COMENIUS UNIVERSITY IN BRATISLAVA
FACULTY OF MATHEMATICS, PHYSICS AND INFORMATICS

ADAPTIVE SKILL ACQUISITION IN HIERARCHICAL REINFORCEMENT LEARNING

Dissertation thesis

Mgr. Juraj Holas 2021

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Adaptive Skill Acquisition in Hierarchical Reinforcement Learning

Dissertation thesis

Mgr. Juraj Holas

Study program:	Computer Science
Field of study:	Computer Science
Department:	Department of Applied Informatics
Supervisor:	prof. Ing. Igor Farkaš, Dr.

Bratislava, 2021

29650199

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Mgr. Juraj Holas
Študijný program:	informatika (Jednoodborové štúdium, doktorandské III. st., denná forma)
Študijný odbor:	informatika
Typ záverečnej práce:	dizertačná
Jazyk záverečnej práce:	anglický
Sekundárny jazyk:	slovenský

Názov: Adaptive Skill Acquisition in Hierarchical Reinforcement Learning
Adaptívna akvizícia zručností v hierarchickom učení posilňovaním

Anotácia: Učenie posilňovaním (RL) je teoretický rámec zahŕňajúci efektívne metódy strojového učenia, založený na priamej interakcii s prostredím. Hierarchická úroveň RL predstavuje zostávajúcu výzvu zameranú na riešenie zložitých problémov založenú na rozklade problému na nižšiu a vyššiu úroveň kontroly. Adaptívne objavovanie a získavanie zručností predstavuje v tomto smere rozumný a flexibilný prístup.

Cieľ: 1. Urobte prehľad a kritické porovnanie existujúcich prístupov ku hierarchickému RL.
2. Navrhnite hierarchický RL model založený na adaptávnej akvizícii zručností.
3. Implementujte a otestujte model na vybratých RL úlohách. Prediskutujte silné a slabé stránky tohto prístupu.

Literatúra: Barto, A. G., & Mahadevan, S. (2003). Recent advances in hierarchical reinforcement learning. Discrete Event Dynamic Systems, 13(4), 341-379.
Sutton, R. S., Precup, D., & Singh, S. (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. Artificial intelligence, 112(1-2), 181-211.
Florensa, C., Duan, Y., & Abbeel, P. (2017). Stochastic neural networks for hierarchical reinforcement learning. arXiv:1704.03012.

Kľúčové slová: hierarchické učenie posilňovaním, adaptívny model, neurónové siete

Školiteľ:	prof. Ing. Igor Farkaš, Dr.
Katedra:	FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry:	prof. Ing. Igor Farkaš, Dr.
Dátum zadania:	21.01.2017

Dátum schválenia: 16.02.2017

prof. RNDr. Rastislav Kráľovič, PhD.
garant študijného programu

.... študent

.... školiteľ

29650199

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Mgr. Juraj Holas
Study programme:	Computer Science (Single degree study, Ph.D. III. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Dissertation thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Adaptive Skill Acquisition in Hierarchical Reinforcement Learning

Annotation: Reinforcement learning (RL) is a theoretical framework involving powerful machine learning methods, based on direct interaction with an environment. Hierarchical RL is a remaining challenge aiming at solving difficult problems, based on problem decomposition to a lower and higher levels of control. Adaptive skill discovery and acquisition represents a sensible and flexible approach in this direction.

Aim: 1. Compile an overview and provide a critical comparison of existing approaches to hierarchical RL.
2. Design a hierarchical RL model based on adaptive acquisition of skills.
3. Implement and test the model functioning on chosen RL tasks. Discuss its advantages and limitations.

Literature: Barto, A. G., & Mahadevan, S. (2003). Recent advances in hierarchical reinforcement learning. Discrete Event Dynamic Systems, 13(4), 341-379.
Sutton, R. S., Precup, D., & Singh, S. (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. Artificial intelligence, 112(1-2), 181-211.
Florensa, C., Duan, Y., & Abbeel, P. (2017). Stochastic neural networks for hierarchical reinforcement learning. arXiv:1704.03012.

Keywords: hierarchical reinforcement learning, adaptive model, neural networks

Tutor:	prof. Ing. Igor Farkaš, Dr.
Department:	FMFI.KAI - Department of Applied Informatics
Head of department:	prof. Ing. Igor Farkaš, Dr.
Assigned:	21.01.2017
Approved:	16.02.2017

prof. RNDr. Rastislav Kráľovič, PhD.
Guarantor of Study Programme

.... Student

.... Tutor

"The most exciting phrase to hear in science, the one that heralds new discoveries, is not 'Eureka!', but 'That's funny...' "

— Isaac Asimov

Acknowledgements

I would like to express my gratitude to my thesis supervisor, prof. Ing. Igor Farkaš, Dr., for his guidance, patience, and advice throughout my postgraduate journey. My sincere thanks also goes to the whole Faculty of Mathematics, Physics and Informatics of Comenius University in Bratislava, for providing soil for my academic growth, and to the local team of colleagues who always provided me with valuable insights.
A special thanks goes to my beloved one, who always kept me going, and helped me pass the toughest parts of this road.

I hereby declare that I wrote this work by myself, only with the help of the referenced literature.

....

In Bratislava, 30. 4. 2021 Mgr. Juraj Holas

Abstrakt

Učenie posilňovaním (reinforcement learning, RL) predstavuje skupinu metód strojového učenia, ktoré sú schopné učiť sa priamou interakciou s prostredím, bez predpripravených trénovacích dát. Navyše, náročnosť mnohých úloch, plynúca z ich vnútornej hierarchickej štruktúry, inšpirovala záujem o oblasť hierarchického RL, ktoré vnáša dekompozíciu úlohy priamo do výpočtových modelov. Modely HRL zvyčajne pozostávajú z nižšej úrovne, tzv. *schopností*, ktoré vykonávajú jednoduchšie behaviorálne úkony, a vyššej úrovne, ktorá využíva tieto schopnosti na riešenie hlavnej, kognitívne náročnejšej úlohy.

Napriek tomu, že rôzne modely HRL využívajú odlišné prístupy k architektúre, identifikácia a učenie schopností ostáva kľúčovou výzvou tejto oblasti. Väčšina výskumu sa zameriava práve na tento problém, využívajúc širokú škálu automatizovaných metód. Väčšina z nich pritom vytvára schopnosti, ktoré sú predtrénované a počas učenia hlavnej úlohy ostávajú nemenné, čo však môže vyústiť do suboptimálnych riešení.

V tejto práci prinášame návrh nového prístupu adaptívneho učenia schopností (*Adaptive Skill Acquisition*, ASA), ktorého cieľom je práve problém suboptimálnych predtrénovaných hierarchií. Táto metóda je navrhnutá ako univerzálny komponent, ktorý môže obohatiť existujúce metódy o novú funkcionalitu. ASA počas učenia analyzuje správanie hlavného agenta a identifikuje potenciálne schopnosti, ktoré mu chýbajú na efektívne splnenie úlohy. Tieto chýbajúce schopnosti sú následne natrénované a integrované do hierarchického systému, čo umožní zlepšiť jeho celkovú úspešnosť. Okrem tohto nového prístupu prinášame taktiež aj prehľad a analýzu existujúcich metód tradičného aj hierarchického RL.

Experimenty, ktoré boli vykonané v dvoch fundamentálne odlišných prostrediach, demonštrujú širokú použiteľnosť metódy ASA. Pridanie novej schopnosti do hierarchie významne zvýšilo celkovú úspešnosť modelu, pričom agenti využvajúci ASA konzistetne dosahovali lepšie výsledky než bez neho. Testy s jednotlivých častí ASA ukázali vysokú robustnosť komponentu na identifikáciu chýbajúcich schopností, avšak poukázali aj na to, že zložitejšie stratégie na integráciu nových schopností neprekonali základnú metódu. Porovnávacie testy tiež potvrdili, že ASA prekonáva predchádzajúci podobne zameraný model.

