</i>

WORK AND RESEARCH EXPERIENCE

Graduate Intern Irvine Sensors Corporation	2019 <i>Irvine, California</i>
Graduate Student Researcher University of California, Irvine	2016–2020 <i>Irvine, California</i>
Graduate Tech Intern Intel Corporation	2016–2017 <i>Toronto, Canada</i>
Researcher Tohoku University	2013–2015 <i>Sendai, Japan</i>
Research Engineer The University of Hyogo	2010–2013 <i>Himeji, Japan</i>

TEACHING EXPERIENCE

Teaching Assistant University of California, Irvine	2015–2023 <i>Irvine, California</i>
---	---

PUBLICATIONS

G. Detorakis, T. Bartley, E. Neftci, *Contrastive Hebbian Learning with Random Feedback Weights*, Neural Networks, No. 114, pp. 1–14, 2019.

G. Detorakis, T. Bartley, E. Neftci, *Random Contrastive Hebbian Learning as a Biologically Plausible Learning Scheme*, to appear in OCNS 2018, Seattle (WA, USA).

G. Detorakis, T. Bartley, R. Parise, S. Sheik, C. Augustine, S. Paul, B. U. Pedroni, N. Dutt, J. Krichmar, G. Cauwenberghs, and E. Neftci, *Three-factor Embedded Learning on Neuromorphic Systems*, COSYNE, Denver (CO, USA), 2018.

G. Detorakis, T. Bartley, R. Parise, S. Sheik, C. Augustine, S. Paul, B. Pedroni, N. Dutt, J. Krichmar, G. Cauwenberghs and E. Neftci, *Embedded Learning on Neuromorphic Systems: Towards a Unified Computing Framework*, Neuro Inspired Computational Elements (NICE) Workshop, (Portland, OR), 2018.

G. Detorakis, T. Bartley, R. Parise, C. Augustine, S. Paul, E. Neftci, *Embedded learning on neuromorphic systems: Towards a Unified Computing Framework*, IEEE International Conference on Computer Aided Design (ICCAD), Hardware and Algorithms for Learning on a Chip (HALO) Workshop, 2017.

T. Bartley, M. Elkoussy, A. Anandkumar, and A. Chandramowliswaran, “Minimizing Communication for Tensor Decompositions,” *SC16, The International Conference for High Performance Computing, Networking, Storage and Analysis*, (Salt Lake City, UT), Nov. 2016.

C. Shao, T. Nakayama, Y. Hata, T. Bartley, Y. Nonomura, S. Tanaka, and M. Muroyama, “A Multiple Sensor Platform with Dedicated CMOS-LSIs for Robot Applications,” *NEMS 2016, The 11th Annual IEEE International Conference on Nano/Micro Engineered and Molecular Systems*, (Sendai and Matsushima Bay, Japan), Apr. 2016.

S. Asano, M. Muroyama, T. Bartley, T. Nakayama, U. Yamaguchi, H. Yamada, Y. Hata, Y. Nonomura, and S. Tanaka, “3-Axis Fully-Integrated Surface-Mountable Differential Capacitive Tactile Sensor by CMOS Flip-Bonding,” *MEMS 2016, The 29th IEEE International Conference on Micro Electro Mechanical Systems*, (Shanghai, China), Jan. 2016.

T. Bartley, S. Tanaka, Y. Nonomura, T. Nakayama, Y. Hata, and M. Muroyama, “Sensor Network Serial Communication System with High Tolerance to Timing and Topology Variations,” *The IEEE Sensors Conference*, (Busan, South Korea), Nov. 2015.

S. Asano, M. Muroyama, T. Bartley, T. Kojima, T. Nakayama, U. Yamaguchi, H. Yamada, Y. Nonomura, Y. Hata, H. Funabashi, and S. Tanaka, “Flipped CMOS-diaphragm capacitive tactile sensor surface mountable on flexible and stretchable bus line,” *18th International Conference on Solid-State Sensors, Actuators and Microsystems Transducers 2015* (Anchorage, AK), pp. 97–100, June 2015.

T. Bartley, S. Tanaka, Y. Nonomura, T. Nakayama, and M. Muroyama, “Delay Window Blind Oversampling Clock and Data Recovery Algorithm with Wide Tracking Range,” *The IEEE International Symposium on Circuits and Systems* (Lisbon, Portugal), pp. 1598–1601, May 2015.

M. Muroyama, M. Makihata, S. Tanaka, T. Kojima, Y. Nakano, T. Bartley, T. Nakayama, U. Yamaguchi, H. Yamada, Y. Nonomura, Y. Hata, H. Funabashi, and M. Esashi, “Practical Application of MEMS-LSI Integration Technology: Tactile Sensor Network,” (Japanese) *InterLab* No. 110, pp. 17–23, Mar. 2014.

O. Nizhnik, K. Higuchi, K. Maenaka, and T. Bartley, “Energy-efficient, 0.1 nJ per conversion temperature sensor with time-to-digital converter and 1 deg C accuracy in -6 to 64 deg C range,” *The IEEE Sensors Conference*, (Baltimore, MD), pp. 1–5, Nov. 2013.

M. Nii, T. Tanaka, Y. Matsumoto, T. Bartley, U. Maksudi, O. Nizhnik, K. Sonoda, H. Takao, K. Maenaka, and K. Higuchi, “Heart Rate Extraction Hardware from ECG Data,” *Transactions of Japanese Society for Medical and Biological Engineering* Vol. 51, No. Supplement p. M-159, Sep. 2013.

H. Takao, K. Maenaka, K. Higuchi, O. Nizhnik, O. Vinluan, U. Maksudi, and T. Bartley, “ASIC for Monitoring of Human Motion,” *Transactions of Japanese Society for Medical and Biological Engineering* Vol. 51, No. Supplement p. M-157, Sep. 2013.

ABSTRACT OF THE DISSERTATION

Hierarchical Reinforcement Learning with Model-Based
Planning for Finding Sparse Rewards

By

Travis D. Bartley

Doctor of Philosophy in Electrical and Computer Engineering

University of California, Irvine, 2023

Assistant Professor Yasser Shoukry, Co-Chair
Professor Fadi Kurdahi, Co-Chair

Reinforcement learning (RL) has proven useful for a wide variety of important applications, including robotics, autonomous vehicles, healthcare, finance, gaming, recommendation systems, and advertising, among many others. In general, RL involves training an agent to make decisions based on a reward signal. One of the major challenges in the field is the sparse reward problem, which occurs when the agent receives rewards only occasionally during the training process. This can make conventional RL algorithms difficult to train since the agent does not receive enough feedback to learn the optimal policy. Model-based planning is one potential solution to the sparse reward problem since it enables an agent to simulate their actions and predict the outcome far into the future. However, planning can be computationally expensive or even intractable when too many time steps are required to be internally simulated, due to combinatorial explosion.

To address these challenges, this thesis presents a new RL algorithm that uses a hierarchy of model-based (manager) and model-free (worker) policies to take advantage of the unique advantages of both. The worker takes guidance from the manager in the form of a goal or selected policy. The worker is computationally efficient and can respond to changes or uncertainty in the environment to carry out its task. From the manager's perspective, this

abstracts away the trivially small state transitions, reducing the depth needed for tree search, and greatly improving the efficiency of planning.

Two different applications were used for evaluation of the hierarchical agent. The first is a maze navigation environment, with continuous-state dynamics and unique episodes. This makes the environment extremely challenging for both model-based and model-free algorithms. The performance of the agent was evaluated on multiple platforms for the random maze task, including DeepMind Lab. For the second demonstration, the proposed algorithm was compared against other algorithms with the Arcade Learning Environment, which is a popular RL benchmark. In comparison with state-of-the-art algorithms, the proposed hierarchical algorithm is shown to have a faster convergence and greater sample efficiency on several tasks. Overall, the proposed hierarchical approach is a potential solution to the sparse rewards problem, and may enable RL algorithms to be applied to a wider range of tasks, ultimately leading to better outcomes in various applications.

Chapter 1

Introduction

1.1 The Sparse Reward Problem

The sparse reward problem is a common challenge in RL where an agent is trying to learn a task, but the rewards it receives for its actions are sparse or infrequent. In some environments, the rewards are sparse, meaning that the agent only receives a reward for completing the task or reaching a certain milestone, while all other actions receive a reward of zero. This makes it difficult for the agent to learn the optimal policy because it may not receive any feedback for a long time, making it hard to distinguish between good and bad actions. This is especially problematic in complex environments where the optimal policy is not obvious. The sparse reward problem can lead to slow learning, or even the failure of the agent to learn anything useful. This is also related to the credit assignment problem, where the agent may not immediately know which of its actions led to a particular reward or penalty.

To overcome this challenge, researchers have developed a number of techniques, such as shaping the reward signal, using a proxy reward, or using intrinsic motivation to encourage exploration. These techniques can provide more frequent feedback to the agent, making it

easier to learn the optimal policy even in the presence of sparse rewards. However, these techniques may not be applicable to all problems.

1.2 Thesis Outline

This dissertation presents a new solution to the sparse reward problem by using hierarchical RL (HRL) that exploits temporal abstraction. The primary contribution of this research is a hierarchical agent which uses Monte Carlo tree search (MCTS) methods for the high-level controller (manager) and a model-free policy for the low-level controller (worker). The combination of the two elements allows the manager to plan over long time horizons, which in turn enables the agent to find rewards that may be sparse and distant. At the same time, the worker is highly efficient for acting on the environment at base level time steps, which avoids the heavy computational demands of using MCTS alone. The result is an agent that is highly capable of solving a variety of tasks, computationally efficient, and sample efficient.

Chapter 2 provides a background of the most seminal works of research in RL and HRL, including the algorithms used by the proposed agent. Chapter 3 introduces a basic version of the hierarchical agent, where the manager based on MCTS plans over a feature-engineered discrete 2D space and provides proxy rewards to the worker, which is based on A3C and acts on a continuous 2D space. Chapter 4 shows a more advanced version of the hierarchical agent which uses the “diversity is all you need” (DIAYN) method, which eliminates the need for any feature engineering. This performance of this version of the agent is demonstrated on the random maze task of the DeepMind Lab platform. Chapter 5 demonstrates the hierarchical agent successfully solving a variety of general tasks in the Atari Learning Environment and discusses the advantages and drawbacks of the approach in comparison to other state-of-the-art RL baseline algorithms. Finally, a summary of the findings and future directions of this research are presented in Chapter 6.

Chapter 2

Background

RL is a type of machine learning (ML) that involves an agent learning how to make decisions in an environment by maximizing a cumulative reward signal [1]. The agent interacts with the environment by taking actions and receiving feedback in the form of rewards or punishments. The goal of the agent is to learn the optimal policy that maps states to actions, in order to maximize the cumulative reward over time. The basic steps involved in RL are as follows:

1. Define the problem. First, the problem needs to be defined in terms of the environment, the agent, and the rewards.
2. Define the state space, action space, and reward function. The state space defines all possible states of the environment, the action space defines all possible actions that the agent can take, and the reward function defines the reward that the agent receives for each action, given the state of the environment.
3. Determine the policy. The policy is a mapping of states to actions that the agent uses to make decisions. The goal is to learn the optimal policy that maximizes the cumulative reward over time.

