Representation Learning for Hierarchical Reinforcement Learning

Lorenzo Steccanella

DOCTORAL THESIS UPF / 2023

THESIS SUPERVISORS
Dr. Anders Jonsson

Dept. of Information and Communication Technologies

Universitat Pompeu Fabra Barcelona

Al mio papà.

Abstract

The concept of hierarchy has a strong appeal to researchers in the field of artificial intelligence. Humans cope with the complexity of the world by thinking in a hierarchical manner, and this same faculty is sought to be imparted to autonomous agents.

Over the past few years, Reinforcement Learning (RL) methods have achieved remarkable success, largely facilitated by the use of deep learning models, reaching human-level performance in several domains, including Atari games and complex board games such as Go and Chess. Nevertheless, it is still a challenge for RL agents to solve environments with sparse rewards and long time horizons.

Hierarchical Reinforcement Learning (HRL) has the potential to simplify the solution of such environments. The idea behind HRL is to decompose a complex decision-making problem into smaller, manageable sub-problems, allowing an agent to learn more efficiently and effectively.

In this thesis, we aim to contribute to the field of HRL through the study of state space partition representations. We aim to discover representations that allow decomposing a complex state space in a set of small interconnected partitions. We start our work by presenting which are the properties of ideal state space partitions for HRL and then proceed to explore different methods for creating such partitions. We present algorithms able to leverage such representations to learn more effectively in sparse reward settings. Finally, we show how to combine the learned representation with Goal-Conditioned Reinforcement Learning (GCRL) and additionally we present state representations useful for GCRL.

Resumen

El concepto de jerarquía tiene un fuerte atractivo para los investigadores en el campo de la inteligencia artificial. Los humanos hacen frente a la complejidad del mundo pensando de manera jerárquica, y se busca impartir esta misma facultad a los agentes autónomos. En los últimos años, los métodos de Reinforcement Learning (RL) han logrado un éxito notable, facilitado en gran medida por el uso de modelos de Deep Learning, alcanzando un rendimiento de nivel humano en varios dominios, incluidos los juegos de Atari y los juegos de mesa complejos, como Go y el Ajedrez. Sin embargo, sigue siendo un desafío para los agentes de RL resolver entornos con escasas recompensas y horizontes a largo plazo. El método Hiearchical Reinforcement Learning (HRL) tiene el potencial de simplificar la solución de dichos entornos. La idea detrás de HRL es descomponer un problema complejo de toma de decisiones en subproblemas más pequeños y manejables, lo que permite que un agente aprenda de manera más eficiente y efectiva. En esta tesis, pretendemos contribuir al campo del HRL a través del estudio de las representaciones de partición del espacio de estado. Nuestro objetivo es descubrir representaciones que permitan descomponer un espacio de estado complejo en un conjunto de particiones interconectadas. Comenzamos nuestro trabajo presentando cuáles son las propiedades de las particiones de espacio de estado ideales para HRL y luego procedemos a explorar diferentes métodos para crear dichas particiones. Presentamos algoritmos capaces de aprovechar tales representaciones para aprender de manera más efectiva en entornos de escasa recompensa. Finalmente, mostramos cómo combinar la representación aprendida con el método Goal-Conditioned Reinforcement Learning (GCRL) y, adicionalmente, presentamos representaciones de estado útiles para GCRL.

Contents

Abstract **v**

Resumen **vi**

List of Figures **xiii**

List of Tables **xvi**

List of Algorithms **xviii**

1 Introduction **1**

1.1 Thesis Structure 3

1.2 Summary of Contributions 5

1.2.1 List of Talks 6

1.2.2 Upcoming Publications 6

I Background **7**

2 Reinforcement Learning and Markov Decision Processes **9**

2.1 Markov Decision Process 9

2.1.1 State Space 10

2.1.2 Action Space 10

2.1.3 Transition Dynamics 11

2.1.4 Rewards 11

2.1.5 Policies 11

2.1.6 Optimality Criteria 12

2.2 Value Functions and Bellman Equations 13

2.3 Dynamic Programming Algorithms 15
2.3.1 Policy Evaluation 15
2.3.2 Policy Improvement 15
2.3.3 Policy Iteration 16
2.3.4 Value Iteration 16
2.3.5 Generalized Policy Iteration 16
2.4 Tabular Reinforcement Learning 17
2.4.1 Monte Carlo methods 17
2.4.2 Temporal Difference Learning 18
2.4.3 Q-learning 19
2.5 Deep Reinforcement Learning 20
2.5.1 Deep Q Network 21
2.5.2 Policy Gradient Methods 23
2.6 Goal Conditioned Reinforcement Learning 28
2.6.1 Setting 28
2.6.2 Goal Conditioned Supervised Learning 28

3 Hierarchical Reinforcement Learning 29
3.1 Introduction 29
3.2 Four-Room Task 30
3.3 Temporally Extended Actions 31
3.4 Semi-Markov Decision Processes 31
3.4.1 SMDP Q learning 33
3.5 Structure 33
3.6 Optimality 34
3.7 Bottleneck States 36
3.8 Options 37

II State Space Partitioning and Option Learning 39

4 State Space Partitioning 41
4.1 State Space Partitions 42
4.2 State Space Partitions Properties 43

5 Hierarchical reinforcement learning for exploration and transfer 47

5.1 Methodology 49
5.1.1 Task MDPs 49
5.1.2 Invariant SMDP 49
5.1.3 Option MDPs 50
5.1.4 Algorithm 51
5.1.5 Solving tasks 52
5.1.6 Controllability 54
5.2 Implementation 55
5.2.1 Manager 55
5.2.2 Worker 56
5.3 Experiments 56
5.3.1 Exploration 58
5.3.2 Transfer Learning 58
5.3.3 Controllability 60
5.4 Discussion 60
5.5 Conclusion 62

6 Hierarchical Representation Learning for Markov Decision Processes 63

6.1 Contribution 65
6.1.1 Compression Function 65
6.1.2 Hierarchical Representation 66
6.1.3 Controllability 68
6.1.4 Transfer 68
6.2 Experimental Results 69
6.2.1 Learning a Compression Function 70
6.2.2 Hierarchical Reinforcement Learning 71
6.2.3 Experiments 77
6.2.4 Additional Empirical Evaluation 78
6.3 Hyperparameters 79
6.4 Conclusion 81

7 Distance Based Representation for Hierarchical Reinforcement Learning 83

7.1 Minimum Action Distance 84

7.1.1 Learning Minimum Action Distance from Adjacency Matrix . . . 85
7.1.2 Symmetric embeddings . . . 86
7.1.3 Asymmetric semi-norm embeddings . . . 89
7.2 Minimum Action Distance State Space Partitions . . . 91
7.2.1 Options Representation . . . 93
7.3 Bottleneck State Discovery . . . 94
7.4 Preliminary Results . . . 95
7.5 Conclusion . . . 99

III Representation Learning for Goal Conditioned Reinforcement Learning 101

8 State Representation Learning For Goal Conditioned Reinforcement Learning 103
8.1 Contribution . . . 104
8.1.1 Learning Transition Models . . . 105
8.1.2 Latent space planning . . . 106
8.1.3 Reward Shaping . . . 107
8.2 Experimental Results . . . 108
8.2.1 Dataset Collection and Domain Description . . . 109
8.2.2 Learning a State Embedding . . . 111
8.2.3 Learning Dynamics . . . 112
8.2.4 Experiments . . . 112
8.3 Conclusion . . . 114

9 Conclusions and Future Work 117

Bibliography 120

List of Figures

2.1 The reinforcement learning framework, in which an agent takes a series of actions, each of which generates a reward and a new state. 10

2.2 Actor-Critic Architecture (Sutton and Barto 2018). 27

3.1 The four-room task in which an agent (A) has to reach the goal (G). Green blocks ($D1$, $D2$, $D3$, $D4$) represents doorways that the agent has to cross to move between rooms ($R1$, $R2$, $R3$, $R4$) in the environment. 31

3.2 An example of hierarchy decomposing the four-room task. X cells represent terminal states. 34

3.3 The policy shown in the left diagram is recursively optimal but not hierarchically optimal. The policy in the right diagram is hierarchically optimal but not recursively optimal. The shaded cells indicate states where the two definitions of optimality disagree (Dietterich 2000; Ghavamzadeh and Mahadevan 2002). 36

5.2 Key-door-treasure-1 (a) and Montezuma's Revenge (b) with compression function superimposed. 58

5.3 Results in Key-door-treasure-1. 59

5.4 Results in Key-door-treasure-2, reward for all objects. . . . 59

5.5 Results in Montezuma's Revenge with controllability. 60

5.6 Results of transfer learning, with reward given for all objects. 61

6.1 Results on Key-Door0 gridworld environment. The first row represents the environments, and the second row illustrates the corresponding learned deterministic compression functions, where different colors represent different regions $z \in \mathcal{Z}$. 69
6.2 Results on geometric variations of the NineRooms0 grid world environments. The first row represents the environments, and the second row illustrates examples of the corresponding learned deterministic compression functions, where different colors represent different regions $z \in \mathcal{Z}$. . . . 70
6.3 Absolute error of the compression function, evaluated on increasing replay memory size. 72
6.4 The discovered invariant SMDP. 74
6.5 Results in the KeyDoor environment. 74
6.6 Results on the variations of nine room gridworld environments where the goal (green square) is placed at an increasing distance from the agent (blue square). From left to right: NineRoom1, NineRoom2, NineRoom3. 77
6.7 Results of the compression function in the MountainCar environment (axes represent location and velocity); different colors represent different regions $z \in \mathcal{Z}$. 79

7.1 Top: a simple grid world where an agent has to pick up a key and open a door (key and door positions are fixed). Bottom: the learned state embedding ϕ on $\mathbb{R}^2$. The state (x, y, has_key) is composed of the agent's location and whether or not it holds the key. 90
7.2 Left: a simple grid world where an agent has to pick up a key (key position is fixed). Right: the learned Centrality measure. The state $(x, y, haskey)$ is composed of the agent location and whether or not it holds the key. 95
7.3 Mean squared error of the MAD distance learned on the EmptyGridWorld respect to the ground truth L1 distance . 96
7.4 EmptyGridWorld results, where we compare Q learning on the original MDP $\mathcal{M}$ (FLAT) against SMDP Q learning on the SMDP $\mathbb{S}$ (SKILL-SMDP). 98

7.5 KeyDoorGridWorld results, where we compare Q learning on the original MDP $\mathcal{M}$ (FLAT) against SMDP Q learning on the SMDP $\mathbb{S}$ (SKILL-SMDP). 99

8.1 Evaluation Tasks. Top row: MountainCar-v0, CartPole-v0, AcroBot-v1 and Pendulum-v0. Bottom row: GridWorld and SawyerReachXYZEnv-v1. 109
8.2 Results in the classic RL control suite. 111
8.3 Results in multi-goal environments. 113

List of Tables

5.1 Hyperparameters used in the experiments. 62

6.1 Hyperparameters used to train HRL and HRL-Transfer agents. 80
6.2 Hyperparameters used to train SIL and DQN-PER agents. . 81

8.1 Dataset description. 110

List of Algorithms

2.1 Tabular TD(0) for estimating V^π (Sutton and Barto 2018) . 19
2.2 Tabular Q-learning (Watkins and Dayan 1992) 20

5.1 InvariantHRL 53
5.2 Manager 55
5.3 RunOption 57

6.1 Manager 76

7.1 MAD State Space Partition embedded in an Episodic RL framework 92

8.1 Plan-Dist 107

Chapter 1

Introduction

The concept of hierarchy has a strong appeal to researchers in the field of artificial intelligence. Humans cope with the complexity of the world by thinking in a hierarchical manner (Botvinick et al. 2015; Botvinick, Niv, and Barto 2009; Eckstein and Collins 2020; Tenenbaum et al. 2011), and this same faculty is sought to be imparted to autonomous agents.

Reinforcement learning (RL) (Sutton and Barto 2018) is a powerful framework for decision-making in complex and uncertain environments. However, when faced with large and complex state spaces, traditional RL can become intractable, leading to the need for more sophisticated techniques. One such technique is Hierarchical Reinforcement Learning (HRL) (Barto and Mahadevan 2003), which breaks down a complex task into smaller, more manageable subtasks, each with its own policy. In HRL the agent must be able to recognize and isolate the relevant information at each level of abstraction and use this information to make informed decisions and learn a successful policy. This requires an appropriate model of the state space, including the partitioning of the state space into smaller, more manageable regions.

State space partitioning is a crucial component of HRL (Wen et al. 2020), as it determines how the task is decomposed into subtasks, and how the policies for these subtasks are coordinated. The process of state space partitioning involves defining the boundaries between different subspaces, and mapping states in the original state space to the corresponding subspaces. This mapping is referred to as state abstraction.

Several approaches to state space partitioning have been proposed in literature, including clustering-based and topological approaches (Ciosek and Silver 2015; Ecoffet et al. 2021; Mannor et al. 2004; Menache, Mannor, and Shimkin 2002; Nachum et al. 2018b; Vezhnevets et al. 2017), and model-based approaches (Asadi and Huber 2004; Castro and Precup 2010; Castro and Precup 2012; Ferns, Panangaden, and Precup 2004, 2011; Infante, Jonsson, and Gómez 2022; Li et al. 2021; Ravindran and Barto 2002, 2004; Wan and Sutton 2022). Clustering-based and topological approaches involve grouping similar states together into the same subspace, using metrics based on features of a state or based on the topological structure of the state space. Model-based approaches are based on the dynamics of the environment and involve defining subspaces based on the behavior of the system in response to different actions and rewards. These methods often require prior knowledge, struggle with large and continuous Markov Decision Processes or fail to meet other requirements discussed in Chapter 4.

In addition to state space partitioning, we will also be focusing on another important aspect of HRL, namely goal-conditioned reinforcement learning.

Goal-conditioned reinforcement learning (Liu, Zhu, and Zhang 2022) involves learning policies that are conditioned on a specific goal or set of goals, rather than on the entire state space. This can be particularly useful in HRL (Nachum et al. 2018a,b), where the goal of a subtask may be specified in terms of high-level objectives, rather than low-level state information. By learning goal-conditioned policies, HRL systems can better capture the structure of the problem and can be more effective at solving complex tasks.

The choice of state space partition and goal-conditioned reinforcement learning method can have a significant impact on the performance of the HRL system. For example, if the partition is too fine-grained or the goals are too specific, the subtasks may be too complex to be learned effectively, while if the partition is too coarse or the goals are too general, the subtasks may not be primitive enough to capture the structure of the problem. It is therefore important to carefully consider the trade-off between the granularity of the partition and the specificity of the goals, and the ability of the HRL system to learn the subtasks effectively.

The goal of this thesis is to advance the area of Hierarchical Reinforcement Learning (HRL) by addressing the challenges of state space partitioning and goal-conditioned reinforcement learning. The aim is to create new HRL algorithms that can effectively utilize and discover state space partitions either through interaction with the environment or by utilizing a pre-existing offline dataset. This thesis places a particular emphasis on reward-free representations, as these have the potential to enhance the transferability of the learned representations across multiple tasks in the environment. To achieve these aims, we will build upon existing HRL literature and incorporate recent developments in deep reinforcement learning.

1.1 Thesis Structure

The remainder of this thesis is organized into three main parts: Background (Chapters 2 and 3), State Space Partitioning and Options Learning (Chapters 4, 5, 6 and 7), and Representation Learning for Goal Conditioned Reinforcement Learning(Chapter 8). In the first part, we introduce the methods on which our work is based and recent related work. In the second part, we detail our contributions to Hierarchical Reinforcement Learning. In the third part, we detail our contribution to learning representation useful for Goal-Conditioned Reinforcement Learning.

I Background

- In Chapter 2 we delve into the background of Reinforcement Learning (RL). The chapter covers the fundamental concepts of Markov Decision Processes (MDPs) and the application of Dynamic Programming and Reinforcement Learning methods for solving them. Additionally, we provide an overview of relevant Deep Reinforcement Learning algorithms and introduce the concept of Goal-Conditioned Reinforcement Learning (GCRL).
- In Chapter 3, we provide an introduction to Hierarchical Reinforcement Learning (HRL). To start, we present the notion of Temporally Extended (TE) Actions and extend the traditional Markov Decision Process (MDP) to a Semi-Markov Decision Process (SMDP), allowing for the consideration of TE actions. Additionally, we introduce

the framework of Options as described by Sutton, Precup, and Singh (1999).

II State Space Partitioning and Options Learning

- In Chapter 4, the topic of state space partitioning is addressed. The properties necessary for an effective representation in Hierarchical Reinforcement Learning are emphasized and discussed in detail.
- In Chapter 5, we present our first contribution to the field of Hierarchical Reinforcement Learning (HRL). One of the challenges in Reinforcement Learning is solving sparse-reward domains where significant exploration is required before obtaining a reward. Our contribution aims to address this challenge by introducing a novel HRL framework based on the compression of an invariant state space that is common to a range of tasks. The framework consists of dividing the original problem into smaller subtasks, which focus on moving between the state partitions induced by the compression. By doing so, the number of decisions required before obtaining a reward is reduced, facilitating exploration. The results of our experiments indicate that the proposed algorithm is capable of solving complex sparse-reward domains, and it demonstrates transferability by quickly solving new, previously unseen tasks.
- In Chapter 6, we present a novel method for learning hierarchical representations of Markov decision processes. Our method works by partitioning the state space into subsets and defining subtasks for performing transitions between the partitions. We formulate the problem of partitioning the state space as an optimization problem that can be solved using gradient descent given a set of sampled trajectories, making our method suitable for high-dimensional problems with large state spaces. We empirically validate the method, by showing that it can successfully learn a useful hierarchical representation in a navigation domain. Once learned, the hierarchical representation can be used to solve different tasks in the given domain, thus generalizing knowledge across tasks.
- In Chapter 7, we introduce the Minimum Action Distance (MAD) metric and its properties. We show how the MAD can be learned from sampled trajectories in an environment, used to partition the

state space and to learn a set of options. Our preliminary results demonstrate the effectiveness of this method, as it leads to faster learning and improved exploration compared to flat agents.

III Representation Learning for Goal Conditioned Reinforcement Learning

- In Chapter 8, we present a novel state representation for reward-free Markov decision processes. The idea is to learn, in a self-supervised manner, an embedding space where distances between pairs of embedded states correspond to the minimum number of actions needed to transition between them. We show how this representation can be leveraged to learn goal-conditioned policies, providing a notion of similarity between states and goals and a useful heuristic distance to guide planning and reinforcement learning algorithms. Finally, we empirically validate our method in classic control domains and multi-goal environments, demonstrating that our method can successfully learn representations in large and/or continuous domains.

Finally, we present our conclusion and future work in Chapter 9.

1.2 Summary of Contributions

The research conducted during the completion of this thesis resulted in the following published works:

- **Chapter 5:** Lorenzo Steccanella, Simone Totaro, Damien Allonsius, Anders Jonsson. Hierarchical reinforcement learning for efficient exploration and transfer. 4th Workshop on Lifelong Learning (LifelongML) at ICML.

- **Chapter 6:** Lorenzo Steccanella, Simone Totaro, Anders Jonsson. Hierarchical Representation Learning for Markov Decision Processes. Workshop on Generalization in Planning (GenPlan) at IJCAI.

- **Chapter 8:** Lorenzo Steccanella, Anders Jonsson. State Representation Learning for Goal-Conditioned Reinforcement Learning.

Workshop on Bridging the Gap Between AI Planning and Reinforcement Learning (PRL) at ICAPS/IJCAI and ECML PKDD 2022.

1.2.1 List of Talks

The work presented in this thesis has been disseminated to the wider academic community through the following talks:

- Eastern European Machine Learning Summer School 2020. Talk about Hierarchical reinforcement learning for efficient exploration and transfer.
- Arizona State University, AAIR lab. Talk about Representations for Hierarchical reinforcement learning.
- Deep Learning Barcelona Symposium. Talk about State Representation Learning for Goal-Conditioned Reinforcement Learning.

1.2.2 Upcoming Publications

The results presented in Chapter 7 are based on our recent work, and we plan to submit it for publication in the upcoming months. Furthermore, we are working on a journal article that collects our works on representation learning for Hierarchical Reinforcement Learning.

Part I

Background

Chapter 2

Reinforcement Learning and Markov Decision Processes

In this chapter, we will introduce the notation and concepts behind Markov decision processes (MDPs) and a class of algorithms called Reinforcement Learning for computing optimal behaviors.

2.1 Markov Decision Process

A Markov Decision Process (MDP) (Puterman 2014) is a framework for decision-making with stochasticity. MDP can be seen as stochastic extensions of finite automata and also as Markov processes augmented with actions and rewards.

Formally, a Markov Decision Process is a tuple $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, r \rangle$ where:

- $\mathcal{S}$ is the state space, which contains all possible states the system may be in.
- $\mathcal{A}$ is the action space, which contains all possible actions the agent may take when interacting with the system.
- $\mathcal{P} : \mathcal{S} \times \mathcal{A} \mapsto \Delta(\mathcal{S})$ is the transition dynamics. Here $\Delta(\mathcal{S})$ is the probability simplex on $\mathcal{S}$, i.e. the set of all probability distributions over $\mathcal{S}$. For each state s, action a and next state s', $\mathcal{P}(s, a, s')$ indicates the probability of ending up in state s' after doing action

a in state s. We sometimes write the conditional probability of transitioning into state s' as $\mathcal{P}(s'|s,a)$.

- $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto \mathbb{R}$ is the reward function. The value $r(s,a,s')$ gives the amount of "reward" associated with transitioning into state s' when taking action a from state s.

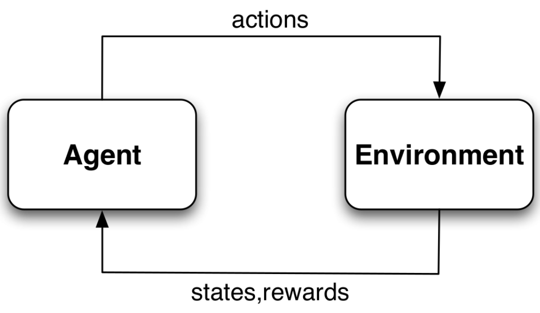


Figure 2.1: The reinforcement learning framework, in which an agent takes a series of actions, each of which generates a reward and a new state.

2.1.1 State Space

In an MDP we define states in $\mathcal{S}$ as a complete description of the state of the world. There is no information about the world that is hidden from the state. Some environments, like Atari, have discrete state space while others, like robotics, have continuous state space.

We represent states as finite vectors or matrices. For instance, a visual observation could be represented by the RGB matrix of its pixel values; the state of a robot might be represented by a vector of its joint angles and velocities.

2.1.2 Action Space

Actions in $\mathcal{A}$ can be used to control the system state and different environments allow different kinds of actions. The set of actions that can be applied in some particular state $s \in \mathcal{S}$, is denoted $\mathcal{A}(s)$, where $\mathcal{A}(s) \subseteq \mathcal{A}$. Some environments, like Atari and Go, have discrete action spaces, where only a finite number of moves are available to the agent. Other environments, like the control of a robot in a physical world, have continuous action spaces. In continuous spaces, actions are real-valued vectors.

2.1.3 Transition Dynamics

The transition function $\mathcal{P} : \mathcal{S} \times \mathcal{A} \mapsto \Delta(\mathcal{S})$ describes the dynamics of our environment.

To define an order in which actions occur, we will define a discrete global time, $t = 1, 2, ...$ For example, the notation s_t denotes the state at time t and s_{t+1} denotes the state at time $t + 1$.

The system being controlled is Markovian if the result of an action does not depend on the previous action and visited states, but only depends on the current state, i.e.

$$P(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, ...) = \mathcal{P}(s_{t+1}|s_t, a_t) = \mathcal{P}(s_t, a_t, s_{t+1}). \qquad (2.1)$$

The idea of Markovian dynamics is that the current state s provides enough information to make an optimal decision.

2.1.4 Rewards

Along with this thesis, we considered deterministic reward functions that map a particular transition (s, a, s'), obtained by applying action a in state s and transitioning to state s', to a real-valued reward $r(s, a, s')$. The reward function is used to define how the system i.e. the MDP, should be controlled.

2.1.5 Policies

Given an MDP, a policy is a computable function that outputs for each state $s \in \mathcal{S}$ an action $a \in \mathcal{A}(s)$. A policy can be either deterministic or stochastic.

A deterministic policy is a function $\pi : \mathcal{S} \to \mathcal{A}$ that directly maps a state to an action to take when in that state.

A stochastic policy $\pi : \mathcal{S} \to \Delta(\mathcal{A})$ assigns a distribution over actions to each state. As with the dynamics, we write the action distribution that

π assigns to state s as $\pi(\cdot|s)$, but the conditional probability of action a in state s when executing policy π as $\pi(a|s)$.

At each time step the policy π outputs an action $a_t \sim \pi(\cdot, s_t)$ and the action is performed. Based on the transition function $\mathcal{P}$ and reward function r a transition is made to state s_{t+1} with probability $\mathcal{P}(s_t, a_t, s_{t+1})$ and a reward $r_t = r(s_t, a_t, s_{t+1})$ is received.

2.1.6 Optimality Criteria

In the previous sections, we have defined the environment (the MDP) and the agent (i.e. the controlling element, or policy). Before we can talk about algorithms for computing optimal policies, we have to define what the agent optimizes.

We refer to a trajectory τ as a sequence of states and actions in the environment $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \ldots, s_T)$ of length T. The return associated with a trajectory $R(\tau)$ is the sum of (discounted) rewards:

$$R(\tau) = \sum_{t=0}^{T} \gamma^t r_t, \tag{2.2}$$

where $\gamma \in (0, 1]$ is the discount factor that defines how much we take into account immediate reward and future reward at each time step. We also define $R_t(\tau)$ as the return received from time step t.

$$R_t(\tau) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... = \sum_{k=t}^{T} \gamma^{k-t} r_k. \tag{2.3}$$

The standard objective for solving an MDP is to find a policy that maximizes the return:

$$\pi^* \in \operatorname*{argmax}_{\pi \in \Pi} \mathbb{E}\left[R(\tau)\right] \tag{2.4}$$

Episodic MDPs

In Episodic MDPs there exists a specific subset of states $\mathcal{S}_e \in \mathcal{S}$ denoted as terminal states where the process ends. The agent in this setting interacts with the MDP over $k = 1, ..., K$ episodes. We refer to this setting by augmenting the MDP tuple with the set of terminal states $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, r, \mathcal{S}_e \rangle$.

The return associated with a trajectory $R(\tau)$ in an episodic MDP is usually defined as above, with $T < \infty$ and possibly undiscounted, $\gamma = 1$.

Infinite Horizon MDPs

In the infinite horizon setting the system is modeled as an infinite horizon where $T = \infty$ and with a discount factor $\gamma \in (0, 1)$. The discount factor $\gamma < 1$ ensures that even with an infinite horizon the sum of the discounted rewards obtained is finite.

2.2 Value Functions and Bellman Equations

Most learning algorithms compute the optimal policy by learning the value function $V : \mathcal{S} \to \mathbb{R}$ or the action-value function $Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. Throughout this section, we will consider the infinite-horizon discounted return setup $\mathbb{E}\left[R(\tau)\right] = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right]$, but the formulation can be simply extended to the episodic case by means of including the time-dependency in the equations.

$$V^\pi(s) = \mathbb{E}\left[R(\tau) | s_0 = s\right] = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s\right], \tag{2.5}$$

$$Q^\pi(s, a) = \mathbb{E}\left[R(\tau) | s_0 = s, a_0 = a\right] = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s, a_0 = a\right]. \tag{2.6}$$

In words, $V^\pi(s)$ gives the expected return when starting in state s, and acting according to π. Similarly, $Q^\pi(s, a)$ gives the expected return starting in state s, taking action a, and acting according to π.

An important property of the value function is that it can be calculated recursively in terms of the so-called Bellman Equation (Bellman 1957)

$$\begin{aligned}
V^{\pi}(s) &= \mathbb{E}\left[R_t(\tau)|s_t = s\right] \\
&= \mathbb{E}\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots |s_t = s\right] \\
&= \mathbb{E}\left[r_t + \gamma V^{\pi}\left(s_{t+1}\right)|s_t = s\right] \\
&= \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in S} \mathcal{P}(s,a,s')(r(s,a,s') + \gamma V^{\pi}(s')).
\end{aligned} \tag{2.7}$$

The solution of an MDP consists of the optimal policy π^*. This optimal policy can be learned through the computation of an optimal value function such that $V^*(s) \geq V^{\pi}(s)\ \forall s \in \mathcal{S}$:

$$\begin{aligned}
V^*(s) &= \max_{\pi} V^{\pi}(s) \\
&= \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}\left(s,a,s'\right)\left(r\left(s,a,s'\right) + \gamma V^*\left(s'\right)\right).
\end{aligned} \tag{2.8}$$

In the same way, we can define an optimal action-value function as:

$$\begin{aligned}
Q^*(s,a) &= \max_{\pi} Q^{\pi}(s,a) \\
&= \sum_{s' \in S} \mathcal{P}\left(s,a,s'\right)\left(r\left(s,a,s'\right) + \gamma \max_{a'} Q^*\left(s',a'\right)\right).
\end{aligned} \tag{2.9}$$

The optimal action-value function plays a fundamental role in the majority of RL algorithms:

$$\begin{aligned}
Q^*(s,a) &= \sum_{s' \in \mathcal{S}} \mathcal{P}\left(s,a,s'\right)\left(r\left(s,a,s'\right) + \gamma V^*\left(s'\right)\right), \\
V^*(s) &= \max_{a} Q^*(s,a)
\end{aligned} \quad . \tag{2.10}$$

Another important function is the so-called advantage function $A : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ defined as:

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s). \tag{2.11}$$

The advantage function $A^{\pi}(s,a)$ corresponding to a policy π describes how much better it is to take a specific action a in state s, compared to simply following the policy π.