Kľúčové slová: hierarchické učenie posilňovaním, schopnosti, adaptívne učenie schopností

Abstract

Reinforcement learning is a class of powerful machine learning methods capable of learning by direct interaction with an environment instead of pre-collected datasets. At the same time, the nature of many tasks with an inner hierarchical structure has evoked interest in hierarchical RL approaches that introduced the two-level decomposition directly into computational models. These methods are usually composed of lower-level controllers – *skills* – providing simple behaviors, and high-level controller which uses the skills to solve the overall task.

While various models of hierarchical reinforcement learning use different architectures, the skill discovery and acquisition remains the principal challenge of this field. Most of the relevant research is focused on resolving this issue, using a broad spectrum of automated methods. Majority of them produce skills that are pre-trained and fixed before the main learning process starts, which may lead to suboptimal skill set, and thus inefficient solution of the overall task.

In this thesis we propose the *Adaptive Skill Acquisition* framework (ASA) aimed to resolve the problem of inefficient hierarchy. It is designed as a universal pluggable component capable of augmenting the existing solutions by new functionality. ASA can observe the high-level controller during its training and identify skills that it lacks to successfully learn the task. These missing skills are subsequently trained and integrated into the hierarchy, enabling better performance of the overall architecture. Besides our new approach, we also provide a review and analysis of available methods for both traditional and hierarchical reinforcement learning.

The conducted experiments on two fundamentally different environments demonstrate the broad applicability of ASA. Embedding the new skills into the hierarchy significantly improves the performance of the overall model, and the ASA-enabled agents exhibit consistent advantage to the baseline. Further ablation tests reveal that the identification of a missing skill is exceptionally robust even with imperfect data, but on the other hand, the elaborate strategies for skill integration do not outperform the baseline ones. A comparative study also confirms that ASA can surpass the previous similarly-oriented model.

Keywords: hierarchical reinforcement learning, skills, adaptive skill acquisition

Contents

Introduction

As an approach inspired by the natural knowledge acquisition process, the reinforcement learning (RL) is gaining significant interest in both research studies and practical applications. This biologically motivated approach aims for learning by direct interaction with the environment, providing an on-line improvement of the learned model. At every time-step an agent (in machine learning context) receives an observation – e.g. sensory input; executes the selected action – e.g. motor activation; and receives a reward signal. However, this reward signal reflects not only the quality of the last action, but instead it can describe agent's success over longer period of time. Additionally, in sparse-reward environments, this reward signal can be withheld for a long time – as an extreme, yet not rare case, we can imagine an agent playing a chess game, receiving the reward only at the end of the match. The reward signal must hence be propagated to earlier actions, adjusting them accordingly to their contribution to the overall success or failure.

This uneasy task of reward propagation initiated a research field of reinforcement learning. The key early contributions by Bellman (1957a), and Howard (1964) provided the fundamental concepts upon which all subsequent research has been founded. Successful algorithms such as SARSA and Q-learning were developed based on the dynamic programming (DP).

DP-based approaches could however only cope with discrete state spaces. To address this issue, function approximators were introduced into the RL framework, enabling it to work in continuous environments as well. Further leap was accomplished by Williams (1992) REINFORCE breakthrough followed by popular actor-critic architecture, which allowed continuous action spaces as well. Subsequent research based on these revolutionary ideas brought impressive results, with applications vastly surpassing human-level performance in numerous cases.

Despite the recent progress in traditional or 'flat' RL, these approaches still struggle to solve a task composed of several layers of abstraction. An example may feature a walking robot tasked with solving a simple maze. A two-layer hierarchy can be observed in such case: the first layer learning the basic motion skills such as walking and turning; and the second layer may be using these skills to more efficiently solve the higher cognitive task of navigation within the maze, freeing itself of the peculiarities of locomotion. To render such a problem tractable, hierarchical reinforcement learning (HRL) has been introduced. Despite the fact that research in this area began only recently within the machine learning community, a diverse variety of approaches has already been proposed. The common underlying feature is the usage of *skills* – actions that are temporally extended in time – forming an implicit or explicit hierarchy within the task. Acquisition of these skills is the key answer to be resolved in HRL.

The most popular approach for acquiring the skills is to train them in a specialised pre-training phase. Once it has finished, the skills are ready to be used in the overall hierarchy, and the main training can begin. However, this two-phased process comes with a caveat. If the pre-training phase is not absolutely optimal, it can generate an inefficient set of skills. Some skills can be malformed, or, even worse, a certain useful skill can be missing altogether. When such incomplete skill set is used in the subsequent main training, the agent is doomed to find a suboptimal solution.

As our contribution to the field, we introduce the *Adaptive Skill Acquisition* framework (ASA). It is tailored to address the problem of a missing skill, and solve it even after the main training has started. We designed it as a pluggable component, which can be deployed onto almost any existing HRL architecture, or those yet to come. While the HRL agent is being trained, ASA can automatically identify that a useful skill is missing, train the new skill, and incorporate it into the hierarchy. The agent then resumes its training with this enriched hierarchy, and can solve the overall task more efficiently. The implementation of our approach is also accessible online[1].

In chapter 1 we provide the necessary theoretical concepts underlying all RL tasks – the Markov Decision Process, its components and properties. Progress in solving traditional reinforcement learning is covered in chapter 2, including the fundamental concepts and distinction between different methods. We describe the most important algorithms since the early research until current state-of-the-art methods. Chapter 3 is dedicated to hierarchical reinforcement learning. Besides assessing common concepts and motivation, we also provide a concise survey of HRL research comparing individual methods. In chapter 4 we present our *Adaptive Skill Acquisition* approach, covering in details all of its components. In order to evaluate this approach, chapter 5 describes the conducted experiments and presents the overall results, which show the performance of our model. Finally, in chapter 6 we discuss the functionality of the overall model, and present some possibilities for improvements and future work.

[1]at https://github.com/holasjuraj/asa

Chapter 1

Markov Decision Process

In order to understand the reinforcement learning in its sheer complexity, first we have to understand the underlying building blocks. The theoretical foundation is represented by the model of Markov Decision Process, which we will describe in this chapter.

The proper definition is that Markov Decision Process (MDP) is a type of a problem, to which Reinforcement Learning provides a toolkit of solutions. More formally, MDP is an extension of a *Markov Process* – a series of successive states $s_0, s_1, s_2, \ldots$ for which the Markov property holds:

$$p(s_{t+1}|s_t) = p(s_{t+1}|s_0, \ldots, s_t) \tag{1.1}$$

In its original form, this means that at any given time, s_t fully represents the state of the system and consideration of historical states would yield no additional information to any computations.[1] An alternation of simple Markov Process is a *Markov Reward Process*, which introduces a concept of immediate (possibly non-deterministic) reward r_t at each state s_t, along with a discount factor $\gamma \in (0, 1]$. This enables us to compute the overall score, or *gain*, of a particular Markov sequence, as a total discounted reward from the time t onwards:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \tag{1.2}$$

The introduction of less-than-one γ value into the equation helps to prevent infinite sums, and hence enables us to evaluate infinite-horizon Markov sequences. In addition, setting the γ discount factor describes whether the agent prefers immediate actions ($\gamma \to 0$) or overall gain ($\gamma \to 1$). Finally, *Markov Decision Process* adds an option to make actions that influence further states. To summarize a formal definition, MDP is a tuple $\langle S, A, P, p_0, R, \gamma \rangle$, where:

- S is a finite set of states
- A is a finite set of actions

[1]This concept is not strictly honored in some real-life applications, as we will discuss later.