4. Agent interacts with environment. The agent takes actions in the environment, and the environment responds by transitioning to a new state and providing a reward signal.
5. Update the policy. The agent uses the reward signal to update its policy, in order to improve its decision-making over time.
6. Repeat. The agent continues to interact with the environment, updating its policy based on the feedback it receives, until it converges on an optimal policy.

This classical RL paradigm is formalized as in Figure 2.1. The basic structure is that the agent observes the state S_t and reward R_t , and produces some action on the environment A_t . The environment then transitions according to its internal transition model, which may be deterministic or stochastic, and produces the next state and reward for the next time step, S_{t+1} and reward R_{t+1} .

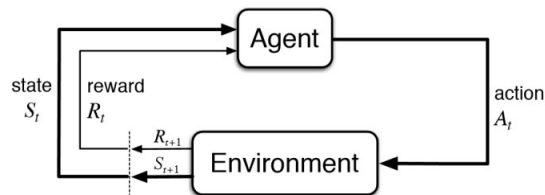


Figure 2.1: The reinforcement learning paradigm. [1]

HRL is a subfield of RL which focuses on breaking down a complex task into smaller subtasks and learning how to perform these subtasks in an hierarchical manner to achieve the overall goal. This can make the learning process more efficient and easier for RL agents, as well as provide a structure for representing and learning about the task.

As illustrated in Figure 2.2, there are typically two levels of decision making in HRL: a high-level controller that selects subtasks and a low-level controller that executes them. The high-level controller can learn to coordinate and plan long-term goals while the low-level controller focuses on learning the optimal control policy for each subtask. The high-level and low-level controllers are also referred to in this text as “manager” and “worker,” respectively.

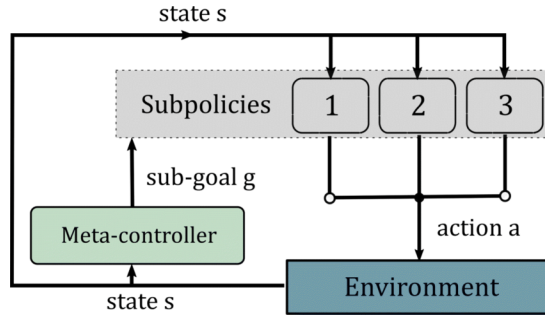


Figure 2.2: Hierarchical reinforcement learning example. [2]

RL and HRL are often used in domains where there is no known optimal solution, or where the solution is too complex to be determined by traditional methods. It has been applied to a wide range of real-world applications, including games, robotics, and autonomous driving. HRL has shown promising results in terms of faster convergence and improved performance compared to flat RL, especially in domains that benefit from using a structured approach to problem solving.

2.1 Markov Decision Processes

To further expand on Figure 2.1, a Markov decision process (MDP) is a framework used to model decision-making in situations where outcomes are partially uncertain and depend on the actions of a decision maker [1]. In an MDP, a decision maker interacts with an environment by choosing actions at each step, and the environment responds by transitioning to a new state and providing a reward or penalty to the decision maker.

The key assumption in an MDP is that the future state and reward depend only on the current state and action taken, and not on any previous states or actions. This is known as the Markov property. This property allows MDPs to be represented compactly as a state space, action space, transition function, and reward function.

More formally, an MDP can be expressed as a tuple: (S, A, T, R) , where S is the set of

states, A is the set of possible actions, T is the transition function, and R is the reward function. Upon initialization, the environment produces state s , and the agent produces action a , which results in a new state s' and reward according to the transition function T and reward function R , respectively.

The goal of solving an MDP is to find a policy, which is a mapping from states to actions, that maximizes the expected cumulative reward over time. This can be done using algorithms such as value iteration or policy iteration, which compute the optimal value function or policy for the MDP. MDPs have many applications, including in robotics, finance, and artificial intelligence.

2.2 Semi-Markov Decision Processes

A Semi-Markov Decision Process (SMDP) is an extension of the standard MDP framework, as illustrated in Figure 2.3 [3]. While in an MDP, a state is considered to be fully observable and a transition from one state to another occurs instantaneously. In an SMDP, states may have different durations, and transitions between states may take variable amounts of time. In an SMDP, the time spent in a state is a random variable, and the probability of transitioning from one state to another depends not only on the current state and action but also on the amount of time spent in the current state.

The state space in an SMDP is typically composed of pairs of (state, time) and the transitions are described by a set of transition probabilities that depend on the state, the action taken, and the amount of time spent in the current state. The reward function is typically defined as a function of the state and time, rather than just the state.

SMDPs are used to model decision-making problems in which the time spent in a state is important, such as resource allocation problems or production scheduling problems. Optimal

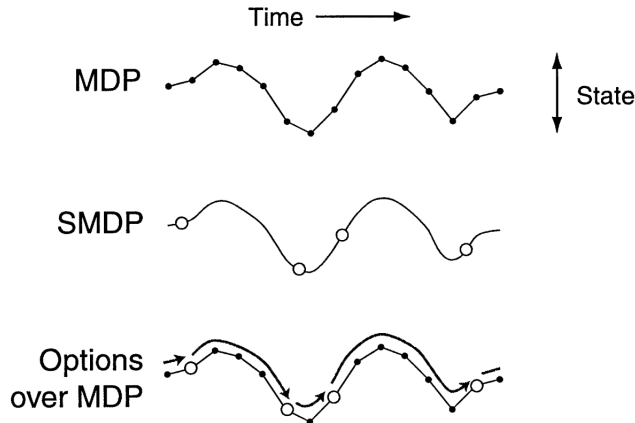


Figure 2.3: The state trajectory of an MDP is made up of small, discrete-time transitions, whereas that of an SMDP comprises larger, continuous-time transitions. Options enable an MDP trajectory to be analyzed in either way. [3]

control algorithms, such as the Semi-Markov Decision Process algorithm, can be used to find the optimal policy in an SMDP, taking into account the duration of the state and the expected reward over time.

The fact that there may be different durations between state transitions in an SMDP allows for temporal abstraction. The closely related concept of options enable an MDP trajectory to be analyzed in either coarse or fine time scales. This can be utilized in HRL to enable the high-level controller to function at a coarse timescale, whereas the low-level controller functions at a fine timescale. Using a dedicated module for the fine time horizon allows for computationally efficient behavior when seeking short-term goals. At the same time, a separate module for planning over coarse time horizon allows the agent to see the big picture and connect the dots to achieving a distant reward.

Options have been used together with MCTS using predefined symbolic goals like *GoToMovableOption* [6]. However, the drawback of this approach is that the symbolic goals are not learned autonomously must be feature engineered. Another important work is the option critic algorithm, which uses the intra-option policy gradient and termination gradient to learn options from scratch [7].

2.3 Model-Free RL

Model-Free RL allows for end-to-end navigation methods, where there is no clear separation between the exploration and exploitation phases. These methods have gained traction due to recent advancements [8, 9, 10]. The appeal of such approaches is generality. The manual extraction of input features and the manual design of map-building and path-planning algorithms can be avoided by instead formulating a reward function as a proxy for the navigation objective. The RL agent then learns intermediate map representations that enable it to maximize its reward.

End-to-end learning is a hot topic in the Deep Learning field for taking advantage of Deep Neural Network’s (DNNs) structure, composed of several layers, to solve complex problems. Similar to the human brain, each DNN layer (or group of layers) can specialize to perform intermediate tasks necessary for such problems. The appeal of End to End models is that the intermediate steps to solve a task, i.e. sensors abstraction, world model, behavior generation, etc. are automatically learned by the multi-layered model.

A recent investigation reveals significant shortcomings of A3C, which is the state-of-the-art DRL navigation algorithm [11]. Experiments show that while the algorithm is able to efficiently exploit map information when trained and tested on the same map, it is unable to do so when trained and tested on different maps. Even when tested and trained on the same map, the performance of the algorithm deteriorates when the placement of the destination is randomized. These observations indicate that A3C does not learn to solve the general problem of maze navigation, but only the much narrower task of navigating in environments it has been trained on. DRL also shows very poor ability to generalize over multiple tasks, as in Atari 2600 games [12]. It was shown that the performance of these algorithms drastically deteriorates after the positions of objects were subtly perturbed. This indicates that even state-of-the-art DRL models still rely on exploiting brittle stimulus-response associations.

While DRL agents may outperform humans on narrow tasks they have previously trained on, there is still a long way to go until human-level generalization across tasks is achieved.

There are several limitations of model-free end to end learning. A huge amount of data is necessary. In the RL context, this means many episodes need to be played out. Also, it can be difficult to understand, improve or modify such a system. It may not be clear what a certain neuron has learned or what its function is in the context of the whole network. If some structural change must be applied, such as increasing the input dimensions, the whole neural network (NN) must be replaced and trained all over again. Another limitation is that some sub-tasks may have highly efficient solutions. If modules are used to solve sub-tasks and cannot be integrated into the DNN, then the system cannot be considered end to end anymore. Finally, it may be unfeasible to validate end to end systems. The potential number of input/output/state tuples can be big enough to make validation of such systems impossible. This especially important for some applications, such as self-driving vehicles, where safety is a top priority.

Additionally, end to end may not work for some applications, as shown in [13] “We have demonstrated that end-to-end learning can be very inefficient for training NN models composed of multiple non-trivial modules. End-to-end learning can even break down entirely; in the worst case none of the modules manages to learn. In contrast, each module is able to learn if the other modules are already trained and their weights frozen. This suggests that training of complex learning machines should proceed in a structured manner, training simple modules first and independent of the rest of the network.”

Agent57

Agent57 is a state-of-the-art model-free RL agent that outperforms the standard human benchmark on all 57 Atari games [14]. To achieve this result, a NN was trained which

parameterizes a family of policies ranging from very exploratory to purely exploitative. The agent has an adaptive mechanism to choose which policy to prioritize throughout the training process. Additionally, Agent57 has a parameterization of the architecture that allows for more consistent and stable learning.

2.4 Model-Based RL

To achieve transfer learning in video game playing, Kansky *et. al.* [12] argue that 1) **object-based representations** can be used to exploit the structure of the domain, 2) a **causal model of the environment** is necessary for planning, and 3) **goal-oriented planning** enables generalization to new environments composed of familiar elements. To this end, the authors developed a Probabilistic Graphical Model (PGM) named Schema Networks, which was able to substantially generalize beyond its training experience. These findings need not be limited to PGMs, but can be readily applied create new DRL algorithms which can transfer experiences between similar tasks.

The approach proposed in this thesis is to solve the exploitation phase for general problem solving using DRL, by adopting the above three conclusions that were reached by Kansky *et. al.*. To exploit the structure of the domain, we use object-based representations from the environment state. Using environment states directly could be avoided by using a vision system for detecting and tracking entities in an image, e.g. a Capsule Network [15]. Object-based representations enable the agent to model the environment accurately using, for example, an interaction network [16]. This, in turn, could allow the agent to plan its actions according to a Monte Carlo tree search (MCTS) [4].

2.4.1 Model-Based Planning

Model-based planning is a category of RL algorithms that utilize a model of the environment to make decisions. There are two main approaches to model-based planning in RL: planning with a learned model and planning with a known model.