2.3 Dynamic Programming Algorithms

When the reward function and transition probabilities are known we can use dynamic programming (DP) algorithms to solve an MDP. Two core DP methods are policy iteration (Howard 1960) and value iteration (Bellman 1958). Along this section, we will consider MDPs with finite state and action spaces ($|\mathcal{S}| < \infty, |\mathcal{A}| < \infty$).

2.3.1 Policy Evaluation

In Policy Evaluation we are interested in computing the value function for an arbitrary policy V^π.

Consider a sequence of approximate value functions $V_0, V_1, \dots$. The initial approximation V_0, is chosen arbitrarily (except that the terminal state, if any, must be given value 0), and each successive approximation is obtained by using the Bellman equation as an update rule:

$$V_{k+1}(s) \leftarrow \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in S} \mathcal{P}(s,a,s')(r(s,a,s') + \gamma V_k(s')), \tag{2.12}$$

for all $s \in \mathcal{S}$. Clearly, $\lim_{k\to\infty} V_k = V^\pi$ is a fixed point for this update rule because the Bellman equation for V^π assures us of equality in this case.

2.3.2 Policy Improvement

The reason for computing the value function for a policy V^π is to help find better policies. In policy improvement, a new policy π_{k+1} is constructed that is guaranteed to be at least as good as the current policy π_k:

$$\begin{aligned} \pi_{k+1}(s) &= \arg\max_{a \in \mathcal{A}} Q^{\pi_k}(s,a) \\ &= \arg\max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}\left(s,a,s'\right)\left(r\left(s,a,s'\right) + \gamma V^{\pi_k}\left(s'\right)\right), \end{aligned} \tag{2.13}$$

for all $s \in \mathcal{S}$. Note that here we consider the special case of deterministic policies, but these ideas extend to stochastic policies (Sutton and Barto 2018).

2.3.3 Policy Iteration

Once a policy, π, has been improved using V^{π} to yield a better policy, π', we can then compute $V^{\pi'}$ and improve it again to yield an even better π''. We can thus obtain a sequence of monotonically improving policies and value functions:

$$\pi_0 \xrightarrow{\text{E}} V^{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} V^{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} V^*,$$

where $\xrightarrow{\text{E}}$ denotes a policy evaluation and $\xrightarrow{\text{I}}$ denotes a policy improvement. Each policy is guaranteed to be a strict improvement over the previous one (unless it is already optimal). Because a finite MDP has only a finite number of deterministic policies, this process must converge to an optimal policy and the optimal value function in a finite number of iterations. This way of finding an optimal policy is called Policy Iteration(PI).

2.3.4 Value Iteration

Value Iteration (VI) directly updates the value function by taking the maximum expected future return over all possible actions at each state. The policy is then extracted from the updated value function. This approach can be more efficient than Policy Iteration as it does not require a full policy evaluation before each policy improvement step.

In VI, the value function is updated as follows:

$$V_{k+1}(s) \leftarrow \max_{a \in \mathcal{A}} \sum_{s' \in S} \mathcal{P}(s, a, s')(r(s, a, s') + \gamma V_k(s')), \tag{2.14}$$

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

The VI algorithm continues until convergence, i.e., until $V_k(s)$ no longer changes after an iteration.

2.3.5 Generalized Policy Iteration

Policy Iteration consists of two processes: policy evaluation (making the value function consistent with the current policy) and policy improvement (making the policy greedy with respect to the current value function). The two processes alternate until convergence to the optimal value function V^*

and policy π^*. In VI, only one iteration of policy evaluation is performed between each policy improvement. We use the term Generalized Policy Iteration (GPI) to refer to the general idea of letting the policy evaluation and policy improvement processes interact, independent of the granularity and other details of the two processes.

2.4 Tabular Reinforcement Learning

In the previous section, we saw how to solve an MDP assuming that the transition model $\mathcal{P}$ and reward function r are given.

Reinforcement Learning (RL) is primarily concerned with how to obtain an optimal policy π^* when such a transition model and reward function are unknown.

The lack of a model generates a need to sample the MDP to gather statistical knowledge about this unknown model. Many model-free RL techniques exist that probe the environment by doing actions, thereby estimating the value and action-value functions.

RL approaches can be classified as being either on-policy or off-policy. We distinguish between the behavior policy, which is the policy used to act in the environment and the target policy, which is the policy updated. On-policy methods use the current policy both as a behavior policy to generate samples and as the target policy to update, while off-policy methods use a behavior policy to generate samples and update a different target policy with the generated samples.

This section will review model-free methods for solving MDPs relevant to the work presented in this thesis. We will consider a finite number of states $|\mathcal{S}| < \infty$ and actions $|\mathcal{A}| < \infty$ and a tabular representation that stores each value of the value function V or action-value function Q in a lookup table.

2.4.1 Monte Carlo methods

Monte Carlo (MC) methods are an on-policy approach that aims to solve reinforcement problems with episodic tasks, where no model of the environment exists. It is an iterative approach and converges as the number of episodes tends towards infinity toward the optimal policy.

MC methods keep frequency counts $N(s)$ and sums of returns $R(s)$ for each state $s \in \mathcal{S}$ and base their values on these estimates. MC methods only require samples to estimate the average sample returns. For example, in MC policy evaluation, for each state s_t visited, the value function $V^{\pi}(s_t)$ is estimated as:

$$V(s_t) = R(s_t)/N(s_t). \tag{2.15}$$

2.4.2 Temporal Difference Learning

An alternative to Monte Carlo methods is Temporal Difference (TD) (Sutton 1988). In contrast with MC methods TD learning can be applied to infinite horizon problems.

Temporal difference (TD) differs from the Monte Carlo methods in the policy evaluation step. Instead of using the total cumulative reward, the methods calculate a temporal error, which is the difference between the new estimate of the value function and the old estimate of the value function, by considering the reward received at the current time step and using it to update the estimated value of the next state. This kind of update reduces the variance but increases the bias in the estimate of the value function.

The Bellman Equation for the value function according to a policy π can be written as:

$$V^{\pi}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in S} \mathcal{P}\left(s, a, s'\right)\left(r\left(s, a, s'\right) + \gamma V^{\pi}\left(s'\right)\right), \tag{2.16}$$

where the value of the next state is an expectation over next states. Since we do not know the transition probabilities, the agent can sample a state s' from that expectation, removing the need of a transition model.

$$V^{\pi}(s) = \mathbb{E}\left[r(s, a, s') + \gamma V^{\pi}\left(s'\right)\right]. \tag{2.17}$$

Now since we are sampling the next state s' and learning the Value function V^{π} at the same time, the equation will be violated. This is in fact called the TD error :

$$\mathrm{TD} = r(s, a, s') + \gamma V^{\pi}\left(s'\right) - V^{\pi}(s). \tag{2.18}$$

TD(0) is a member of the family of TD learning algorithms (Sutton 1988). It estimates V^{π}, in an online, incremental fashion without the need for a transition model just by sampling on-policy transitions (s, a, r, s') and performing the following update rule:

$$V_{k+1}(s) \leftarrow V_k(s) + \alpha_k \left[r + \gamma V_k(s') - V_k(s)\right] , \tag{2.19}$$

where $\alpha_k \in (0, 1]$ is the learning rate that determines how much the values get updated.

Algorithm 2.1 Tabular TD(0) for estimating V^{π} (Sutton and Barto 2018)

```
Input: the policy $\pi$ to be evaluated
Initialize $V(s)$ arbitrarily (e.g., $V(s) = 0$, for all $s \in \mathcal{S}$)
for each episode do
    $s \leftarrow$ initial state of the episode
    for each step of episode do
        $a \leftarrow$ action given by $\pi$ for $s$
        Take action $a$, observe $r, s'$
        $V(s) \leftarrow V(s) + \alpha_k [r + \gamma V(s') - V(s)]$
        $s \leftarrow s'$
    end
     until  $s$  is terminal
end
```

2.4.3 Q-learning

Q-learning (Watkins and Dayan 1992) is the most famous RL algorithm for off-policy learning. Q-learning uses a behavior policy to sample transitions (s, a, r, s') from the environment and incrementally estimates Q-values for the state action tuple (s, a). The agent observes a transition (s, a, r, s') and the update takes place on the Q-value of action a in the state s from which this action was executed.

The update rule for this algorithm is:

$$Q_{k+1}(s, a) \leftarrow Q_k(s, a) + \alpha \left(r + \gamma \max_{a'} Q_k(s', a') - Q_k(s, a)\right). \tag{2.20}$$

Q-learning is considered an off-policy method because it uses two separate policies to learn: the behavior policy and the target policy.

The behavior policy, also known as the exploration policy, is used to control the agent's actions in the environment and gather data. This policy is often stochastic, meaning that it selects actions randomly with some probability, to encourage exploration and avoid getting stuck in suboptimal solutions.

The target policy used in Q-learning is the greedy policy:

$$\max_{a' \in \mathcal{A}(s')} Q\left(s', a'\right),$$

which, unlike the behavior policy, does not explore.

Algorithm 2.2 Tabular Q-learning (Watkins and Dayan 1992)

```
Initialize $Q$ arbitrarily
**for** *each episode* **do**
    $s \leftarrow$ initial state of the episode
    **for** *each step of episode* **do**
        $a \leftarrow$ action based on $Q$ and an exploration strategy
        Take action $a$, observe $r, s'$
        $Q(s,a) \leftarrow Q(s,a) + \alpha_k \left(r + \gamma \cdot \max_{a' \in \mathcal{A}(s')} Q\left(s', a'\right) - Q(s,a)\right)$
        $s \leftarrow s'$
    **end**
    until $s$ is terminal
**end**
```

2.5 Deep Reinforcement Learning

The methods we presented so far rely on a tabular representation of the value function V or the action-value function Q. These methods are affected by the curse of dimensionality (Bellman 1957), limiting their applicability to finite state and action spaces. In real world problems, the state space can be large or even continuous and it is not feasible to visit all possible states nor storing large V or Q tables due to memory constraints.

To overcome these restrictions the RL community has proposed the use of deep neural networks (DNNs) (LeCun, Bengio, and Hinton 2015) as function approximations, leading to the Deep Reinforcement Learning (DRL) revolution (Li 2017).

Many of the successes in DRL are based on scaling up prior work in RL to high-dimensional problems, replacing value tables and/or the policy with DNNs. The ability of DNNs to approximate non-linear functions and to extract relevant features from raw inputs allows DRL to generalize over unseen states.

2.5.1 Deep Q Network

Deep Q Network (DQN) (Mnih et al. 2013) combines Q-learning (see Section 2.4.3) with Deep Neural Network function approximation to approximate the optimal action-value function Q^*.

Let us define the neural network parametrized action-value function as Q_θ with parameters θ. DQN updates the parameters θ performing stochastic gradient descent with the squared TD-error loss:

$$\mathcal{L}(\theta) = \mathbb{E}_{(s,a,r,s')\sim D}\left[\left(r_t + \gamma \max_{a'} Q_{\theta^-}\left(s', a'\right) - Q_\theta\left(s, a\right)\right)^2\right], \tag{2.21}$$

where the gradient is:

$$\nabla_\theta \mathcal{L}\left(\theta\right) = -2\mathbb{E}_{(s,a,r,s')\sim D}\left[\left(r_t + \gamma \max_{a'} Q_{\theta^-}\left(s', a'\right) - Q_\theta\left(s, a,\right)\right) \nabla_\theta Q_\theta\left(s, a\right)\right]. \tag{2.22}$$

Here D is a dataset of transitions (s, a, r, s') that is referred to as the replay memory and Q_θ^- is a copy of Q_θ updated at a slower timescale.

The training of the neural network encompasses two additional mechanisms, called experience replay and target network.

Experience Replay: The idea of experience replay is to store transitions (s, a, s', r) in a buffer D and uniformly sampling mini-batches of transitions from D.

Target Network: The second mechanism is a frozen target network that is used to alleviate the fact that we are trying to calculate our loss based on a moving target that changes while learning. Two networks with the same structure, but different weights are used: θ for the Q-network and

θ^- for the target network. The Q-network is regularly updated according to the loss function from equation 3.1, while the target network is updated by copying the parameters of the Q-network to the target network $\theta^- = \theta$ every C time steps. Thus, the weights of the target network θ^- are held frozen for C time steps. The target network θ^- is updated much less frequently than θ in order to increase the stability of learning.

Double DQN

Double DQN (DDQN) (Van Hasselt, Guez, and Silver 2016) has been proposed as an improvement over DQN by fixing the problem of overestimation of Q values.

The new update rule is:

$$\mathcal{L}(\theta) = \mathbb{E}_{(s,a,r,s')\sim D}\left[\left(r + \gamma Q_{\theta^-}(s', \arg\max_{a'} Q_\theta(s', a')) - Q_\theta(s,a)\right)^2\right]. \tag{2.23}$$

The DDQN update rule uses the Q-network to select the action and the target network to evaluate the action. This decoupling of the selection and evaluation processes helps to reduce over-estimation and results in more stable and accurate action value estimates.

Prioritized Experience Replay

One of the possible improvements over DQN is Prioritized Experience Replay (Schaul et al. 2015a)(PER). PER does improve the way experience is sampled. The main idea is that we prefer to learn from transitions (s, a, s', r) that do not fit well over our current approximation of the Q value.

We can define an error e of a sample (s, a, s', r) as a distance between $Q(s, a)$ and its target $T(s, a, r, s')$:

$$e = |Q(s_t, a_t) - T(s, a, r, s')|, \tag{2.24}$$

where $T(s, a, r, s')$ in the case of DDQN would be:

$$T(s, a, r, s') = r + \gamma Q_{\theta^-}(s', \arg\max_{a'} Q_\theta(s', a')). \tag{2.25}$$

This error is then converted to a priority of each sample (s, a, s', r):

$$p = (e + \epsilon)^{\vartheta}, \tag{2.26}$$

where ϵ is a small positive constant that ensures that no transition has zero priority. The parameter ϑ, $0 \leq \vartheta \leq 1$ controls the relative difference between high and low error. It determines how much prioritization is used. With $\vartheta = 0$, we would get the uniform case.

Priority is translated to the probability of being chosen for replay. A sample i has a probability of being picked during the experience replay determined by:

$$P_i = \frac{p_i}{\sum_k p_k}. \tag{2.27}$$

The algorithm is simple - during each learning step we will sample a batch of transitions using this probability distribution and train our network on it.

2.5.2 Policy Gradient Methods

Policy Gradient (PG) methods try to learn a parametrized policy directly without consulting a value function (Sutton et al. 2000) or by using it as a critic for the policy (Konda and Tsitsiklis 1999).

Let us consider a parametrized policy π_θ with parameters θ and some performance measure of this policy $J(\theta)$. PG methods seek to maximize the performance of this measure by performing approximate gradient ascent on J.

$$\theta^* = \arg\max_{\theta} J(\theta), \tag{2.28}$$

with update rule:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \widehat{\nabla J(\boldsymbol{\theta}_t)}, \tag{2.29}$$

where $\widehat{\nabla J(\boldsymbol{\theta}_t)}$ is a stochastic estimate whose expectation approximates the gradient of the performance measure with respect to its argument $\boldsymbol{\theta}_t$.

All methods that follow this general schema are called policy gradient methods, whether or not they also learn an approximate value function.

One advantage of Policy Gradient algorithms is their stable convergence property due to the policy updating directly and thus improving smoothly at each time step. In comparison, Value-Based algorithms update the value function at each time step. A small change in the value function can lead to a drastic change in the policy often resulting in big oscillations during training.

Policy Gradient algorithms can deal with infinite and continuous action spaces. Instead of determining a Q-value for each possible discrete action, the action can be estimated directly with the parametrized policy π_θ. These algorithms offer a natural way to learn stochastic policies, promoting exploration. This is in contrast with many value-based methods, where an explicit exploration strategy, such as ϵ-greedy, must be devised to explore the environment. The major disadvantage of Policy Gradient algorithms is that they often converge to a local maximum (Sutton and Barto 2018).

Policy Gradient Theorem

Let us consider the infinite horizon case where we can define our performance measure as:

$$J(\theta) = \sum_{s \in \mathcal{S}} d^{\pi_\theta}(s) V^{\pi_\theta}(s) = \sum_{s \in \mathcal{S}} \sum_{a \in \mathcal{A}} \pi_\theta(a|s) Q^{\pi_\theta}(s, a), \tag{2.30}$$

where $d^{\pi_\theta}(s) = \lim_{t \to \infty} P(s_t = s | s_0, \pi_\theta)$ is the stationary distribution of the Markov chain for π_θ.

Computing the gradient $\nabla_\theta J(\theta)$ could be hard since it depends on both the policy π_θ and the transition dynamics $\mathcal{P}$ and as we have seen in RL the environment dynamics are usually unknown.

The Policy Gradient theorem (Sutton et al. 2000) provides a nice reformulation of the gradient that does not involve the derivative of the state distribution d^{π_θ}.

$$\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \sum_{s\in\mathcal{S}} d^{\pi_\theta}(s) \sum_{a\in\mathcal{A}} Q^{\pi_\theta}(s,a)\pi_\theta(a \mid s) \\
&\propto \sum_{s\in\mathcal{S}} d^{\pi_\theta}(s) \sum_{a\in\mathcal{A}} Q^{\pi_\theta}(s,a)\nabla_\theta \pi_\theta(a \mid s) \\
&= \sum_{s\in\mathcal{S}} d^{\pi_\theta}(s) \sum_{a\in\mathcal{A}} Q^{\pi_\theta}(s,a)\pi_\theta(a \mid s)\frac{\nabla_\theta \pi_\theta(a \mid s)}{\pi_\theta(a \mid s)} \\
&= \mathbb{E}\left[Q^{\pi_\theta}(s,a)\nabla_\theta \log \pi_\theta(a|s)\right].
\end{aligned} \tag{2.31}$$

The main disadvantage of this approach is the high variance and subsequent algorithms have been proposed to reduce the variance while keeping the bias unchanged (Greensmith, Bartlett, and Baxter 2004; Weaver and Tao 2013; Williams 1992).

REINFORCE

REINFORCE algorithm (Williams 1992) proposes to update a policy through gradient ascent and Monte Carlo sampling leveraging the following equality:

$$Q^{\pi_\theta}(s_t,a_t) = \mathbb{E}[R_t|s_t,a_t]. \tag{2.32}$$

The gradient of $J(\theta)$ then becomes:

$$\begin{aligned}
\nabla_\theta J(\theta) &= \mathbb{E}[Q^{\pi_\theta}(s,a)\nabla_\theta \log \pi_\theta(a|s)] \\
&= \mathbb{E}[R_t \nabla_\theta \log \pi_\theta(a_t|s_t)].
\end{aligned} \tag{2.33}$$

Baseline

The policy gradient theorem can be trivially generalized to include a comparison of the action value to an arbitrary baseline $b(s)$:

$$\nabla J(\theta) \propto \sum_{s\in\mathcal{S}} d^{\pi_\theta}(s) \sum_{a\in\mathcal{A}} (Q^{\pi_\theta}(s,a) - b(s))\nabla_\theta \pi_\theta(a \mid s). \tag{2.34}$$

The baseline can be any function that does not vary with a.

$$\sum_a b(s)\nabla\pi_\theta(a|s) = b(s)\nabla\sum_a \pi_\theta(a|s) = b(s)\nabla 1 = 0. \tag{2.35}$$

In general, the baseline leaves the expected value of the update unchanged, but it can have a large effect on its variance. One natural choice for the baseline is an estimate of the value function V.

Actor Critic Methods

Actor-Critic methods (Konda and Tsitsiklis 1999) combines policy gradient and TD learning. The Critic represents the value function V^π approximated through TD learning, while the Actor represents the current policy updated through policy gradient with baseline $b(s) = V^\pi(s)$ (see Fig. 2.2).

In Actor-Critic algorithms we use a parametrized critic V_ϕ and update the parameters ϕ through gradient descent minimizing the following TD learning loss:

$$\mathcal{L}_{Critic} = \frac{1}{2}\left(V_\phi(s_t) - (r_t + V_\phi(s_{t+1}))\right)^2, \tag{2.36}$$

while the Actor defines a separate neural network to parametrize the policy π_θ and performs gradient descent with baseline $b(s) = V_\phi(s)$ to improve the policy :

$$\mathcal{L}_{Actor} = -(R_t - V_\phi(s_t))\log\pi_\theta(a_t|s_t). \tag{2.37}$$

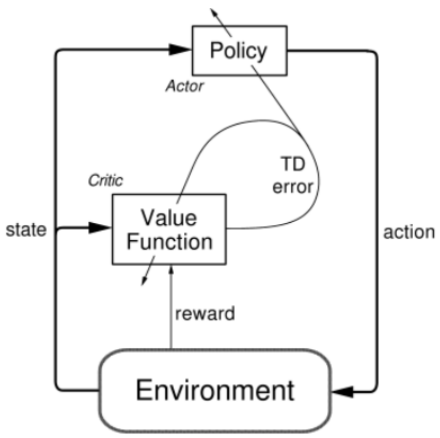


Figure 2.2: Actor-Critic Architecture (Sutton and Barto 2018).

Self Imitation Learning

Self Imitation Learning (SIL) (Oh et al. 2018) is an actor-critic algorithm that leverages an agent's past good experiences to improve exploration.

The basic idea introduced in this paper is to use a Prioritized Experience Replay buffer to exploit past good experiences that can indirectly drive better exploration.

To this end, authors propose to store past episodes with cumulative rewards: $\mathcal{D} = \{(s_t, a_t, R_t)\}$ where s_t, a_t are a state and an action at time-step t, and $R_t = \sum_{k=t}^{\infty} \gamma^{k-t} r_k$ is the discounted sum of rewards. To exploit only good state-action pairs in the replay buffer the following off-policy loss is used:

$$\begin{aligned} \mathcal{L}^{sil} &= \mathbb{E}_{(s,a,R)\in\mathcal{D}}[\mathcal{L}^{sil}_{\text{Actor}} + \beta^{sil}\mathcal{L}^{sil}_{\text{Critic}}] \\ \mathcal{L}^{sil}_{\text{Actor}} &= -\log \pi_\theta(a|s)\,(R - V_\phi(s))_+ \\ \mathcal{L}^{sil}_{\text{Critic}} &= \frac{1}{2}\left((R - V_\phi(s))_+\right)^2, \end{aligned} \tag{2.38}$$

where $(\cdot)_+ = \max(\cdot, 0)$, π_θ, V_ϕ are the Actor and the Critic parameterized respectively by θ and ϕ, and $\beta \in \mathbb{R}^+$ is a hyperparameter controlling the importance of $\mathcal{L}^{sil}_{\text{Critic}}$.

If the return in the past is greater than the agent's value estimate ($R > V_\theta$), the agent learns to choose the same action chosen in the past in the given state. Otherwise, $(R \leq V_\theta)$, and such a state action pair is not used to update the parameter due to the $(\cdot)_+$ operator. This encourages the agent to imitate its own decisions in the past only when such decisions resulted in larger returns than expected.

2.6 Goal Conditioned Reinforcement Learning

An alternative setting that we consider in our work is the setting of Goal-Conditioned Reinforcement Learning (GCRL) (Liu, Zhu, and Zhang 2022). Different from standard RL, GCRL augments the observation with an additional goal that the agent is required to achieve when making a decision in an episode (Andrychowicz et al. 2017; Schaul et al. 2015b).

2.6.1 Setting

In GCRL the agent is required to master multiple tasks simultaneously. To tackle such a challenge we augment the MDP tuple $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, r, \mathcal{G}, p_g \rangle$, where $\mathcal{G} \subseteq \mathcal{S}$ denotes the space of goals describing the tasks and p_g represents the desired goal distribution of the environment and the reward function $r : \mathcal{S} \times \mathcal{A} \times \mathcal{G} \to \mathbb{R}$ is defined on goals $\mathcal{G}$. Therefore the objective of GCRL is to reach goal states via a goal-conditioned policy $\pi : \mathcal{S} \times \mathcal{G} \to \Delta(\mathcal{A})$.

2.6.2 Goal Conditioned Supervised Learning

When we consider the goal space to be equal to the state space $G = S$ we can treat any trajectory $\tau = \{s_0, a_0, ..., a_t, s_{t+1}\}$ and any sub-trajectory $\tau_{i,j} = \{s_i, a_i, ..., a_j, s_{j+1}\}$ as a successful trial for reaching their final states. Goal Conditioned Supervised Learning (GCSL) (Ghosh et al. 2019) iteratively performs behavioral cloning on sub-trajectories collected in a dataset $\mathcal{D}$ by learning a policy π conditioned on both the goal and the horizon h.

Formally, the policy is performing maximum-likelihood estimation (MLE) via supervised learning.

$$\arg\max_\theta \mathbb{E}_{((s,g,h),a)\sim\mathcal{D}}[\log \pi_\theta(a|s, g, h)]. \tag{2.39}$$

Chapter 3

Hierarchical Reinforcement Learning

In this chapter, we will review Hierarchical Reinforcement Learning (HRL) approaches with a focus on algorithms that are of interest to the work presented in this thesis.

3.1 Introduction

Recently, Deep Reinforcement Learning (DRL) has achieved significant success in many domains (Bellemare et al. 2020; Mnih et al. 2013; Silver et al. 2016; Vinyals et al. 2019). Nevertheless, it is still a challenge for DRL agents to solve environments with sparse rewards and long time horizons, such as Minecraft (Guss et al. 2019; Johnson et al. 2016).

Hierarchical reinforcement learning holds the promise to reduce the complexity of solving long horizon and sparse rewards environments (Bacon, Harb, and Precup 2017; Barto and Mahadevan 2003; Dayan and Hinton 1992; Dietterich 2000; Kaelbling 1993; Nachum et al. 2018a; Parr and Russell 1997; Sutton, Precup, and Singh 1999; Vezhnevets et al. 2017; Wen et al. 2020). HRL works by reducing complex problems to a smaller set of interrelated problems. The smaller problems are solved separately and the results are re-combined to find a solution to the original problem. This hierarchical decomposition can achieve multi-level control where long-horizon planning and high-level meta-learning guide the lower-level controllers. The modularization of hierarchical structures also allow transferability and interpretability.

3.2 Four-Room Task

Throughout this Chapter, we will use a simple four-room task as a running example to help illustrate concepts.

In the four-room task in Figure 3.1, the four rooms are labeled $R1$, $R2$, $R3$, $R4$, the agent is represented as the red block labeled as A in room $R1$ and the doorways are the green blocks labeled $D1$, $D2$, $D3$, $D4$. The goal of the agent is to reach the blue cell labeled G where the episode terminates. Each cell represents a possible agent position and the position is uniquely described by the position in each room and the room id $(x, y, room\_id)$. The rooms have the same dimensions and similar positions in each room are assumed to be described by the same (x, y) coordinates. The agent can move one step in any of the four cardinal directions. When an action is taken there is an 80% chance that the agent will move in the intended direction and a 20% chance that it will stay in place.

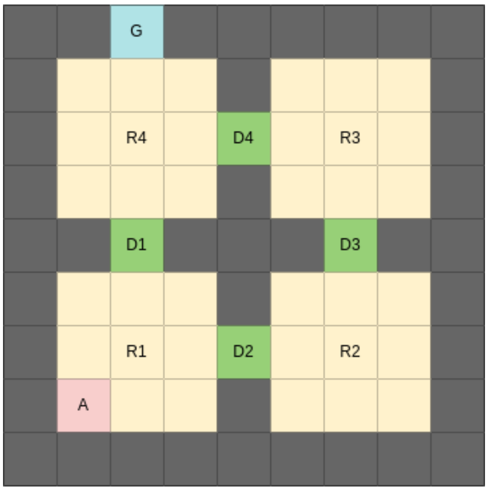


Figure 3.1: The four-room task in which an agent (A) has to reach the goal (G). Green blocks ($D1$, $D2$, $D3$, $D4$) represents doorways that the agent has to cross to move between rooms ($R1$, $R2$, $R3$, $R4$) in the environment.