- P is a probability distribution describing state transition: $p(s_{t+1}|s_t, a_t) \sim P$
- p_0 is a probability distribution of an initial state: $p(s_0) \sim p_0$
- R is a (possibly non-deterministic) reward function: $r_t = R(s_t, a_t)$
- γ is a discount factor, $\gamma \in (0, 1]$

An agent in an MDP system follows an episodic flow, where in each time step t it obtains a state s_t, performs an action a_t, and receives a reward signal r_t, forming a history: $[s_0, a_0, r_0, s_1, a_1, r_1, \ldots]$, as depicted in figure 1.1. An agent is responsible for choosing actions, while the environment provides new states and rewards, according to P, p_0 and R respectively. These iterations repeat until the environment terminates the process, forming a complete episode (or trajectory, rollout) of the agent. Alternatively, in the case of infinite-horizon environments, the trajectory may continue indefinitely.

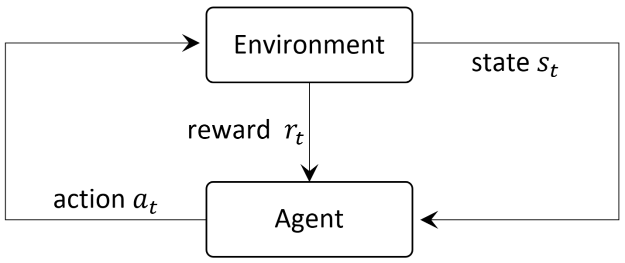


Figure 1.1: Schema of a single episode in Markov Decision Process.

1.1 Policy and its value functions

In every time step, the agent chooses an action a as a reaction to the presented state s. This process of selection is described by agent's *policy*. The policy may be deterministic, yielding the same action every time when presented with a particular state s, or non-deterministic which draws an action from a specified probability distribution, given state s. Notation is $\pi(s)$ for deterministic policies or $\pi(a|s)$ generally[2].

As an adjustment of a gain from Markov Reward Process, MDP uses state- and action-value functions. *State-value function* (or simply value function) defines the expected gain from state s, supposing that agent is following the specified policy π:

$$V_\pi(s) = E_\pi[G_t|s_t = s] \tag{1.3}$$

Alternatively, *action-value function* (or Q-function) defines the expected gain from state s, supposing that agent *firstly chooses action a*, and then follows specified policy π:

$$Q_\pi(s, a) = E_\pi[G_t|s_t = s, a_t = a] \tag{1.4}$$

[2]A deterministic policy is, after all, only a special case of a broader class of non-deterministic ones.

As presented by Bellman (1957a), both state- and action-value functions can be expressed recursively using immediate reward and expected value of the next step. This approach allows us to approximate their values iteratively. The Bellman Expectation Equation[3] states that:

$$\begin{aligned} V_\pi(s) &= E_\pi[r_t + \gamma V_\pi(s_{t+1}) | s_t = s] \\ Q_\pi(s,a) &= E_\pi[r_t + \gamma Q_\pi(s_{t+1}, a_{t+1}) | s_t = s, a_t = a] \end{aligned} \tag{1.5}$$

These equations became a cornerstone for almost all subsequent Reinforcement Learning methods, as we will discuss in chapter 2.

1.2 Alternatives of MDP

In addition to the classical MDP problem, many off-springs were introduced to help formally describe various real-world problems. These diminish the constraints pertaining to classical MDP, in order to address a specific problem, and they can often be combined. However, all of them need to be accounted for when designing an algorithm, as they usually introduce additional complexity to the problem.

1.2.1 Continuous state and action spaces

Most common generalization of MDP removes the constraint of having *finite* sets of states and actions. Instead, they can be infinite and even non-discrete, allowing the introduction of both continuous state representation and continuous actions. This version of MDP is frequently used in robotics, where a physical agent moves in an obviously continuous environment, and its actions typically include applying torque to the joints, drawn from a continuous interval as well. It is also common to see environments in which only the state-space is continuous but the action-space is discrete, such as controlling a game with a joystick (four discrete actions) given a raw pixel data from screen as the input (high-dimensional continuous state).

1.2.2 Partially observable MDP

In a classical setup of an MDP problem, an agent receives state s_t which fully represents the state of the environment. However, this may not be always possible to achieve, as the environment may be too vast, or simply due to limitations of agent's sensors. An example would be a robot with a front-facing camera that is not able to see the state of an environment behind him. Such setup is called a *Partially observable MDP*, or POMDP.

[3]There is another variant called Bellman Optimality Equation, both of which are often referred to as simply Bellman equation.

As a formalization of this concept, a *space of observations* O is introduced. In each time step, agent receives an observation o with probability $\Omega(o|s)$, and a probability distribution of states – called a *belief* – must be maintained. Having a belief instead of an actual state, the whole problem can be easily viewed as an action selection in a belief space, which can be solved by the traditional MDP methods.

1.2.3 Semi-MDP

In the real-time applications, time flow is also continuous, suggesting that time-steps in discrete-time MDP should be broken down to infinitesimals and form a continuous action selection. This, however, is not achievable in real life, as every action selection and action performance takes non-zero time.

In contrast to classical MDP, *Semi-Markov Decision Process* (SMDP) considers the varying *length* of each action, and so it models the continuous-time scenario a bit better. Formal definitions are altered to reflect this concept:

- Transitional distribution P yields a probability of a transition to the next state s' at time τ: $p(s', \tau|s, a)$
- Reward function R considers the duration of an action: $r = R(s, a, \tau)$
- Discount factor γ is scaled with respect to duration: γ^τ
- Value functions are adjusted accordingly: $V_\pi(s) = E_\pi[R(s, a, \tau) + \gamma^\tau V_\pi(s')]$

$$\begin{aligned} V_\pi(s) &= E_\pi[R(s, a, \tau) + \gamma^\tau V_\pi(s')] \\ Q_\pi(s, a) &= E_\pi[R(s, a, \tau) + \gamma^\tau Q_\pi(s', a')] \end{aligned}$$

1.2.4 Universal MDP

The *Universal MDP* (UMDP) is a subclass of the MDP family, even though its name suggests the opposite relation. Its *universality* lies in its ability to reach any arbitrary goal in the environment, given that the goal is specified to the agent. Most common example of such an approach can be found in the field of robotics, when the robot is trained to move to any location specified by the user.

In UMDP, the state space is partitioned into two components: $S^+ = S \times G$, where $s \in S$ is the actual state of the environment, and $g \in G$ represents the goal state. Although G can be constructed arbitrarily, the most common case is to see $G \subseteq S$, e.g. $s = [\textit{position of robot}, \textit{joints torques}]$, $g = [\textit{desired position of robot}]$. Unlike state s which changes with each step, the goal g is chosen at the beginning of the trajectory and then stays fixed until that trajectory terminates.

The concept of UMDP is useful in spatial tasks, where the goal state can be easily formulated by specifying desired coordinates. However, the MDP covers a wide variety of problems where no such formulation is feasible, or might not exist at all. Such problems range from collecting a resource in a spatial environment (e.g. gather coins), through stabilising a closed-loop system (e.g. balance a two-wheeled robot), to so-called *multi-armed bandits* – single-action environments for reward maximisation (e.g. advertising placement on websites). Furthermore, the final goal state certainly cannot be specified in the infinite-horizon environments.