In planning with a learned model, the agent learns the dynamics of the environment through experience and then uses this learned model to plan ahead. This approach has the advantage of being able to handle complex and partially observable environments, but it can be computationally expensive to learn the model.

In planning with a known model, the agent has access to a pre-defined model of the environment, which is often provided by the designer of the environment. This approach is computationally efficient but requires the designer to have a good understanding of the environment.

In either case, the agent can use the model to simulate possible future states and evaluate the expected value of taking different actions. This allows the agent to choose actions that maximize expected future rewards. Overall, model-based planning can be a powerful approach to RL, particularly when the environment is complex and uncertain. However, it requires the agent to have a good model of the environment, which can be difficult to acquire in practice.

Latent Action Partition

Latent Action Partition (LaP³) is a path planning method which extends the LaMCTS [17] method to path planning. The method uses a latent representation of the search space and adaptive region partitioning to reduce dimensionality. The method works by searching over the trajectory space and recursively partitioning the space into subregions based on

trajectory reward.

The disadvantage of this approach is that dimensionality reduction is performed after a large number of states have already been generated. To perform planning efficiently, the partitioning must first be performed to clean up the search space. This is in contrast with the algorithm presented in this paper, where trivial states are abstracted away before being stored in the tree data structure. In this case, planning can be performed efficiently at any time, without any need to perform partitioning to clean up the search space.

Automatically Generating Abstractions for Planning

In general, problems can be hierarchically broken down, where lower-level tasks are solved without violating the conditions set in higher levels. The authors in [18] propose a framework for this hierarchical deconstruction of tasks. They propose assigning a level to each literal, with level 0 being the complete ground state and higher levels being abstractions. Any plan at level i can only access literals with level i or higher.

The authors also introduce the ordered monotonicity property, which specifies the conditions necessary for a successful hierarchy. The property has three conditions: operators at level $i - 1$ must preserve their order in level i , the addition of an operator at level $i - 1$ is only allowed if it achieves some precondition for an operator at level i and if that precondition has level $i - 1$, and if an operator at level $i - 1$ changes a literal with level i , that operator must exist at level i . The authors provide some sufficient conditions for the property, which can be used to organize literals in a topological order to create an abstraction hierarchy.

2.4.2 Monte Carlo Tree Search

Monte Carlo tree search (MCTS) [4] is a versatile method used for planning, and methods with world champion-level performance on discrete Markov Decision Processes (MDPs) such as chess and Go rely on MCTS.

MCTS uses a tree data structure, which is used to contain the various possible actions and their resulting states. Each node in the tree represents a particular state s , and each edge represents a particular action a from state s , which leads to a new state s' . The branching factor is the number of children at each node. In this case, the branching factor is the action space of the environment. The tree is initialized with a single node, the root, which represents the initial state of the episode.

MCTS consists of four stages: selection, expansion, rollout and backup.

Selection Starting at the root node, a tree policy based on the action values attached to the edges of the tree traverses the tree to select a leaf node.

Expansion On some iterations (depending on details of the application), the tree is expanded from the selected leaf node by adding one or more child nodes reached from the selected node via unexplored actions.

Simulation From the selected node, or from one of its newly-added child nodes (if any), simulation of a complete episode is run with actions selected by the rollout policy. The result is a Monte Carlo trial with actions selected first by the tree policy and beyond the tree by the rollout policy.

Backup The return generated by the simulated episode is backed up to update, or to initialize, the action values attached to the edges of the tree traversed by the tree policy in this iteration of MCTS. No values are saved for the states and actions visited by the rollout policy beyond the tree. Figure 2.4 illustrates this by showing a backup from the terminal state of the simulated trajectory directly to the state-action node in the tree where the rollout policy began (though in general, the entire return over the simulated trajectory is backed up to this state-action node).

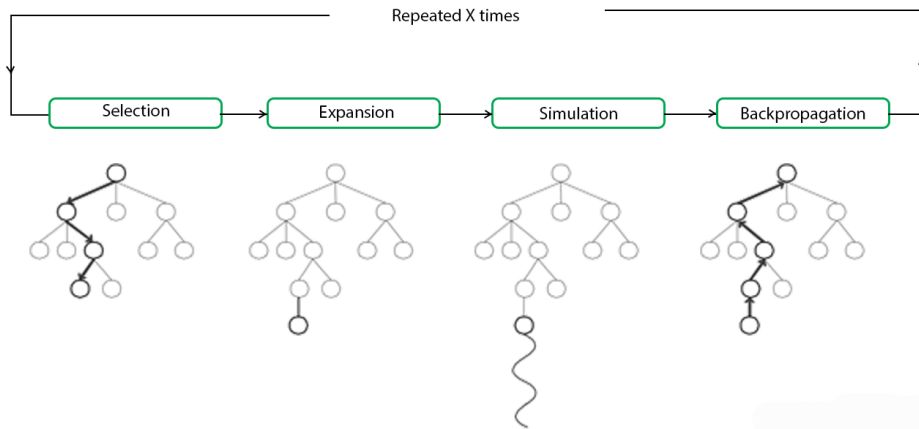


Figure 2.4: Monte Carlo tree search method. [4]

UCT Algorithm

Upper Confidence bounds applied to Trees (UCT) is an algorithm that deals with a flaw in MCTS where a sub-optimal action with only a few bad potential outcomes may be chosen instead of a better action. Regarding action selection in the framework of the multi-armed bandit problem, the tradeoff faced by the agent is exploration vs exploitation, i.e. whether the agent should seek more information about the expected payoffs or the agent should choose the action with the highest expected payoff. Balancing these two, UCT uses upper confidence bounds to select a node. Child node j is selected which maximizes the UCT evaluation:

$$UCT_j = X_j + C * \sqrt{\frac{\ln(n)}{n_j}},$$

where X_j is the ratio of the child, n is the number of times the parent has been visited, n_j is the number of times the child has been visited, and C is a constant to adjust the amount of exploration.

PUCT

The probabilistic upper confidence bound for trees (PUCT) algorithm is a popular algorithm used in MCTS [9]. The PUCT algorithm works by constructing a search tree to represent the state space of the problem. Each node in the tree represents a state, and the edges represent possible actions that can be taken from that state. The algorithm selects nodes to expand and simulate using a combination of two factors: the value estimate of the state and the exploration bonus.

The value estimate is based on the expected reward for taking a certain action from a certain state, and is updated as the algorithm explores the tree. The exploration bonus is a term that encourages the algorithm to explore new, untried actions in order to find the best solution. The exploration bonus is calculated using the upper confidence bound formula, which balances the exploration-exploitation trade-off.

As in standard MCTS, PUCT repeats the following steps until a certain stopping criterion is met: selection, expansion, simulation, and backpropagation.

The PUCT algorithm is able to efficiently find the optimal solution by balancing exploration and exploitation, and by using the Monte Carlo approach to estimate the value of states.

The AlphaGo system combined deep neural networks and MCTS to achieve state-of-the-art performance in the game of Go [19]. PUCT was one of the key algorithmic innovations that

made this possible, and the paper provides a detailed explanation of the algorithm and its implementation.

MuZero

The MuZero algorithm [20], by combining a tree-based search with a learned model, achieves superhuman performance in a range of challenging and visually complex domains, without any knowledge of their underlying dynamics. The MuZero algorithm learns an iterable model that produces predictions relevant to planning: the action-selection policy, the value function and the reward.

There is also a variant of the MuZero algorithm called MuZero Unplugged [21]. It is improved by the Reanalyse algorithm which uses model-based policy and value improvement operators to compute new improved training targets on existing data points, allowing efficient learning entirely from demonstrations without any environment interactions, as in the case of offline RL.

2.5 Hierarchical Reinforcement Learning

The benefits of HRL over standard RL are the following [22]:

- **Faster learning of complex tasks:** HRL can help speed up the learning process of complex tasks by breaking them down into smaller sub-tasks, making the learning process more efficient.
- **Improved exploration:** HRL can help an agent explore its environment more effectively by guiding it to focus on the most promising areas of the environment.

- Generalization to new tasks: HRL can learn a hierarchical policy that can be generalized to new, unseen tasks more easily than a flat RL agent, which has to learn a new policy from scratch.
- Robustness to environmental changes: HRL can adapt to changes in the environment more effectively by reusing learned sub-tasks and modifying them as necessary, instead of starting the learning process from scratch.
- Increased interpretability: HRL can provide more insight into the decision-making process of the agent by decomposing it into smaller sub-tasks, which can help make the policy more interpretable and explainable.
- Reduced dimensionality: HRL can help reduce the dimensionality of the state and action space, making it easier to learn and optimize the policy.
- Facilitation of transfer learning: HRL can facilitate transfer learning, where an agent can use knowledge learned from one task to improve its performance on another related task.

Standard RL planning suffers from the curse of dimensionality when the action space is too large or when the state space is unfeasible to enumerate. Humans simplify the problem of planning in such complex conditions by abstracting away details which are not relevant at a given time and decomposing actions into hierarchies. Several researchers have proposed to model the temporal-abstraction in RL by composing some form of hierarchy over actions space [23, 24, 25]. By modeling actions as hierarchies, researchers extended the primitive action space by adding abstract actions. Options framework [23], refer the abstract actions as options, MAXQ [23] refer to them as tasks and Hierarchical Abstract Machines (HAM) [25] refers to them as choices.

The common theme among these papers is to extend the Markov Decision Process (MDP) to a Semi-Markov Decision Process (SMDP), where actions can take multiple time steps. As

compared to MDP, which only allow actions of a discrete time-steps, SMDP allows modeling temporally abstract actions of varying length over a continuous time. As represented in first two trajectories of figure below. By extending the action space of the MDP over primitive and abstract actions, hierarchical RL approaches superimpose MDPs and SMDPs.

HRL is appealing because the abstraction of actions facilitate accelerated-learning and generalization while exploiting the structure of the domain. Faster learning is possible because of the more compact representation. The original MDP is broken into sub-MDP with fewer states and fewer actions. The abstraction of states hides irrelevant details and hence reduces the number of states. For example, in the Taxi Domain introduced in [23], if the agent is learning to navigate to a location it does not matter if the passenger is being picked or dropped. Details about location of passenger are irrelevant and hence the state space is reduced.

Better generalization is possible because of the abstracted actions. In the taxi domain, because an abstract action is defined, called *NavigationNavigation*, agent learns a policy to navigate the taxi to a location. Once that policy is learned for navigation to pick up a passenger, the same policy can be leveraged when then agent is navigating to drop the passenger.

Two important promises of HRL are prior-knowledge and transfer-learning. A complex task in HRL is decomposed into hierarchy (usually by humans). Hence, it is easier for humans to provide some prior on actions from their domain knowledge. Different levels of hierarchy encompass different knowledge and hence ideally it would be easier to transfer that knowledge across different problems.

One minor limitation of HRL is that all the hierarchical methods converge to hierarchically optimal policy, which can be a sub-optimal policy. For example in the taxi domain, if the hierarchy decomposition states first navigate to the passenger location and then navigate to

the fuel location, the HRL agent will find an optimal policy to do that in exactly that order. This policy might be sub-optimal given an initial state which is closer to the fuel location. This limitation is an artifact of restricting the action space while solving sub-MDPs. If the full action space is available in all the MDPs, the exponential increase in computational overhead makes the learning infeasible.