3.3 Temporally Extended Actions

HRL employs actions that persist for multiple time steps. Once selected, a temporally extended (TE) action hides the multistep transition and reward detail until termination. As an example, consider the four-room task (Figure 3.1) where we can define a temporally extended action in room $R1$ that moves the agent from $R1$ to $D1$. This TE action can be seen as a subtask that the agent needs to solve to reach its goal G. Moreover, we can learn a "local" policy to perform this TE action that will consider a smaller MDP comprising only the states of $R1 \cup D2$.

Special case TE actions that terminate in one time step are just ordinary actions, and we refer to them as primitive actions.

3.4 Semi-Markov Decision Processes

We can extend the MDP formulation to include temporally extended actions, and MDPs that include TE actions are called semi-Markov Decision Processes (SMDPs) (Puterman 2014).

We define a random variable $N \geq 1$ to be the number of time steps that a TE action a takes to complete, starting in state s and terminating in state s'.

A discrete-time SMDP is a tuple $\mathbb{S} = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, r \rangle$ where $\mathcal{S}$ represent the state space, $\mathcal{A}$ represent the action space that can include temporally extended actions and primitive actions.

The transition function $\mathcal{P} : \mathcal{S} \times \mathcal{A} \rightarrow \Delta(\mathcal{S} \times N)$ gives the probability of the TE action a terminating in state s' after N steps, having been initiated in state s.

$$\mathcal{P}(s, a, s', N) = P(s_{t+N} = s' | s_t = s, a_t = a). \tag{3.1}$$

The reward function $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \times N \rightarrow \mathbb{R}$ accumulates single-step rewards according to the optimality criteria selected. Here we consider the infinite horizon discounted case where:

$$r(s, a, s', N) = \mathbb{E}\left[\sum_{n=0}^{N-1} \gamma^n r_{t+n} | s_t = s, a_t = a, s_{t+N} = s'\right]. \tag{3.2}$$

The value function V^π can also be generalized for SMDPs:

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in S, N} \mathcal{P}(s, a, s', N)(r(s, a, s', N) + \gamma^N V^\pi(s')). \tag{3.3}$$

The optimum value function for an SMDP can then be defined:

$$V^*(s) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}, N} \mathcal{P}(s, a, s', N)(r(s, a, s', N) + \gamma^N V^*(s')). \tag{3.4}$$

Similarly, we can define the optimal Q-function Q^*:

$$Q^*(s, a) = \sum_{s' \in S, N} \mathcal{P}(s, a, s', N)(r(s, a, s', N) + \gamma^N V^*(s')), \tag{3.5}$$

where $V^*(s') = \max_{a'} Q^*(s', a')$.

3.4.1 SMDP Q learning

All the methods developed for solving Markov decision processes in Chapter 2 for reinforcement learning using primitive actions can be applied to problems using TE actions.

Here we highlight the extension of Q learning to SMDPs that is of interest in our work. We refer to it as SMDP Q learning (Sutton, Precup, and Singh 1999).

Under the SMDP model, decisions are taken at certain decision times spaced by random time intervals, we observe transition (s, a, r, N, s') and the update rule for SMDP Q-learning becomes:

$$Q_{k+1}(s,a) \leftarrow Q_k(s,a) + \alpha_k \left(r + \gamma^N \max_{a'} Q_k(s',a') - Q_k(s,a) \right), \quad (3.6)$$

where the update is performed on a transition from state s to s' under TE action a that has taken time N and received reward $r = \sum_{n=0}^{N-1} \gamma^n r_{t+n}$. As in Chapter 2 α_k is the learning rate.

3.5 Structure

Temporally extended actions together with SMDPs naturally lead to a hierarchical structure. With TE actions we may be able to learn a policy with less effort than it would take to solve the problem using primitive actions.

As an example consider the four-room task and the four *room−leaving* temporally extended actions in Figure 3.2. The four TE actions are applicable in any room and when successful move the agent from the actual room to a neighboring room. Consider the TE action to leave the room through the north doorway. When selected in a room $R1$ this action will move the agent through the doorway $D1$ to room $R4$.

TE actions themselves in this case are policies for smaller MDPs of the size of a single room and can be "transferred" to any room in the four-room task.

At the high level, we can use the $room - leaving$ TE actions to reduce the task to an SMDP with 4 states to represent only the rooms $R1$, $R2$, $R3$, $R4$.

The parent-child relationship between SMDPs that we have seen leads to task-hierarchies (Dietterich 2000). A task-hierarchy is a directed acyclic graph of sub-tasks. The root-node is the top-level SMDP that can invoke its child-node SMDP/MDP policies as TE actions.

The benefit of decomposing a large MDP through task hierarchies is that it will hopefully lead to state abstraction opportunities to help reduce the complexity of the problem. An abstracted state space is smaller than the state space of an original MDP.

In HRL the task-hierarchy is usually assumed to be given and automatically discovering hierarchies for HRL is an active area of research (Bacon, Harb, and Precup 2017; Bar, Talmon, and Meir 2020; Barto and Mahadevan 2003; Jinnai et al. 2019a,b; Machado, Bellemare, and Bowling 2017; Mannor et al. 2004; Nachum et al. 2018a,b; Vezhnevets et al. 2017; Wan and Sutton 2022).

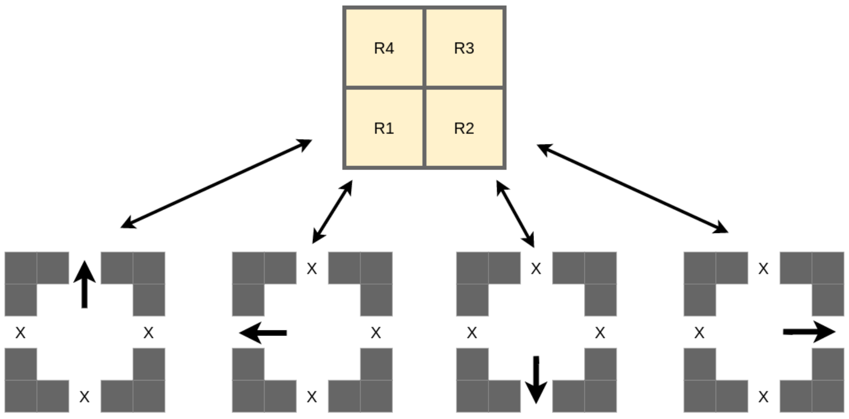


Figure 3.2: An example of hierarchy decomposing the four-room task. X cells represent terminal states.

3.6 Optimality

In Hierarchical Reinforcement Learning we have to depart from the notion of optimal policy π^* we have seen in Chapter 2 since we can not guarantee that the decomposed problem will in general yield the optimal solution. It depends on the problem and the quality of the decomposition in terms of the TE actions available and the structure of the task-hierarchy.

- **Hiearchical Optimal**: A hierarchical optimal policy for MDP $\mathcal{M}$ is a policy that archives the highest cumulative reward among all policies consistent with the given hierarchy (Dietterich 2000).

- **Recursively Optimal**: In recursive optimality (RO) the final policy is optimal given the policies learned by its sub-task children. In RO the sub-task policies to reach the goal terminal state are context-free and only locally optimal ignoring the needs of their parent tasks. This formulation has the advantage that sub-tasks can be re-used in various contexts, but they may not, therefore, be optimal in each situation (Dietterich 2000).

Consider the simple example in Figure 3.3 that demonstrates the difference between recursively and hierarchically optimal policies (Dietterich 2000; Ghavamzadeh and Mahadevan 2002). Suppose a robot starts somewhere in the left room and it must reach the goal G in the right room. In addition to three primitive actions, North, South and East, the robot have a high level task $room - leaving$ in the left room and a high-level task $go - to - goal$ in the right room. Task $room - leaving$ terminates when the robot exits the left room and $go-to-goal$ terminates when the robot reaches the goal G. The arrows in Figure 3.3(a) shows the locally optimal policy within each room. The arrows in the left room seek to exit the room by the shortest path. The arrows in the right room follow the shortest path to the goal. However, the resulting policy is not hierarchically optimal. Figure 3.3(b) shows the hierarchically optimal policy that would always exit the left room by the upper door. This policy would not be locally optimal because the states in the shaded region would not follow the shortest path to a doorway.

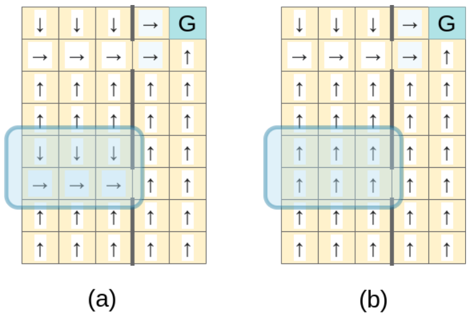


Figure 3.3: The policy shown in the left diagram is recursively optimal but not hierarchically optimal. The policy in the right diagram is hierarchically optimal but not recursively optimal. The shaded cells indicate states where the two definitions of optimality disagree (Dietterich 2000; Ghavamzadeh and Mahadevan 2002).

3.7 Bottleneck States

Bottlenecks states have been the first response to the need of discovering TA actions automatically (Barto and Mahadevan 2003). Bottlenecks have been defined as those states which appear frequently on successful trajectories to a goal but not on unsuccessful ones (McGovern and Barto 2001) or as nodes that allow for densely connected regions of the interaction graph to reach other such regions (Menache, Mannor, and Shimkin 2002; Şimşek and Barto 2004).

An alternative point of view is through the network centrality measure called betweenness (Şimşek and Barto 2008). Betweenness centrality measures the fraction of shortest paths passing through a given vertex of the graph. Nodes with high betweenness are deemed more important as they would appear more frequently on the shortest path, therefore more likely to be optimal trajectories to a goal.

The canonical HRL domain for motivating the bottleneck concept has been mainly concerned with the four-room task (Sutton, Precup, and Singh 1999) (see Figure 3.1) where we can identify four bottleneck states $D1$, $D2$, $D3$, $D4$. These states connect two nearby rooms, so they connect strongly connected regions.

3.8 Options

The options framework is at the crossroads of MDPs and SMDPs. It considers a base MDP overlaid with variable-length TE actions represented as options. It is shown by Sutton, Precup, and Singh (1999) (Theorem 1) how an MDP and a pre-defined set of options form a semi-Markov Decision Process. Most of the theory on SMDPs can thus be reused for options. As opposed to the usual theory of SMDPs that treats TE actions as indivisible and opaque decision units, the options framework allows us to look at the structure within.

Options formalize temporally extended actions as a triplet $\langle \mathcal{I}^o, \pi^o, \beta^o \rangle$, where $\mathcal{I}^o \subseteq \mathcal{S}$ is an initiation set, $\pi^o : \mathcal{S} \to \Delta(\mathcal{A})$ is a policy, and $\beta^o : \mathcal{S} \to [0, 1]$ is a termination condition.

An option o can be executed in a state $s \in \mathcal{S}$ only if $s \in \mathcal{I}^o$. This allows restricting the starting states only to a subset of $\mathcal{S}$. Once chosen, the option generates a trajectory τ following the policy π^o. The option terminates according to a function β^o.

Based on the termination condition we distinguish two types of options:

1. Markov options: the termination function β^o depends only on the current state s.

2. Semi-Markov options: the termination function β^o depends on some other factors apart from the current state s (i.e. the full history of states).

Given the definition of option, it is easy to see that a primitive action does fit in this formalism. A primitive action a is an option $\langle \mathcal{I}^o = \mathcal{S}, \pi^o(s, a) = 1.0, \beta^o(s) = 1 \rangle$ for all $s \in \mathcal{S}$.

Highlighting the formalism of a primitive action as an option, we can see that the option policies π^o can call options, allowing the possibility to create hierarchical structures of an arbitrary depth.

Part II

State Space Partitioning and Option Learning

Chapter 4

State Space Partitioning

The description of state space partitioning in HRL traces back to the work of Dietterich (2000) that defines two conditions under which state abstraction for Hierarchical Reinforcement Learning can be introduced:

- Subtask Irrelevance.
- Abstract actions that "funnel" the agent to a small subset of states.

Subtask Irrelevance consists of the elimination of irrelevant variables for a subtask. As an example consider the task-hierarchy (see Figure 3.2) on the four-room task (see Figure 3.1). At the low level, we have four $room - leaving$ temporally extended (TE) actions, while at the high level, we have an SMDP with 4 states to represent only the rooms $R1$, $R2$, $R3$, $R4$. The four-room MDP state space is described by three variables $(x, y, room\_id)$. Subtask Irrelevance allows us to ignore the $room\_id$ state variable at the low level and ignore the (x, y) state variables at the high level. This condition opens up the possibility to transfer and reuse subtasks and can be exploited in state space partitioning by partitioning the state space into subtasks that repeat across the MDP.

The second condition highlighted by Dietterich (2000) is the funneling state abstraction described as a type of state abstraction where TE actions move the environment from a large number of initial states to a small number of resulting states. Funnelling can be observed in the four-room environment. $Room - leaving$ TE actions move the agent from any position in a room to the respective doorway. Funnelling allows the four-room task to be state-abstracted at the root node to just 4 states (see Figure 3.2), because irrespective of the starting position in each room, the TE actions have the ability to move the agent via doorways and reach the goal state.

Based on these guiding principles in this chapter we will highlight the desired characteristics of a state partition that can be used for Hierarchical Reinforcement Learning.

4.1 State Space Partitions

State space partitions on an MDP allow a large problem to be broken down into sub-problems, which could be tackled and solved independently. Sub-problem solutions could then be "stitched" together to (approximately) solve the entire problem. Wen et al. (2020) proved that if these sub-problems are relatively small and repeated, this approach can lead to large computational gains.

Definition 1. *Given an episodic MDP $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, r, \mathcal{S}_e \rangle$, where $\mathcal{S}_e$ is a set of terminal states, consider a partition of the non-terminal states $S \setminus \{\mathcal{S}_e\}$ into L disjoint subsets $\mathcal{Z} = \{\mathcal{S}_i\}_{i=1}^{L}$, i.e. $\mathcal{S} \setminus \mathcal{S}_e = \mathcal{S}_1 \cup \cdots \cup \mathcal{S}_L$ and $\mathcal{S}_i \cap \mathcal{S}_j = \emptyset$ for each pair $(\mathcal{S}_i, \mathcal{S}_j) \in \mathcal{Z}^2$.*

We define an induced subMDP $\mathcal{M}_i = \langle \mathcal{S}_i \cup \mathcal{E}_i, \mathcal{A}, \mathcal{P}_i, r_i, \mathcal{E}_i \rangle$ as follows:

- *$\mathcal{S}_i$ is the internal state set, and the action space is still $\mathcal{A}$.*
- *The exit state set $\mathcal{E}_i$ is defined as $\mathcal{E}_i = \{e \in \mathcal{S} \backslash \mathcal{S}_i : \exists (s,a) \in \mathcal{S}_i \times \mathcal{A}$ s.t. $P(e \mid s,a) > 0\}$. Is important to notice that the exit state set $\mathcal{E}_i$ will belong to a different partition in $S_j \in \mathcal{Z}$ with $j \neq i$ that is reachable in one step from some state in $\mathcal{S}_i$.*
- *The state space of $\mathcal{M}_i$ is $\mathcal{S}_i \cup \mathcal{E}_i$.*
- *$\mathcal{P}_i : \mathcal{S}_i \times \mathcal{A} \to \Delta(\mathcal{S}_i \cup \mathcal{E}_i)$ and $r_i : \mathcal{S}_i \times \mathcal{A} \times (\mathcal{S}_i \cup \mathcal{E}_i) \to \mathbb{R}$ are respectively the restriction of $\mathcal{P}$ and r to domain $\mathcal{S}_i \times \mathcal{A}$.*
- *The subMDP $\mathcal{M}_i$ terminates once it reaches a state in $\mathcal{E}_i$ (i.e., an exit state).*

4.2 State Space Partitions Properties

We identify several properties that a state partition should have in order to define a good representation for Hierarchical Reinforcement Learning.

- **Induced subMDP must be easy to solve**:

  The maximum size of an induced subMDP M is defined as:

  Definition 2. $M = \max_i |S_i \cup \mathcal{E}_i|$.

  If M is small, all subMDPs have small size $|\mathcal{S}_i \cup \mathcal{E}_i| \leq M$, so they would be relatively easy to solve. This definition characterizes the hardness in terms of state space size of a subMDP. Complexity results with tabular representation have shown that finite MDPs can be solved in polynomial time in the size of the state space and action space (Littman, Dean, and Kaelbling 2013) when the transition matrix $\mathcal{P}$ is known, proving that the smaller the state space size is, the easier it is to solve the MDP.

  Note that this is the easiest and most general definition of hardness since it does not take into account the reward r_i nor transition probability $\mathcal{P}_i$ inside the subMDPs.

- **Equivalent SubMDPs**:

  Two subMDPs $\mathcal{M}_i$ and $\mathcal{M}_j$ are equivalent if there is a bijection between S_i, $\mathcal{E}_i$ and S_j, $\mathcal{E}_j$, such that the subMDPs have the same transition probabilities and rewards at internal states.

  Definition 3. *Two subMDPs $\mathcal{M}_i$ and $\mathcal{M}_j$ are equivalent if there is a bijection $f : \mathcal{S}_i \cup \mathcal{E}_i \to \mathcal{S}_j \cup \mathcal{E}_j$ s.t. $f(\mathcal{S}_i) = S_j, f(\mathcal{E}_i) = \mathcal{E}_j$, and, through f, the subMDPs have the same transition probabilities and rewards at internal states.*

  Note that the constraints $f(\mathcal{S}_i) = \mathcal{S}_i$ and $f(\mathcal{E}_i) = \mathcal{E}_j$ ensure that each internal (or exit) state in $\mathcal{M}_i$ is mapped to each internal (or exit) state in $\mathcal{M}_j$.

  Let $K \leq L$ be the number of equivalence classes of subMDPs induced by a particular partition $\mathcal{Z}$ of $\mathcal{M}$. When there is no repeatable structure, $K = L$. When the partition produces repeatable structure, $K < L$.

A desirable state partition $\mathcal{Z}$ is such a partition where $K \ll L$. With such a partition we can learn to solve a single subMDP and reuse the solution on many partitions of the state space $\mathcal{S}$.

- **Bottleneck states**:

  The set of all exit states for a given partition is defined as:

  Definition 4. $\mathcal{E} = \cup_i^L \mathcal{E}_i$.

  If $|\mathcal{E}|$ is small, intuitively we have a few states that connect the sub-problems. We can think of these as "bottleneck" states in $\mathcal{M}$, which have been shown before to enable computationally efficient planning (see e.g. (McGovern and Barto 2001; Şimşek and Barto 2008; Solway et al. 2014; Stolle and Precup 2002; Sutton, Precup, and Singh 1999)).

  We highlight that a trade-off exists between the maximal size of an induced subMDP M and the size of the set of all exit states $|\mathcal{E}|$. It is desirable for M to be small to simplify the solving process for every subMDP. However, trivially partitioning every single state as an independent subMDP can result in a significant increase in the size of $|\mathcal{E}|$.

- **Strongly connected SubMDPs**

  Another desired property of a partition $\mathcal{Z}$ is that it should induce strongly-connected subMDPs.

  Definition 5. *A subMDP $\mathcal{M}_i$ is strongly connected if for each pair of states $s_i, s_j \in \mathcal{S}_i \cup \mathcal{E}_i$ there exists a policy π which, when starting in s_i, reaches s_j with a positive probability.*

  This property ensures that when we enter a subMDP $\mathcal{M}_i$ we can choose a policy π that with positive probability will let us reach a desired exit state $e \in \mathcal{E}_i$. As an example consider the partition in Fig 4.1. Here the partition creates regions that are not strongly connected.

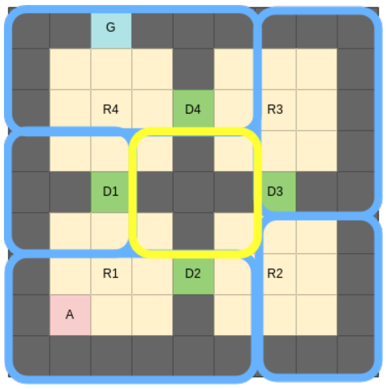


Figure 4.1: The four-room task with a state space partition superimposed in blue. The partition highlighted in yellow violates the strongly connected condition.

Chapter 5

Hierarchical reinforcement learning for exploration and transfer

In this chapter, we present our first contribution: a novel hierarchical reinforcement learning framework based on the compression of an invariant state space that is common to a range of tasks.

We use a fixed, state-dependent compression function to define a hierarchical decomposition of complex, sparse-reward tasks. The agent defines subtasks which consist in navigating across state-space partitions by jointly learning the policy of each temporally extended action. The compression function makes it possible to use tabular methods at the top level to effectively explore large state spaces even in sparse reward settings. Furthermore, we show that our method is suitable for transfer learning across tasks that are defined by introducing additional learning components.

State Space Partition properties

Here we summarize and briefly discuss only the state space partition properties (see Chapter 4) of the methodology presented in this chapter. We will follow this procedure for each chapter of Part II.

Pros:

✓ **Small SubMDPs**: The compression function $f : \mathcal{S}^{\mathcal{I}} \rightarrow \mathbb{N}_{+}$ proposed in this chapter is assumed to have access to the invariant part of the state space $\mathcal{S}^{\mathcal{I}}$ (see Section 5.1.2). The invariant part of the state space $\mathcal{S}^{\mathcal{I}}$ is defined as the part of the state space that is common to multiple tasks in the environment. This is achieved for example in a grid world environment by assuming that we have access to the (x, y) position of the agent in the grid, and the compression of $\mathcal{S}^{\mathcal{I}}$ is simply achieved by an integer division of these two coordinates $f(x, y) = \lfloor x/d_{int} \rfloor + \lfloor y/d_{int} \rfloor$. Tuning the hyperparameter d_{int} allows us to control the maximum size M(see Section 4.2) of each induced subMDP.

✓ **Equivalent subMDPS**: The compression function $f : \mathcal{S}^{\mathcal{I}} \rightarrow \mathbb{N}_+$ creates different regions z only for the invariant part of the state space $\mathbb{S}^{\mathcal{I}}$. This implies that we will have equivalent SubMDPs (see Section 4.2). As we can see in the Key-Treasure example (Figure 5.1) we consider to be in the same subMDP either if the agent collected or didn't collect the key.

Cons:

× **Prior knowledge of invariant state space and compression function**: The main drawback of the state partitioning approach presented in this chapter is that it needs prior knowledge about the invariant state space $\mathcal{S}^{\mathcal{I}}$ and compression function f.

× **Bottleneck exit states**: The compression function presented here is not able to create partition regions that minimize the set of all exit states $\mathcal{E}$ (see Section 4.2). In this work, we consider solely state space partitions that simply divide the invariant state space features into equal-sized partitions. This limits the need for prior knowledge of the MDP with the drawback of not being able to represent more complex partitions that take into account bottleneck exit states.

× **Strongly connected subMDPs**: Partitioning the state space simply by an integer division over the invariant state space features does not guarantee that each subMDP state space $\mathcal{S}_z \cup \mathcal{E}_z$ is going to be strongly connected(see Section 4.2).

5.1 Methodology

5.1.1 Task MDPs

We assume that each task $\mathcal{T}$ is described by an MDP $\mathcal{M}^{\mathcal{T}} = \langle \mathcal{S}^{\mathcal{I}} \times \mathcal{S}^{\mathcal{T}}, \mathcal{A}^{\mathcal{I}} \cup \mathcal{A}^{\mathcal{T}}, \mathcal{P}^{\mathcal{I}} \cup \mathcal{P}^{\mathcal{T}}, r^{\mathcal{T}} \rangle$. Crucially, the state-action space $\mathcal{S}^{\mathcal{I}} \times \mathcal{A}^{\mathcal{I}}$ as well as the transition kernel $\mathcal{P}^{\mathcal{I}} : \mathcal{S}^{\mathcal{I}} \times \mathcal{A}^{\mathcal{I}} \to \Delta(\mathcal{S}^{\mathcal{I}})$ are *invariant*, i.e. shared among all tasks. On the other hand, the state-action space $\mathcal{S}^{\mathcal{T}} \times \mathcal{A}^{\mathcal{T}}$, reward function $r^{\mathcal{T}} : \mathcal{S}^{\mathcal{T}} \times \mathcal{A}^{\mathcal{T}} \to \mathbb{R}$ and transition kernel $\mathcal{P}^{\mathcal{T}} : (\mathcal{S}^{\mathcal{I}} \cup \mathcal{S}^{\mathcal{T}}) \times \mathcal{A}^{\mathcal{T}} \to \Delta(\mathcal{S}^{\mathcal{T}})$ are task-specific. We assume that actions in $\mathcal{A}^{\mathcal{I}}$ incur zero reward, and that states in $\mathcal{S}^{\mathcal{I}}$ are unaffected by actions in $\mathcal{A}^{\mathcal{T}}$. Task $\mathcal{T}$ is only coupled to the invariant MDP through the transition kernel $\mathcal{P}^{\mathcal{T}}$, since the effect of actions in $\mathcal{A}^{\mathcal{T}}$ depend on the states in $\mathcal{S}^{\mathcal{I}}$.

5.1.2 Invariant SMDP

We further assume that the agent has access to a partition $\mathcal{Z} = \left\{\mathcal{S}_i^{\mathcal{I}}\right\}_{i=1}^{L}$ of the invariant state space, i.e. $\mathcal{S}^{\mathcal{I}} = \mathcal{S}_1^{\mathcal{I}} \cup \cdots \cup \mathcal{S}_L^{\mathcal{I}}$ and $\mathcal{S}_i^{\mathcal{I}} \cap \mathcal{S}_j^{\mathcal{I}} = \emptyset$ for each pair $(\mathcal{S}_i^{\mathcal{I}}, \mathcal{S}_j^{\mathcal{I}}) \in \mathcal{Z}^2$. Even though each element of $\mathcal{Z}$ is a subset of $\mathcal{S}^{\mathcal{I}}$, we often use lower-case letters to denote elements of $\mathcal{Z}$, and we refer to each element $z \in \mathcal{Z}$ as a *region*. We use the partition $\mathcal{Z}$ to form an SMDP over the invariant part of the state-action space. This SMDP is defined as $\mathbb{S}^{\mathcal{I}} = \langle \mathcal{Z}, \mathcal{O}, \mathcal{P}_{\mathcal{Z}} \rangle$, where $\mathcal{Z}$ is the set of regions, $\mathcal{O}$ is a set of options, and $\mathcal{P}_{\mathcal{Z}} : \mathcal{Z} \times O \to \Delta(\mathcal{Z})$ is a transition kernel.

We first define the set of neighbors of a region $z \in \mathcal{Z}$ as:

$$\mathcal{N}(z) = \{z' : \exists (s, a, s') \in z \times \mathcal{A}^{\mathcal{I}} \times z', \mathcal{P}^{\mathcal{I}}(s'|s, a) > 0\}.$$

Hence, neighbors of z can be reached in one step from some state in z. For each neighbor $z' \in \mathcal{N}(z)$, we define an option $o_{z,z'} = \langle \mathcal{I}_z, \pi_{z,z'}, \beta_z \rangle$ whose subtask is to reach region z' from z. Hence, the initiation set is z, i.e. $\mathcal{I}_{\ddagger} = \mathcal{S}_z$, the termination function is $\beta_z(s) = 0$ if $s \in z$ and $\beta_z(s) = 1$ otherwise, and the policy $\pi_{z,z'}$ should reach region z' as quickly as possible.

The set of options available to the agent in the region $z \in \mathcal{Z}$ is $\mathcal{O}_z = \{o_{z,z'} : z' \in \mathcal{N}(z)\} \subseteq O$, i.e. all options that can be initiated in z and that transition to a neighbor of z. Note that the option sets $\mathcal{O}_z$ are disjoint, i.e. each region z has its own set of admissible options. The transition kernel $\mathcal{P}_{\mathcal{Z}}$ determines how successful the options are; ideally, $\mathcal{P}_{\mathcal{Z}}(z'|z, o_{z,z'})$

should be close to 1 for each pair of neighboring regions (z, z'), but can be smaller to reflect that $o_{z,z'}$ sometimes ends up in a region different from z'.