Chapter 2

Reinforcement Learning

After properly defining MDPs as a problem, we can now investigate the methods used to *solve* this problem, i.e. various Reinforcement Learning principles and algorithms.

2.1 Fundamental concepts

When applying the MDP theoretical base into real-life cases, Reinforcement Learning (RL) did not follow a rather linear path. Instead, a variation of different concepts was introduced, each of them aiming to resolve a particular real-world challenge that previous methods could not. These concepts could be – and often were – combined to make the algorithms more robust. This highly non-linear trajectory of research brought up many different (usually binary) decompositions of RL algorithms: model-based vs. model-free, static vs. active RL, on-policy vs. off-policy, sums vs. samples, discrete vs. continuous, etc.

We will try to cover the most important ones, in order to better understand more complex algorithms as well as our motivation for further work.

2.1.1 Model usage

A model in RL is complete knowledge of the underlying MDP, especially the transitional distribution P and the reward function R. Generally speaking, if an agent has full access to these properties it is a *model-based* agent. Such agents can then use the P and R distributions for computations without actually interacting with the environment. On the other hand, if an agent can only sample the transitions and rewards by querying the environment, it is *model-free*.

There is a branch of model-free algorithms in which the agent builds its own *approximation of the model* by observing transitions and rewards it experienced (Kober and Peters, 2012; Deisenroth et al., 2013). This approach allows to use some model-based methods to bootstrap the learning process even in a setup that is actually model-free. One of the first examples was brought by Sutton (1990) in a form of *Dyna-Q* algorithm, while recent research successfully applied this approach even to large-scale problems (Gu et al., 2016).

2.1.2 Sampling vast distributions

Even if the model is known to the agent, it may not be feasible to fully use it. As we have shown in chapter 1, most of the calculations involve expected values over a distribution – it may either be a transitional distribution $P(s'|s, a)$, a non-deterministic reward function $R(s, a)$, non-deterministic policy $\pi(a|s)$, or frequently a joint distribution of all aforementioned.

Exact computations of these expected values require expanding them into sums (or integrals) such as $\sum_{s \in S} P(s'|s, a).V(s')$. These become computationally impractical in cases of high cardinality of S or A, or if these are infinite and continuous. In these scenarios, which form the majority of today's research, sampling of underlying distributions is used instead of exhaustive sums. As a result, various approaches use sampled data to improve approximated variable iteratively.

2.1.3 Policy evaluation

In the task of *policy evaluation*, also called a prediction task or static RL, we are given a MDP and a fixed policy π and we aim to find the value function V_π or Q_π of this policy. In the simplest setup of finite model-based static RL, the value function can be obtained by starting with an arbitrary function $V^{(0)}$ and iteratively applying the Bellman expectation equation. After expanding the expectations into sums, we directly get:

$$\begin{aligned} \forall s \in S : V^{(0)}(s) &\leftarrow 0 \\ \forall s \in S : V^{(k+1)}(s) &\leftarrow \sum_{a \in A} \pi(a|s) \left(r(s, a) + \sum_{s' \in S} P(s'|s, a) V^{(k)}(s') \right) \end{aligned} \tag{2.1}$$

After the finite number of iterations, this algorithm is guaranteed to converge into the true value function of the given policy, i.e. $V^{(n)} = V_\pi$, similarly for the action-value function Q_π. Algorithms implementing this idea are usually based on dynamic programming, with computational complexity $O\left(|A||S|^2\right)$ per iteration[1] suitable only for small setups.

Due to their high computational complexity, dynamic programming approaches tend to turn unfeasible in case of high cardinality of S or A. In such cases, as well as when the setup is model-free, we can compute the value functions by simply sampling the P and R. The agent is interacting with the environment, generating the history $[s_0, a_0, r_0, s_1, a_1, r_1, \ldots]$ in each

[1]For action-value function, the complexity even grows to $O\left(|A|^2|S|^2\right)$.

episode. Using this historical data, we can update an approximated value function V. Two main approaches are used for such updates: *Monte-Carlo* and *Temporal Difference*.

In **Monte-Carlo** (MC) update method, the agent has to complete the whole episode and reach a terminal state first. Only then, after episode termination, the gains G_t are computed for each time-step t, which are used as an unbiased sample of true value function V_π. Following the traditional α-step approximation scheme, we obtain:

$$V(s_t) \leftarrow V(s_t) + \alpha \left(G_t - V(s_t)\right) \tag{2.2}$$

The drawback of MC methods lies in high variance of G_t as it is subject to a long trajectory of (possibly non-deterministic) actions. This aspect prevails especially in long-horizon environments.

Temporal Difference (TD) approach, as opposed to MC, is able to update value function even during the episode, allowing it to be used for infinite-horizon MDPs. It uses an idea of Bellman equation, which says that $V(s_t)$ can be approximated by $r_t + \gamma V(s_{t+1})$. TD learning uses this term (called *TD target*) as a sample of V_π:

$$V(s_t) \leftarrow V(s_t) + \alpha \left(r_t + \gamma V(s_{t+1}) - V(s_t)\right) \tag{2.3}$$

The TD target is no longer an unbiased sample of true V_π, as it bootstraps from the current imperfect value of V. Despite this drawback, it can be proven that TD-evaluation does converge to V_π correctly. Additionally, thank to much lower variance of TD target in comparison with gains in MC, TD-based methods are more widespread. The term of *TD error*:

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{2.4}$$

was used in subsequent works, as we will discuss in sections 2.2 and 2.3.

TD-λ approach comes as a combination of MC and TD methods. TD takes a single-step lookahead, considering only one reward r_t, while on the other side of the spectrum MC takes full-episode lookahead, considering all future rewards in form of G_t. A possibility in between is to take n-step lookahead, considering actual rewards of n steps $r_t, \dots, r_{t+n-1}$. The novel TD-λ technique combines all aforementioned versions – it forms the target (sample of V_π) as a weighted average of all n-step lookahead values, for $n \in [1, \infty)$.

The key component in TD-λ learning lies in defining an *n-step gain* quantity combining exact rewards from first n steps, and approximating the rest by V function:

$$G_t^{(n)} = \left(\sum_{k=0}^{n-1} \gamma^k r_{t+k} \right) + \gamma^n V(s_{t+k}) \tag{2.5}$$

Note that $G_t^{(1)}$ is equal to standard TD target performing single-step lookahead, while $G_t^{(\infty)}$ represents gain G_t used for a full-episode lookahead in MC method.

If we take a weighted average of $G_t^{(n)}$ values for all n we get the *TD-λ target*:

$$
\begin{aligned}
G_t^{\lambda} &= (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}G_t^{(n)} \\
V(s_t) &\leftarrow V(s_t) + \alpha\left(G_t^{\lambda} - V(s_t)\right)
\end{aligned}
\tag{2.6}
$$

where the $\lambda \in [0,1]$ hyperparameter will tune how much we prefer longer lookaheads ($\lambda \to 1$) over shorter ones ($\lambda \to 0$). By choosing λ correctly we can greatly reduce the variance in comparison with MC methods, while introducing a smaller bias than in standard TD-evaluation.

Although we have discussed mostly state-value function approximation in this section, the same methods with only slight adjustments can be used for action-value function approximation as well.

2.1.4 Policy optimization

In the previous section we presented various methods for evaluating an existing or fixed policy. *Policy optimization* methods (also called the control task or active RL) do not work with a fixed policy, but rather actively use the evaluations to *improve* a suboptimal policy, i.e. to actually teach the agent to perform a given task.