Max-Q Learning

The Max-Q framework has a clear hierarchical decomposition of tasks, while the options framework do not have clear hierarchy. Options framework achieves temporal abstraction of actions, Max-Q framework additionally also achieves state abstractions. While there has been an attempt on discovering and transferring the Max-Q hierarchies [26], learning Max-Q hierarchies directly from the trajectories is still an open problem. For large and complex problem it might be a challenge to provide the task hierarchy or options and their termination conditions.

Chapter 3

Hierarchical Reinforcement Learning Agent for Path Planning

3.1 Introduction

Path planning is an important optimization tool not just for reinforcement learning, but many other applications such as biology [27], chemistry [28] and robotics [29]. In general, the objective is to find the best trajectory $\mathbf{x} = (s_0, a_0, s_1, a_1, \dots, s_n)$ in the search space $\Omega : \mathbf{x}^* = \arg \max_{x \in \Omega} f(\mathbf{x})$, where $f(\mathbf{x})$ is the reward.

The ability to plan is vital for solving tasks where rewards are sparse and may require several sequential actions. For a reinforcement agent to solve such tasks, it is useful to have an internal model of the environment, so that the agent can simulate their actions and predict the outcome, allowing them to plan action sequences that maximize total reward farther into the future. Model-based planning provides two main benefits: 1) it enables the agent to find sparse rewards more consistently, and 2) it enables generalization to unfamiliar scenarios [12]. On the other hand, planning can require many time steps to be simulated,

making it computationally expensive.

In environments with continuous state space, real-valued positions and short time steps, the branching factor is large, and the length of the sequence \mathbf{x}^* can be huge if rewards are sparse and distant from the agent. The MCTS method will need to be modified and enhanced in order to navigate in more challenging environments.

The sparse reward problem has been a problem for RL agents since it is very difficult for conventional agents to find rewards that require long sequences of actions. This is because in training it is difficult to assign credit to actions that are temporally distant from the reward and yet we ultimately necessary to reach the reward. This is known as the credit assignment problem.

The general maze navigation problem can be thought of as a sequential multi-task problem. In this case, each task is a different random maze composed of a fixed set of entities (walls, agent, destination). After learning the principles of navigation by training on some set of mazes, the agent should be able to transfer its knowledge to solve new unseen mazes efficiently. Therefore, some form of transfer learning between tasks [30] is required for a general solution to the random maze navigation problem.

To make MCTS-based planning tractable for navigating environments with a state space that is unfeasible to enumerate, such as a 2D or 3D space with real-valued positions, we first note that many of these states may have only minor differences. In an environment with short time steps, the vast majority of states are just empty space, and the distance between s_0 and s_1 after just a single action a_0 is tiny. It would be unnecessary and inefficient to plan at the level of single time steps, as the tree data structure would become astronomical in size. Instead, we wish for MCTS to plan only over the most interesting states by fast forwarding through many of these uninteresting states.

To accomplish this we propose to use options [3] for temporal abstraction. Using this frame-

work, we construct a hierarchical agent with separate components for the planning function (manager) and the acting function (worker). The worker takes semantic goals from the manager in the form of a goal or selected option. The worker is efficient at acting on single time steps and can react to small changes in the environment state to carry out its task. This frees up the manager from the minutiae of many small state transitions in the environment, allowing it to plan more efficiently over extended time periods.

This approach is inspired by Figure 2.3 [3], where options enable a Markov Decision Process to be analyzed on both coarse and fine time horizons. Using a dedicated module for the fine time horizon allows for computationally efficient behavior when seeking short-term goals. At the same time, a separate module for planning over coarse time horizon allows the agent to see the big picture and connect the dots to achieving a distant reward.

3.1.1 Waterworld Random Maze Task

For evaluating the hierarchical learning algorithm, we use a 2D random maze environment as shown in Figure 3.1. In each episode, a new maze appears, which requires the agent to generalize to solving any maze, rather than overfitting to solving a particular maze. The player (blue dot) must navigate from a random position in a random maze to the target (green dot) before time runs out. The environment has complex continuous-state dynamics, including inelastic collisions with walls which can be utilized by a skill player to conserve velocity through turns. There are four possible actions, (up, down, left, right), which apply thrust to the player sprite in the corresponding direction.

The mazes are generated according to a pseudorandom algorithm [31]. In total, the pre-generated pool consists of 2,342 mazes, each 9x9 tiles in size with 31 wall tiles. The reason for using a fixed number of wall tiles was so that the observation vector is of constant size. Using a fixed-size observation vector is convenient and simplifies the de-

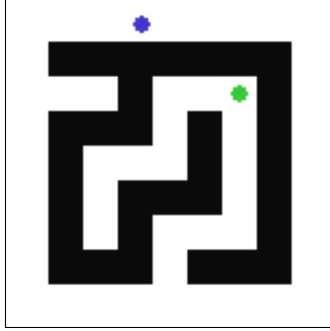


Figure 3.1: Waterworld environment.

sign of the agent. The observation vector consists of the (x, y) coordinates of each entity:

$$(x_{\text{player}}, y_{\text{player}}, x_{\text{target}}, y_{\text{target}}, x_{\text{wall0}}, y_{\text{wall0}}, \dots, x_{\text{wall30}}, y_{\text{wall30}}).$$

3.1.2 Gridworld Random Maze Task

To facilitate planning, a simplified navigation environment called Gridworld is provided to the manager. The environment consists of 3 elements, as shown in Figure 3.2: walls (black), target (green) and player (blue). At each step in the internally-simulated environment, the manager may take one of the following actions: move up, move down, move left, move right, none.

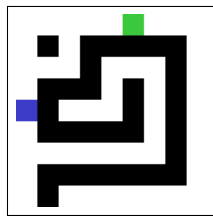


Figure 3.2: Gridworld random maze task.

The GridWorld environment makes the 2D path planning problem very easy by making the number of possible states small (with a discrete grid of only 9×9 tiles), and therefore the maximum length of \mathbf{x}^* was small. This makes it tractable to use MCTS without any enhancements or modifications.

In the actual Waterworld environment, navigation is more challenging than the GridWorld environment. In environments with continuous spaces, real-valued positions and short time steps, the branching factor becomes much larger, and the length of the sequence \mathbf{x}^* can be huge if rewards are sparse and distant from the agent. The MCTS agent from the previous chapter will need to be modified and enhanced in order to navigate in more challenging environments.

3.2 Methods

3.2.1 Hierarchical Agent with Feature Crafted Goals

As an initial foray into solving the continuous-space random maze task using a hierarchical RL agent, we use some feature crafting (Gridworld) to create a proof of concept agent in this section. The section after this will describe how to learn worker policies to avoid this feature crafting.

The hierarchy consists of a manager and a worker, which work together by solving certain functions necessary for solving the overall environment, as shown in Figure 3.3.

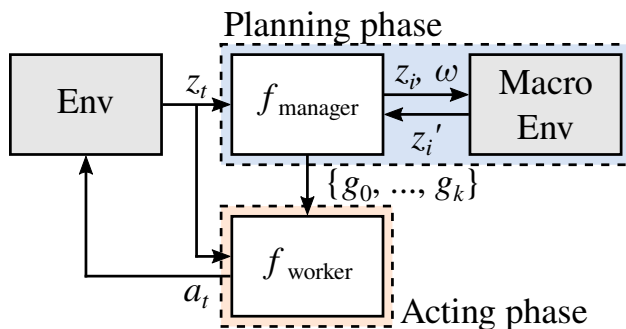


Figure 3.3: Hierarchical agent with goal-based planning.

The manager has access to the state of the environment and a discrete model of the environment (denoted macro env). The model of the environment was feature crafted to allow

MCTS to be performed over tiles of the environment (denoted z_i) instead of raw environment states (denoted z_t). This dramatically reduces the branching factor and therefore the search space for MCTS. MCTS takes the initial state of the episode, and searches paths in macro env tile by tile until the target tile is reached. The overall plan then produced by f_{manager} is denoted as G , which is defined as the sequence of subgoals, $G = g_0, g_1, \dots, g_n$. The detailed algorithm for the hierarchical agent is presented in Algorithm 1.

Algorithm 1 Pseudocode for manager function

```

// Assume Euclidean distance between two latent environment states  $d(x, y)$ 
// Assume do_planning initialized to TRUE before first execution
 $r_{wt} \leftarrow r_{\text{tick}}$ 
if do_planning then
    do_planning  $\leftarrow$  FALSE
     $\mathbf{G} \leftarrow \text{tree\_search}(s_t)$  // Perform MCTS to produce goal matrix
end if
if  $d(\mathbf{g}_0, s_t) < \text{goal\_met\_th}$  then
     $r_{wt} \leftarrow r_{wt} + r_{\text{goal\_met}}$  // Reward for reaching goal
end if
if  $d(\mathbf{g}_0, s_t) > \text{goal\_div\_th}$  then
    do_planning  $\leftarrow$  TRUE
     $r_{wt} \leftarrow r_{wt} + r_{\text{goal\_div}}$  // Punishment for diverging from goal
end if

```

We extend the option-critic architecture by using MCTS to plan over options, rather than using the output of the manager policy π_{Ω} directly. In this case, the edges of the tree correspond to options in a SMDP, rather than actions in a MDP. In the expansion phase of MCTS, a candidate option is carried out with worker policy $\pi_{\omega}(a | s)$ acting on a simulated environment until the goal is reached. The tree is then expanded with a new node representing the state resulting from that option. There is no training necessary for the manager, as it simply enumerates new child nodes until the goal is found. Since the space is discrete, it can easily enumerate over all tiles of the maze if necessary.

An illustration of the planning process is shown in Figure 3.4. The tree is shown on the left with edges representing actions and vertices representing the resulting states from macro env. The shortest path to the goal is shown in the branch to the right. The right side of the

figure shows the path in the maze that corresponds to the tree. The arrows are color coded to options. For example, blue represents the move left goal, and purple represents the move up goal.

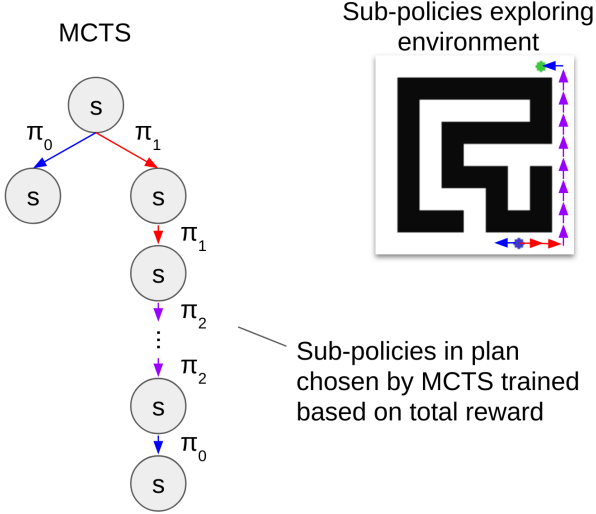


Figure 3.4: Illustration Monte Carlo tree search planning process for maze solving.