5.1.3 Option MDPs

We do not assume that the policy $\pi_{z,z'}$ of each option $o_{z,z'}$ is given; rather, the agent has to learn the policy $\pi_{z,z'}$ from experience. For this purpose, we define an option-specific MDP $\mathcal{M}_{z,z'} = \langle \mathcal{S}_z, \mathcal{A}^{\mathcal{I}}, \mathcal{P}_z, r_{z,z'} \rangle$ associated with option $o_{z,z'}$. Here, the state space $\mathcal{S}_z = z \cup \mathcal{N}(z)$ consists of all states in the region z, plus all the neighboring regions of z. The set of actions $\mathcal{A}^{\mathcal{I}}$ are those of the invariant part of the state-action space. All the states in $\mathcal{N}(z)$ are terminal states. The transition kernel $\mathcal{P}_z$ is a projection of the invariant transition kernel $\mathcal{P}^{\mathcal{I}}$ onto the state-action space $z \times \mathcal{A}^{\mathcal{I}}$ involving non-terminal states, and is defined as:

$$\mathcal{P}_z(s'|s,a) = \begin{cases} \mathcal{P}^{\mathcal{I}}(s'|s,a), & \text{if } s' \in z, \\ \sum_{s'' \in s'} \mathcal{P}^{\mathcal{I}}(s''|s,a) & \text{if } s' \in \mathcal{N}(z). \end{cases}$$

Hence, the probability of transitioning to a neighbor s' of z is the sum of probabilities of transitioning to any state in s'.

In the definition of $\mathcal{M}_{z,z'}$, the state-action space $\mathcal{S}_z \times \mathcal{A}^{\mathcal{I}}$ and transition kernel $\mathcal{P}_z$ are shared among all options in $\mathcal{O}_z$. They only differ in the reward function $r_{z,z'} : z \times \mathcal{A}^{\mathcal{I}} \times S_z \to \mathbb{R}$ defined on triples (s, a, s'), i.e. it depends on the resulting next state s'. The theory of MDPs easily extends to this case. Specifically, the reward function $r_{z,z'}$ is defined as

$$r_{z,z'}(s,a,s') = \begin{cases} +1, & \text{if } s' = z', \\ -0.1, & \text{if } s' \in \mathcal{N}(z) \setminus \{z'\}. \end{cases} \tag{5.1}$$

In other words, successfully terminating in the region z' is awarded a reward of $+1$ while terminating in a region different from z' is penalized with a reward of -0.1. If the option can't terminate within a time limit of 100 steps the same negative reward of -0.1 is given. In practice, option $o_{z,z'}$ can compute the policy $\pi_{z,z'}$ indirectly by maintaining a value function $V_{z,z'}$ associated to the option MDP $\mathcal{M}_{z,z'}$.

5.1.4 Algorithm

In practice, we do not assume that the agent has access to the invariant SMDP $\mathbb{S}^{\mathcal{I}} = \langle Z, \mathcal{O}, \mathcal{P}_{\mathcal{Z}} \rangle$. Instead, the agent can only observe the current state $s \in \mathcal{S}^{\mathcal{I}}$, select an action $a \in \mathcal{A}^{\mathcal{I}}$, and observe the next state $s' \sim \mathcal{P}^{\mathcal{I}}(\cdot|s, a)$. Rather than observing regions in $\mathcal{Z}$, the agent has oracle access to a compression function $f : \mathcal{S}^{\mathcal{I}} \to \mathbb{N}_+$ from invariant states to non-negative integers. Each region z has an associated integer ID $ID(z)$ and is implicitly defined as $z = \{s \in \mathcal{S}^{\mathcal{I}} : f(s) = ID(z)\}$. To identify regions, the agent has to repeatedly query the function f on observed states and store the integers returned. By abuse of notation, we often use z to denote both a region in $\mathcal{Z}$ and its associated ID.

Our algorithm iteratively grows an estimate of the invariant SMDP $\mathbb{S}^{\mathcal{I}} = \langle \mathcal{Z}, \mathcal{O}, \mathcal{P}_{\mathcal{Z}} \rangle$. Initially, the agent only observes a single state $s \in \mathcal{S}$ and associated region $z = f(s)$. Hence the state space $\mathcal{Z}$ contains a single region z, whose associated option set $O_{\mathcal{Z}}$ is initially empty. In this case, the only alternative available to the agent is to *explore*. For each region z, we add an exploration option $o_z^e = \langle \mathcal{I}_z, \pi_z^e, \beta_z \rangle$ to the option set $\mathcal{O}$. This option has the same initiation set and termination condition as the options in O_z, but the policy π_z^e is an exploration policy that selects actions at random or implements a more advanced exploration strategy.

Once the agent discovers a neighboring region z' of z, it adds region z' to the set $\mathcal{Z}$ and the associated option $o_{z,z'}$ to the option set O. The agent also maintains and updates a directed graph whose nodes are regions and whose edges represent the neighbor relation. Hence next time the agent visits region z, one of its available actions is to select the option $o_{z,z'}$. When option $o_{z,z'}$ is selected, it chooses actions using its policy $\pi_{z,z'}$ and simultaneously updates $\pi_{z,z'}$ based on the rewards of the option MDP $\mathcal{M}_{z,z'}$. Figure 5.1 shows an example representation discovered by the algorithm.

Algorithm 5.1 shows the pseudocode of the algorithm. As explained, $\mathcal{Z}$ is initialized with the region z of the initial state s, and $\mathcal{O}$ is initialized with the exploration option o_z^e. In each iteration, the algorithm selects an option o which is applicable in the current region z. This option then runs from the current state s until terminating in a state s' whose associated region z' is different from z. If this is the first time region z' has been observed, it is added to $\mathcal{Z}$ and the exploration option $o_{z'}^e$ is appended to

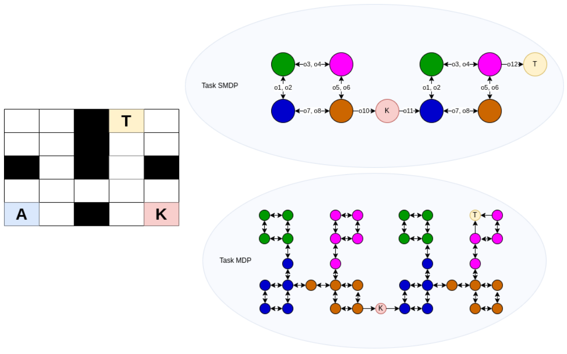


Figure 5.1: Example of an SMDP learned on top of a simple grid world Key-treasure environment. In this environment, agent A has to collect the key K and open the treasure T, black cells represent walls. Each different region is represented with a different color and as we can notice we have the same regions either if we do have or we don't have the key.

$\mathcal{O}$. If this is the first time z' has been reached from z, the option $o_{z,z'}$ is appended to $\mathcal{O}$. The process then repeats from state s' and region z'.

The subroutine GetOption that selects an option o in the current region z can be implemented in different ways. If the aim is just to estimate the invariant SMDP $\mathbb{S}^{\mathcal{I}} = \langle \mathcal{Z}, \mathcal{O}, \mathcal{P}_{\mathcal{Z}} \rangle$, the optimal choice of option is that which maximizes the chance of discovering new regions or, alternatively, that which improves the ability of options to successfully solve their subtasks. If the aim is to solve a task $\mathcal{T}$, the optimal choice of option is that which maximizes the reward of $\mathcal{T}$. On the other hand, the subroutine RunOption executes the policy of the option while simultaneously improving the associated option policy.

5.1.5 Solving tasks

Recall that each task $\mathcal{T}$ is defined by a task MDP $\mathcal{M}^{\mathcal{T}} = \langle \mathcal{S}^{\mathcal{I}} \times \mathcal{S}^{\mathcal{T}}, \mathcal{A}^{\mathcal{I}} \cup \mathcal{A}^{\mathcal{T}}, \mathcal{P}^{\mathcal{I}} \cup \mathcal{P}^{\mathcal{T}}, r^{\mathcal{T}} \rangle$. Given an estimate $\mathbb{S}^{\mathcal{I}} = \langle \mathcal{Z}, \mathcal{O}, \mathcal{P}_{\mathcal{Z}} \rangle$, we define an associated task SMDP $\mathbb{S}^{\mathcal{T}} = \langle \mathcal{Z}^{\mathcal{T}}, \mathcal{O} \cup \mathcal{O}^{\mathcal{T}}, \mathcal{P}_{\mathcal{Z}} \cup \bar{\mathcal{P}}^{\mathcal{T}}, r^{\mathcal{T}} \rangle$. Here, $\mathcal{O}^{\mathcal{T}}$ is a set of task-specific options whose purpose is to change the task state in $\mathcal{S}^{\mathcal{T}}$, and $\bar{\mathcal{P}}^{\mathcal{T}}$ is the transition kernel corresponding to these options. The state space is $\mathcal{Z}_{\mathcal{T}} = \mathcal{Z} \times \mathcal{S}^{\mathcal{T}}$, i.e. a state $(z, s) \in \mathcal{Z}^{\mathcal{T}}$ consists of a region z

Algorithm 5.1 InvariantHRL

```
Input: Action set $\mathcal{A}^{\mathcal{I}}$, oracle compression function $f$
$s \leftarrow$ initial state, $z \leftarrow f(s)$
$\mathcal{Z} \leftarrow \{z\}$, $\mathcal{O} \leftarrow \{o^e_z\}$
while within budget do
  $o \leftarrow$ GetOption$(z, \mathcal{O})$
  $s' \leftarrow$ RunOption$(s, o, A^{\mathcal{I}})$, $z' \leftarrow f(s')$
  if $z' \notin \mathcal{Z}$ then
    $\mathcal{Z} \leftarrow \mathcal{Z} \cup \{z'\}$
    $\mathcal{O} \leftarrow \mathcal{O} \cup \{o^e_{z'}\}$
  end if
  if $o_{z,z'} \notin \mathcal{O}$ then
    $\mathcal{O} \leftarrow \mathcal{O} \cup \{o_{z,z'}\}$
  end if
  $s \leftarrow s'$, $z \leftarrow z'$
end while
```

and a task state s.

As before, we do not assume that the agent has access to options in $\mathcal{O}^{\mathcal{T}}$. Instead, the agent has to discover from experience how to change the task state in $\mathcal{S}^{\mathcal{T}}$. For this purpose, we redefine the exploration option o^e_z of each region z so that it has access to actions in $\mathcal{A}^{\mathcal{T}}$. When selected in state (z, s), o^e_z may terminate for one of two reasons: either the current region changes, i.e. the next state is (z', s) for some neighbor z' of z, or the current task state changes, i.e. the next state is (z, s') for some task state s'. In the latter case, the agent will add an option $o^{s,s'}_z$ to $\mathcal{O}_{\mathcal{T}}$ which is applicable in (z, s) and whose subtask is to reach state (z, s'). Option $o^{s,s'}_z$ has an associated option MDP $\mathcal{M}^{s,s'}_z$, analogous to $\mathcal{M}_{z,z'}$ except that it assigns positive reward to (z, s').

To solve task $\mathcal{T}$, the agent need to maintain and update a high-level policy $\pi^{\mathcal{T}} : \mathcal{Z}^{\mathcal{T}} \to \Delta(\mathcal{O} \cup \mathcal{O}^{\mathcal{T}})$ for the task SMDP $\mathbb{S}^{\mathcal{T}}$. In a state (z, s), policy $\pi^{\mathcal{T}}$ has to decide whether to change regions by selecting an option in $\mathcal{O}$, or to change task states by selecting an option in $\mathcal{O}^{\mathcal{T}}$. Because of our previous assumption on the reward $r^{\mathcal{T}}$, only options in $\mathcal{O}^{\mathcal{T}}$ will incur a non-zero reward, which has to be appropriately discounted after applying each option. Note that in Algorithm 1, policy $\pi^{\mathcal{T}}$ plays the role of the subroutine `GetOption`.

The transition kernel $\bar{\mathcal{P}}^{\mathcal{T}}$ measures the ability of task options in $\mathcal{O}^{\mathcal{T}}$ to successfully solve their subtasks. Hence, $\bar{\mathcal{P}}^{\mathcal{T}}((z, s')|(z, s), o_z^{s,s'})$ should be close to 1, but is lower in case option $o_z^{s,s'}$ sometimes terminates in the wrong state. In our experiments, however, the agent performs model-free learning and never estimates the transition kernel $\bar{\mathcal{P}}^{\mathcal{T}}$.

5.1.6 Controllability

According to the definition of the option reward function $r_{z,z'}$ in (5.1), option $o_{z,z'}$ is equally rewarded for reaching any boundary state between regions z and z'. However, all boundary states may not be equally valuable, i.e. from some boundary states the options in $O_{z'}$ may have a higher chance of terminating successfully. To encourage option $o_{z,z'}$ to reach valuable boundary states and thus make the algorithm more robust to the choice of compression function f, we add a reward bonus when the option successfully terminates in a state s' belonging to region z'.

One possibility is that the reward bonus depends on the value of state s' of options in the set $\mathcal{O}_{z'}$. However, this introduces a strong coupling between options in the set O: the value function $V_{z,z'}$ of option $o_{z,z'}$ will depend on the value functions of options in $\mathcal{O}_{z'}$, which in turn depend on the value functions of options in neighboring regions of z', etc. We want to avoid such a strong coupling since learning the option value functions may become as hard as learning a value function for the original invariant state space $\mathcal{S}^{\mathcal{I}}$.

Instead, we introduce a reward bonus which is a proxy for controllability, by counting the number of successful applications of subsequent options after $o_{z,z'}$ terminates. Let K be the number of options that are selected after $o_{z,z'}$, and let $G \leq K$ be the number of such options that terminate successfully. We define a controllability coefficient ρ as

$$\rho(z) = \frac{G}{K}. \tag{5.2}$$

We then define a modified reward function $\bar{r}_{z,z'}$ which equals $r_{z,z'}$ except when $o_{z,z'}$ terminates successfully, i.e. $\bar{r}_{z,z'}(s, a, s') = r_{z,z'}(s, a, s') + \rho(z)$ if $s' \in z'$. In experiments, we use a fixed horizon $K = 10$ after which we consider successful options transitions as not relevant. In practice, the algorithm has to wait for 10 more options before assigning a reward to the last transition of option $o_{z,z'}$.

5.2 Implementation

In this section, we describe the implementation of our algorithm. We distinguish between a *manager* in charge of solving the task SMDP $\mathbb{S}^{\mathcal{T}}$, and *workers* in charge of solving the option MDPs $\mathcal{M}_{z,z'}$ (or $\mathcal{M}_z^{s,s'}$ for task options).

5.2.1 Manager

Since the space of regions Z is small, the manager performs tabular SMDP Q-learning (Sutton, Precup, and Singh 1999) over the task SMDP $\mathbb{S}^{\mathcal{T}}$. This procedure is shown in Algorithm 6.1. Similar to Algorithm 1, the task state space $\mathcal{S}^{\mathcal{T}}$ and option set $\mathcal{O}^{\mathcal{T}}$ are successively grown as the agent discovers new states and transitions.

Algorithm 5.2 MANAGER

```
 1: **Input**: Task action set $\mathcal{A}^{\mathcal{T}}$, invariant SMDP $\mathbb{S}$
 2: $z \leftarrow$ initial region, $s \leftarrow$ initial task state
 3: $\mathcal{S}^{\mathcal{T}} \leftarrow \{s\}$, $\mathcal{O}^{\mathcal{T}} \leftarrow \emptyset$
 4: $\pi^{\mathcal{T}} \leftarrow$ initial policy
 5: while within budget do
 6:     $o \leftarrow$ GETOPTION($\pi^{\mathcal{T}}, (z,s), \mathcal{O} \cup \mathcal{O}^{\mathcal{T}}$)
 7:     $(z', s'), r \leftarrow$ RUNOPTION($(z,s), o, \mathcal{A}^{\mathcal{I}} \cup \mathcal{A}^{\mathcal{T}}$)
 8:     UPDATEPOLICY($\pi^{\mathcal{T}}, (z,s), o, r, (z', s')$)
 9:     if $s' \notin \mathcal{S}^{\mathcal{T}}$ then
10:         $\mathcal{S}^{\mathcal{T}} \leftarrow \mathcal{S}^{\mathcal{T}} \cup \{s'\}$
11:     end if
12:     if $o_z^{s,s'} \notin \mathcal{O}_{\mathcal{T}}$ then
13:         $\mathcal{O}^{\mathcal{T}} \leftarrow \mathcal{O}^{\mathcal{T}} \cup \{o_z^{s,s'}\}$
14:     end if
15:     $(z,s) \leftarrow (z', s')$
16: end while
```

5.2.2 Worker

The worker associated with option $o_{z,z'} \in \mathcal{O}$ (resp. $o_z^{s,s'} \in \mathcal{O}_{\mathcal{T}}$) should learn a policy $\pi_{z,z'}$ (resp. $\pi_z^{s,s'}$) that allows the manager to transition between two abstract states z, z' (resp. task states s, s'). We use Self-Imitation Learning (SIL) (Oh et al. 2018) which benefits from an exploration bonus coming from the self-imitation component of the loss function. Moreover, since the critic update is off-policy, one can relabel failed transitions in order to speed up learning of the correct option behavior, similar to Hindsight Experience Replay (Andrychowicz et al. 2017).

The architecture is made of two separate neural networks, one for the policy $\pi^{\theta}_{z,z'}$, parameterized on θ, and one for the value function $V^{\psi}_{z,z'}$, parameterized on ψ. The agent minimizes the loss in (5.3) via mini-batch stochastic gradient descent, with on-policy samples:

$$L(\theta, \psi) = L(\hat{\eta}_{\theta}) + \alpha H^{\pi} + L(\hat{V}_{\psi}). \tag{5.3}$$

Algorithm 5.3 shows the algorithm for executing an option. All options act in a region z and stop as soon as the region changes (for options of type $o_z^{s,s'}$ we also have to track the task state). In each iteration, the policy π of the option selects an action in state s from action set A, and this action is simulated to obtain a next state s' and reward r_t. The option then uses the modified reward function $r_z(s, a, s')$ to compute its local reward $\bar{r}$, and adds the transition $(s, a, \bar{r}, s')$ to a buffer B. Once the region changes, execution stops, and the policy and value function of the worker is updated using the subroutine UPDATEWORKER.

5.3 Experiments

To evaluate the proposed algorithm we use two benchmark domains: a Key-door-treasure grid world, and a simplified version of Montezuma's Revenge where the agent only has to pick up the key in the first room. In both domains, the invariant part of the state consists of the agent's location, and the compression function f imposes a grid structure on top of the location (cf. Figure 5.2) by performing an integer division over the (x, y) coordinates of the agent position $f(x, y) = \lfloor x/d_{int} \rfloor + \lfloor y/d_{int} \rfloor$. Results are averaged over 5 seeds and each experiment is run for 4e5 all

Algorithm 5.3 RunOption

```
1: Input: region $z$, action set $\mathcal{A}$
2: Retrieve buffer $B$ and policy $\pi$ from memory
3: $s \leftarrow$ current state in $z$
4: while $f(s) = z$ do
5:     $a \leftarrow \textsc{SelectAction}(s, \mathcal{A}, \pi)$
6:     $s', r_t \leftarrow \textsc{Simulate}(s, a)$
7:     $\bar{r} \leftarrow \bar{r}_z(s, a, s')$
8:     $B \leftarrow B \cup (s, a, \bar{r}, s')$
9:     $s \leftarrow s'$
10: end while
11: $\textsc{UpdateWorker}(B)$
12: return $f(s)$, $\sum_t r_t$
```

the agents have been trained with the choice of hyperparameters in Figure 5.1.

In the Key-door-treasure domain, we make the reward progressively more sparse. In the simplest setting the agent obtains a reward in each intermediate goal state, while in the hardest setting the agent obtains a reward only in the terminal state. We also tested the transfer learning ability of our algorithm in new tasks generated by moving the position of the Key, Door and Treasure objects.

In Montezuma's Revenge, we evaluate whether our controllability proxy helps transition between regions. Montezuma does present an ideal environment to test this since imposing a grid set of regions on it does not respect the structural semantics of the environment and transitioning to the wrong state in another region may cause the agent to fall and die.

Key-door-treasure is a stochastic variant of the original domain (Oh et al. 2018) taking random actions with probability 20%. The agent has a budget of 300 time steps. We define two variants and randomly generate multiple tasks by changing the location of the Key, Door and Treasure. In Key-door-treasure-1 (Oh et al. 2018) the key is in the same room as the door, while in Key-door-treasure-2 the key is in a different room, making exploration harder.

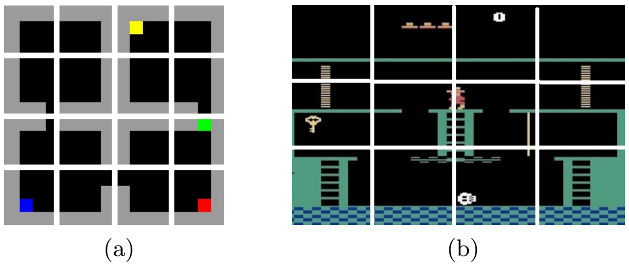


Figure 5.2: Key-door-treasure-1 (a) and Montezuma's Revenge (b) with compression function superimposed.

5.3.1 Exploration

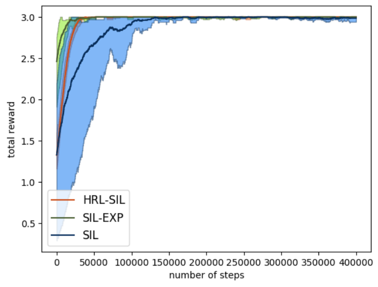
To investigate the exploration advantage of the proposed algorithm, we compare it against SIL (Oh et al. 2018) and against a version of SIL augmented with count-based exploration (Strehl and Littman 2008) that gives an exploration bonus reward $r_{exp}(s,a) = \beta/\sqrt{N(s)}$, where $N(s)$ is the visit count of state s and β is a hyperparameter. In the figures, our algorithm is labeled HRL-SIL, while SIL and SIL-EXP refer to SIL without/with the exploration bonus.

In Key-door-treasure-1 (Figure 5.3) we observe that when the reward is given for every object, all the algorithms perform well, while by making the reward more sparse, our algorithm clearly outperforms the others, because of its ability to act on different timescales through the compressed state space and the options action space.

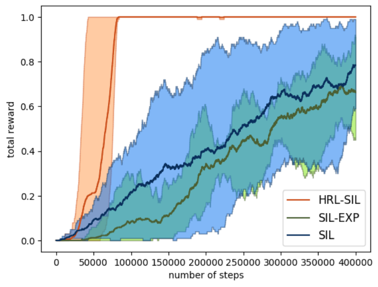
We further investigate this in Key-door-treasure-2 (Figure 5.4) where the key and door are placed in different rooms. This makes exploration harder, and SIL struggles even in the setting with intermediate rewards, only learning to pick up the key, while SIL-EXP slowly learns to open the door and get the treasure thanks to the exploration bonus.

5.3.2 Transfer Learning

To investigate the transfer ability of the algorithm, we train 'HRL-SIL' subsequently on a set of tasks and compared it to 'SIL-EXP'. In the first task, the goal is just to pick up a key and open a door. Once trained



(a) Reward for all objects.



(b) Reward for treasure only.

Figure 5.3: Results in Key-door-treasure-1.

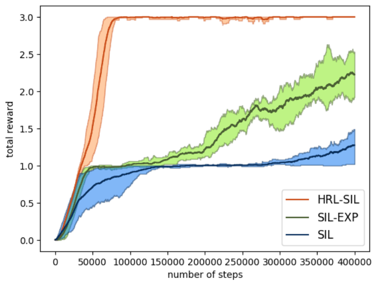


Figure 5.4: Results in Key-door-treasure-2, reward for all objects.

on this task, the agent is presented with a more complex task that also involves a treasure. The third task is the same as the second with the location of the objects mirrored.

Our agent is evaluated by resetting the manager policy from task to task, while 'SIL-EXP' is evaluated by clearing the Experience Replay buffer between every task. We omit 'SIL' since it always performs worse than 'SIL-EXP'. From Figure 5.6 we observe that the learned set of options $\mathcal{O}$ and set of regions $\mathcal{Z}$ transfer well across tasks. In contrast, 'SIL-EXP' struggles to solve new tasks. In the figure, 'NO-TRANSFER-HRL-SIL' and 'NO-TRANSFER-SIL-EXP' refer to the versions that relearn tasks from scratch.

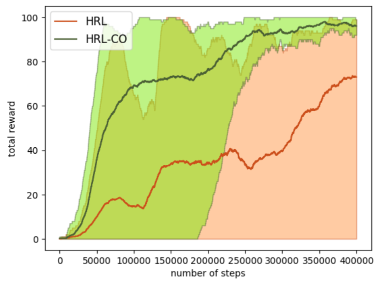


Figure 5.5: Results in Montezuma's Revenge with controllability.

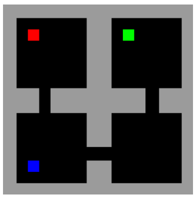
5.3.3 Controllability

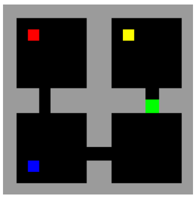
Lastly, we test whether the controllability proxy helps transition successfully between regions. We compare two versions of our algorithm, one with controllability ('HRL-CO') and one without ('HRL'), in the first room of Montezuma's Revenge with the task of collecting the key. This environment is challenging since the agent could learn unsafe transitions that lead to successful moves between regions but subsequently dying. As we can see from Figure 5.5 the controllability proxy does indeed help in learning successful and safe transitions between regions, outperforming the simpler reward scheme of 'HRL'.

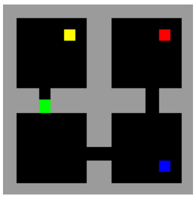
5.4 Discussion

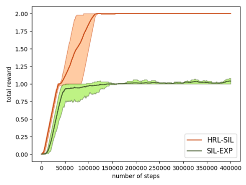
In spite of the encouraging results in Section 5.3, the current version of the proposed algorithm has several limitations. In this section, we discuss potential future improvements aimed at addressing these limitations.

Invariant state-action space The current version of the algorithm assumes that the agent has prior knowledge of the invariant part of the state-action space, i.e. $\mathcal{S}^{\mathcal{I}} \times \mathcal{A}^{\mathcal{I}}$. In some applications, this seems like a reasonable assumption, e.g. in environments such as MineCraft or DeepMind Lab where the agent has access to a basic set of actions and is later asked to solve specific tasks.

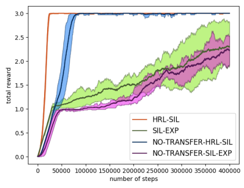

(a) Env 0


(b) Env 1

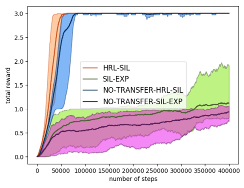

(c) Env 2



(d) results in Env 0



(e) results in Env 1



(f) results in Env 2

Figure 5.6: Results of transfer learning, with reward given for all objects.

Compression function The algorithm also assumes that the agent has access to a compression function f which maps invariant states to regions. In case such a function is not available, the agent would need to automatically group states into regions. We believe that the algorithm is reasonably robust to changes in the compression function, but an important feature is that neighboring states should be grouped into the same region. Dilated recurrent neural networks (Chang et al. 2017) are designed to maintain constant information during a given time period, similar to the idea of remaining in a given region for multiple timesteps, and have been previously applied to hierarchical reinforcement learning (Vezhnevets et al. 2017).

Option policies Another limitation of the algorithm is that it needs to learn a large number of policies that scale as the number of regions times the number of neighbors. In large-scale experiments, it would be necessary to compress the number of policies in some way. Since regions are mutually exclusive, in principle one could use a single neural network to represent the policy of $|\mathcal{Z}|$ different options. However, in preliminary experiments such a representation suffers from catastrophic forgetting, struggling to maintain the optimal policy of a given option while training the policies

Hyperparameters	**Value**
Architecture	- FC(64) - FC(64)
Learning rate	0.0007
N steps per iteration	6
Entropy reg (α)	0.01
SIL update per iteartion (M)	SIL: 4, HRL: [1, 4]
SIL batch size	512
SIL loss weight	1
SIL value loss weight (β^{sil})	0.01
Replay buffer size	10^4
Exponent for prioritization	0.6
Bias correction, prioritized replay	0.4
Manager ϵ-greedy	[0.05, 0.005]
Count exploration β	0.2
Observation in Key-door-treasure	(x, y, inventory)
Observation in Montezuma	(x, y)

Table 5.1: Hyperparameters used in the experiments.

of other options. We believe that a more intelligent compression scheme would be necessary for the algorithm to scale, potentially sharing a single policy among a carefully selected subset of options.

5.5 Conclusion

We presented a hierarchical reinforcement learning algorithm that decomposes the state space using a compression function and introduces subtasks that consist in moving between the resulting partitions. Furthermore, we illustrated that the algorithm can successfully solve relatively complex sparse-reward domains. As discussed in Section 5.4, there are many opportunities for extending the work in the future.

Chapter 6

Hierarchical Representation Learning for Markov Decision Processes

In this chapter, we present a novel method for learning hierarchical representations of Markov decision processes. Our method works by partitioning the state space into subsets and defining subtasks for performing transitions between the partitions. At the high level, we use model-based planning to decide which subtask to pursue next from a given partition. We formulate the problem of partitioning the state space as an optimization problem that can be solved using gradient descent given a set of sampled trajectories, making our method suitable for high-dimensional problems with large state spaces. We empirically validate the method, by showing that it can successfully learn useful hierarchical representations in domains with high-dimensional states. Once learned, the hierarchical representation can be used to solve different tasks in the given domain, thus generalizing knowledge across tasks.