The goal of policy optimization is to produce a policy that maximizes average overall gain, i.e. to find the optimal policy $\pi_* = \arg\max_{\pi\in\Pi}(E_\pi[G_0])$. For such policy a Bellman Optimality Equation holds:

$$
\begin{aligned}
V_*(s) &= \max_{a\in A}\left(r(s,a) + \gamma\sum_{s'\in S}P(s'|s,a)V_*(s')\right) \\
Q_*(s,a) &= r(s,a) + \gamma\sum_{s'\in S}P(s'|s,a)\max_{a'\in A}(Q_*(s',a'))
\end{aligned}
\tag{2.7}
$$

Greedy policy construction

Given any action-value function Q, we can always construct a deterministic policy acting greedily with respect to that function, i.e. given state s, always choose the action a with highest value $Q(s,a)$:

$$\pi_{\text{greedy}}(s) = \arg\max_{a\in A}(Q(s,a)) \tag{2.8}$$

This simple policy construction allows us to construct more successful policies by optimizing the action-value function instead of actual policy.

Convergence by Contraction mapping theorem

A wide span of RL algorithms can be found to have a common underlying structure that can be decomposed into two interchanging steps:

1. *Policy evaluation* acquiring more precise value function of current policy

2. *Policy improvement* generating more effective policy from computed value function

This iterative framework will converge to optimal policy π_* and optimal value function V_* or Q_*. This can be proved using the *Banach fixed-point theorem* (Banach, 1922), also known as *Contraction mapping theorem.*

Definition. *For a metric space X a function $f : X \rightarrow X$ is called a contraction mapping if and only if there exists $k \in [0, 1)$ such that $\|f(x), f(y)\| \leq k\|x, y\|$ for all $x, y \in X$.*

Theorem (Banach fixed-point theorem)**.** *Let X be a metric space and f a contraction mapping. Then f defines a unique fixed-point $x_* \in X$ for which $f(x_*) = x_*$ holds. Furthermore, x_* can be found as $x_* = \lim_{n\rightarrow\infty} x_n$ where x_0 is any point in X and $x_{n+1} = f(x_n)$.*

We will review the two interchanging steps to explain how they help to converge to optimal policy. First, policy evaluation of a fixed policy π is a contraction mapping in the space of all action-value functions Q, where Bellman Expectation Equation is the function f, true Q_π is the fixed point, and γ represents a contraction factor k. Subsequent contraction mapping can be constructed in a space of all policies Π. Starting with any policy π_n, we can construct a new one by evaluating Q_π and acting greedily with respect to it, i.e. $\pi_{n+1} = \pi_{\text{greedy}}[Q_\pi]$. Such function $f : \Pi \rightarrow \Pi$ is a contraction mapping, with a fixed point in optimal policy π_*. Therefore, by starting with an arbitrary policy π_0, alternating evaluation of Q_{π_n} and construction of greedy π_{n+1} will converge to a single optimal policy (Singh et al., 2000).

While we outlined the proof using the Bellman expectation equation and Greedy policy construction, similar approach can be applied to different algorithms too, as long as the two fundamental steps fulfill the contraction property.

2.1.5 Exploration vs. exploitation

The Contraction mapping theorem entails a theoretical guarantee of successful convergence. However, if theory is applied in practice, every state's $Q(s, \cdot)$ value has to be updated infinitely many times. Even if we put an upper limit on number of iterations (resulting in a sufficient approximation), choosing *every* state is unfeasible in tasks with high-cardinality S. In such cases we can sample states by following agent's trajectories, but we need to ensure the trajectories satisfy two key functions:

- *Exploration:* select as many states as possible to evaluate Q on majority of its domain
- *Exploitation:* select "good" states as often as possible to improve the policy

These two approaches are in opposition to each other and increasing the priority of one diminishes the function of the other. A careful balance between those, however, will not only fulfill the contraction property, but can also significantly speed up the learning process.

Simple, yet efficient exploration strategy – *ε-greedy policy* – uses the idea of random perturbations of classical greedy policy. With a probability of $(1 - \varepsilon)$, greedy action is chosen, while with ε probability, a random action is chosen:

$$\pi(a|s) = \begin{cases} (1 - \varepsilon) + \frac{\varepsilon}{|A|} & \text{if } a = \arg\max_{a' \in A} (Q(s, a')) \\ \frac{\varepsilon}{|A|} & \text{otherwise} \end{cases} \tag{2.9}$$

If the ε factor is decreasing over time by a specific schedule, the ε-greedy policy satisfies the contraction condition, due to the *GLIE* property (Greedy in the Limit with Infinite Exploration, Singh et al. (2000)). Hence, convergence to optimal policy is still guaranteed.

If the task at hand features continuous action space A, then *Gaussian exploration* can be applied to any chosen deterministic policy π:

$$\pi_{Gauss}(a|s) = \mathcal{N}\left(\pi(s), \sigma^2\right) \tag{2.10}$$

where σ can be fixed, but is preferable to be decreasing with time to ensure the GLIE property.

2.2 Discrete-space algorithms

We discussed various fundamental concepts of Reinforcement Learning in the previous section, outlining most of the key aspects and mathematical principles. With those covered, we can now join these building blocks – each time in different way – to present the basic algorithms used for RL control.

As mentioned before, RL-control algorithms commonly share the underlying structure of interchanging value function evaluation and policy improvement. In the earlier stages of research, more attention was paid to the former – this only changed after the *REINFORCE* breakthrough by Williams (1992).

The algorithms mentioned in this section are mostly suitable for discrete state- and action-space setups, as they traditionally store computed $Q(s, a)$ values in a form of a lookup table. To manage continuous (or high-cardinality) state spaces, and especially action spaces, more advanced algorithms have to be incorporated.

Policy iteration

The algorithm of *Policy iteration* (Howard, 1964) represents the fundamental basic of the RL-control task. Without explicitly mentioning it, we already covered all its essential parts in section 2.1.4. Policy iteration consists of two alternating steps:

1. Evaluate Q_π by iteratively applying Bellman expectation equation in an inner loop, until convergence for every state-action pair
2. Create a new policy by acting greedily with respect to Q_π

Value function is evaluated for *every* state in each iteration, hence no explicit exploration is needed. On the other hand, this approach limits the algorithm for usage only on small-scale tasks. Furthermore, it is a strictly model-based approach, unfeasible if the P or R are unknown or too complex.

Value iteration

The Policy iteration was aiming to learn true Q_π of the current policy in every iteration of the main loop, which led to exhaustive iterative application of Bellman's equation in an inner loop. The *Value iteration* (Bellman, 1957b), on the other hand, resolves this inefficiency.

Instead of finding the *true* Q_π, Value iteration only brings its approximation Q of Q_π *one step closer* to the true value by applying the Bellman's equation once. Policy improvement is again achieved by acting greedily w.r.t. new Q.[2]

Value iteration brought significantly faster convergence rates in comparison to Policy iterations, nonetheless other disadvantages of the approach remained.

SARSA

Policy- and Value-iteration algorithms are performing value function evaluation for every state $s \in S$ in each iteration, and thus are limited to small-cardinality state-space tasks. SARSA approach breaks this limitations by using samples of P and R, obtained as agent's trajectories.

The name hints the basic idea: agent starts in state s, performs action a gaining reward r, leading to state new state s' from which it will perform next action a'. The collection of s, a, r, s', a' is used in SARSA update, occurring after each step of the agent, directly following Bellman expectation equation:

$$\begin{aligned} Q(s,a) &\leftarrow Q(s,a) + \alpha\left(r + \gamma Q(s',a') - Q(s,a)\right) \\ Q(s,a) &\leftarrow Q(s,a) + \alpha\delta_t \qquad \text{where } \delta_t \text{ is a TD error} \end{aligned} \tag{2.11}$$

[2]The two steps can be equivalently rewritten into single one, where we iteratively apply Bellman *optimality* equation instead, having an implicit greedy policy. Such approach was used in Bellman's original paper.