There are a few tuneable hyperparameters for this hierarchical agent. The threshold within which a goal is considered met is named `g_met_th`, which was set to the diameter of the goal, which is half the tile width. The threshold above which the worker policy has diverged from the goal is named `g_div_th`, which signals that planning must be done again to give the worker a more feasible (nearby) goal. `g_div_th` was set to twice the tile width for all experiments.

The plan from the manager is then fed as input to the worker, denoted as f_{worker} . The worker is given a sequence of the next 3 goal positions, along with the environment state. The worker is trained to reach the intermediate goals, one at a time. When the agent comes close to the next goal a positive reward is given, denoted as r_{goal_met} , and set to 1 in the experiments. When the worker goes beyond a certain threshold distance to the goal, a negative reward value is given, denoted as r_{goal_div} , and set to 1 in the experiments. and the planning process is restarted to provide new goals that are more easily reached. The worker

policies are trained using the A3C algorithm in Figure 2 [8].

Algorithm 2 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread

```

// Assume global shared parameter vectors  $\theta$  and  $\theta_v$  and global shared counter  $T = 0$ 
// Assume thread-specific parameter vectors  $\theta'$  and  $\theta'_v$ 
Initialize thread step counter  $t \leftarrow 1$ 
while  $T \leq T_{\max}$  do
  Reset gradients:  $d\theta \leftarrow 0$  and  $d\theta_v \leftarrow 0$ 
  Synchronize thread-specific parameters  $\theta' = \theta$  and  $\theta'_v = \theta_v$ 
   $t_{\text{start}} = t$ 
  Get state  $s_t$ 
  while not terminal  $s_t$  and  $t - t_{\text{start}} < t_{\max}$  do
    Perform  $a_t$  according to policy  $\pi(a_t|s_t; \theta')$ 
    Receive reward  $r_t$  and new state  $s_{t+1}$ 
     $t \leftarrow t + 1$ 
     $T \leftarrow T + 1$ 
  end while
   $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \end{cases}$  // Bootstrap from last state
  for  $i = t - 1, \dots, t_{\text{start}}$  do
     $R \leftarrow r_i + \gamma R$ 
    Accumulate gradients wrt  $\theta' : d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i; \theta')(R - V(s_i; \theta'_v))$ 
    Accumulate gradients wrt  $\theta'_v : d\theta_v \leftarrow d\theta_v + \partial(R - V(s_i; \theta'_v))^2 / \partial \theta'_v$ 
  end for
  Perform asynchronous update of  $\theta$  using  $d\theta$  and of  $\theta_v$  using  $d\theta_v$ 
end while

```

3.3 Results

The goal-based hierarchical agent was able to solve the continuous-space random maze task. An illustration of the how the overall agent performs is shown in Figure 3.5. The total sequence of goals G provided by the manager is shown as red dots. The worker is then trained to reach the tile goals one at a time in order until the the target is reached.

The results of the feature-engineered hierarchical agent are shown in Figure 3.6. For comparison with a model-free agent, we also show the results of an A3C agent. The A3C agent

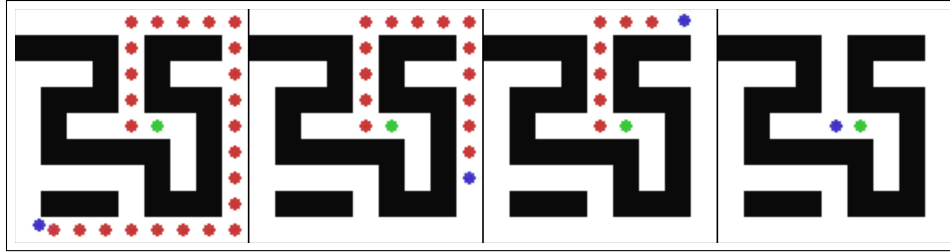


Figure 3.5: Planning results and illustration of goals (red) from a random episode.

is able to learn some simple sweeping and searching behaviors, but is unable to solve the random mazes consistently. This is because each episode is randomized, and policies that were learned on the past maze are not useful for solving the next maze. In other words, the ideal stimulus-response associations are unique to each maze. Without a model of the environment and the ability to plan, the agent is unable find a general solution.

In comparison, the hierarchical agent quickly reaches a high performance in solving the mazes consistently in a small number of steps. As the worker learns to reach intermediate goals, the performance of the agent quickly improves.

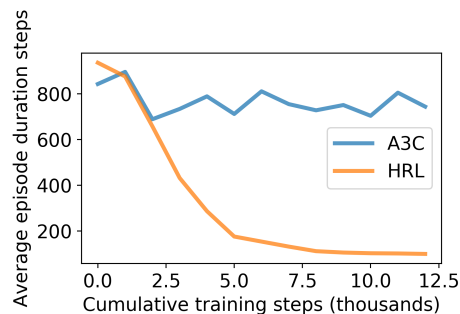


Figure 3.6: Training results for A3C agent

The results show that the HRL approach can learn to solve the random maze task in more efficiently than a model-free reinforcement learning algorithm. This is because the HRL approach is able to generalize to new mazes and can adapt to changes in the environment. These results suggest that the use of a hierarchical reinforcement learning approach can improve the performance of reinforcement learning algorithms on tasks with complex dynamics and long time horizons.

3.4 Discussion

The goals-based hierarchical agent was able to solve the Waterworld environment well. However some human feature-engineering was required to get the agent to work successfully. Ideally, we would like the agent to learn discretization strategies itself, rather than be provided with a suitable discrete model. This is the topic of the next chapter.

Chapter 4

Learning Diverse Policies for Path Planning

4.1 Introduction

To avoid human feature engineering as in the previous chapter, it would be better to have the worker automatically learn policies that are suitable for exploring the environment. This can provide the manager with suitably discrete space to search, without providing a specialized model of the environment. To achieve this, we use “diversity is all you need” (DIAYN) to train a set of skills for the agent [5].

The path planning problem involves discovering high reward trajectories, which requires optimizing a high-dimensional reward function. MCTS algorithms can be highly efficient at this [17].

We propose a new path planning method that improves MCTS by making the sampling function more efficient. We use options [3] over diverse policies [5] to provide dimensionality

reduction in the MCTS sampling space via temporal abstraction. Together in a hierarchical agent, the MCTS-based manager plans over the different states resulting from the diverse worker policies. The worker policies automatically learn behaviors which are distinct from each other and search the sampling space in different directions. Since the DIAYN policies are temporally extended sequences, and have diverse behaviors, they lead to more interesting and meaningful samples for MCTS. This makes exploration more efficient and greatly extends the distance over which path planning can be performed by MCTS, allowing the agent to find sparse rewards that may be very far away.

4.2 Hierarchical Agent with Learned Diverse Policies

The method for the hierarchical agent using DIAYN together with MCTS is shown in Algorithm 3.

Algorithm 3 Pseudocode for Path Planning Hierarchical Agent.

Input: number of rounds T , Environment Oracle: $f(\mathbf{x})$, Dataset \mathcal{D} , DIAYN Sampling Model $h(\mathbf{x})$
Parameters: Number of worker policies N_{policies} , DIAYN re-training interval N_{retrain} , $N_{\text{worker_samples}}$, UCB parameter C_p
Pretrain $h(\cdot)$ on \mathcal{D} when $\mathcal{D} \neq \emptyset$
Draw N_{init} samples uniformly from $\mathcal{S}_0 = \{(\mathbf{x}_i, f(\mathbf{x}_i))\}_{i=1}^{N_{\text{init}}} \subset \Omega$
for $t = 0, \dots, T - N_{\text{init}} - 1$ **do**
 if t divides N_{retrain} **then**
 Train the sampling model $h(\cdot)$ using samples $\mathcal{S} \cup \mathcal{D}$
 end if
 // Perform UCB MCTS using samples from $h(\cdot)$
end for

First, the worker policies can be pretrained according to DIAYN as in Figure 4.1. Samples from the environment were recorded into a dataset \mathcal{D} . Pretraining is not strictly required for the agent to successfully learn to solve the task, but it is recommended since untrained worker policies are random and not diverse which will cause MCTS to waste computational resources.

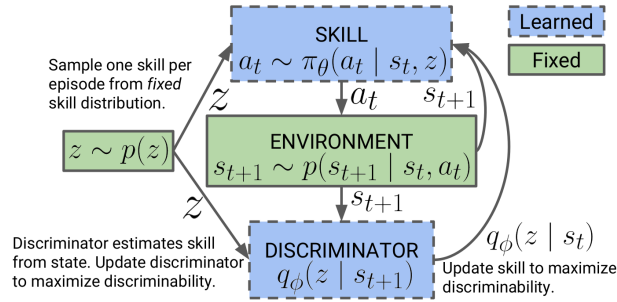


Figure 4.1: DIAYN Algorithm. [5]

```

while not converged do
  Sample skill  $z \sim p(z)$  and initial state  $s_0 \sim p_0(s)$ 
  for  $t \leftarrow 1$  to  $steps\_per\_episode$  do
    Sample action  $a_t \sim \pi_\theta(a_t | s_t, z)$  from skill.
    Step environment:  $s_{t+1} \sim p(s_{t+1} | s_t, a_t)$ .
    Compute  $q_\phi(z | s_{t+1})$  with discriminator.
    Set skill reward  $r_t = \log q_\phi(z | s_{t+1}) - \log p(z)$ 
    Update policy ( $\theta$ ) to maximize  $r_t$  with SAC.
    Update discriminator ( $\phi$ ) with SGD.

```

Figure 4.2: DIAYN Algorithm: the discriminator is updated to better predict the skill, and the skill is updated to visit diverse states that make it more discriminable. [5]

MCTS then proceeds as usual, with the exception that the expansion step is carried out by worker policies, as selected by the manager.

There are a few tuneable parameters for this method. First is $N_{policies}$. This is the number of worker policies, which will search the environment in different directions. For 2D navigation, 4 is enough to cover the cardinal directions and provides enough diverse policies to explore the space efficiently. More policies can provide more granularity, or provide more behaviors which may be required for a more rich environment. In the implementation of the worker policies, the first two layers are shared by all policies. The output of these first layers are then fed to the individual policy heads.

The $N_{retrain}$ parameter determines how many samples should lapse before the worker policies are retrained on the new data. If the initial dataset provides enough episodes, it may not be necessary to fine tune the worker policies. If the environment is non-stationary, periodically

retraining the worker may make it more efficient.

The number of rollout samples is determined by $N_{\text{worker_samples}}$. This sets the amount of temporal abstraction (and dimensionality reduction) provided by the worker policies. If this number is small, MCTS planning will occur more often, and the tree will be loaded with more nodes. If this number is large, the worker policies will act for a long time without the direction of the manager, which may result in wandering. For the experiments in this paper, this number was set to 100 samples.

4.3 Results

To demonstrate the efficiency of the hierarchical navigation agent, we use a random maze task from Deepmind Lab. The environment has some movement dynamics and short time steps which make rewards very sparse, and requires the agent to make a very long action sequences before reaching the goal.

4.3.1 Deepmind Lab Random Maze Task

DeepMind Lab [32] is a 3D learning environment which provides a suite of challenging 3D navigation and puzzle-solving tasks for learning agents. Its primary purpose is to act as a tested for research in AI, especially DRL. The DeepMind Lab environment also has some basic physics such as acceleration and momentum.