State Space Partition properties

Here we summarize and briefly discuss the state space partitions properties (see Chapter 4) of the methodology presented in this chapter.

Pros:

✓ **Small SubMDPs**: The parametrized compression function $f_\psi : S \rightarrow \Delta(Z)$ proposed in this chapter is able to control and balance the size of each subMDP by tuning the number of regions $|\mathcal{Z}|$ to fit on the state space $\mathcal{S}$. Moreover, the second term in the loss proposed (see eq. 6.1) constrains the representation to balance the probabilities of belonging to regions, forcing the region size to be balanced and allowing us to control the maximum size M(see Section 4.2) of each induced subMDP.

✓ **Bottleneck exit states**: The compression function presented here implicitly favors partitions that minimize the set of all exit states $\mathcal{E}$ (see Section 4.2) as we can see in Figure 6.2 and 6.1. From loss 6.1 we can notice that the two terms $\mathcal{L}_\mathcal{Z}$, $\mathcal{L}_H$ will be minimized when the compression function f_ψ clusters strongly connected states together and at the same time balance the probabilities of belonging to regions. By tuning the weights w_H we could allow different sizes of regions and better match clusters that minimize the set of all exit states $\mathcal{E}$.

✓ **Strongly connected subMDPs**: By minimizing the first term in loss 6.1 we as well incentive the partitions that induce strongly connected subMDPs (see Section 4.2)

Cons:

× **Prior knowledge on number of regions**: The main drawback of the state partitioning approach presented in this chapter is that he needs prior knowledge about the number of regions $|\mathcal{Z}|$

× **Equivalent subMDPs**: The state partition presented in this chapter is not able to identify equivalent subMDPs(see Section 4.2)

× **Dependency to the behavior policy used to collect the dataset**: The learned representation demonstrates a coupling to the behavior policy used to collect the dataset, which indirectly defines the distance between states (i.e. which states cluster together).

6.1 Contribution

In this section, we present our main contribution, a method for learning a hierarchical representation of a given MDP.

6.1.1 Compression Function

The first step is to learn a compression function from MDP states to regions. We first define a set $\mathcal{Z}$ of regions (see Chapter 4) that will represent the partitions of the state space. Without loss of generality, the elements of $\mathcal{Z}$ are simply integers, i.e. $\mathcal{Z} = \{0, \dots, |\mathcal{Z}|-1\}$, where $|\mathcal{Z}|$ is an input parameter of the method. Our goal is to learn a parameterized compression function $f_\psi : S \to \Delta(\mathcal{Z})$ that maps MDP states to probability distributions over regions. Ideally, f_ψ should be *deterministic*, but the learning framework we consider favors probabilistic compression functions.

Intuitively, for regions to represent partitions of the state space, on a given trajectory the region should remain the same most of the time, and only change occasionally. We formalize this intuition as a loss term, which will later be part of the objective that the learner attempts to minimize. Let $\chi = (s_t, a_t, r_t, s_{t+1})$ be a transition, and let $\mathcal{D} = \{\chi_1, \dots, \chi_m\}$ be a set of transitions. The loss associated with $\mathcal{D}$ is given by

$$\mathcal{L}_\mathcal{Z}(\mathcal{D}) = -\sum_{\chi\in\mathcal{D}} \sum_{z\in\mathcal{Z}} f_\psi(z|s_t) \log f_\psi(z|s_{t+1}).$$

Here, $-\sum_{z\in\mathcal{Z}} f_\psi(z|s_t) \log f_\psi(z|s_{t+1})$ is the cross-entropy loss for consecutive states s_t and s_{t+1} in χ, measuring the distance between the distributions $f_\psi(\cdot|s_t)$ and $f_\psi(\cdot|s_{t+1})$.

On its own, the above loss term will not yield a meaningful compression function, since it can be minimized by mapping all states to the same region. To ensure that all regions appear in the compression, we define a second loss term equivalent to the negative entropy of the compression function across the same set of transitions $\mathcal{D}$. Given a region $z \in \mathcal{Z}$, let $F(z|\mathcal{D}) = \frac{1}{|\mathcal{D}|} \sum_{\chi\in\mathcal{D}} f_\psi(z|s_t)$ be the average probability of being in z across the first state s_t of each transition $\chi \in \mathcal{D}$. We define a loss term

$$\mathcal{L}_H(\mathcal{D}) = -H(F(\cdot|\mathcal{D})) = \sum_{z\in\mathcal{Z}} F(z|\mathcal{D}) \log F(z|\mathcal{D}),$$

where $H(F(\cdot|\mathcal{D}))$ is the entropy of the function $F(\cdot|\mathcal{D})$. This loss is minimized when the probabilities of regions are uniform, i.e. each region is equally likely.

Finally, as already stated, we would like the compression function f_ψ to be as deterministic as possible. For this reason, we define a third loss term equivalent to the entropy of the compression function for individual states. We use the same set of transitions $\mathcal{D}$, and define this loss term as

$$\begin{aligned}\mathcal{L}_D(\mathcal{D}) &= \frac{1}{|\mathcal{D}|}\sum_{\chi\in\mathcal{D}} H(f_\psi(\cdot|s_t)) \\ &= -\frac{1}{|\mathcal{D}|}\sum_{\chi\in\mathcal{D}}\sum_{z\in\mathcal{Z}} f_\psi(z|s_t)\log f_\psi(z|s_t).\end{aligned}$$

This loss term is minimized when the compression function $f_\psi(\cdot|s_t)$ is deterministic, i.e. assigns probability 1 to a single region, for the first state s_t of each transition $\chi \in \mathcal{D}$.

The overall loss function $\mathcal{L}(\mathcal{D})$ is a combination of the three individual loss terms, i.e.

$$\mathcal{L}(\mathcal{D}) = \mathcal{L}_{\mathcal{Z}}(\mathcal{D}) + w_H\mathcal{L}_H(\mathcal{D}) + w_D\mathcal{L}_D(\mathcal{D}), \tag{6.1}$$

where w_H and w_D are weights that we can tune to determine the relative importance of each loss term. Note that for $w_D = -1$, $\mathcal{L}_{\mathcal{Z}}(\mathcal{D}) + w_D\mathcal{L}_D(\mathcal{D})$ is the average Kullback-Leibler divergence between $f_\psi(\cdot|s_t)$ and $f_\psi(\cdot|s_{t+1})$; however, our intention is to use positive values of w_D.

To train our compression function we use experience replay (Mnih et al. 2013) to randomize the transitions in $\mathcal{D}$. We first sample a set of trajectories using some exploration policy, which constitutes our memory. However, learning directly from consecutive transitions along the same trajectory is inefficient, due to the strong correlations between the samples. Instead, we form the set of transitions $\mathcal{D}$ by randomly sampling individual transitions from the memory. Randomizing the sampled transitions this way breaks the correlations and therefore reduces the variance of the updates.

6.1.2 Hierarchical Representation

Once we have learned a compression function f_ψ for a given MDP $\mathcal{M} = \langle\mathcal{S}, \mathcal{A}, \mathcal{P}, r\rangle$, we use it to define a set of options $\mathcal{O}$ and an SMDP $\mathbb{S}$. First, we introduce a *deterministic* compression function $g : \mathcal{S} \to \mathcal{Z}$, defined in each state s as $g(s) = \arg\max_z f_\psi(z|s)$. Given a region $z \in \mathcal{Z}$, let $\mathcal{S}_z$ be the subset of states that map to z, i.e. $\mathcal{S}_z = \{s \in \mathcal{S} : g(s) = z\}$.

Our algorithm then uses the compression function in an online manner, by exploring the environment and finding *region transitions*, i.e. consecutive states s_t and s_{t+1} such that $g(s_t) \neq g(s_{t+1})$. Let $\mathcal{Y} \subseteq \mathcal{Z} \times \mathcal{Z}$ be the subset of pairs of distinct regions (z, z') that appear as region transitions while exploring, i.e. there exist two consecutive states s_t and s_{t+1} such that $g(s_t) = z$ and $g(s_{t+1}) = z'$. For each pair $(z, z') \in \mathcal{Y}$, we introduce an option $o_{z,z'} = \langle \mathcal{I}^z, \pi_{z,z'}, \beta_z \rangle$ whose purpose is to perform a region transition from z to z'. Option $o_{z,z'}$ is applicable in region z, i.e. $\mathcal{I}^z = \mathcal{S}_z$ and terminates as soon as we reach a region different from z, i.e. $\beta_z(s) = 0$ if $s \in \mathcal{S}_z$ and $\beta_z(s) = 1$ otherwise.

To learn the policy $\pi_{z,z'}$ of option $o_{z,z'}$, we define an option-specific Markov decision process $\mathcal{M}_{z,z'} = \langle \mathcal{S}_z, \mathcal{A}, \mathcal{P}_z, r_{z,z'} \rangle$. Note that $\mathcal{M}_{z,z'}$ needs only be defined for states in $\mathcal{S}_z$, since option $o_{z,z'}$ always terminates outside this set. The local reward function $r_{z,z'}$ is defined for each state-action pair as $r_{z,z'}(s, a, s') = r(s, a, s')$, i.e. equal to the environment reward. We also introduce a bonus $+1$ for terminating in a state s such that $g(s) = z'$. As a consequence, the policy $\pi_{z,z'}$ has the incentive to leave region z, and prefers to transition to region z' whenever possible.

Let $\mathcal{O} = \{o_{z,z'} : (z, z') \in \mathcal{Y}\}$ be the set of options for performing region transitions and let $\mathcal{O}_z = \{o \in \mathcal{O} : I^z = \mathcal{S}_z\}$ be the subset of options applicable in region z. We define an SMDP $\mathbb{S} = \langle \mathcal{Z}, \mathcal{O}, \mathcal{P}_\mathcal{Z}, r_\mathcal{Z} \rangle$, i.e. the high-level choices of the learning agent are to select region transitions to perform. Once the individual option policies have been trained, exploration is typically more efficient since the single decision of which option to execute results in a state that is many steps away from the initial state. In addition, one can approximate the SMDP policy as $\pi : \mathcal{Z} \to \Delta(\mathcal{O})$, i.e. the choice of which option to execute only depends on the current region. This has the potential to significantly speed up learning if $|\mathcal{Z}| \ll |\mathcal{S}|$.

The system is trained using a Manager-Worker architecture (Dayan and Hinton 1993). The Manager performs tabular Value Iteration over the SMDP. The motivation for using tabular learning is that the number of regions $|\mathcal{Z}|$ is typically small, even if states are high-dimensional. On the other hand, the Worker uses off-policy value-based methods to learn the policies of the options $o_{z,z'}$.

6.1.3 Controllability

As in Chapter 5.1.6 we introduced a reward bonus which is a proxy for controllability, by counting the number of successful applications of subsequent options after $o_{z,z'}$ terminates. Let K be the number of options that are selected after $o_{z,z'}$, and let $G \leq K$ be the number of such options that terminate successfully. We define a controllability coefficient ρ as

$$\rho(z) = \frac{G}{K}. \tag{6.2}$$

We then define a modified reward function $\bar{r}_{z,z'}$ which equals $r_{z,z'}$ except when $o_{z,z'}$ terminates successfully, i.e. $\bar{r}_{z,z'}(s, a, s') = r_{z,z'}(s, a, s') + \rho(z)$ if $s' \in z'$. In experiments, we use a fixed horizon $K = 4$ after which we consider successful option transitions as irrelevant.

6.1.4 Transfer

The hierarchical representation in the form of the SMDP $\mathbb{S}$ defined above can be used to transfer knowledge between tasks. Concretely, we assume that the given MDP $\mathcal{M}$ can be extended to form a task by adding states and actions. Imagine for example that $\mathcal{M}$ models a navigation problem in a given environment. A task can be defined by adding objects in the environment that the learning agent can manipulate, while navigation is still part of the task.

Formally, given an MDP $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, r \rangle$, a task $\mathcal{T}$ is an MDP $\mathcal{M}^{\mathcal{T}} = \langle \mathcal{S} \times \mathcal{S}^{\mathcal{T}}, \mathcal{A} \cup \mathcal{A}^{\mathcal{T}}, \mathcal{P} \cup \mathcal{P}^{\mathcal{T}}, r \cup r^{\mathcal{T}} \rangle$. The states in $\mathcal{S}^{\mathcal{T}}$ represent information about task-specific objects, and the actions in $\mathcal{A}^{\mathcal{T}}$ are used to manipulate these objects. The transition kernel $\mathcal{P}^{\mathcal{T}} : (\mathcal{S} \times \mathcal{S}^{\mathcal{T}}) \times \mathcal{A}^{\mathcal{T}} \rightarrow \Delta(\mathcal{S}^{\mathcal{T}})$ governs the effects of the actions in $\mathcal{A}^{\mathcal{T}}$, which may depend on the states of the original MDP (e.g. the location of the agent). Finally, the reward function $r^{\mathcal{T}} : (\mathcal{S} \times \mathcal{S}^{\mathcal{T}}) \times \mathcal{A}^{\mathcal{T}} \rightarrow \mathbb{R}$ models the reward associated with actions in $A^{\mathcal{T}}$.

To solve a task, we can replace the MDP $\mathcal{M}$ with the learned SMDP $\mathbb{S} = \langle \mathcal{Z}, \mathcal{O}, \mathcal{P}_{\mathcal{Z}}, r_{\mathcal{Z}} \rangle$, forming a task SMDP $\mathbb{S}^{\mathcal{T}} = \langle \mathcal{Z} \times \mathcal{Z}^{\mathcal{T}}, \mathcal{O} \cup \mathcal{A}^{\mathcal{T}}, \mathcal{P}_{\mathcal{Z}} \cup P^{\mathcal{T}}, r_{\mathcal{Z}} \cup r^{\mathcal{T}} \rangle$. Here, the options in O are used to navigate in the original

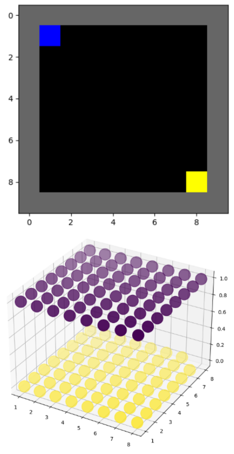


Figure 6.1: Results on Key-Door0 gridworld environment. The first row represents the environments, and the second row illustrates the corresponding learned deterministic compression functions, where different colors represent different regions $z \in \mathcal{Z}$.

state space S, while the actions in $A^{\mathcal{T}}$ are used to manipulate the task-specific objects. If the policies of the options in $\mathcal{O}$ have been previously trained, the task SMDP $\mathbb{S}^{\mathcal{T}}$ can significantly accelerate learning compared to the task MDP $\mathcal{M}^{\mathcal{T}}$. To ensure that the learning agent can navigate to individual objects inside a partition of $\mathcal{Z}$, we consider states in $S^{\mathcal{T}}$ to be different regions; hence our algorithm will automatically add options for manipulating objects.

6.2 Experimental Results

The experiments are designed to answer the following questions:

- Is the learned compression function suitable for learning a hierarchy?
- Does the learned hierarchy transfer across different tasks in the same environment?
- How does our HRL algorithm compare against state-of-the-art flat

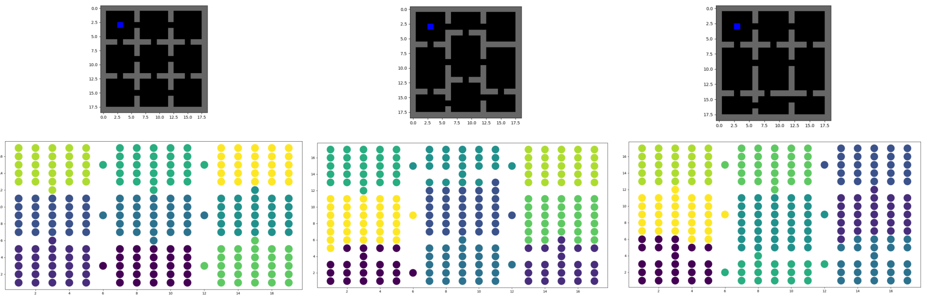


Figure 6.2: Results on geometric variations of the NineRooms0 grid world environments. The first row represents the environments, and the second row illustrates examples of the corresponding learned deterministic compression functions, where different colors represent different regions $z \in \mathcal{Z}$.

algorithms, such as Self Imitation Learning (Oh et al. 2018) and Double-DQN with Prioritized Experience Replay (Schaul et al. 2015a)?

6.2.1 Learning a Compression Function

We designed two different empty navigation environments without tasks, KeyDoor0 (c.f. Figure 6.1), with grid size 10×10, where an agent (blue square) always starts in position (1, 1), has to collect a key (yellow square). And NineRooms0 (c.f. Figure 6.2), a nine rooms grid environment with grid size 19×19 where at each episode the agent is placed at a random initial position, which promotes exploration. For all the environments the states are (x, y)-positions which are mapped to images, and the discrete action space is $\mathcal{A} = \{up, down, left, right\}$.

The first step of our procedure consists in a pre-training phase where we form a replay memory of trajectories. We use a random exploration policy to repeatedly generate trajectories from the random initial states, using a fixed episode length of 100. During this phase, we can vary the number of trajectories generated to test the robustness of the approach.

We then use the replay memory and a number of regions $|\mathcal{Z}| = 2$ for KeyDoor0 and $|\mathcal{Z}| = 9$ for NineRoom0 to train the compression function f_ϕ using the AdamW optimizer (Loshchilov and Hutter 2017) by minimizing the loss in (6.1) over 4000 iterations, randomly sampling a set of 32 transitions $\mathcal{D}$ from the replay memory in each iteration. The learned compression functions for 1000 trajectories are shown in Figures 6.1 and 6.2, respectively.

To asses the robustness of our procedure, in Figure 6.2 we evaluate how the compression function changes by introducing different geometries of the NineRooms0 environment. As we can see, if we make the room sizes imbalanced, the resulting compression function does not exactly match the shape of the rooms, due to the second term $\mathcal{L}_H$ of the loss in (6.1), which promotes all regions z to be equally likely. However, the resulting compression function still partitions the states and translates into a correct SMDP.

In Figure 6.3, we evaluate how the size of the replay memory affects the accuracy of the compression function in terms of the absolute error deviation with respect to a correct representation. For this experiment, we use the left-most room in Figure 6.2 with balanced room sizes and vary the number of trajectories in the replay memory. When the replay memory contains at least 200 trajectories, the procedure converges to an absolute error very close to 0, while less than 200 trajectories result in an increasing absolute error.

In the supplementary material, we present additional experiments with learning a compression function in the MountainCar environment. We also list all hyperparameters of the algorithm.

6.2.2 Hierarchical Reinforcement Learning

Following the pre-training phase, we can use the learned compression function to solve any task in the same environment. In what follows we use the compression function learned in the left-most room in Figure 6.2 with a replay memory of 1000 trajectories. We distinguish between a *manager* in charge of solving the task SMDP $\mathbb{S}^{\mathcal{T}}$ and *workers* in charge of solving the option MDPs $\mathcal{M}_{z,z'}$.

Manager

Our algorithm iteratively grows an estimate of the SMDP $\mathbb{S}$. Initially, the agent only observes a single state $s \in \mathcal{S}$ and associated region $z = g(s)$.

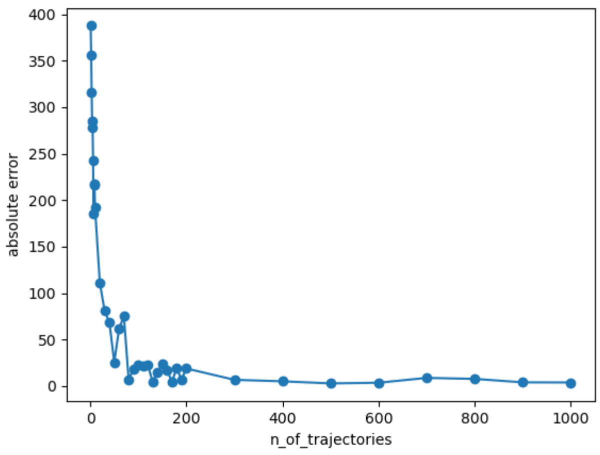


Figure 6.3: Absolute error of the compression function, evaluated on increasing replay memory size.

Hence the state space $\mathcal{Z}$ contains a single region z, whose associated option set $\mathcal{O}_z$ is initially empty. In this case, the only alternative available to the agent is to *explore*. For each region z, we add an exploration option $o_z^e = \langle \mathcal{I}^z, \pi_z^e, \beta_z \rangle$ to the option set O. This option has the same initiation set and termination condition as the options in O_z, but the policy $\pi_z^{exploration}$ is an exploration policy that selects actions uniformly at random, terminating when it leaves region z or exhausts a given budget.

Once the agent discovers a neighboring region z' of z, it adds z' to the set $\mathcal{Z}$ and the associated option $o_{z,z'}$ to the option set $\mathcal{O}$. The agent also maintains and updates a directed graph whose nodes are regions and whose edges represent the neighbor relation. Hence next time the agent visits region z, one of its available actions is to select option $o_{z,z'}$. When option $o_{z,z'}$ is selected, it chooses actions using its policy $\pi_{z,z'}$ and updates $\pi_{z,z'}$ based on the rewards of the option MDP $\mathcal{M}_{z,z'}$. Figure 6.4 shows an example representation discovered by the algorithm.

Algorithm 6.1 shows the pseudo-code of the algorithm. As explained, $\mathcal{Z}$ is initialized with the region z of the initial state s, and $\mathcal{O}$ is initialized with the exploration option $o_z^{exploration}$. In each iteration, the algorithm selects an option o which is applicable in the current region z. If we transition to a new region z', it is added to $\mathcal{Z}$ and the exploration option $o_{z'}^{exploration}$ and transition option $o_{z,z'}$ are appended to $\mathcal{O}$. The process then repeats from the next state s'.

The subroutine GetOption that selects an option o in the current region z can be implemented in different ways; in our case, we use an ϵ-greeedy policy.

Since the set of region $\mathcal{Z}$ is small, the manager performs tabular Value Iteration over the task SMDP $\mathbb{S}^{\mathcal{T}}$.

In order to recognize new goal states, while exploring we define any terminal state in the environment as a new region z; hence the manager will introduce options for reaching this terminal state.

Planning with a learned SMDP

We have seen how the state space of the task SMDP $\mathbb{S}^{\mathcal{T}}$ is discovered online given a compression function $g(s)$. In order to apply a model-based method to this learned compression function, we still need to be able to estimate the transition kernel $P^{\mathcal{T}}$ and reward function $r^{\mathcal{T}}$.

Estimating the transition probability associated with an option $o_{z,z'}$ of our task SMDP is not easy, since the policy $\pi_{z,z'}$ is trained online while exploring the environment, making the transition probability non-stationary. In order to alleviate the cost of estimating the transition probability, we assume that $o_{z,z'}$ will become deterministic once the training phase terminates, i.e. $\widehat{P}^{\mathcal{T}}(z'|z, o_{z,z'}) = 1$. Though this is an approximation, the aim of option $o_{z,z'}$ is precise to reach region z', and constructing the SMDP is intended to simplify high-level decision-making.

On the other hand, for each state-option pair (z, o) of the task SMDP $\mathbb{S}^{\mathcal{T}}$, we estimated the task SMDP reward $r^{\mathcal{T}}$ as an average of the reward encountered in the environment:

$$\widehat{r}^{\mathcal{T}}(s, o) = \frac{\sum_{i=1}^{C(z,o)} R_i(z, o)}{C(z, o)}, \tag{6.3}$$

where $C(z, o)$ counts the number of times the state-option pair (z, o) has been observed, and R_i is the cumulative reward obtained while applying option o for the i-th time.

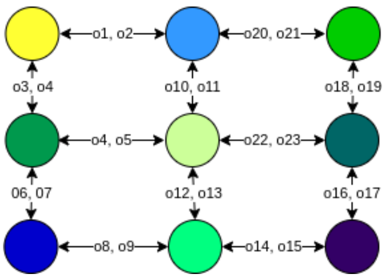


Figure 6.4: The discovered invariant SMDP.

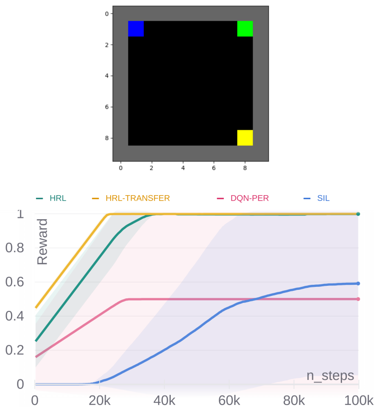


Figure 6.5: Results in the KeyDoor environment.

Since the model changes over time, the subroutine UPDATEPOLICY updates the Q values of the Manager at regular intervals by applying value iteration on the learned SMDP $\widehat{\mathbb{S}}^{\mathcal{T}}$.

Workers

The workers are in charge of learning the policies of each option $o_{z,z'}$ in O, allowing the manager to transition between two regions z, z'. We use Double DQN (Van Hasselt, Guez, and Silver 2016), a version of DQN that addresses the overestimation of Q-values, combined with Prioritized Experience Replay (PER) (Schaul et al. 2015a) that improves the way experience is sampled from the Experience Replay. The rewards that the worker observes are defined in Section 6.1.2 and implemented in the routine TrainOption from Algorithm 6.1.

Since Double DQN is able to evaluate Q-values off-policy, one can relabel failed transitions to speed up learning of the correct option behavior, similar to Hindsight Experience Replay (Andrychowicz et al. 2017). The architecture is made of a neural network Q_θ parametrized on θ, and a frozen target network $Q_{\bar{\theta}}$ used to alleviate the non-stationarity of the targets $\mathrm{T_Q} = r(s,a) + \gamma \max_{a'} Q_\theta(s', a')$.

The parameters of the neural network are updated as:

$$\theta \leftarrow \theta + \alpha \left(\mathrm{T_D} - Q_\theta(s,a)\right) \nabla_\theta Q_\theta(s,a), \tag{6.4}$$

where $\mathrm{T_D}$ is the target value computed as:

$$\mathrm{T_D} = r\,(s,a) + \gamma Q_{\bar{\theta}} \left(s_{t+1}, \underset{a}{\operatorname{argmax}} Q_\theta\,(s_{t+1}, a) \right). \tag{6.5}$$

The target network is then updated with Polyak updates (Heess et al. 2015):

$$\bar{\theta} = \omega\theta + (1-\omega)\,\bar{\theta}, \tag{6.6}$$

where $\omega \in [0,1]$

Algorithm 6.1 MANAGER

```
1: Input: environment $e$, previously discovered SMDP $\mathbb{S}$ in case of transfer learning, compression function $g$
2: $s \leftarrow initialstate$
3: $z \leftarrow g(s)$
4: if $z \notin \mathcal{Z}$ then
5:     $\mathcal{Z} \leftarrow \mathcal{Z} \cup \{z\}$
6:     $\mathcal{O} \leftarrow o_z^{exploration}$
7: end if
8: $\pi^{\mathcal{T}} \leftarrow$ initial policy
9: $o \leftarrow$ None
10: while within budget do
11:     if $o$ is None or Terminate then
12:         $o \leftarrow \textsc{GetOption}(\pi^{\mathcal{T}}, z, \mathcal{O})$
13:         $R = 0$
14:     end if
15:     $s', r, done \leftarrow e(o(s))$
16:     $\textsc{TrainOption}(o, s, r, s', done)$
17:     $R = R + r$
18:     $z' \leftarrow g(s')$
19:     if $z' \notin \mathcal{Z}$ then
20:         $\mathcal{Z} \leftarrow \mathcal{Z} \cup \{z'\}$
21:         $\mathcal{O} \leftarrow \mathcal{O} \cup \{o_{z'}^{e}, o_{z,z'}\}$
22:     end if
23:     if $z \neq z'$ then
24:         $\textsc{UpdatePolicy}(\pi^{\mathcal{T}}, z, o, R, z')$
25:         $o \leftarrow \textsc{GetOption}(\pi^{\mathcal{T}}, z', O)$
26:     end if
27:     if $s'$ is terminal and $s'$ not in $\mathcal{Z}$ then
28:         $\mathcal{O} \leftarrow \mathcal{O} \cup \{o_z^{z,s'}\}$
29:         $\mathcal{Z} \leftarrow \mathcal{Z} \cup \{s'\}$
30:     end if
31:     $(z, s) \leftarrow (z', s')$
32: end while
```

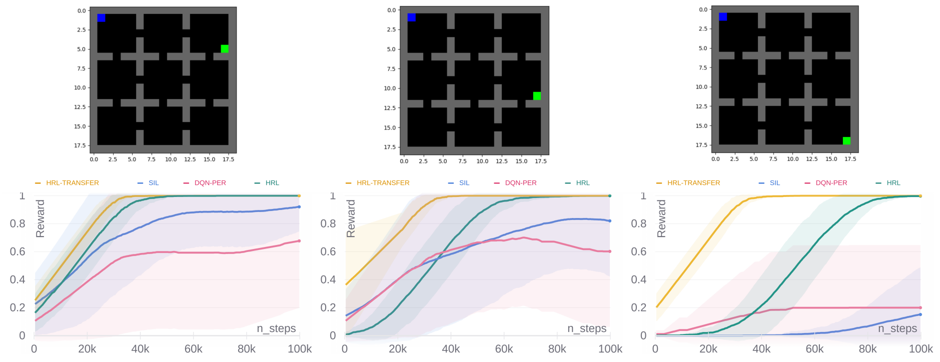


Figure 6.6: Results on the variations of nine room gridworld environments where the goal (green square) is placed at an increasing distance from the agent (blue square). From left to right: NineRoom1, NineRoom2, NineRoom3.