Given the fundamentals analyzed earlier, SARSA approximates Q_π by sampling the P, R instead of exhaustive sums, and it does so using Temporal Difference learning (i.e. one-step lookahead). After evaluation (or rather approximation) of Q_π, the policy improvement is made by applying ε-greedy policy to current Q. Because SARSA no longer performs full state-space sweeps, an exploratory policy is needed to cover sufficient area of state space, hence traditional greedy policy cannot be used.

The idea of SARSA was first published by Rummery and Niranjan (1994) under the name '*Modified Connectionist Q-Learning*' as an extension of Q-learning for function approximators. However, when applied to a simple lookup table case, it ideologically precedes Watkins' Q-Learning.

Q-learning

A subtle but important drawback of SARSA lays in its Q-function estimate. Because of the usage of exploratory – i.e. inherently non-optimal – policy, the Q-function estimate also determines non-optimal policy. This can only change after long period of learning, when the ε-schedule brings exploration in limit to zero.

To overcome this downside, Q-learning (Watkins and Hellaby, 1989) leverages a concept of *off-policy learning*. The key idea, adopted in later algorithms as well, is to teach a *target policy* π while following a different *behavior policy* μ. Typically, π aims to reach the optimal policy π_*, while μ can be non-optimal and provides sufficient exploration.

Similarly to SARSA, Q-learning also collects s, a, r, s', but the update is performed utilizing Bellman optimality equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma \left(\max_{a' \in A} Q(s',a') \right) - Q(s,a) \right) \tag{2.12}$$

The actions here are drawn from an exploratory policy μ, while Q is approximating the value of Q_*, i.e. value function of the optimal policy. The choice of μ is much more liberal, as long as it provides satisfactory balance between exploration and exploitation.

Q-learning successfully tackles various problems of previous methods, and with the help of function approximators[3] it can also be used for continuous state-space environments, achieving astonishing performance (Mnih et al., 2015). However, it cannot be efficiently used in continuous action-space environments, due to the need of maximization over A, which turns into a non-convex optimization. This remains to be the main disadvantage of the algorithm. While there were attempts to overcome this drawback (Gaskett et al., 1999), eventually a different approach of policy gradient methods was widely adopted, as we present in the following section.

[3]discussed in section 2.3.1

2.3 Continuous-space algorithms

A large number of practical applications of RL use a continuous state space or action space. A robot control is typically continuous in both space and action spaces, and game environments may employ discrete state space with the number of states as high as 10^{170} (Go board game) or more. Such environments cannot be effectively captured in the form of lookup tables, due to memory and computational limitations. We would like to address this problem here, presenting more advanced approaches, including state-of-the-art algorithms of recent years.

2.3.1 Function approximators

A rather straightforward way to overcome the memory limitations of lookup tables is to use standardized function approximators in their place. Instead of storing Q value for each state separately, approximators allow us to generalize from seen states to the unseen ones. Using this approach, it is possible to scale-up the previous methods to large problems.

All discrete-space methods described previously use a greedy or ε-greedy policy after evaluating Q. By definition, this requires maximization over all actions to obtain an optimal action: $\max_{a \in A} Q(s, a)$. Therefore, if we wanted to introduce continuous action space into these methods, such maximization would turn into a non-convex optimization over A, being a computationally complex problem on its own. As a consequence, simpler methods – even with the help of function approximators – can be used in continuous *state*-space environments, but not in continuous *action*-space ones.

Approximator models

Almost any of the modern approximation models can be used, from linear models through decision trees up to most frequently used neural networks. Though, a few limitations still hold. Most importantly, the model has to be able to process non-stationary, non-i.i.d. data.[4] This condition arises from the exploration–exploitation framework: the shift from exploration to exploitation phase effectively causes the distribution from which the states are drawn to change over time (from the perspective of the underlying function approximator). A certain degree of plasticity is hence needed.

In terms of input/output, approximators are usually implemented to compute the vector $Q(s, \cdot)$ given the state s. In such case, dimension of input is equal to dimension of S, while the dimension of output is $|A|$. Alternative approach can be chosen, in which the input is composed of a tuple $\langle s, a \rangle$, and the output is simply scalar $Q(s, a)$. This is usually used if

[4]This condition is not always honored in practical algorithms, but in such cases, careful precautions are taken to stabilise the training process.

the actions logically form a sort of proceeding sequence, while the former is preferred if the actions are unrelated to each other.

Application to algorithms

The general goal of using approximators in basic RL algorithms is to find a parameter vector ψ of parametrized function $Q_\psi(s,a)$ that best approximates $Q_\pi(s,a)$, i.e. it minimizes the mean-squared cost function:

$$J(\psi) = E_\pi\left[\left(Q_\pi(s,a) - Q_\psi(s,a)\right)^2\right] \tag{2.13}$$

We can use traditional stochastic gradient descent to minimize this function, yielding a parameter-update step as follows:

$$\Delta\psi = \alpha\left(Q_\pi(s,a) - Q_\psi(s,a)\right)\nabla_\psi Q_\psi(s,a) \tag{2.14}$$

The last gradient term is model-specific, according to the chosen approximator. The term $Q_\pi(s,a)$ is unknown and can be sampled in several ways, the choice of which determines the resulting algorithm. For Monte-Carlo learning we use gain G_t as a sample of Q_π, for TD learning (i.e. continuous SARSA) we use TD target, and Q-learning uses the maximized target. The resulting parameter updates for these three methods are, respectively:

$$\begin{aligned}
\Delta\psi &= \alpha\left(G_t - Q_\psi(s,a)\right)\nabla_\psi Q_\psi(s,a) \\
\Delta\psi &= \alpha\left(r + \gamma Q_\psi(s',a') - Q_\psi(s,a)\right)\nabla_\psi Q_\psi(s,a) = \alpha\,\delta_t\,\nabla_\psi Q_\psi(s,a) \\
\Delta\psi &= \alpha\left(r + \gamma\left(\max_{a'\in A} Q_\psi(s',a')\right) - Q_\psi(s,a)\right)\nabla_\psi Q_\psi(s,a)
\end{aligned} \tag{2.15}$$

Convergence

Unfortunately, usage of function approximators instead of lookup tables often causes the violation of the contraction property of evaluating Q. In some cases, when alternative TD learning uses *linear* function approximators, the contraction is still guaranteed, but for all known *non-linear* approximators this does not apply. As a consequence, non-linear methods can theoretically diverge into $\psi = \pm\infty$.

DQN and Experience replay

The divergence of non-linear models does not allow the direct naïve usage of e.g. neural networks, as the most popular tool in recent years. However, various batch methods successfully prevent catastrophic divergence.

In their iconic work, Mnih et al. (2015) presented an approach able to surpass human-level performance in playing classic Atari 2600 games, gaining interest in both research community and public. Their model *Deep Q-Network* (DQN) was based on Q-learning, using a convolutional neural network Q_ψ to approximate over Q. In order to stabilise the weights ψ, which would otherwise diverse to $\pm\infty$, the authors implemented two key stabilisation concepts: *experience replay* and *target network*.

Experience replay helps to decorrelate the trajectories, hence presenting the network with nearly i.i.d. data. To achieve this, authors perform a mini-batch learning on recently collected data.

In each step, the agent stores the transition (or *experience*) tuple $e_t = \langle s_t, r_t, a_t, s_{t+1}\rangle$ into a dataset D. Then, in the parameter-update step, the Q_ψ network is trained on a minibatch of transitions drawn from uniform distribution $U(D)$ over D. This allows the network to train also on slightly older data, and greatly decorrelates the training examples it receives. The dataset D is set to a fixed capacity, discarding the oldest transitions once it is full.