Similar to the Waterworld random maze task, DeepMind lab contains a random maze task located in the repository at

`game_scripts/levels/demos/random_maze.lua`

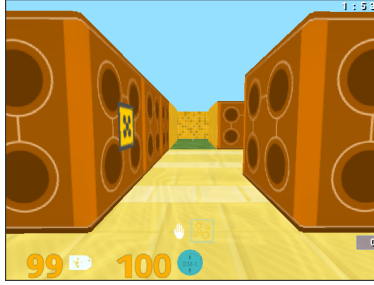


Figure 4.3: DeepMind Lab environment.

Mazes were randomized for each episode in the testing and training in the experiments. A few selected random mazes are shown below in Figure 4.4. As before, the mazes are size 9×9 with continuous valued positions within each tile. The mazes are large enough to require thousands of time steps to traverse.



Figure 4.4: Textual representation of 4 selected random mazes in DeepMind Lab navigation task. The goal is represented by "G" and walls represented by "*".

The complex dynamics together with the reasoning and planning required to solve large mazes make the environment challenging for both model-free and model-based reinforcement algorithms. Model-free agents perform well when presented with the same maze repeatedly. However, they learn a brittle policy that does not perform well when presented with a different maze. Model-free agents are unable to extract the principles of how the environment works so that they can generalize to new instances. On the other hand, the continuous state space is challenging for model-based planning agents, as planning would have to be performed over very long action sequences.

The worker policies were trained on dataset \mathcal{D} , and their learned behaviors are illustrated in Figure 4.5. The policies learn to search the space in different directions.



Figure 4.5: Worker policies trained according to DIAYN. Each color represents a different policy. The plot shows the trajectories of the agent over 10 mini-episodes of 100 frames for each different policy.

The training loss for the DIAYN worker is shown below in Figures 4.6–4.9.

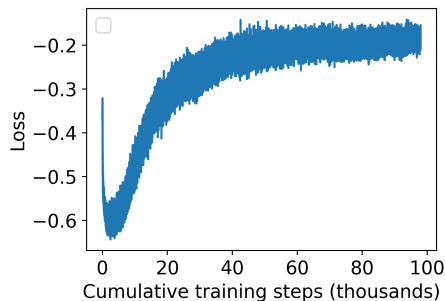


Figure 4.6: Training loss of worker DIAYN actor.

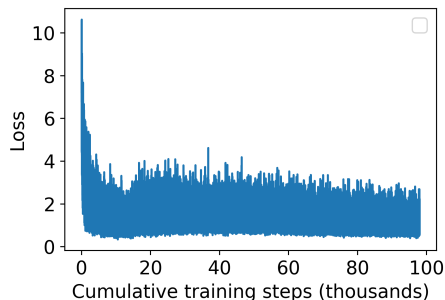


Figure 4.7: Training loss of worker DIAYN critic1.

The average time to solve a maze is shown in Figure 4.10. The plot shows the agent is able to find the goal reliably after about 40k training steps. This is around the same time the DIAYN worker training loss reaches a minimum. After training, the MCTS agent was able to solve all random mazes on the first trial.

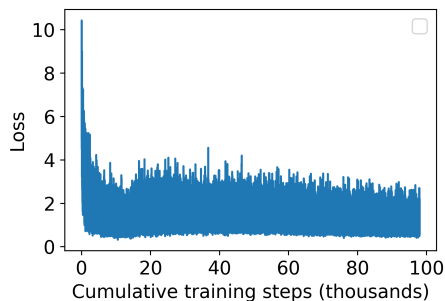


Figure 4.8: Training loss of worker DIAYN critic2.

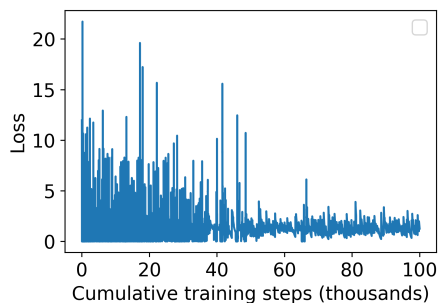


Figure 4.9: Training loss of worker DIAYN discriminator.

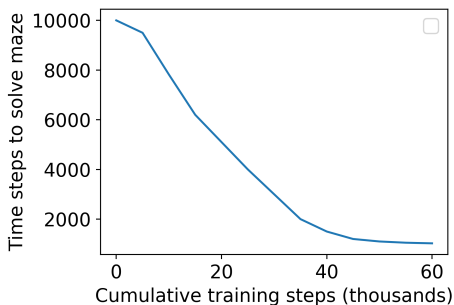


Figure 4.10: Maze solving performance of hierarchical agent, as measured in average time to solve maze per episode.

4.4 Discussion

This chapter presented a hierarchical reinforcement learning algorithm that uses a manager-worker architecture. The manager uses MCTS to select actions and the worker policies are trained using DIAYN. The worker policies are then used by the manager during the expansion step of MCTS to search the environment in different directions.

Chapter 5

Hierarchical Reinforcement Learning Agent for General Tasks

5.1 Introduction to OpenAI Gymnasium Atari Tasks

For evaluating the agent, Atari environments are simulated via the Arcade Learning Environment [33]. This popular benchmark provides a diverse set of reference environments, each with unique agent-environment interactions and rules. Contained in the set is 57 Atari 2600 environments simulated through Stella and the Arcade Learning Environment. There is a broad range of complexity for agents to learn. This set is part of the Gymnasium (formerly Gym) open source Python library for benchmarking reinforcement learning algorithms.

Observation Space The observation issued by an Atari environment may be: the RGB image that is displayed to a human player, a grayscale version of that image or the state of the 128 Bytes of RAM of the console. For all experiments below, the state of the 128 Bytes of RAM is used. Although this is not the same exact challenge a human would face, it is

possible to use a vision model that can give a sufficient approximation of the RAM state. The scope of this paper is the development and evaluation of the RL policy, and so the vision model is outside the scope.

Rewards The exact reward dynamics depend on the environment and are usually documented in the respective game’s manual. Each game has a different range of possible rewards.

Stochasticity Atari games use a pseudorandom number generator to produce variability in gameplay. Given a fixed random seed, Atari games are entirely deterministic [33]. Therefore, in an unmodified Atari environment, agents could achieve optimal performance by memorizing an optimal sequence of actions while completely ignoring observations from the environment. To avoid this, ALE implements some elements to introduce stochasticity [34]. One such element is sticky actions, where instead of always simulating the action passed to the environment, there is a small probability that the previously executed action is used instead.

Another element that produces stochasticity in the gameplay is stochastic frame skipping. In each environment step, the action is repeated for a random number of frames. This behavior may be altered by setting the keyword argument `frameskip`. If `frameskip` is an integer, frame skipping is deterministic, and in each step the action is repeated `frameskip` many times. If `frameskip` is a tuple, the number of skipped frames is chosen uniformly at random between `frameskip[0]` (inclusive) and `frameskip[1]` (exclusive) in each environment step.

Different games in the ALE suite may have sticky actions, frame skipping, neither or both, depending on their implementation. Each time an episode is presented, it may be different than any previous episodes. This variability creates challenge for the RL agent to solve the task. Agents must be able to generalize to perform well on ALE tasks.

5.2 Hierarchical Agent for General Task Solving

A number of improvements and modifications were made to the hierarchical agent in the previous chapters to make it more adapted for solving the Atari environments. The first is that the PUCT variant of MCTS is used. The second alteration is to the NN architecture to make it suitable for the RAM observation states. Finally, hyperparameters such as the worker policy termination were tuned based on the performance in the various Atari tasks.

5.2.1 PUCT

The manager component of the HRL agent is once again implemented using a Monte Carlo tree search algorithm, which uses random simulations to evaluate the potential value of different actions. This algorithm is used to learn and execute high-level plans based on the output of the DIAYN worker.

The PUCT variant of the MCTS algorithm was used [9], as described in Section 2.4.2. As in standard MCTS, the algorithm proceeds with the four stages as outlined in 2.4.2: Selection, Expansion, Simulation and Backup. The key difference with PUCT is that in each time step in the Selection stage, an action is selected according to the statistics in the search tree, $a_t = \arg \max_a (Q(s_t, a) + u(s_t, a))$ using a variant of the PUCT algorithm: $u(s, a) = c_{\text{puct}} P(s, a) \frac{\sqrt{\sum_b N_r(s, b)}}{1 + N_r(s, a)}$, where c_{puct} is a constant determining the level of exploration. This search control strategy initially prefers actions with a high prior probability and low visit count (dominated by left term), but asymptotically prefers actions with high reward value (dominated by right term).

5.2.2 NN Architecture

The NN architecture of the HRL agent based on the previous chapters has been implemented for solving the ALE suite in the following ways:

The DIAYN worker consists of a deep neural network with 3 hidden layers, each containing 128 neurons. The input layer has 128 neurons, one for each byte of RAM observation. The output layer has the same number of neurons as the number of possible actions in the game.

The activation function for the hidden layers is a rectified linear unit (ReLU), which is generally fast to calculate and has been found to work well in many RL tasks. The output layer uses a softmax activation function, which allows the outputs to be interpreted as probabilities. The layers in the DIAYN worker are fully connected.

Overall, this implementation of the HRL agent is a deep neural network with multiple layers that are fully connected, using ReLU and softmax activation functions, and optimized using Adam. The DIAYN worker provides sub-plans to the MCTS component, which would be used to learn and execute high-level plans. The parameters of the model, such as the weights and biases of the neurons, are optimized using Adam.

5.2.3 Worker Policy Termination

In the HRL algorithm described above, it is necessary to determine when to terminate the DIAYN worker, i.e. how many timesteps it should run, denoted by $N_{\text{worker_samples}}$. As before, the approach taken for this problem is to use a predefined maximum number of timesteps for the DIAYN worker, and to terminate it once this number of timesteps has been reached. This maximum number of timesteps is determined through experimentation and testing, by trying different values and observing the effect on the agent’s performance. Each game may have a different optimal value, balancing computational efficiency with overall performance.

5.3 Results

5.3.1 Experimental Procedure

The games selected for comparison were the ones with results available from the baselines repository, as shown in Appendix A: BeamRider, Breakout, Enduro, Pong, Seaquest, and SpaceInvaders. These games were selected to be diverse and challenging, in order to provide a good test of the agent’s learning abilities.

The agent was trained on the selected games by providing it with the state of the 128 bytes of RAM of the console as the input observation. This allows the agent to learn the dynamics of the game and develop strategies for playing it. During training, the agent was evaluated periodically to measure its performance on the games. This was done by running the agent on the games and measuring its score and other relevant metrics and saving them to a log file. This evaluation process occurred at an interval of every 1,000 training samples. The maximum score attained during training was used to measure its performance for comparison with other RL algorithms, to evaluate the effectiveness of the hierarchical algorithm.

Hyperparameter Tuning Grid search was used to tune the hyperparameters of the algorithm, such as the number of worker policies and the duration of the worker rollout. In grid search, a range of values for each hyperparameter was specified, and the algorithm is trained and evaluated for each combination of hyperparameter values within the specified range. The combination of hyperparameter values that yields the best performance on the games was then selected as the optimal set of hyperparameters, as shown in the results below.