6.2.3 Experiments

In our experiments, we evaluate the performance of our agent in two environments, on a KeyDoor environment in Figure 6.5 where an agent has to collect a key (yellow square) and open a door (green square) and a NineRooms environment in Figure 6.6 where an agent has to reach the goal (green square). In both environments, the initial position is fixed to $(1, 1)$. In the NineRooms environment, we defined three variants where the goal is positioned at an increasing distance from the initial state to make exploration harder, c.f. NineRooms1, NineRooms2 and NineRooms3 in Figure 6.6. Results are averaged over 5 seeds and each experiment is run for 100,000 iterations. Even though the compression function is given, the goal location is unknown, so the agent has to explore the environment in order to find the goal location for the first time.

We set the maximum number of steps in the environment to 40 for the KeyDoor environment and to 200 for NineRooms environment, making exploration hard, especially in NineRooms3 where the goal is at the maximum distance from the initial state. Results in Figures 6.5 and 6.6 show the total reward with a running average smoothing of 100 episodes and shaded standard deviation. In the KeyDoor environment, the agent receives a reward of +1 only once it opens the door (green square) with the key (yellow square) while in NineRooms the agent receives a reward of +1 when it reaches the goal position (green square) and a reward of 0 elsewhere.

We compared our algorithm against state of the art flat reinforcement learning agents designed to perform well in sparse reward settings, namely Self Imitation Learning (SIL) (Oh et al. 2018) and Double DQN with Prioritized Experience Replay (DQN-PER) (Schaul et al. 2015a), implemented on top of the Reinforcement Learning framework Machin (Li 2020).

We refer to "HRL" as our algorithm in which the task SMDP $\mathcal{S}^{\mathcal{T}}$ has to be learned online while exploring, but the compression function g is given. "HRL-TRANSFER" refers to our algorithm where the agent is first pretrained in order to learn the options on NineRooms0 in Figure 6.1 without any task and then exposed to NineRooms1, NineRooms2, NineRooms3 in sequence. In this case, the algorithm benefits from the transfer of the SMDP $\mathbb{S}$ while the manager policy (i.e. Q-values) is reset to 0 after training in each environment.

We can observe that the HRL algorithm learns faster than SIL and DQN-PER in all the environments. SIL and DQN-PER both rely only on random exploration, but once they find a positive reward, they can exploit it. In contrast, the exploration of HRL and HRL-TRANSFER is aided by the hierarchical structure. Both SIL and DQN-PER present high variance, and for some seeds, they are not even able to solve the task, given the budget of 100,000 iterations. We can also observe that HRL-TRANSFER does improve over HRL, and we would argue that this improvement could be larger if we choose harder tasks where the option policies for transitioning between regions become harder to learn.

6.2.4 Additional Empirical Evaluation

In this section, we present an additional empirical evaluation of our approach to learning a compression function, complementing the analysis reported in the previous sections.

Concretely, we evaluate our approach in the MountainCar environment, in which the state consists of the current location and velocity of the agent. In this environment, we collected a replay memory consisting of 200 trajectories of length 200 using a sub-optimal policy that can reach the goal state and can approximately cover all the state space, and learned a compression function with 20 regions.

In Figure 6.7 we show the result of this compression function where different colors represent different regions $z \in \mathcal{Z}$. Note that the compression function is able to cluster together states that are close in the environment, i.e. states where the car is at a similar position and velocity. In particular, states with low velocity near the center are *not* very similar to states with high velocity in the same location, and this is captured by the compression function.

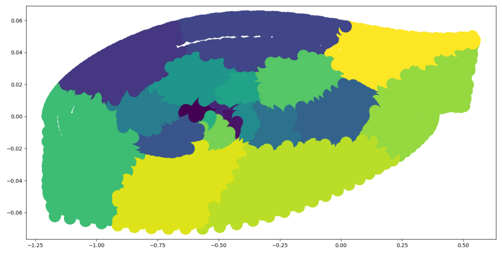


Figure 6.7: Results of the compression function in the MountainCar environment (axes represent location and velocity); different colors represent different regions $z \in \mathcal{Z}$.

6.3 Hyperparameters

Table 6.1 reports the values of the hyperparameters used to train the compression function and the HRL agent.

Table 6.2 reports the value of the hyperparameters used to train the DQN-PER and SIL agents.

Hyperparameters	***Value***
Worker Hyperparameters	

Neural Network Architecture	CONV1(32, (7, 7), (1, 1))
	FC1(32)
	FC2(32)

Activation Function	Relu
Learning rate	0.001
Optimizer	Adam
E-Greedy decay	0.9998
Batch size	100
Target network poliak update	0.05
Discount Factor	0.95
Replay buffer size	$5 * 10^5$
Replay type:	PrioritizedExperience Replay
Exponent for prioritization	0.6
Bias Correction	0.1
Manager Hyperparameters	
E-Exploration to learn the model	0.995
Discount Factor	0.95
Compression Function Hyperparameters	

Neural Network Architecture	CONV1(16, (1, 1), (1, 1))
	BatchNorm2D(16)
	CONV2(32, (5, 5), (1, 1))
	BatchNorm2D(32)
	CONV3(32, (3, 3), (1, 1))
	BatchNorm2D(32)
	FC1(64)
	BatchNorm1D(64)
	FC1(1)

Activation Function	Selu
w_H, w_D on GridWorld	0.2, 0.1
w_H, w_D on Mountain Car	2, 0.1
Learning rate	0.001
Optimizer	AdamW
Batch size	32, 64
Epochs	4000

Table 6.1: Hyperparameters used to train HRL and HRL-Transfer agents.

Hyperparameters	***Value***
DQN-PER Hyperparameters	
Same as Worker	
SIL Hyperparameters	
Neural Network Architecture	Same as Worker
Discount Factor	0.95
Replay type:	Prioritized Experience Replay
Exponent for prioritization	0.6
Bias Correction	0.1
Additional Hyperparameters	Same as reported in Self Imitation Learning paper

Table 6.2: Hyperparameters used to train SIL and DQN-PER agents.

6.4 Conclusion

We present a novel method for learning a hierarchical representation from sampled transitions in high-dimensional domains. The idea is to generate regions that partition the original state space and introduce options for performing transitions between regions. Experiments show that the learned representation can successfully be used to solve multiple tasks in the same environment, significantly speeding up learning compared to a flat learner.

An important direction for future work is to sample trajectories using a more informed exploration policy, since learning the compression function depends on having a variety of trajectories in different states. Another possible extension is to interleave representation learning with policy improvement, which may successively improve the quality of the sampled trajectories. Yet another possibility is to correct the compression function in states from which some region transitions are not possible.

Chapter 7

Distance Based Representation for Hierarchical Reinforcement Learning

In this chapter, we present a novel method for learning a state space partition of Markov decision processes. Our method partitions the state space by selecting representative centroids based on the notion of Minimum Action Distance (MAD) and defines subtasks for performing transitions between these centroids' regions. At the high level, we use SMDP Q-learning (Sutton, Precup, and Singh 1999) to decide which subtask or action to pursue next from a given centroid region or state. We formulate the problem of learning the MAD as a constrained optimization problem, and we show how it is possible to transform the constraints into a penalty term and solve the new objective via stochastic gradient descent. Preliminary results on grid world environments validate the method, by showing that it can learn options that accelerate learning.

State Space Partition properties

Here we summarize and briefly discuss only the state space partition properties (see Chp. 4) of the methodology presented in this chapter.

Pros:

✓ **Small SubMDPs**: The state space partitioning based on MAD proposed in this chapter is able to directly balance the size of sub-MDP by setting the hyperparameter d_t (see Section 7.2). This allows us to control the maximum size M(see Section 4.2) of each induced subMDP.

✓ **Strongly connected subMDPs**: By construction of the state space partitions each subMDP will be strongly connected (see Section 4.2). Clustering with the MAD distance d_{MAD} ensures that there will always be a path from any state belonging to a region to any other state of that region.

Cons:

× **Bottleneck exit states**: As we can see in Section 7.3, the MAD can be used to identify bottleneck states, however, the state space partitioning presented in this chapter does not favor partitions that allow having bottleneck exit states, and we leave this as future work.

× **Equivalent subMDPs**: The state partition presented in this chapter is not able to identify equivalent subMDPs (see Section 4.2).

7.1 Minimum Action Distance

In this section, we describe the notion of Minimum Action Distance, and we derive useful ways of computing this measure on finite MDPs and continuous or large MDPs.

We start by introducing some notation. We refer to the Minimum Action Distance (MAD) as the minimum number of actions to transition between any pair of states $(s, s') \in \mathcal{S}^2$.

Definition 6. *(Minimum Action Distance) Let $T(s' \mid \pi, s)$ be the random variable denoting the first time step in which state s' is reached in the MDP when starting from state s and following policy π. Then $d_{MAD} : \mathcal{S}^2 \to \mathbb{R}^+$ is defined as:*

$$d_{MAD}(s, s') := \min_{\pi} \min \left[T\left(s' \mid \pi, s\right) \right].$$

Similarly, we refer to the Shortest Path Distance (SPD) $d_{SPD}(s, s')$ as the expected minimum number of actions to transition between any pair of states $(s, s') \in \mathcal{S}^2$.

Definition 7. *(Shortest Path Distance) Let $T(s' \mid \pi, s)$ be the random variable for the first time step in which state s' is reached in the MDP. Then $d_{SPD} : \mathcal{S}^2 \to \mathbb{R}^+$ is defined as:*

$$d_{SPD}(s, s') := \min_{\pi} \mathbb{E}\left[T\left(s' \mid \pi, s\right)\right].$$

Note that SPD differs from MAD in the change from expected reaching time $\mathbb{E}\left[T\left(s' \mid \pi, s\right)\right]$ to minimum reaching time $\min\left[T\left(s' \mid \pi, s\right)\right]$ and the two definitions match in the case of deterministic MDP.

We can observe that the MAD is an asymmetric distance function (Mennucci 2013) and must satisfy the following properties:

- $d_{MAD} \geq 0$ and $\forall s \in \mathcal{S}$, $d_{MAD}(s, s) = 0$.
- $d_{MAD}(s, s') = d_{MAD}(s', s) = 0$ implies $s = s'$.
- $d_{MAD}(s, s') \leq d_{MAD}(s, s'') + d_{MAD}(s'', s')$ $\forall (s, s', s'') \in \mathcal{S}$.

7.1.1 Learning Minimum Action Distance from Adjacency Matrix

In discrete and finite MDPs we can compute the state-transition graph $G = (V, E)$ of an MDP. In this section, we will revise how to learn the minimum action distance from the graph adjacency matrix.

A state-transition graph $G = (V, E)$ of an MDP $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, r \rangle$ is a graph with nodes representing the states in the MDP and the edges representing state adjacency in the MDP. More precisely, $V = \mathcal{S}$, $e(s, s') \in E$ iff $\exists a\, \mathcal{P}(s, a, s') > 0$. An adjacency matrix represents a graph with a square matrix of size $|\mathcal{S}| \times |\mathcal{S}|$ with (i, j)-value being 1 if $e(s_i, s_j) \in E$ and 0 otherwise.

$$A^G_{ij} = \begin{cases} 0 & s_i = s_j \quad or \quad e(s_i, s_j) \notin E \\ 1 & e(s_i, s_j) \in E \end{cases} \qquad i, j = 1, \ldots, |\mathcal{S}|. \tag{7.1}$$

Having access to the adjacency matrix A^G we can simply compute the minimum action distance by using the Floyd-Warshall algorithm (Floyd 1962; Roy 1959; Warshall 1962).

The Floyd-Warshall algorithm compares all possible paths through the graph between each pair of vertices. It is able to do this with $\Theta\left(|V|^3\right)$ comparisons in a graph, even though there may be up to $\Omega\left(|V|^2\right)$ edges in the graph, and every combination of edges is tested. It does so by incrementally improving an estimate on the shortest path between two vertices until the estimate is optimal.

This dynamic programming procedure relies on having access to the edge weights, which in the case of MAD reduces to having access to the adjacency matrix A^G of $a_{i,j} = 1$ when s_i, s_j are connected by an edge.

Thanks to this we can define the shortest path on the A^G by just first computing the base cases:

$$d(s_i, s_j) = \begin{cases} 0 & s_i = s_j \\ 1 & a_{i,j} = 1 \\ \infty & a_{i,j} = 0 \quad and \quad s_i \neq s_j \end{cases} \qquad i, j = 1, \ldots, |\mathcal{S}|, \tag{7.2}$$

and subsequently computing the recursive case leveraging the triangle inequality property:

$$d(s_i, s_j) = \min(d(s_i, s_j), d(s_i, s_k) + d(s_k, s_j)) \quad \forall (s_i, s_j, s_k) \in \mathcal{S}^3. \tag{7.3}$$

Note that in case we do not have access to the adjacency matrix A^G this can be retrieved by interacting with the environment by visiting all the (s, s') transitions.

7.1.2 Symmetric embeddings

The Minimum Action Distance between states is a priori unknown and is not directly observable in continuous and/or noisy state spaces where we cannot simply enumerate the states and construct the adjacency matrix of the MDP. Instead, we will approximate an upper bound using the distances between states observed on trajectories. We introduce the notion of Trajectory Distance (TD) as follows:

Definition 8. *(Trajectory Distance) Given any trajectory $\tau = s_0, ..., s_n \sim \mathcal{M}$ collected in an MDP $\mathcal{M}$ and given any pair of states along the trajectory $(s_i, s_j) \in \tau$ such that $0 \leq i \leq j \leq n$, we define $d_{TD}(s_i, s_j \mid \tau)$ as*

$$d_{TD}(s_i, s_j \mid \tau) = (j - i), \tag{7.4}$$

i.e. the number of decision steps required to reach s_j from s_i on trajectory τ.

We start by observing that given any state trajectory $\tau = \{s_0, ..., s_n\}$, choosing any pair of states $(s_i, s_j) \in \tau$ with $0 \leq i \leq j \leq n$, their distance along the trajectory represents an upper bound of the MAD.

$$d_{MAD}(s_i, s_j) \leq d_{TD}(s_i, s_j \mid \tau). \tag{7.5}$$

Given a dataset of trajectories $\mathcal{D}$ collected by any unknown behavior policy, we can retrieve the MAD d_{MAD} by solving the following constrained optimization problem:

$$\begin{aligned} \min_{\theta} \quad & \sum_{\tau \in \mathcal{D}} \sum_{(s,s') \in \tau} (d_\theta(s, s') - d_{TD}(s, s' \mid \tau))^2, \\ \text{s.t.} \quad & d_\theta(s, s') \leq d_{TD}(s, s' \mid \tau) \quad \forall \tau \in \mathcal{D}, \forall (s, s') \in \tau, \\ & d_\theta(s_i, s_j) \leq d_\theta(s_i, s_k) + d_\theta(s_k, s_j), \quad \forall (s_i, s_j, s_k) \in \mathcal{S}_{\mathcal{D}}^3 \end{aligned} \tag{7.6}$$

where $(s, s') \in \tau$ refers to a one-step transition (i.e. $d_{TD}(s, s'|\tau) = 1$) in the trajectory $\tau \in \mathcal{D}$ while $(s_i, s_j, s_k) \in \mathcal{S}_{\mathcal{D}}^3$ indicates all the combinations of 3 states contained in the trajectory dataset $\mathcal{S}_{\mathcal{D}}$.

Note that the first constraint in 7.6 imposes an upper bound on one-step transitions, i.e. it says that two states (s, s') at distance one along a trajectory $d_{TD}(s, s'|\tau) = 1$ are either the same state $s = s'$ or they must satisfy $d_{MAD} = 1$. This allows us to approximate the MAD without having to identify whether two states along a trajectory are the same state or not.

As we can notice the second constraint in 7.6 imposes the triangle inequality that we have seen to be a property of the Minimum Action Distance. Moreover, this second constraint implies that we have to calculate it for all the combinations of 3 states contained in $\mathcal{S}_{\mathcal{D}}$ which can become intractable for large state spaces.

To address this issue we rely on an alternative formulation based on embedding the MAD in a parametric embedding space $\phi_\theta : \mathcal{S} \to \mathbb{R}^{dim}$ where a chosen distance metric that respects the triangle inequality (e.g. any norm $||\cdot||_p$) can be used to enforce the triangle inequality constraint.

Our goal is to learn a parametric state embedding $\phi_\theta : \mathcal{S} \to \mathbb{R}^{dim}$ such that the distance d between any pair of embedded states approximates the Minimum Action Distance.

We first show how to favor symmetric embeddings since it allows us to use norms as distance functions, e.g. the L1 norm $d(z, y) = ||z-y||_1$. Later we discuss possible ways to extend it to asymmetric distance functions. If we use symmetric embeddings we will have that for any pair of states $(s_i, s_j) \in \mathcal{S}$,

$$d(\phi_\theta(s_i), \phi_\theta(s_j)) \approx \min(d_{MAD}(s_i, s_j), d_{MAD}(s_j, s_i)). \tag{7.7}$$

We then formulate the problem of learning this embedding as a constrained optimization problem:

$$\begin{aligned} \min_\theta \quad & \sum_{\tau \in \mathcal{D}} \sum_{(s_i, s_j) \in \tau} \left(\|\phi_\theta(s_i) - \phi_\theta(s_j)\|_p - d_{TD}(s_i, s_j \mid \tau)\right)^2, \\ \text{s.t.} \quad & \left\|\phi_\theta(s) - \phi_\theta(s')\right\|_p \leq d_{TD}(s, s' \mid \tau) \quad \forall \tau \in \mathcal{D}, \forall (s, s') \in \tau. \end{aligned} \tag{7.8}$$

Intuitively, the objective is to make the embedded distance between pairs of states as close as possible to the observed trajectory distance while respecting the upper bound constraints. Without constraints, the objective is minimized when the embedding matches the expected Trajectory Distance $\mathbb{E}\left[d_{TD}\right]$ between all pairs of states observed on trajectories in the dataset $\mathcal{D}$. In contrast, constraining the solution to match the minimum TD with the upper-bound constraints $\|\phi_\theta(s) - \phi_\theta(s')\|_p \leq d_{TD}(s, s' \mid \tau)$ allows us to approximate the MAD. The precision of this approximation depends on the quality of the given trajectories.

To make the constrained optimization problem tractable, we relax the hard constraints in (7.8) and convert them into a penalty term to retrieve a simple unconstrained formulation. Moreover, we rely on sampling $(s_i, s_j, d_{TD}(s_i, s_j \mid \tau))$ and $(s, s', d_{TD}(s, s' \mid \tau))$ from the dataset of trajectories $\mathcal{D}$ making this formulation amenable for gradient descent and to fit within the optimization scheme of neural networks.

$$\mathcal{L} = \mathbb{E}_{(s_i, s_j, d_{TD}(s_i, s_j \mid \tau)) \sim \mathcal{D}} \left[\gamma^{d_{TD}(s_i, s_j \mid \tau)} (\|\phi_\theta(s_i) - \phi_\theta(s_j)\|_p - d_{TD}(s_i, s_j \mid \tau))^2 \right] + C, \quad (7.9)$$

where C is our penalty term defined as

$$C = \mathbb{E}_{(s, s', d_{TD}(s, s' \mid \tau)) \sim \mathcal{D}} \left[\max\Big(0, \|\phi_\theta(s) - \phi_\theta(s')\|_p - d_{TD}(s, s' \mid \tau)\Big)^2 \right]. \quad (7.10)$$

The penalty term C introduces a quadratic penalization of the objective for violating the upper-bound constraints $\|\phi_\theta(s) - \phi_\theta(s')\|_p <= d_{TD}(s, s' \mid \tau)$, while the term $\gamma^{d_{TD}(s_i, s_j \mid \tau)} \in (0, 1]$ prioritizes small trajectory distances (i.e. distances between states that are close along a trajectory). Intuitively, this makes sense since there is more uncertainty regarding the MAD of pairs of states that are further apart on a trajectory. In Figure 7.1 we can see a symmetric MAD representation learned in a simple grid world.

7.1.3 Asymmetric semi-norm embeddings

In the previous section, we have seen how it is possible to define the MAD embedding problem with the use of norms $|| \cdot ||_p$. While the formulation is useful to understand how it is possible to remove the triangle inequality constraint in 7.6 the Minimum Action Distance is naturally asymmetric and we would like embedding that preserves this asymmetry.

A norm is a function $\| \cdot \| : \mathcal{X} \to \mathbb{R}$ satisfying, $\forall x, y \in \mathcal{X}, \alpha \in \mathbb{R}^+$:

- **N1** (Pos. def.). $\|x\| > 0$, unless $x = 0$.
- **N2** (Pos. homo.). $\alpha\|x\| = \|\alpha x\|$, for $\alpha \geq 0$.
- **N3** (Subadditivity). $\|x + y\| \leq \|x\| + \|y\|$.
- **N4** (Symmetry). $\|x\| = \| - x\|$.

An asymmetric semi-norm satisfies, **N2**, **N3** but not necessarily **N1**, **N4**.

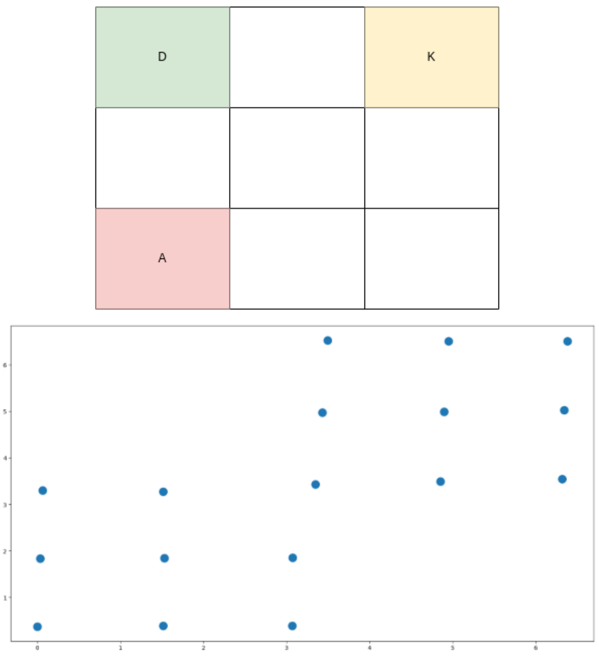


Figure 7.1: Top: a simple grid world where an agent has to pick up a key and open a door (key and door positions are fixed). Bottom: the learned state embedding ϕ on $\mathbb{R}^2$. The state (x, y, has_key) is composed of the agent's location and whether or not it holds the key.

A convex function $f : \mathcal{X} \to \mathbb{R}$ is a function satisfying **C1** : $\forall x, y \in \mathcal{X}, \alpha \in [0, 1] : f(\alpha x + (1 - \alpha)y) \leq \alpha f(x) + (1 - \alpha)f(y)$. The commonly used ReLU activation, $\text{relu}(x) = \max(0, x)$, is convex.

Is easy to observe that any **N2** and any **N3** function is convex and thus that any asymmetric semi-norm is convex.

Motivated by this relationship between convex functions and norms Pitis et al. (2020), introduced Wide Norms a parametric distance that models symmetric and asymmetric norms.

A Wide Norm is any combination of symmetric/asymmetric semi-norms. They are based on the Mahalanobis norm of $x \in \mathbb{R}^{dim}$, parametrized by $W \in \mathbb{R}^{m \times n}$, defined as $||x||_W = ||Wx||_2$.

Symmetric Wide Norms are then defined as:

$$||x||_{WN} = \text{maxmean}_i \left(\|W_i x\|_2\right) \text{ where } W_i \in \mathbb{R}^{m_i \times n} \text{ with } m_i \leq n.$$

Where maxmean indicates:

$$\text{maxmean}\left(x_1, x_2, \ldots, x_n\right) = \alpha \max\left(x_1, x_2, \ldots, x_n\right) + (1-\alpha)\,\text{mean}\left(x_1, x_2, \ldots, x_n\right)$$

While asymmetric Wide Norms are defined as:

$$||x|_{WN} = ||W relu(x :: -x)||_2 \text{ where } W_i \in \mathbb{R}^{m_i \times n} \text{ with } m_i \leq n.$$

We can then use the parametrized Wide Norm distance to constrain the triangular inequality on the embedding space:

$$\mathcal{L} = \mathbb{E}_{(s_i, s_j, d_{TD}(s_i, s_j|\tau)) \sim \mathcal{D}} \left[\gamma^{d_{TD}(s_i, s_j|\tau)} (||\phi_\theta(s_i) - \phi_\theta(s_j)|_{WN} - d_{TD}(s_i, s_j \mid \tau))^2\right] + C, \tag{7.11}$$

where C is our penalty term defined as

$$C = \mathbb{E}_{(s, s', d_{TD}(s, s'|\tau)) \sim \mathcal{D}} \left[\max\left(0, ||\phi_\theta(s) - \phi_\theta(s')|_{WN} - d_{TD}(s, s' \mid t)\right)^2\right]. \tag{7.12}$$

7.2 Minimum Action Distance State Space Partitions

In the previous section, we have seen how it is possible to learn the MAD from a dataset of trajectory. In this section, we will use the MAD to partition the state space of an MDP.

Given an MDP $\mathcal{M}$ and a distance threshold hyperparameter $d_t \in [0, \max_{(s_i, s_j)} d_{MAD}(s_i, s_j)]$, we propose to partition the state space $\mathcal{S}$ online by acting in $\mathcal{M}$ (see Algorithm 7.1), collecting transitions (s, a, s', r) and adding centroids c to a list C when $d_{MAD}(c, s) > d_t$, $\forall s \in \mathcal{S}$. Note that d_{MAD} is learned in an online manner, simply by minimizing Loss 7.11 on an increasing dataset of collected trajectories $\mathcal{D}$. It is worth highlighting that it is enough to have an accurate d_{MAD} for every $(s_i, s_j) \in \mathcal{S}$

where $d_{MAD}(s_i, s_j) \leq (d_t + 1)$ in order to create a correct d_t-partition of the state space.

The set of centroids C together with the MAD distance defines our deterministic state space partition function $g : \mathcal{S} \rightarrow C$:

$$g(s, C) = \min\{c \in C \mid d_{MAD}(c, s) \leq \min_{c' \in C} d_{MAD}(c', s)\} \tag{7.13}$$

Algorithm 7.1 MAD State Space Partition embedded in an Episodic RL framework

```
Input: distance treshold $d_t$
$C = \emptyset, \mathcal{T} = \emptyset$
while within episodes budget do
  $t = \emptyset$
  $s \leftarrow$ initial state
  if $C = \emptyset$ then
    $C \leftarrow \{s\}$
  else if $d_{MAD}(c, s) > d_t, \forall c \in C$ then
    $C \leftarrow C \cup \{s\}$
  end if
  while episode not terminate do
    $a \leftarrow agent(s)$
    $(s', r, done) \leftarrow env(a)$
    if $d_{MAD}(c, s) > d_t, \forall c \in C$ then
      $C \leftarrow C \cup \{s'\}$
    end if
    $t = t \cup \{(s, a, s', r, done)\}$
    $s \leftarrow s'$
  end while
  $\mathcal{T} = \mathcal{T} \cup \{t\}$
  Train $d_{MAD}$ on $\mathcal{D}$
  Refine $C$
end while
```

7.2.1 Options Representation

The agent then uses the compression function in an online manner, by exploring the environment and finding centroid region transitions, i.e. consecutive states s_t and s_{t+1} such that $g(s_t, C) \neq g(s_{t+1}, C)$. Once a new centroid region c' is discovered the agent adds it to the set of centroids C and the associated options $o_{\cdot,c'}$ to the option set $\mathcal{O}$. The agent also updates and maintains a directed graph whose nodes are centroid regions and whose edges represent the option neighbor relation. Hence, the next time that the agent visits centroid region c, it knows which options are available.

We refer to $\mathcal{Y} \subseteq C \times C$ to be the subset of pairs of distinct regions (c, c') that appear as region transitions while exploring. By abuse of notation, we often use c to denote both the centroid state and the centroid region.