The experience replay technique established a new standard subsequently used in many other methods (Andrychowicz et al., 2017; Levy et al., 2018). One of the notable variants is *Priority experience replay* (Schaul et al., 2016), in which the transitions are not chosen from dataset D uniformly, but proportionally to their TD-error. This prioritises the transitions with greatest potential for learning improvement, and was shown to accelerate training.

Usage of **target network** further stabilises the learning by decorrelation of network's inputs and targets. We can see from equation 2.15 that the targets, towards which Q_ψ is optimised, are highly dependent on the Q_ψ itself. To break this linkage, authors introduce a second neural network $Q_{\overline{\psi}}$ to act in the target term of the learning rule. Consequently, the parameter update slightly changes:

$$\Delta\psi = \alpha\left(r + \gamma\left(\max_{a'\in A} Q_{\overline{\psi}}(s', a')\right) - Q_\psi(s,a)\right)\nabla_\psi Q_\psi(s,a) \quad \text{where } \langle s,a,r,s'\rangle \sim U(D) \tag{2.16}$$

The target network $Q_{\overline{\psi}}$ is set to change at a slower rate than policy network Q_ψ. A common practice, used also in this paper, is to periodically set the weights of the target network to the current weights of the main network, e.g. perform $\overline{\psi} \leftarrow \psi$ once every C episodes. This ensures stability of the targets, while also ensuring that "fresh" targets are provided for the Q_ψ network.

2.3.2 Policy gradient methods

Taking one step further, we approach methods suitable for environments which are continuous in both state- *and* action-spaces. The key idea is to represent the policy itself as a parametrized function approximator[5] $\pi_\theta(s,a)$ and optimize it with respect to suitable objective function. Several options for such function are possible, the simplest one being the expected gain:

[5] We use terms $\pi_\theta(s,a)$ and $\pi_\theta(a|s)$ interchangeably.

$J(\theta) = E_{\pi_\theta}[G_t] = E_{\pi_\theta}\left[Q_{\pi_\theta}(s_t, a_t)\right]$. Other options include average-value $J_{av}(\theta) = E_{\pi_\theta}\left[V_{\pi_\theta}(s)\right]$ or average-reward $J_{ar}(\theta) = E_{\pi_\theta}[r_t]$. All of these satisfy the conditions and guarantees in this chapter.

While optimizing this function, we will make use of the *log-derivative trick*:

$$\begin{aligned}\nabla_\theta \pi_\theta(s,a) &= \pi_\theta(s,a)\ \frac{1}{\pi_\theta(s,a)}\ \nabla_\theta \pi_\theta(s,a) \\ &= \pi_\theta(s,a)\ \left(\nabla_{\pi_\theta(s,a)} \log \pi_\theta(s,a)\right)\ \nabla_\theta \pi_\theta(s,a) \\ &= \pi_\theta(s,a)\ \nabla_\theta \log \pi_\theta(s,a)\end{aligned} \tag{2.17}$$

In the last step a reverse of chain-rule was applied to combine the two gradient terms. Using this trick, we can compute the gradient of the objective function:

$$\begin{aligned}\nabla_\theta J(\theta) &= \nabla_\theta E_{\pi_\theta}\left[Q_{\pi_\theta}(s_t, a_t)\right] \\ &= \nabla_\theta \left(\int_S \rho_{\pi_\theta}(s) \int_A \left(\pi_\theta(s,a)\ Q_{\pi_\theta}(s,a)\right) \mathrm{d}a\ \mathrm{d}s\right) \\ &= \int_S \rho_{\pi_\theta}(s) \int_A \left(\nabla_\theta \pi_\theta(s,a)\ Q_{\pi_\theta}(s,a)\right) \mathrm{d}a\ \mathrm{d}s \\ &= \int_S \rho_{\pi_\theta}(s) \int_A \left(\pi_\theta(s,a)\ \nabla_\theta \log \pi_\theta(s,a)\ Q_{\pi_\theta}(s,a)\right) \mathrm{d}a\ \mathrm{d}s \\ &= E_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\ Q_{\pi_\theta}(s,a)\right]\end{aligned} \tag{2.18}$$

The probability distribution $\rho_{\pi_\theta}(s)$ describes the probability of the agent appearing in state s, assuming environment transitions P and the policy π_θ. The log-derivative trick helps us obtain an expected value over familiar term, which we can sample. Incorporating this knowledge into gradient-ascend scheme (for maximizing the objective function), we obtain the parameter-update step:

$$\Delta\theta = \alpha\ \nabla_\theta \log \pi_\theta(s,a)\ Q_{\pi_\theta}(s,a) \tag{2.19}$$

Again, we need to substitute $Q_{\pi_\theta}(s,a)$ for a suitable sample or estimate, which leads to different versions of advanced algorithms.

REINFORCE

The breakthrough in the field of continuous RL was brought by Williams in his most notable work (Williams, 1992). In the paper, he introduced all principles described in the previous paragraphs, and suggested the first algorithm according to his findings, bearing a simple name "*REINFORCE*". His work set a ground for most of subsequent algorithms, including current state-of-the-art methods.

The precise form of the REINFORCE algorithm directly followed parameter update $\Delta\theta$ above. Using the Monte-Carlo evaluation, he substituted actual gain G_t as a sample of $Q_{\pi_\theta}(s,a)$:

$$\Delta\theta = \alpha\ \nabla_\theta \log \pi_\theta(s,a)\ G_t \tag{2.20}$$

Actor–Critic architecture

Analogically to the variety of discrete-space RL algorithms, we can obtain different policy-gradient methods by substituting $Q_{\pi_\theta}(s, a)$ with different terms, such as gain, TD target, etc. An interesting approach is to combine a policy-gradient method with function-approximated evaluation, which leads to the so-called *Actor–Critic* architecture. The *actor* is a function approximator π_θ representing the policy itself, while the *critic* is another approximator Q_ψ responsible for evaluation of the states. Firstly proposed by Konda and Tsitsiklis (2000), the A–C architecture was proven to have significantly lower variance of $Q_{\pi_\theta}(s, a)$ estimate, and hence higher convergence rates.

As the name suggests, Actor–Critic architecture represents a class of algorithms. The first degree of freedom is in the choice of actor and critic approximator models. Another choice concerns the time frame of both actor and critic, as they can use either Monte-Carlo, TD or TD-λ approaches to estimate the gain.

A subsequent research (Sutton et al., 2000) showed that subtracting a state-dependent baseline function from Q_{π_θ} can significantly reduce variance, without introducing any bias. The state-value function V_{π_θ} comes as the best candidate for the baseline. Furthermore, as we construct the advantage function as $A_{\pi_\theta}(s, a) = Q_{\pi_\theta}(s, a) - V_{\pi_\theta}(s)$, we can show that the TD error δ_t serves as unbiased sample of $A_{\pi_\theta}(s, a)$:

$$\begin{aligned} A_{\pi_\theta}(s_t, a_t) &= Q_{\pi_\theta}(s_t, a_t) - V_{\pi_\theta}(s_t) \\ &\approx r_t + V_{\pi_\theta}(s_{t+1}) - V_{\pi_\theta}(s_t) \\ &= \delta_t \end{aligned} \tag{2.21}$$

This approach further simplifies the update step to:

$$\Delta\theta = \alpha \, \nabla_\theta \log \pi_\theta(s, a) \, \delta_t \tag{2.22}$$

CACLA

The *Continuous Actor–Critic Learning Automaton* – CACLA (Hasselt, 2012) can be interpreted as a variation of previous Actor–Critic architecture with advantage learning. It aims to resolve one of the traditional drawbacks of gradient-ascend based methods – diminishing parameter update in case of near-zero gradient values, i.e. on plateaus of value function. The key difference is that instead of using actual value of δ_t, CACLA only updates the parameters if $\delta_t > 0$, regardless of its absolute value. As summarized by the authors: "... CACLA only updates its actor when actual improvements have been observed, this avoids slow learning when there are plateaus in the value space and the temporal difference errors are small" (Hasselt, 2012). However, in order for their design to work, the actor choice is narrowed down only to Gaussian exploration policy.