Game	HRL High Score	N_{policies}	$N_{\text{worker_samples}}$
BeamRider	6823	3	20
Breakout	464	3	30
Enduro	708	4	20
SpaceInvaders	1343	4	30
Qbert	19839	5	10
Seaquest	1904	5	10
Pong	20	3	10

Table 5.1: Summary of the results of the hierarchical agent (HRL) on selected games from the Atari Learning Environment.

5.3.2 Atari Games Performance

Table 5.1 shows the performance of the hierarchical agent on several Atari games. Each game was run for 10M time steps. The table also shows the optimal hyperparameters found by grid search: the number of worker policies, N_{policies} , and the duration of worker activity, $N_{\text{worker_samples}}$.

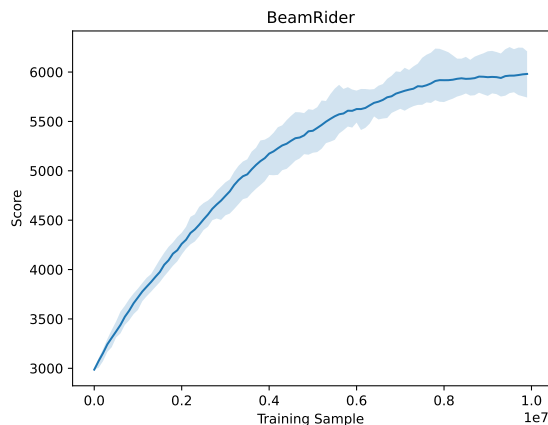


Figure 5.1: Performance of hierarchical agent on BeamRider.

5.4 Discussion

The results of the hierarchical agent show that it was able to successfully solve all 7 of the Atari games from the Baselines data set (Appendix A). In comparison with the state-of-the-

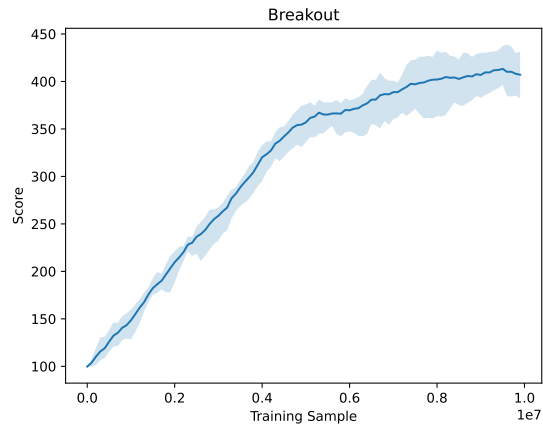


Figure 5.2: Performance of hierarchical agent on Breakout.

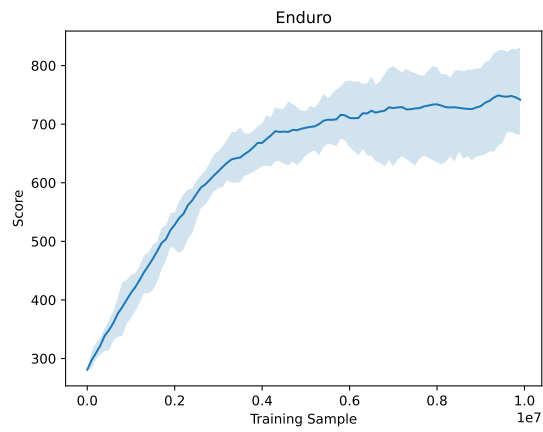


Figure 5.3: Performance of hierarchical agent on Enduro.

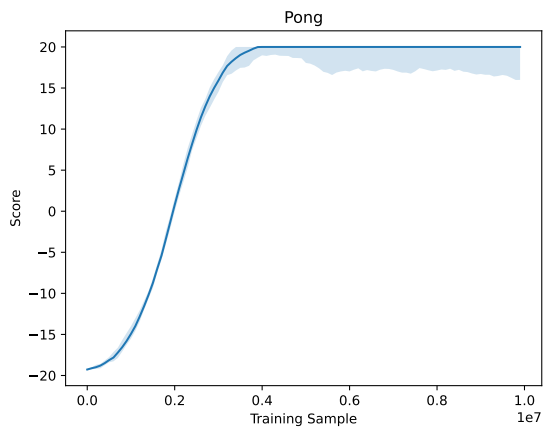


Figure 5.4: Performance of hierarchical agent on Pong.

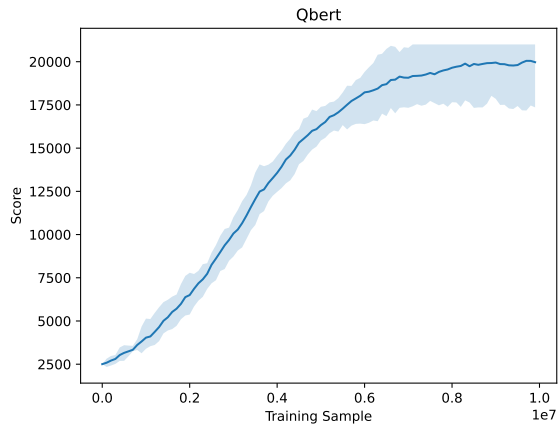


Figure 5.5: Performance of hierarchical agent on Qbert.

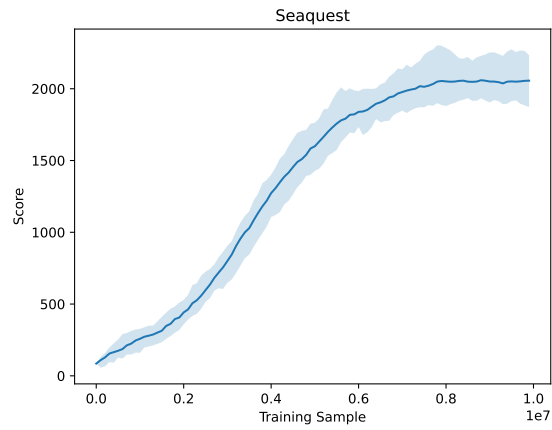


Figure 5.6: Performance of hierarchical agent on Seaquest.

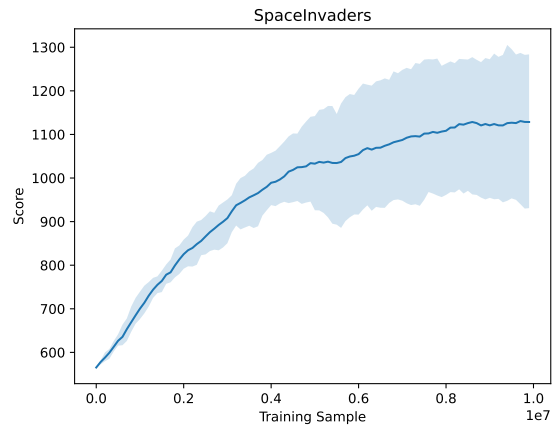


Figure 5.7: Performance of hierarchical agent on SpaceInvaders.

art algorithms shown in the Appendix, the algorithm achieves comparable high scores.

Another interesting result from the evaluation is the steepness of the learning curves. In many of the games the hierarchical agent had the steepest performance gains initially, even if it the high score saturated at a similar value as the baseline algorithms. This suggests that the hierarchical agent may be more sample efficient, allowing the agent to achieve moderate performance with fewer training samples.

Performance Variability There is variability in the performance of common agents used in the OpenAI Baselines repository. this is why post-training agent performance has been reported as a distribution, rather than a point estimate [35]. The graphs show the average performance, as well as the min and max over 6 runs with varied random seeds.

Chapter 6

Conclusion

6.1 Summary

An HRL algorithm was presented which uses MCTS as a high-level controller, and DIAYN as a low-level controller. The combination of model-based planning together with a model-free controller combines the best of both worlds, and is able to efficiently solve the sparse reward problem.

The algorithm was shown to perform well on random maze navigation tasks, as well as general-purpose Atari tasks. In comparison with state-of-the-art RL algorithms, the HRL agent was shown to be competitive in terms of performance. In addition, the algorithm was shown to be sample efficient, achieving moderate results after a small number of training samples.

6.2 Future Directions

There are several potential avenues for future research based on this research. First is that automatic hyperparameter tuning could avoid the expensive sweeping of grid search. Autonomous addition of new worker policies as needed could avoid trying different values and training each variation repetitively.

The other hyperparameter that could be made learnable is the worker rollout duration. This could also be a variable duration termination function. This choice is environment dependent, and there is a tradeoff between overall MCTS search effort and overall performance. If the duration is a single base level time step, the hierarchy collapses, and the efficiency gains of the hierarchical approach are lost. On the other hand, if the duration is too long, the performance of the overall agent may suffer, as time steps are wasted with a suboptimal policy being active.

HRL is a promising area of research in RL, with many challenges and opportunities. Overcoming these challenges can enable the development of more efficient and effective RL algorithms that can be used in a wide range of applications.

Bibliography

- [1] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.
- [2] Nicolas Bougie and Ryutaro Ichise. Hierarchical learning from human preferences and curiosity. *Applied Intelligence*, 52(7):7459–7479, may 2022.
- [3] Richard S. Sutton, Doina Precup, and Satinder Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1):181 – 211, 1999.
- [4] G.M.J.B. Chaslot, M.H.M. Winands, J.W.H.M. Uiterwijk, H.J. Van Den Herik, and B. Bouzy. Progressive strategies for Monte-Carlo Tree Search. *Information Sciences 2007*, page 655–661, 2007.
- [5] Benjamin Eysenbach, Abhishek Gupta, Julian Ibarz, and Sergey Levine. Diversity is all you need: Learning skills without a reward function. *CoRR*, abs/1802.06070, 2018.
- [6] M. de Waard, D. M. Roijers, and S. C. J. Bakkes. Monte Carlo Tree Search with options for general video game playing. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8, Sept 2016.
- [7] Pierre-Luc Bacon, Jean Harb, and Doina Precup. The option-critic architecture. *CoRR*, abs/1609.05140, 2016.
- [8] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33rd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, pages 1928–1937, 2016.
- [9] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, and et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.
- [10] Piotr Mirowski, Razvan Pascanu, Fabio Viola, Hubert Soyer, Andrew J. Ballard, Andrea Banino, Misha Denil, Ross Goroshin, Laurent Sifre, Koray Kavukcuoglu, Dharshan