To learn the options we define a Goal-Conditioned Reinforcement Learning (GCRL) MDP (Andrychowicz et al. 2017; Schaul et al. 2015b) $\mathcal{M}_O = \langle \mathcal{S}, \mathcal{C}, \mathcal{A}, \mathcal{P}, r \rangle$ where C represent the set of goal states, and we change the usual notation by defining $G = C$. In GCRL the observation is augmented with a goal g sampled from G that the agent is required to achieve when taking a decision in an episode, and in our case, each goal is a centroid $c \in C$. The reward $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \times \mathcal{C} \to \mathbb{R}$ is then defined on centroids C:

$$r(s, a, s', c) = \begin{cases} -1, & \text{if } g(s', C) \neq c, \\ 0, & \text{otherwise.} \end{cases} \tag{7.14}$$

Therefore, the objective of GCRL is to reach a centroid state via a centroid-conditioned policy $\pi_{GC} : \mathcal{S} \times C \to \Delta(\mathcal{A})$ that maximizes the expectation of the cumulative return over the goal distribution.

Given a centroid region $c \in C$, let $\mathcal{S}_c$ be the subset of states that map to c, i.e. $\mathcal{S}_c = \{s \in \mathcal{S} : g(s, C) = c\}$. For each pair (c, c') of adjacent centroid regions, we introduce an option $o_{c,c'} = \langle \mathcal{I}_c, \pi_{c,c'}, \beta_c \rangle$ to transition from centroid region c to c'. Option $o_{c,c'}$ is applicable in centroid region c, i.e. $I_c = \mathcal{S}_c$, uses the goal conditioned policy $\pi_{c,c'} = \pi_{GC}(\cdot, c')$, and terminates as soon it reaches the goal centroid region, i.e. $\beta_c(s) = 1$ if $g(s, C) = c'$, or it consumes the budget steps for the option.

We then define $\mathcal{O} = \{o_{c,c'} : (c, c') \in \mathcal{Y}\}$ to be the set of options for performing region transitions, and let $O_c = \{o \in O : \mathcal{I}_c = \mathcal{S}_c\}$ be the subset of options applicable in centroid region c.

Skill SMDP

We are now ready to introduce the SMDP $\mathbb{S}$. We consider skills in RL (Bowling and Veloso 1998; Pickett and Barto 2002; Sutton, Precup, and Singh 1999; Thrun and Schwartz 1994). Skill in this context refers to any mapping from states to actions that could aid in the learning of a task. We define a skill SMDP $\mathbb{S} = \langle \mathcal{S}, \mathcal{A} \cup \mathcal{O}, \mathcal{P}', r \rangle$, where we keep the original state space $\mathcal{S}$, and we consider an extended action space $\mathcal{A} \cup \mathcal{O}$. In this way we endow the agent with the ability to reach centroid partitions $c \in C$ that could help the agent to explore the state space in a more effective way. Note that in this SMDP $\mathbb{S}$ we can retrieve the optimal policy of the original MDP $\mathcal{M}$ since the primitive actions $\mathcal{A}$ are still available in the SMDP $\mathbb{S}$.

7.3 Bottleneck State Discovery

In this section, we show how the MAD can be used to discover bottleneck states based on the graph centrality measure. Here we consider a simple definition of centrality (Bavelas 1950):

$$\text{centrality}(s) = \left(\sum_{s' \in \mathcal{S}} \text{dist}\left(s, s'\right) \right)^{-1}, \tag{7.15}$$

where dist is a generic distance measure. We propose to use the MAD $dist = d_{MAD}$ to discover bottleneck states. In Figure 7.2 we calculate $centrality(s)$ for all states, and as we can see the states with higher centrality measure indeed concentrate around the position of the key that is a bottleneck state since it connects two highly dense regions.

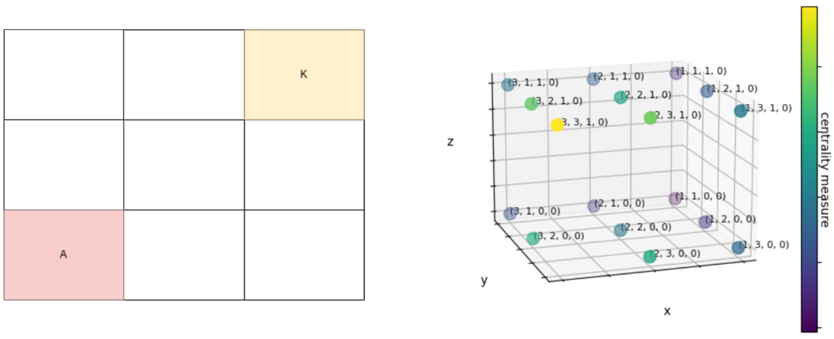


Figure 7.2: Left: a simple grid world where an agent has to pick up a key (key position is fixed). Right: the learned Centrality measure. The state $(x, y, haskey)$ is composed of the agent location and whether or not it holds the key.

7.4 Preliminary Results

We report some preliminary results on using the Minimum Action Distance representation. We learn the minimum action distance using a neural network parametrization, and with Wide Norm distance by minimizing loss 7.11 on an increasing dataset of trajectories $\mathcal{D}$. Using the same dataset $\mathcal{D}$ we learned the goal-conditioned options in an offline manner using DDQN (Van Hasselt, Guez, and Silver 2016) and relabeling transitions in hindsight (Andrychowicz et al. 2017). For each transition $(s, a, s', r, done)$ we transform it in a goal-conditioned transition $(s, a, s', c, r_c, done_c)$ by including a goal $c \in C$ and transforming the reward 7.14 and *done* condition in accordance with the goal.

We experiment on two simple grid worlds: where SMDP tabular Q learning (Sutton, Precup, and Singh 1999) can be used at the SMDP level.

1. **EmptyGridWorld**: In this first environment we considered an empty grid world of size 25 × 25.The agent starts in $s_i = (1, 1)$ and has to reach a goal state placed at position $s_g = (25, 25)$ and only then it receives a reward of +1. The environment is modeled as episodic with a maximum number of steps of 200 after which the

agent is reset to the initial position s_i. The agent acts in the environment by selecting to move UP, DOWN, LEFT or RIGHT and actions are deterministic.

2. **KeyDoorGridWorld**: In this environment, we consider a grid world of size 8×8. The agent starts in $s_i = (1, 1, 0)$ and has to pick up a key positioned at $s_{key} = (8, 8, 0)$ and open a door at position $s_{door} = (8, 1, 1)$ and only then receives a reward of $+1$. The environment is modeled as episodic with a maximum number of steps of 100 after which the agent is reset to the initial position s_i. The agent acts in the environment by selecting to move UP, DOWN, LEFT or RIGHT and actions are deterministic.

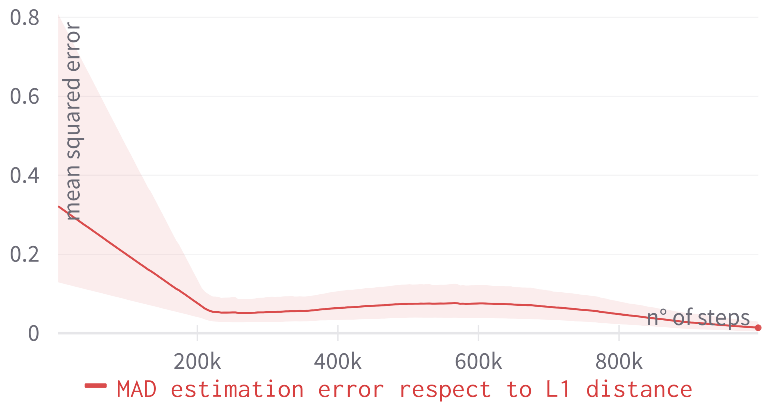


Figure 7.3: Mean squared error of the MAD distance learned on the EmptyGridWorld respect to the ground truth L1 distance

We first analyze whether we can learn the MAD in the EmptyGridWorld where we have a ground truth reference that is the L1-norm. In Figure 7.3 we compute the mean squared error between the MAD distance learned and the ground truth distance over the entire state space $\frac{\sum_{(s_i,s_j)\in\mathcal{S}\times\mathcal{S}} ||\phi_\theta(s_i)-\phi_\theta(s_j)|_{WN}-\|s_i-s_j\|_1)^2}{|\mathcal{S}\times\mathcal{S}|}$. Remember that this distance is learned online and as we can see we are able to correctly learn the MAD state embedding with WN parametric distance.

Figure 7.4 compares a Q learning agent on the original EmptyGridWorld MDP $\mathcal{M}$ (FLAT agent) against SMDP Q learning on the Skill SMDP $\mathbb{S}$ (SKILL-SMDP agent). Distance d_{MAD} and options $\mathcal{O}$ are learned online and centroids C discovered online using the learned d_{MAD}. Both agents use the same ϵ-greedy exploration with $\epsilon_{decay} = 0.9995$ updated at each step as $\epsilon = \epsilon \cdot \epsilon_{decay}$. Results are averaged using a moving window average of 100 episodes and execution is interrupted once it converged to the maximum reward. In Figure 7.4 top, we can observe the maximum reward collected against the number of steps executed in the environment. Both SKILL-SMDP agent and FLAT agent are eventually able to converge to the maximum reward but SKILL-SMDP converges faster needing less samples to learn the optimal policy. In Figure 7.4 bottom, we can observe that both agents are able to find the shortest path to the goal. This is possible since we consider a Skill SMDP with action set $\mathcal{A} \cup \mathcal{O}$.

Figure 7.5 considers the KeyDoorGridWorld environment. As we can see on the top we have similar results where SKILL-SMDP agent outperforms FLAT agent in terms of sample time. Surprisingly on the bottom we can see that SKILL-SMDP has been able to find the shortest path while FLAT agent could not learn the shortest path in the limited exploration time.

EmptyGridWorld

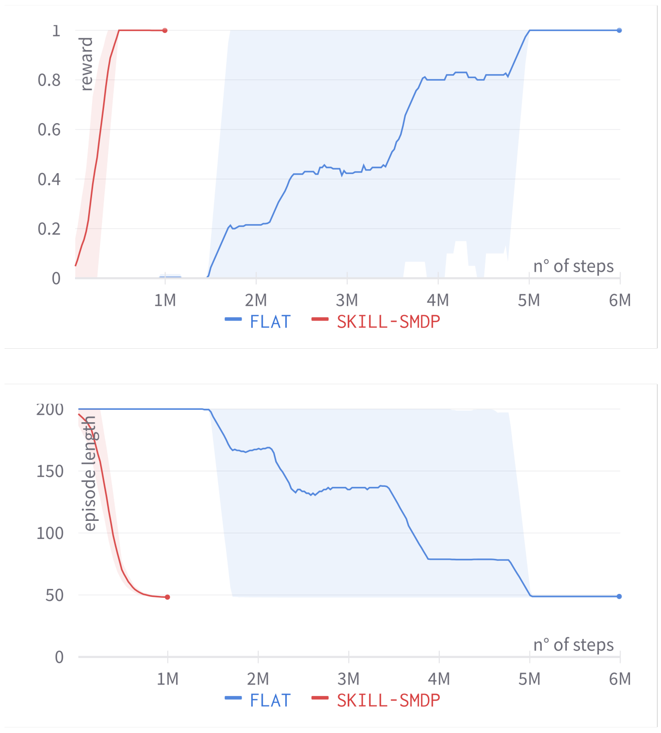


Figure 7.4: EmptyGridWorld results, where we compare Q learning on the original MDP $\mathcal{M}$ (FLAT) against SMDP Q learning on the SMDP $\mathbb{S}$ (SKILL-SMDP).

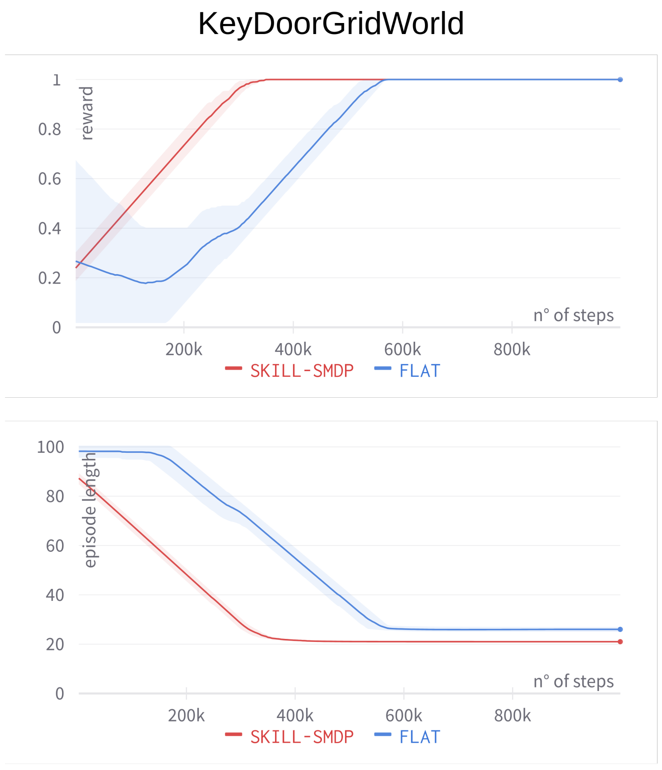


Figure 7.5: KeyDoorGridWorld results, where we compare Q learning on the original MDP $\mathcal{M}$ (FLAT) against SMDP Q learning on the SMDP $\mathbb{S}$ (SKILL-SMDP).

7.5 Conclusion

We presented a novel method to learn a partition of the state space that induces strongly connected subMDPs. The idea is to discover centroids C that partition the state space using a learned Minimum Action Distance d_{MAD} in an online manner. We then introduce options to transition between different centroid regions creating a Skill SMDP $\mathbb{S}$. Preliminary experiments show that the induced skill SMDP can significantly speed up learning compared to a flat learner.

This work is still at a preliminary phase and more work is necessary in order to understand when and how much the introduced options will help in speeding up learning and exploration. Moreover, in this work, we consider using the state space partition only to augment the action space with options that help reach distant states in terms of d_{MAD}. A possible alternative would be to use the learned representation to create a hierarchy as we did in Chapters 5, 6

Part III

Representation Learning for Goal Conditioned Reinforcement Learning

Chapter 8

State Representation Learning For Goal Conditioned Reinforcement Learning

In this chapter, we present a novel state representation for reward-free Markov decision processes. The idea is to learn, in a self-supervised manner, an embedding space where distances between pairs of embedded states correspond to the minimum number of actions needed to transition between them. Compared to previous methods, our approach does not require any domain knowledge, learning from offline and unlabeled data. We show how this representation can be leveraged to learn goal-conditioned policies, providing a notion of similarity between states and goals and a useful heuristic distance to guide planning and reinforcement learning algorithms. Finally, we empirically validate our method in classic control domains and multi-goal environments, demonstrating that our method can successfully learn representations in large and/or continuous domains.

In comparison to the previous chapter where we aim to learn representations useful for Hierarchical Reinforcement Learning, here our focus shifts to learning representations useful for Goal Conditioned Reinforcement Learning.

8.1 Contribution

In this section, we present our main contribution, a method for learning a state representation of an MDP that can be leveraged to learn goal-conditioned policies. The state representation learned here resembles the one that we introduce in 7.1 with just a few differences for the choice of regularizer and norm distance.

We formulate the problem of learning this embedding as a constrained optimization problem, using the symmetric L1 norm as embedding distance:

$$\begin{aligned} \min_{\theta} \quad & \sum_{\tau \in \mathcal{D}} \sum_{(s_i, s_j) \in \tau} \left(\|\phi_\theta(s_i) - \phi_\theta(s_j)\|_1 - d_{TD}(s_i, s_j \mid \tau) \right)^2, \\ \text{s.t.} \quad & \|\phi_\theta(s_i) - \phi_\theta(s_j)\|_1 \leq d_{TD}(s_i, s_j \mid \tau) \quad \forall \tau \in \mathcal{D}, \forall (s_i, s_j) \in \tau. \end{aligned} \tag{8.1}$$

In contrast with the representation we have seen in 7.1, here the constraint is defined over all state pairs $(s_i, s_j) \in \mathcal{D}$.

Intuitively, the objective is to make the embedded distance between pairs of states as close as possible to the observed trajectory distance while respecting the upper bound constraints. Without constraints, the objective is minimized when the embedding matches the expected Trajectory Distance $\mathbb{E}[d_{TD}]$ between all pairs of states observed on trajectories in the dataset $\mathcal{D}$. In contrast, constraining the solution to match the minimum TD with the upper-bound constraints $\|\phi_\theta(s) - \phi_\theta(s')\|_1 \leq d_{TD}(s, s' \mid \tau)$ allows us to approximate the MAD.

To make the constrained optimization problem tractable, we relax the hard constraints in (8.1) and convert them into a penalty term in order to retrieve a simple unconstrained formulation. Moreover, we rely on sampling $(s_i, s_j, d_{TD}(s_i, s_j \mid \tau))$ from the dataset of trajectories $\mathcal{D}$ making this formulation amenable for gradient descent and to fit within the optimization scheme of neural networks.

$$\mathcal{L} = \mathbb{E}_{(s_i, s_j, d_{TD}(s_i, s_j \mid \tau)) \sim \mathcal{D}} \left[\tfrac{1}{d_{TD}(s, s' \mid t)^2} \left(\|\phi_\theta(s_i) - \phi_\theta(s_j)\|_1 - d_{TD}(s_i, s_j \mid \tau) \right)^2 \right] + C, \tag{8.2}$$

where C is our penalty term defined as

$$C = \mathbb{E}_{(s_i, s_j, d_{TD}(s,s'|\tau)) \sim \mathcal{D}} \frac{1}{d_{TD}(s,s'|\tau)^2} \left[\max \left(0, \|\phi_\theta(s) - \phi_\theta(s_i)\|_1 - d_{TD}(s, s' \mid \tau)\right)^2 \right]. \tag{8.3}$$

The penalty term C introduces a quadratic penalization of the objective for violating the upper-bound constraints $\|\phi_\theta(s) - \phi_\theta(s')\|_1 <= d_{TD}(s, s' \mid \tau)$, while the term $\frac{1}{d_{TD}(s,s'|\tau)^2}$ normalizes each sample loss to be in the range $[0, 1]$. The normalizing term also has the effect of prioritizing pairs of states that are close together on a trajectory, while giving less weight to pairs of states that are further apart. Intuitively, this makes sense since there is more uncertainty regarding the MAD of pairs of states that are further apart on a trajectory.

8.1.1 Learning Transition Models

In the previous section, we showed how to learn a state representation that encodes a distance metric between states. This distance allows us to identify states s_t that are close to a given goal state, i.e. $\|\phi_\theta(s_t), \phi_\theta(s_{goal})\|_1 < \epsilon$, or to measure how far we are from the goal state, i.e. $\|\phi_\theta(s_t), \phi_\theta(s_{goal})\|_1$. However, on its own, the distance metric does not directly give us a policy for reaching the desired goal state.

In this section we propose a method to learn a transition model of actions, that combined with our state representation allows us to plan directly in the embedded space and derive policies to reach any given goal state. Given a dataset of trajectories $\mathcal{D}$ and a state embedding $\phi_\theta(s)$, we seek a parametric transition model $\rho_\zeta(\phi_\theta(s), a)$ such that for any triple $(s, a, s') \in \mathcal{D}$, $\rho_\zeta(\phi_\theta(s), a) \approx \phi_\theta(s')$.

We propose to learn this model simply by minimizing the squared error as

$$\min_\zeta \quad \sum_t^{\mathcal{T}} \sum_{s,a,s'}^{t} \left[(\rho_\zeta(\phi_\theta(s), a) - \phi_\theta(s'))^2 \right]. \tag{8.4}$$

Note that in this minimization problem, the parameters θ of our state representation are fixed, since they are considered known and are thus not optimized at this stage.

8.1.2 Latent space planning

The functions ρ_ζ and ϕ_θ together represent an approximate model of the underlying MDP.

We propose a Model Predictive Control algorithm that we call Plan-Dist, which computes a policy to reach a given desired goal state $s_{goal} \in \mathcal{S}$ by unrolling trajectories for a fixed horizon H in the embedded space. Plan-Dist uses the negative distance between the actual state s_t and the goal state s_{goal} as the desired reward function to be maximized, i.e. $r(s) = -\|\phi_\theta(s_t), \phi_\theta(s_{goal})\|_1$. Our algorithm considers discrete action spaces and discretizes the action space otherwise. Plan-Dist samples a number N of action trajectories $T_{N,H}$ from the set of all possible action sequences of length H, $T_{N,H} \subset A_H$. The trajectories are then unrolled recursively in the latent space starting from our actual state s_t and using the transition model $\phi_\theta(s_{t+1}) \approx \rho_\zeta(\phi_\theta(s_t), a_t)$. At time step t, the first action of the trajectory that minimizes the distance to the goal is performed and this process is repeated at each time step until a terminal state is reached (cf. Algorithm 8.1).

Algorithm 8.1 PLAN-DIST

```
1: **Input**: environment $e$, state embedding $\phi_\theta$, transition model $\rho_\zeta$,
   horizon $H$, number $N$ of trajectories to evaluate
2: $s \leftarrow initialstate$
3: $s_{goal} \leftarrow goalstate$
4: $z_{goal} \leftarrow \phi_\theta(s_{goal})$
5: while within budget do
6:    $T_{N,H} \leftarrow$ sample $N$ action sequences of length $H$
7:    $t_{MaxReward} \leftarrow None$
8:    $r_{max} \leftarrow MinReward$
9:    for $t_a \in T_{N,H}$ do
10:      $z = \phi_\theta(s)$
11:      $r = r - \|z, z_{goal}\|_1$
12:      for $a_t \in t_a$ do
13:         $z_{t+1} = \rho_\zeta(z, a_t)$
14:         $r = r - \|z_{t+1}, z_{goal}\|_1$
15:      end for
16:      if $r > r_{max}$ then
17:         $r_{max} = r$
18:         $t_{MaxReward} \leftarrow t_a$
19:      end if
20:   end for
21:   $s' \leftarrow$ apply action $t_{MaxReward}[0]$ in state $s$
22:   $s = s'$
23: end while
```

8.1.3 Reward Shaping

Our last contribution is to show how to combine prior knowledge in the form of goal states and our learned distance function to guide existing reinforcement learning algorithms.

We assume that a goal state is given and we augment the environment reward $r(s, a, s')$ observed by the reinforcement learning agent with Potential-based Reward Shaping (Ng, Harada, and Russell 1999) of the form:

$$\overline{r}(s,a,s') = r(s,a,s') + F(s,\gamma,s'), \tag{8.5}$$

where F is our potential-based reward:

$$F(s,\gamma,s') = -\gamma\|\phi_\theta(s'),\phi_\theta(s_{goal})\|_1 + \|\phi_\theta(s),\phi_\theta(s_{goal})\|_1. \tag{8.6}$$

Here, $\|\phi_\theta(\cdot),\phi_\theta(s_{goal})\|_1$ represents our estimated Minimum Action Distance to the goal s_{goal}. Note that for a fixed goal state s_{goal}, $-\|\phi_\theta(\cdot),\phi_\theta(s_{goal})\|_1$ is a real-valued function of states which is maximized when $-\|\phi_\theta(\cdot),\phi_\theta(s_{goal})\|_1 = 0$.

Intuitively our reward shaping schema is forcing the agent to reach the goal state as soon as possible while maximizing the environment reward $r(s,a,s')$. By using potential-based reward shaping $F(s,\gamma,s')$ we are ensuring that the optimal policy will be invariant (Ng, Harada, and Russell 1999).

8.2 Experimental Results

In this section, we present results from experiments where we learn a state embedding and transition model offline from a given dataset of trajectories. We then use the learned models to perform experiments in two settings:

1. Offline goal-conditioned policy learning: Here we evaluate the performance of our Plan-Dist algorithm against GCSL (Ghosh et al. 2020).

2. Reward Shaping: In this setting we use the learned MAD distance to reshape the reward of a DDQN(Van Hasselt, Guez, and Silver 2016) agent (DDQN-PR) for discrete action environments and DDPG(Lillicrap et al. 2015) for continuos action environment (DDPG-PR), and we compare it to their original versions.

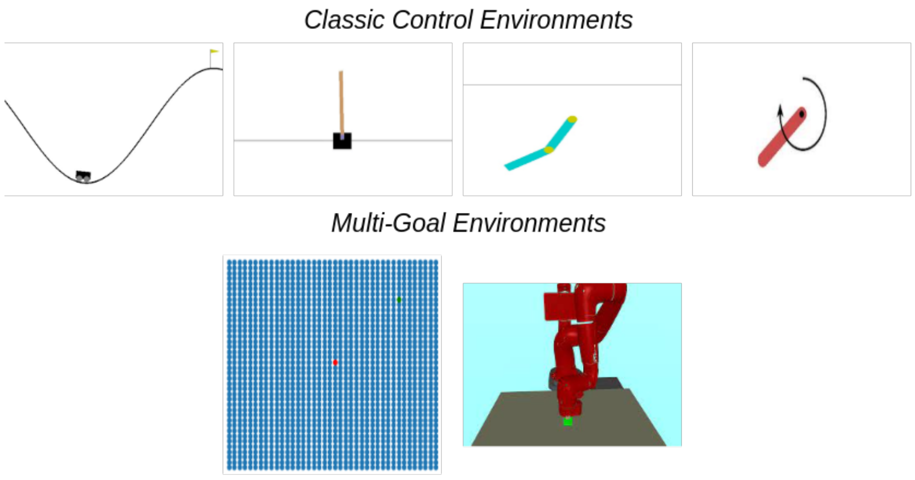


Figure 8.1: Evaluation Tasks. Top row: MountainCar-v0, CartPole-v0, AcroBot-v1 and Pendulum-v0. Bottom row: GridWorld and SawyerReachXYZEnv-v1.

8.2.1 Dataset Collection and Domain Description

We test our algorithms on the classic RL control suite (cf. Figure 8.1). Even though termination is often defined for a range of states, we fix a single goal state among the termination states. These domains have complex dynamics and random initial states, making it difficult to reach the goal state without dedicated exploration. The goal state selected for each domain is:

- MountainCar-v0: [0.50427865, 0.02712902]
- CartPole-v0: [0, 0, 0, 0]
- AcroBot-v1: [-0.9661, 0.2581, 0.8875, 0.4607, -1.8354, -5.0000]
- Pendulum-v0: [1, 0, 0]

Additionally, we test our model-based algorithm Plan-Dist in two multi-goal domains(see. Fig. 8.1):

- A 40x40 GridWorld.

- The multiworld domain SawyerReachXYZEnv-v1, where a multi-jointed robotic arm has to reach a given goal position.

In each episode, a new goal s_{goal} is sampled at random, so the set of possible goal states G equals the entire state space $\mathcal{S}$. These domains are challenging for reinforcement learning algorithms, and even previous work on goal-conditioned reinforcement learning usually considers a small fixed subset of goal states.

In each of these domains, we collect a dataset that approximately covers the state space, since we want to be able to use any state as a goal state. Collecting these datasets is not trivial. As an example, consider the MountainCar domain where a car is on a one-dimensional track, positioned between two mountains. A simple random trajectory will not be enough to cover all the state space since it will get stuck in the valley without being able to move the cart on top of the mountains. Every domain in the classic control suite presents this exploration difficulty and for these environments, we rely on collecting trajectories performed by the algorithms DDQN(Van Hasselt, Guez, and Silver 2016) and DDPG(Lillicrap et al. 2015) while learning a policy for these domains. Note that we use DDPG only in the Pendulum domain, which is characterized by a continuous action space.

In Table 8.1 we report the size, the algorithm/policy used to collect the trajectories, the average reward and the maximum reward of each dataset. Note that the average reward is far from optimal and that both Plan-Dist (our offline algorithm) and GCSL improve over the dataset performance (cf. Figure 8.2).

Environments	***# Trajectories Dataset***	***Algorithm to Collect Trajectories***	***Avg Reward Dataset***	***Max Reward Dataset***
MountainCar-v0	100	DDQN	-164.26	-112
CartPole-v0	200	DDQN	+89.42	+172
AcroBot-v1	100	DDQN	-158.28	-92.0
Pendulum-v0	100	DDPG	-1380.39	-564.90
GridWorld	100	RandomPolicy	–	–
SawyerReach-XYZEnv-v1	100	RandomPolicy	–	–

Table 8.1: Dataset description.

8.2.2 Learning a State Embedding

The first step of our procedure consists in learning a state embedding ϕ_θ from a given dataset of trajectories $\mathcal{D}$. From each trajectory $\tau_i = \{s_0, ..., s_n\} \in \mathcal{D}$ we collect all samples $(s_{i|\tau_i}, s_{j|\tau_i}, d_{TD}(s_{i|\tau_i}, s_{j|\tau_i} \mid \tau_i))$, $0 \leq i \leq j \leq n$, and populate a Prioritized Experience Replay (PER) memory (Schaul et al. 2015a). We use PER to prioritize the samples based on how much they violate our penalty function in (8.2).