- Kumaran, and Raia Hadsell. Learning to navigate in complex environments. *CoRR*, abs/1611.03673, 2016.
- [11] Vikas Dhiman, Shurjo Banerjee, Brent Griffin, Jeffrey Mark Siskind, and Jason J. Corso. A critical investigation of deep reinforcement learning for navigation. *CoRR*, abs/1802.02274, 2018.
- [12] Ken Kansky, Tom Silver, David A. Mély, Mohamed Eldawy, Miguel Lázaro-Gredilla, Xinghua Lou, Nimrod Dorfman, Szymon Sidor, D. Scott Phoenix, and Dileep George. Schema networks: Zero-shot transfer with a generative causal model of intuitive physics. *CoRR*, abs/1706.04317, 2017.
- [13] Tobias Glasmachers. Limits of end-to-end learning. *CoRR*, abs/1704.08305, 2017.
- [14] Adrià Puigdomènech Badia, Bilal Piot, Steven Kapturowski, Pablo Sprechmann, Alex Vitvitskyi, Zhaohan Daniel Guo, and Charles Blundell. Agent57: Outperforming the atari human benchmark. *CoRR*, abs/2003.13350, 2020.
- [15] Sara Sabour, Nicholas Frosst, and Geoffrey E. Hinton. Dynamic routing between capsules. *CoRR*, abs/1710.09829, 2017.
- [16] Peter W. Battaglia, Razvan Pascanu, Matthew Lai, Danilo Jimenez Rezende, and Koray Kavukcuoglu. Interaction networks for learning about objects, relations and physics. *CoRR*, abs/1612.00222, 2016.
- [17] Kevin Yang, Tianjun Zhang, Chris Cummins, Brandon Cui, Benoit Steiner, Linnan Wang, Joseph E. Gonzalez, Dan Klein, and Yuandong Tian. Learning space partitions for path planning, 2022.
- [18] Craig A Knoblock. Automatically generating abstractions for planning. *Artificial Intelligence*, 68(2):243–302, 1994.
- [19] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016.
- [20] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy P. Lillicrap, and David Silver. Mastering atari, go, chess and shogi by planning with a learned model. *CoRR*, abs/1911.08265, 2019.
- [21] Julian Schrittwieser, Thomas Hubert, Amol Mandhane, Mohammadamin Barekatin, Ioannis Antonoglou, and David Silver. Online and offline reinforcement learning by planning with a learned model. *CoRR*, abs/2104.06294, 2021.

- [22] Matthias Hutsebaut-Buysse, Kevin Mets, and Steven Latré. Hierarchical reinforcement learning: A survey and open research challenges. *Machine Learning and Knowledge Extraction*, 4(1):172–221, 2022.
- [23] Thomas G. Dietterich. The maxq method for hierarchical reinforcement learning. In *Proceedings of the Fifteenth International Conference on Machine Learning*, ICML '98, page 118–126, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.
- [24] Richard S. Sutton, Doina Precup, and Satinder P. Singh. Intra-option learning about temporally abstract actions. In *Proceedings of the Fifteenth International Conference on Machine Learning*, ICML '98, page 556–564, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.
- [25] Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. In *Proceedings of the 1997 Conference on Advances in Neural Information Processing Systems 10*, NIPS '97, page 1043–1049, Cambridge, MA, USA, 1998. MIT Press.
- [26] Neville Mehta, Soumya Ray, Prasad Tadepalli, and Thomas Dietterich. Automatic discovery and transfer of maxq hierarchies. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, page 648–655, New York, NY, USA, 2008. Association for Computing Machinery.
- [27] Christian Kroer and Tuomas Sandholm. Sequential planning for steering immune system adaptation. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, IJCAI'16, page 3177–3184. AAAI Press, 2016.
- [28] Marwin H. S. Segler, Mike Preuss, and Mark P. Waller. Learning to plan chemical syntheses. *CoRR*, abs/1708.04202, 2017.
- [29] Thi Thoa Mac, Cosmin Copot, Duc Trung Tran, and Robin De Keyser. Heuristic approaches in robot path planning: A survey. *Robotics and Autonomous Systems*, 86:13–28, 2016.
- [30] Matthew E. Taylor and Peter Stone. Transfer learning for reinforcement learning domains: A survey. *J. Mach. Learn. Res.*, 10:1633–1685, December 2009.
- [31] Nicolas P Rougier. Maze generation algorithm. https://en.wikipedia.org/wiki/Maze_generation_algorithm, 2010.
- [32] Charles Beattie, Joel Z. Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler, Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, Julian Schrittwieser, Keith Anderson, Sarah York, Max Cant, Adam Cain, Adrian Bolton, Stephen Gaffney, Helen King, Demis Hassabis, Shane Legg, and Stig Petersen. Deepmind lab. *CoRR*, abs/1612.03801, 2016.
- [33] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *CoRR*, abs/1207.4708, 2012.

- [34] Marlos C. Machado, Marc G. Bellemare, Erik Talvitie, Joel Veness, Matthew J. Hausknecht, and Michael Bowling. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *Journal of Artificial Intelligence Research*, 61:523–562, 2018.
- [35] Kaleigh Clary, Emma Tosch, John Foley, and David D. Jensen. Let’s play again: Variability of deep reinforcement learning agents in atari environments. *CoRR*, abs/1904.06312, 2019.
- [36] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. Openai baselines. <https://github.com/openai/baselines>, 2017.

Appendix A

OpenAI Baselines for Atari Learning Environment

Below are the results for 7 different RL algorithms, reproduced from the OpenAI Baselines repository [36].








algo	user	mean	Enduro	SpaceInvaders	Qbert	Seaquest	Pong	BeamRider	Breakout
ppo2	 cron	1509.38	350.22	557.28	7012.06	1218.87	13.68	1299.25	114.26
deepq	 cron	1012.69	479.75	459.86	3254.83	1164.08	16.49	1582.34	131.46
acktr	 cron	1211.71	0.0	557.19	4429.3	1201.16	9.56	2171.19	113.58
acer	 cron	1457.36	0.0	656.91	6433.38	1065.98	3.11	1959.22	82.94
ppo2_mpi	 cron	1417.92	207.47	459.89	7184.73	1383.38	13.9	594.45	81.61
a2c	 cron	717.73	0.0	463.06	2047.07	1150.66	1.0	1302.61	59.72
trpo_mpi	 cron	625.67	24.83	457.7	2486.18	710.07	2.82	683.11	14.98

Table A.1: Baseline results for the Atari environments.

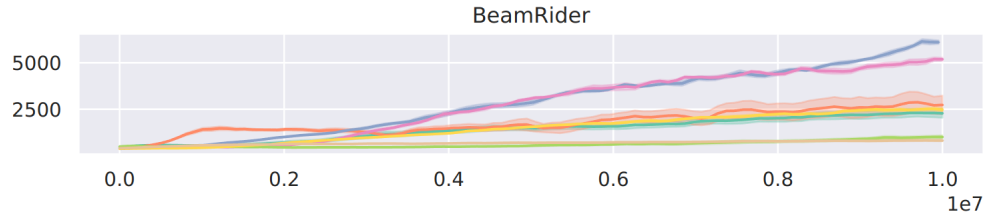


Figure A.1: Performance of baseline algorithms on BeamRider.

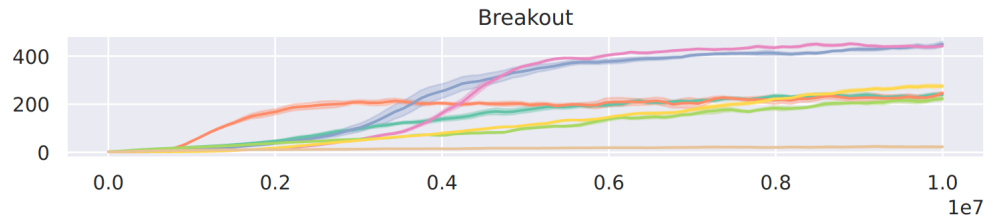


Figure A.2: Performance of baseline algorithms on Breakout.

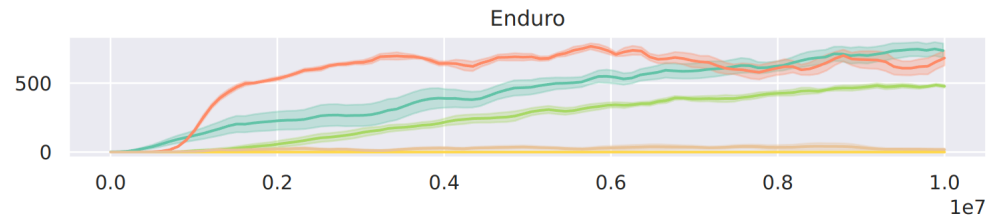


Figure A.3: Performance of baseline algorithms on Enduro.

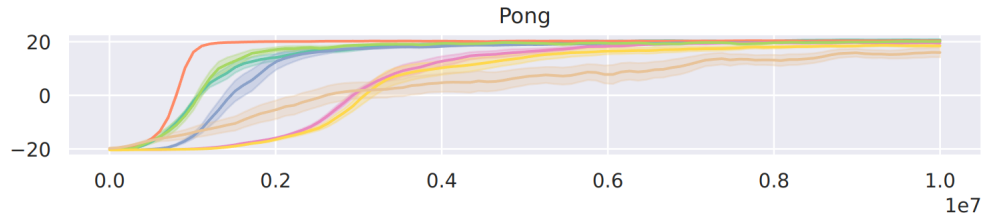


Figure A.4: Performance of baseline algorithms on Pong.

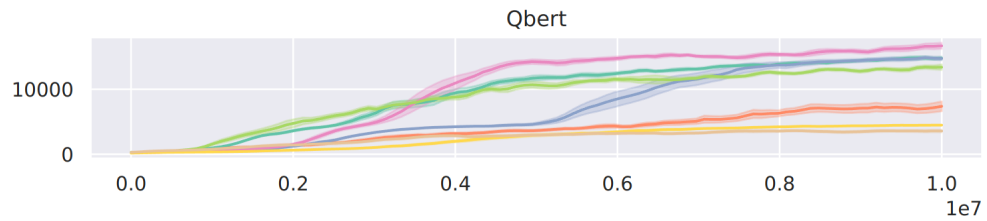


Figure A.5: Performance of baseline algorithms on Qbert.

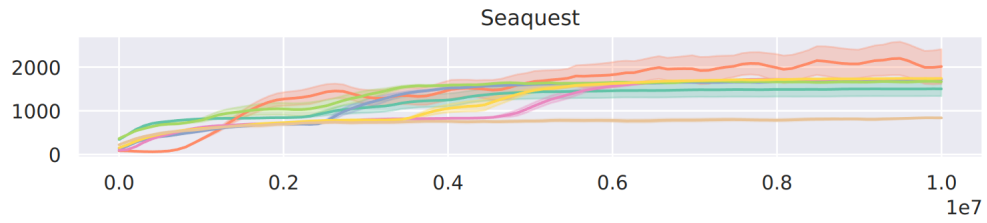


Figure A.6: Performance of baseline algorithms on Seaquest.

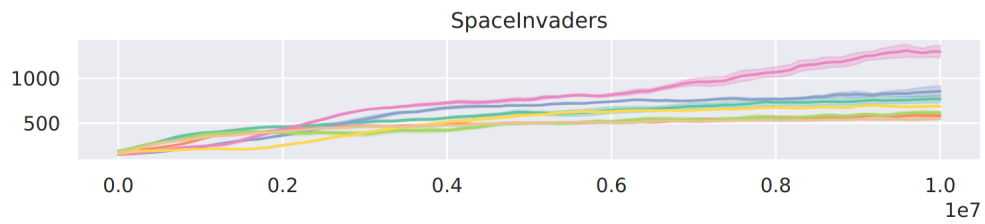


Figure A.7: Performance of baseline algorithms on SpaceInvaders.