We used mini-batches B of size 512 with the AdamW optimizer (Loshchilov and Hutter 2017) and a learning rate of $5 * 10^{-4}$ for 100,000 steps to train a neural network ϕ_θ by minimizing the loss in (8.2). Moreover, we used an embedding dimension of size 64 with an L1 norm as the metric to approximate the MAD distance. Empirically, the L1 norm turns out to perform better than the L2 norm in high-dimensional embedding spaces. These findings are in accordance with theory (Aggarwal, Hinneburg, and Keim 2001).

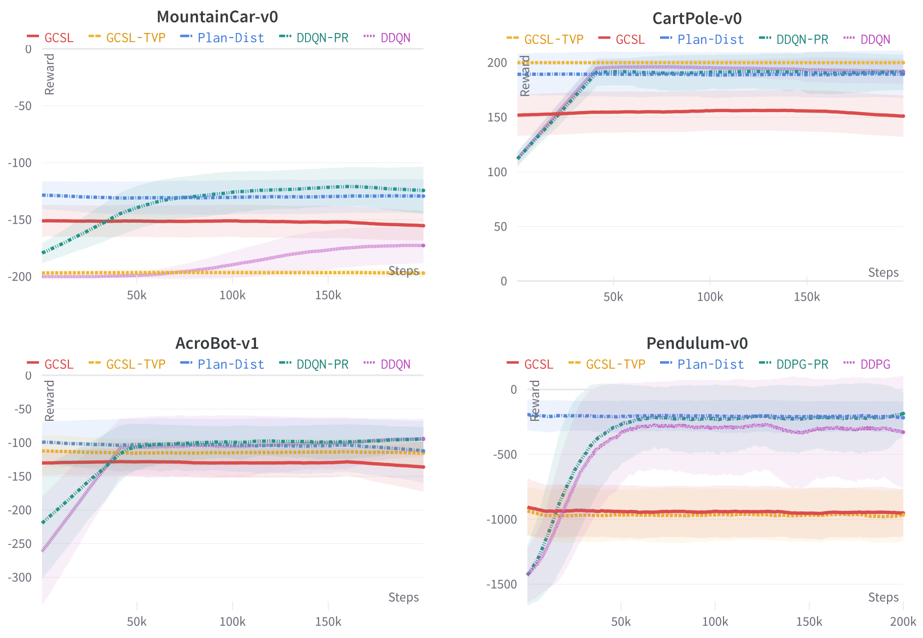


Figure 8.2: Results in the classic RL control suite.

8.2.3 Learning Dynamics

We use the same dataset of trajectories $\mathcal{D}$ to learn a transition model. We collect all the samples (s, a, s') in a dataset $\mathcal{D}_f$ and train a neural network ρ_ζ using mini-batches B of size 512 with the AdamW optimizer (Loshchilov and Hutter 2017) and a learning rate of $5 * 10^{-4}$ for 10,000 steps by minimizing the following loss derived from (8.4):

$$\mathcal{L}(\mathcal{B}) = \sum_{s,s',d_{TD}}^{B} \left[(\rho_\zeta(\phi_\theta(s), a) - \phi_\theta(s'))^2\right] \tag{8.7}$$

8.2.4 Experiments

We compare our algorithm Plan-Dist against an offline variant of GCSL, where GCSL is trained from the same dataset of trajectories as our models ϕ_θ and ρ_ζ. The GCSL policy and the models ϕ_θ and ρ_ζ are all learned offline and frozen at test time.

Ghosh et al. (Ghosh et al. 2020) propose two variants of the GCSL algorithm, a Time-Varying Policy where the policy is conditioned on the remaining horizon $\pi(a|s, g, h)$ (in our experiments we refer to this as GCSL-TVP) and a horizon-less policy $\pi(a|s, g)$ (we refer to this as GCSL).

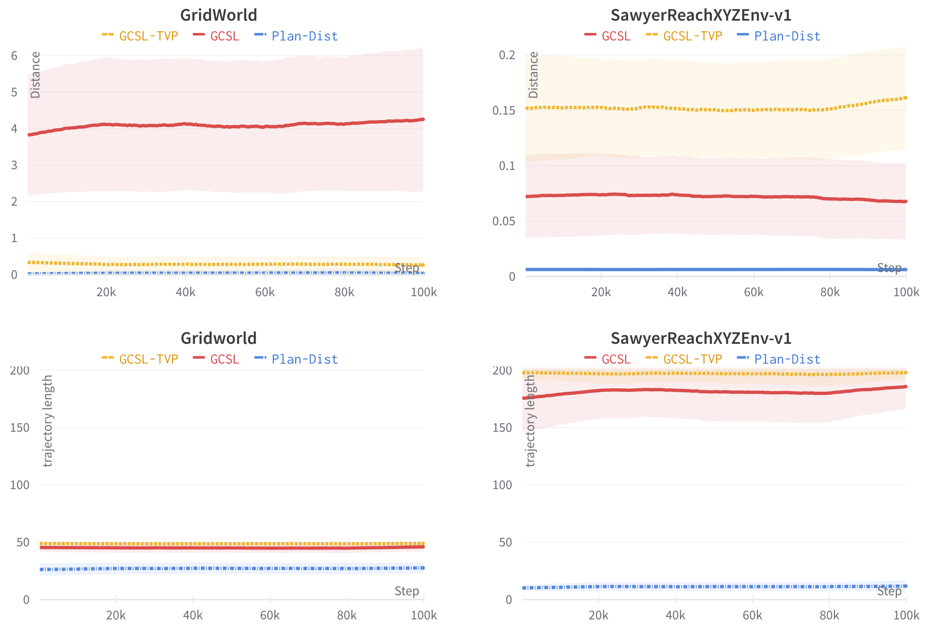


Figure 8.3: Results in multi-goal environments.

We refer to our reward-shaping algorithms as DDQN-PR/DDPG-PR and their original counterpart without reward-shaping as DDQN/DDPG. DDQN is used in domains in which the action space is discrete, while DDPG is used for continuous action domains.

For all the experiments we report results averaged over 10 seeds where the shaded area represents the standard deviation and the results are smoothed using an average window of length 100. All the hyper-parameters used for each algorithm are reported in the appendix.

In the multi-goal environments in Figure 8.3 we report two metrics: the distance to the goal with respect to the state reached at the end of the episode, and the length of the performed trajectory. In both domains, the episode terminates either when we reach the goal state or when we reach the maximum number of steps (50 steps for GridWorld, and 200 steps for SawyerReachXYZEnv-v1). We evaluate the algorithms for 100,000 environment steps.

We can observe that Plan-Dist is able to outperform GCSL, being able to reach the desired goal state with better precision and by using shorter paths. We do not compare to reinforcement learning algorithms in these domains since they struggle to generalize when the goal changes so frequently.

On the classic RL control suite in Figure 8.2 we report the results showing the total reward achieved at the end of each episode. Here we compare both goal-conditioned algorithms and state-of-the-art reinforcement learning algorithms for 200,000 environment steps. Plan-Dist is still able to outperform GCSL in almost all domains while performing slightly worse than GCSL-TVP in CartPole-v0. Compared to DDQN-PR/DDPG-PR, Plan-Dist is able to reach a similar total reward, but in MountainCar-v0, DDQN-PR is eventually able to achieve a higher reward.

The reward shaping mechanism of DDQN-PR/DDPG-PR is not helping in the domains CartPole-v0, Pendulum-v0 and Acrobot-v0. In these domains, it is hard to define a single state as the goal to reach in each episode. As an example, in CartPole-v0 we defined the state $[0, 0, 0, 0]$ as our goal state and we reshape the reward accordingly, but this is not in line with the environment reward that instead cares only about balancing the pole regardless of the position of the cart. While in these domains we do not observe an improvement in performance, it is worth noticing that our reward shaping scheme is not adversely affecting DDQN-PR/DDPG-PR, and they are able to achieve results that are similar to those of their original counterparts.

Conversely, in MountainCar-v0 where the environment reward resembles a goal-reaching objective since the goal is to reach the peak of the mountain as fast as possible, our reward shaping scheme is aligned with the environment objective and DDQN-PR outperforms DDQN in terms of learning speed and total reward on the fixed evaluation time of 200,000 steps.

8.3 Conclusion

We propose a novel method for learning a parametric state embedding ϕ_θ where the distance between any pair of states (s, s') in embedded space approximates the Minimum Action Distance, $\|\phi_\theta(s), \phi_\theta(s')\|_1 \approx d_{MAD}(s, s')$. One limitation of our approach is that we consider symmetric distance functions, while in general, the MAD in an MDP could be asymmetric, $d_{MAD}(s, s') \neq d_{MAD}(s', s)$. In Chapter 7.1 we have already seen how to extend our approach to asymmetric distance embeddings.

While our work focuses on estimating the MAD between states and empirically shows the utility of the resulting metric for goal-conditioned reinforcement learning, the distance measure could be uninformative in a highly stochastic environment where the expected shortest path distance better measures the distance between states. One possible way to approximate this measure using our self-supervised training scheme would be to minimize a weighted version of our objective in (8.1):

$$\min_{\theta} \quad \sum_{\tau \in \mathcal{D}} \sum_{(s,s') \in \tau} 1/d_{TD}^{\alpha}(\left\|\phi_\theta(s) - \phi_\theta(s')\right\|_1 - d_{TD}(s, s' \mid \tau))^2. \qquad (8.8)$$

Here, the term $1/d_{TD}$ is exponentiated by a factor α which decides whether to favor the regression over shorter or longer Trajectory Distances. Concretely, when $\alpha < 1$ we favor the regression over shorter Trajectory Distances, approximating a Shortest Path Distance.

In this work, we focus on single goal-reaching tasks, in order to have a fair comparison with goal-conditioned reinforcement learning agents in the literature. However, the use of our learned distance function is not limited to this setting and we can consider multi-goal tasks, such as reaching a goal while maximizing the distance to forbidden (obstacle) states, reaching the nearest of two goals, and in general any linear and non-linear combination of distances to states given as input.

Lastly, it would be interesting to use this work in the context of Hierarchical Reinforcement Learning, in which a manager could suggest subgoals to our Plan-Dist algorithm.

Chapter 9

Conclusions and Future Work

In conclusion, this thesis has presented an investigation into the area of Hierarchical Reinforcement Learning (HRL) through state space partition and into the area of Goal Conditioned Reinforcement Learning (GCRL).

In Chapter 5 we first presented an HRL algorithm that decomposes the state-action space using a given compression function and introduces subtasks that consist in moving between the resulting partitions. Here we demonstrate how a given compression function over the invariant part of the state-action space allows the discovery of equivalent subMDPs repeated across the state-action space defined by multiple tasks. Experiments demonstrate that the proposed HRL approach outperformed state-of-the-art flat agents in terms of improved exploration and reduced sample complexity on a set of sparse reward environments.

The algorithm assumes prior knowledge of the invariant part of the state-action space, i.e. $\mathcal{S}^{\mathcal{I}} \times \mathcal{A}^{\mathcal{I}}$. In some applications, this seems like a reasonable assumption, e.g. in environments such as MineCraft (Guss et al. 2019; Johnson et al. 2016) or Deep-Mind Lab (Beattie et al. 2016) where the agent has access to a basic set of actions and is later asked to solve specific tasks. But in general, we believe this is a strong assumption, and further research is needed to automatically discover the invariant part of the state-action space.

Another prominent scientific question raised from our first work in

Chapter 5 is how to learn the compression function of the state space that respects some of the properties we highlighted in Chapter 4.

In Chapter 6 we address this question, and we proposed a novel method for learning a hierarchical representation from sampled trajectories in high-dimensional domains. The learned representation exhibits desired properties such as controlling the size of the subMDPs, discovering bottleneck exit states, and creating strongly connected subMDPs. However, it does not have the capability of discovering equivalent subMDPs, it assumes prior knowledge of the number of clusters, and its quality depends on the quality of the behavior policy used to collect the trajectory data. Indeed, the learned representation demonstrates a coupling to the behavior policy used to collect the dataset, which indirectly defines the distance between states (i.e. which states cluster together).

In Chapter 7 we took a different approach and propose to learn an HRL representation directly from the Minimum Action Distance (MAD) metric. The study demonstrated how to estimate the MAD from interactions in the environment and shows how to use the learned MAD to cluster the state space based on the discovery of centroids C. The learned representation demonstrates some desired properties, such as the ability to control the size of each subMDP and the ability to define strongly connected subMDPs. On the contrary, the representation cannot discover bottleneck exit states and equivalent subMDPs.

As we have seen we are interested in equivalent subMDP because this open ups the possibility of transfer learning. We highlight that while the representation is not able to discover equivalent subMDPs the algorithm presented in the experiments can exhibit some transfer learning since the representation based on centroids can be used in conjunction with Goal-Conditioned Reinforcement Learning to learn transitions between partitions, offering the possibility of transfer learning. Indeed, the option policies are all parametrized by the same set of parameters. As a result, learning to move between a pair of centroids can potentially transfer knowledge on how to move between other pairs of centroids.

Contrary to the method presented in Chapter 6 the representation learned here is not dependent on the behavior policy used to collect data and can be learned in an online manner.

The research presented in this chapter is in its early stages. Preliminary results show that augmenting the action space of an agent with options that allow it to reach states that are far in terms of MAD improves sample efficiency and in future work, we are interested in understanding why and when such options are useful. A possible starting point is through the work of Jinnai et al. (2019a,b) or the analysis of Fruit and Lazaric (2017) and Fruit et al. (2017).

Finally, in Chapter 8 we presented a novel state representation for reward-free Markov decision processes. As in Chapter 7 we leverage the notion of MAD to define an embedding space where distances between pairs of embedded states correspond to the MAD to transition between them. We then used this state embedding in the context of Goal Conditioned Reinforcement Learning showing how this representation can be leveraged to learn goal-conditioned policies, providing a notion of similarity between states and goals and a useful heuristic distance to guide planning and reinforcement learning algorithms. We empirically validate our method in classic control domains and multi-goal environments, demonstrating that our method can successfully learn representations in large and/or continuous domains.

In summary, this thesis has explored the field of Hierarchical Reinforcement Learning (HRL) employing state space partitioning and the field of Goal Conditioned Reinforcement Learning (GCRL). Through different approaches, we have presented algorithms to decompose the state-action space into subtasks that can help improve exploration and sample efficiency. However, there is still room for further research in learning hierarchical representation that can discover equivalent subMDPs and transfer knowledge. Moreover, we presented state space representations useful for GCRL.

Bibliography

Aggarwal, Charu C, Alexander Hinneburg, and Daniel A Keim (2001). "On the surprising behavior of distance metrics in high dimensional space". In: *International conference on database theory*. Springer, pp. 420–434.

Andrychowicz, Marcin et al. (2017). "Hindsight experience replay". In: *Advances in Neural Information Processing Systems*, pp. 5048–5058.

Asadi, Mehran and Manfred Huber (2004). "State Space Reduction For Hierarchical Reinforcement Learning." In: *FLAIRS Conference*, pp. 509–514.

Bacon, Pierre-Luc, Jean Harb, and Doina Precup (2017). "The option-critic architecture". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 31. 1.

Bar, Amitay, Ronen Talmon, and Ron Meir (2020). "Option discovery in the absence of rewards with manifold analysis". In: *International Conference on Machine Learning*. PMLR, pp. 664–674.

Barto, Andrew G and Sridhar Mahadevan (2003). "Recent advances in hierarchical reinforcement learning". In: *Discrete event dynamic systems* 13.1-2, pp. 41–77.

Bavelas, Alex (1950). "Communication patterns in task-oriented groups". In: *The journal of the acoustical society of America* 22.6, pp. 725–730.

Beattie, Charles et al. (2016). "Deepmind lab". In: *arXiv preprint arXiv:1612.03801*.

Bellemare, Marc G et al. (2020). "Autonomous navigation of stratospheric balloons using reinforcement learning". In: *Nature* 588.7836, pp. 77–82.

Bellman, RICHARD (1957). “Dynamic programming, princeton univ”. In: *Press Princeton, New Jersey.*

Bellman, Richard (1958). “Dynamic programming and stochastic control processes”. In: *Information and control* 1.3, pp. 228–239.

Botvinick, Matthew et al. (2015). “Reinforcement learning, efficient coding, and the statistics of natural tasks”. In: *Current opinion in behavioral sciences* 5, pp. 71–77.

Botvinick, Matthew M, Yael Niv, and Andew G Barto (2009). “Hierarchically organized behavior and its neural foundations: A reinforcement learning perspective”. In: *Cognition* 113.3, pp. 262–280.

Bowling, Mike and Manuela Veloso (1998). “Reusing learned policies between similar problems”. In: *in Proceedings of the AI* AI-98 Workshop on New Trends in Robotics.* Citeseer.

Castro, Pablo and Doina Precup (2010). “Using bisimulation for policy transfer in MDPs”. In: *Proceedings of the AAAI conference on artificial intelligence.* Vol. 24. 1, pp. 1065–1070.

Castro, Pablo Samuel and Doina Precup (2012). “Automatic construction of temporally extended actions for mdps using bisimulation metrics”. In: *Recent Advances in Reinforcement Learning: 9th European Workshop, EWRL 2011, Athens, Greece, September 9-11, 2011, Revised Selected Papers 9.* Springer, pp. 140–152.

Chang, S et al. (2017). “Dilated Recurrent Neural Networks”. In: *Advances in Neural Information Processing Systems.*

Ciosek, Kamil and David Silver (2015). “Value iteration with options and state aggregation”. In: *arXiv preprint arXiv:1501.03959.*

Dayan, Peter and Geoffrey E Hinton (1992). “Feudal reinforcement learning”. In: *Advances in neural information processing systems* 5.

Dayan, Peter and Geoffrey E Hinton (1993). “Feudal reinforcement learning”. In: *Advances in neural information processing systems*, pp. 271–278.

Dietterich, Thomas G (2000). “Hierarchical reinforcement learning with the MAXQ value function decomposition”. In: *Journal of Artificial Intelligence Research* 13, pp. 227–303.

Eckstein, Maria K and Anne GE Collins (2020). “Computational evidence for hierarchically structured reinforcement learning in humans”. In:

Proceedings of the National Academy of Sciences 117.47, pp. 29381–29389.

Ecoffet, Adrien et al. (2021). "First return, then explore". In: *Nature* 590.7847, pp. 580–586.

Ferns, Norm, Prakash Panangaden, and Doina Precup (2004). "Metrics for Finite Markov Decision Processes." In: *UAI*. Vol. 4, pp. 162–169.

Ferns, Norm, Prakash Panangaden, and Doina Precup (2011). "Bisimulation metrics for continuous Markov decision processes". In: *SIAM Journal on Computing* 40.6, pp. 1662–1714.

Floyd, Robert W. (1962). "Algorithm 97: Shortest Path". In: *Commun. ACM* 5.6, pp. 345–. ISSN: 0001-0782. DOI: 10.1145/367766.368168.

Fruit, Ronan and Alessandro Lazaric (2017). "Exploration-exploitation in mdps with options". In: *Artificial intelligence and statistics*. PMLR, pp. 576–584.

Fruit, Ronan et al. (2017). "Regret minimization in mdps with options without prior knowledge". In: *Advances in Neural Information Processing Systems* 30.

Ghavamzadeh, Mohammad and Sridhar Mahadevan (2002). "Hierarchically optimal average reward reinforcement learning". In: *ICML*, pp. 195–202.

Ghosh, Dibya et al. (2019). "Learning to reach goals via iterated supervised learning". In: *arXiv preprint arXiv:1912.06088*.

Ghosh, Dibya et al. (2020). "Learning to Reach Goals via Iterated Supervised Learning". In: *International Conference on Learning Representations*.

Greensmith, Evan, Peter L Bartlett, and Jonathan Baxter (2004). "Variance Reduction Techniques for Gradient Estimates in Reinforcement Learning." In: *Journal of Machine Learning Research* 5.9.

Guss, William H et al. (2019). "The minerl competition on sample efficient reinforcement learning using human priors". In:

Heess, Nicolas et al. (2015). *Learning Continuous Control Policies by Stochastic Value Gradients*. arXiv: 1510.09142 [cs.LG].

Howard, Ronald A (1960). "Dynamic programming and markov processes." In:

Infante, Guillermo, Anders Jonsson, and Vicenç Gómez (2022). "Globally Optimal Hierarchical Reinforcement Learning for Linearly-Solvable

Markov Decision Processes". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 36. 6, pp. 6970–6977.

Jinnai, Yuu et al. (2019a). "Discovering options for exploration by minimizing cover time". In: *International Conference on Machine Learning*. PMLR, pp. 3130–3139.

Jinnai, Yuu et al. (2019b). "Finding options that minimize planning time". In: *International Conference on Machine Learning*. PMLR, pp. 3120–3129.

Johnson, Matthew et al. (2016). "The Malmo Platform for Artificial Intelligence Experimentation." In: *Ijcai*. Citeseer, pp. 4246–4247.

Kaelbling, Leslie Pack (1993). "Hierarchical learning in stochastic domains: Preliminary results". In: *Proceedings of the tenth international conference on machine learning*. Vol. 951, pp. 167–173.

Konda, Vijay and John Tsitsiklis (1999). "Actor-critic algorithms". In: *Advances in neural information processing systems* 12.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton (2015). "Deep learning". In: *nature* 521.7553, p. 436.

Li, Muhan (2020). *Machin*. `https://github.com/iffiX/machin`.

Li, Siyuan et al. (2021). "Learning subgoal representations with slow dynamics". In: *International Conference on Learning Representations*.

Li, Yuxi (2017). "Deep reinforcement learning: An overview". In: *arXiv preprint arXiv:1701.07274*.

Lillicrap, Timothy P et al. (2015). "Continuous control with deep reinforcement learning". In: *arXiv preprint arXiv:1509.02971*.

Littman, Michael L., Thomas L. Dean, and Leslie Pack Kaelbling (2013). *On the Complexity of Solving Markov Decision Problems*. DOI: `10.48550/ARXIV.1302.4971`.

Liu, Minghuan, Menghui Zhu, and Weinan Zhang (2022). "Goal-conditioned reinforcement learning: Problems and solutions". In: *arXiv preprint arXiv:2201.08299*.

Loshchilov, Ilya and Frank Hutter (2017). "Decoupled weight decay regularization". In: *arXiv preprint arXiv:1711.05101*.

Machado, Marlos C, Marc G Bellemare, and Michael Bowling (2017). "A laplacian framework for option discovery in reinforcement learning". In: *International Conference on Machine Learning*. PMLR, pp. 2295–2304.

Mannor, Shie et al. (2004). “Dynamic abstraction in reinforcement learning via clustering”. In: *Proceedings of the twenty-first international conference on Machine learning*, p. 71.

McGovern, Amy and Andrew G Barto (2001). “Automatic discovery of subgoals in reinforcement learning using diverse density”. In:

Menache, Ishai, Shie Mannor, and Nahum Shimkin (2002). “Q-cut—dynamic discovery of sub-goals in reinforcement learning”. In: *European conference on machine learning*. Springer, pp. 295–306.

Mennucci, Andrea CG (2013). “On asymmetric distances”. In: *Analysis and Geometry in Metric Spaces* 1.2013, pp. 200–231.

Mnih, Volodymyr et al. (2013). “Playing atari with deep reinforcement learning”. In: *arXiv preprint arXiv:1312.5602*.

Nachum, Ofir et al. (2018a). “Data-efficient hierarchical reinforcement learning”. In: *Advances in neural information processing systems* 31.

Nachum, Ofir et al. (2018b). “Near-optimal representation learning for hierarchical reinforcement learning”. In: *arXiv preprint arXiv:1810.01257*.

Ng, Andrew Y, Daishi Harada, and Stuart Russell (1999). “Policy invariance under reward transformations: Theory and application to reward shaping”. In: *Icml*. Vol. 99, pp. 278–287.

Oh, Junhyuk et al. (2018). “Self-imitation learning”. In: *arXiv preprint arXiv:1806.05635*.

Parr, Ronald and Stuart Russell (1997). “Reinforcement learning with hierarchies of machines”. In: *Advances in neural information processing systems* 10.

Pickett, Marc and Andrew G Barto (2002). “Policyblocks: An algorithm for creating useful macro-actions in reinforcement learning”. In: *ICML*. Vol. 19, pp. 506–513.

Pitis, Silviu et al. (2020). “An inductive bias for distances: Neural nets that respect the triangle inequality”. In: *arXiv preprint arXiv:2002.05825*.

Puterman, Martin L (2014). *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons.

Ravindran, Balaraman and Andrew G Barto (2002). “Model minimization in hierarchical reinforcement learning”. In: *Abstraction, Reformulation, and Approximation: 5th International Symposium, SARA*

2002 Kananaskis, Alberta, Canada August 2–4, 2002 Proceedings 5. Springer, pp. 196–211.

Ravindran, Balaraman and Andrew G Barto (2004). “Approximate homomorphisms: A framework for non-exact minimization in Markov decision processes”. In:

Roy, Bernard (1959). “Transitivité et connexité”. In: *Extrait des comptes rendus des séances de l’Académie des Sciences.* `http://gallica.bnf.fr/ark:/12148/bpt6k3201c/f222.image.langFR`. Gauthier-Villars, pp. 216–218.

Schaul, Tom et al. (2015a). “Prioritized experience replay”. In: *arXiv preprint arXiv:1511.05952.*

Schaul, Tom et al. (2015b). “Universal value function approximators”. In: *International conference on machine learning.* PMLR, pp. 1312–1320.

Silver, David et al. (2016). “Mastering the game of Go with deep neural networks and tree search”. In: *nature* 529.7587, p. 484.

Şimşek, Özgür and Andrew Barto (2008). “Skill characterization based on betweenness”. In: *Advances in neural information processing systems* 21.

Şimşek, Özgür and Andrew G Barto (2004). “Using relative novelty to identify useful temporal abstractions in reinforcement learning”. In: *Proceedings of the twenty-first international conference on Machine learning*, p. 95.

Solway, Alec et al. (2014). “Optimal behavioral hierarchy”. In: *PLoS computational biology* 10.8, e1003779.

Stolle, Martin and Doina Precup (2002). “Learning options in reinforcement learning”. In: *International Symposium on abstraction, reformulation, and approximation.* Springer, pp. 212–223.

Strehl, Alexander L and Michael L Littman (2008). “An analysis of model-based interval estimation for Markov decision processes”. In: *Journal of Computer and System Sciences* 74.8, pp. 1309–1331.

Sutton, Richard S (1988). “Learning to predict by the methods of temporal differences”. In: *Machine learning* 3.1, pp. 9–44.

Sutton, Richard S and Andrew G Barto (2018). *Reinforcement learning: An introduction.* MIT press.

Sutton, Richard S, Doina Precup, and Satinder Singh (1999). “Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning”. In: *Artificial intelligence* 112.1-2, pp. 181–211.

Sutton, Richard S et al. (2000). “Policy gradient methods for reinforcement learning with function approximation”. In: *Advances in neural information processing systems*, pp. 1057–1063.

Tenenbaum, Joshua B et al. (2011). “How to grow a mind: Statistics, structure, and abstraction”. In: *science* 331.6022, pp. 1279–1285.

Thrun, Sebastian and Anton Schwartz (1994). “Finding structure in reinforcement learning”. In: *Advances in neural information processing systems* 7.

Van Hasselt, Hado, Arthur Guez, and David Silver (2016). “Deep reinforcement learning with double q-learning”. In: *Thirtieth AAAI conference on artificial intelligence.*

Vezhnevets, Alexander Sasha et al. (2017). “Feudal networks for hierarchical reinforcement learning”. In: *Proceedings of the 34th International Conference on Machine Learning.* JMLR. org, pp. 3540–3549.

Vinyals, Oriol et al. (2019). “Grandmaster level in StarCraft II using multi-agent reinforcement learning”. In: *Nature* 575.7782, pp. 350–354.

Wan, Yi and Richard S Sutton (2022). “Toward Discovering Options that Achieve Faster Planning”. In: *arXiv preprint arXiv:2205.12515.*

Warshall, Stephen (1962). “A Theorem on Boolean Matrices”. In: *J. ACM* 9.1, pp. 11–12. ISSN: 0004-5411. DOI: 10.1145/321105.321107.

Watkins, Christopher JCH and Peter Dayan (1992). “Q-learning”. In: *Machine learning* 8.3-4, pp. 279–292.

Weaver, Lex and Nigel Tao (2013). “The optimal reward baseline for gradient-based reinforcement learning”. In: *arXiv preprint arXiv:1301.2315.*

Wen, Zheng et al. (2020). “On efficiency in hierarchical reinforcement learning”. In: *Advances in Neural Information Processing Systems* 33, pp. 6708–6718.

Williams, Ronald J (1992). “Simple statistical gradient-following algorithms for connectionist reinforcement learning”. In: *Machine learning* 8.3, pp. 229–256.