Deep Deterministic Policy Gradient

Similarly to DQN in discrete action-space environments, the *Deep Deterministic Policy Gradient* (DDPG) by Lillicrap et al. (2016) also showed the great capabilities of training stabilisation, this time in continuous action-space environments. In fact, the authors of DDPG directly followed the recipe for success of DQN.

The core of DDPG consists of Actor–Critic architecture with neural networks both for the actor – π_θ and critic – Q_ψ. Similarly to DQN, the experience replay D is used to decorrelate the training inputs for both actor and critic, and to present near-i.i.d. data to both networks. The critic-target network $Q_{\overline{\psi}}$ remained, but since the actor is now also parametrised, a second actor-target network $\pi_{\overline{\theta}}$ was introduced. As summarised by the authors, having two target networks was necessary to have stable targets in order to consistently train the critic without divergence. Alternating on equation 2.16, DDPG involves $\pi_{\overline{\theta}}$ into critic's learning rule:

$$\Delta\psi = \alpha\left(r + \gamma\left(Q_{\overline{\psi}}(s', \pi_{\overline{\theta}}(s'))\right) - Q_\psi(s,a)\right) \nabla_\psi Q_\psi(s,a) \quad \text{where } \langle s,a,r,s'\rangle \sim U(D) \tag{2.23}$$

For the actor update, the original (not target) networks are used:

$$\Delta\theta = \alpha\ \nabla_\theta \log \pi_\theta(s)\ Q_\psi(s,a) \tag{2.24}$$

To further stabilise the process, authors update the target networks more smoothly than in DQN, using a Polyak-averaged (Polyak and Juditsky, 1992) version of the main networks:

$$\begin{aligned} \overline{\psi} &\leftarrow \tau\psi + (1-\tau)\,\overline{\psi} \\ \overline{\theta} &\leftarrow \tau\theta + (1-\tau)\,\overline{\theta} \end{aligned} \tag{2.25}$$

A batch normalization on the inputs and hidden layers' outputs is also applied as in Ioffe and Szegedy (2015).

Trust Region Policy Optimization

A novel and superiorly effective approach of *Trust Region Policy Optimization* (TRPO) was proposed by Schulman et al. (2015). Based on previous work on *Conservative Policy Iteration* (Kakade and Langford, 2002), they developed a practical algorithm that provides explicit lower bounds on the improvement of expected discounted reward (i.e. gain).

The monotonic improvement guarantee is achieved by indirect optimization of policy, in which authors maximise quantity $L_\pi(\tilde{\pi})$ instead of true gain $E_\pi[G_t]$. First, they express a gain of a *different* policy $\tilde{\pi}$ using gain and advantage function of an existing policy π:

$$\begin{aligned} E_{\tilde{\pi}}[G_t] &= E_\pi[G_t] + E_{\tilde{\pi}}\left[\sum_t \gamma^t A_\pi(s_t,a_t)\right] \\ &= E_\pi[G_t] + \sum_s \rho_{\tilde{\pi}}(s) \sum_a \tilde{\pi}(s,a)\, A_\pi(s,a) \end{aligned} \tag{2.26}$$

This equation implies that any policy update $\pi \rightarrow \tilde{\pi}$ that improves expected advantage at every state s, i.e. $\sum_a \tilde{\pi}(s,a)\, A_\pi(s,a) \geq 0$, is guaranteed to improve the performance of the policy.[6] However, since the gain $E_{\tilde{\pi}}[G_t]$ is difficult to optimise directly, the authors use a surrogate quantity:

$$L_\pi(\tilde{\pi}) = E_\pi[G_t] + \sum_s \rho_\pi(s) \sum_a \tilde{\pi}(s,a)\, A_\pi(s,a) \tag{2.27}$$

drawing state frequencies from current $\rho_\pi(s)$ instead of $\rho_{\tilde{\pi}}(s)$, which they show is a local approximation of $E_{\tilde{\pi}}[G_t]$ and matches it to the first order.

The monotonous nature of improvement in their step is theoretically achievable, but practically it does not work for larger steps due to estimation and approximation errors. To make the improvement steps larger, yet still robust, they introduce a constraint parameter δ on the KL divergence between the new policy and the old policy, i.e. a *trust region* constraint:

$$\begin{aligned} &D_{KL}^{max}(\pi, \tilde{\pi}) \leq \delta \\ \text{where} \quad &D_{KL}^{max}(\pi, \tilde{\pi}) = \max_s D_{KL}\left(\pi(s,\cdot) \,||\, \tilde{\pi}(s,\cdot)\right) \end{aligned} \tag{2.28}$$

Maximizing the surrogate $L_\pi(\tilde{\pi})$ subjected to this constraint yields a powerful algorithm, which offers exceptional performance in a wide variety of tasks. Due to the success of TRPO, it is widely used in other recent works, for instance 3D object grasping task (Kovács, 2017) or locomotion skills acquisition (Florensa et al., 2017). An alternative version called PPO – *Proximal Policy Optimization* (Schulman et al., 2017) – was used to train emergence robust locomotion policies in difficult environments (Heess et al., 2017).

[6]This also illustrates a single step of the contraction mapping in the classic Policy iteration algorithm.

Chapter 3

Hierarchical Reinforcement Learning

Over the period of its research, Reinforcement Learning successfully went through several major milestones. The most important ones included the scaling from discrete to continuous state spaces with the use of function approximators, and from discrete to continuous action spaces with the Williams' REINFORCE breakthrough. Scaling up the framework further, we face new challenges that might be achieved in upcoming years, namely high increase in dimensions and efficient sparse-reward handling.

3.1 Motivation for hierarchization

The idea of *Hierarchical Reinforcement Learning* (HRL) offers a possibility to resolve these open questions. Though coming in different forms, all HRL methods aim to decompose the whole task into hierarchy of easier ones, e.g. first learn how to accelerate, steer, and break the car, *then* learn how to navigate through the city, using the skills acquired earlier. Hierarchization is mainly driven by three principles that need to be addressed in order to scale RL further up.

Curse of Dimensionality

The most notable and difficult-to-overcome problem has acquired a name "*Curse of Dimensionality*", coined by Bellman (1957a). A large number of approaches – including non-RL ones – tried to break this curse[1] with questionable results.

The key idea of this long-intractable problem is that as number of dimensions grow, the volume of underlying space grows exponentially, making it harder and harder to search through. The problem was occurring in earlier applications with discrete spaces too, but it became much more eminent with the rise of continuous methods. For instance, in robotics

[1]Some authors got a bit carried away by talking of "*exorcising the demon of dimensionality*" (Moerman, 2009; Dayan and Hinton, 1993) or other notations.

we often work with up to 20 or more degrees of freedom, yielding 20-dimensional action space, accompanied by enormous state space based on the robot's sensors, which can go to hundreds or thousands in case of visual input. Evaluating all states quickly becomes impossible, and sampling must be ensured in a careful way, as RL requires that data must be collected throughout the whole space.

In robotics or other fields, the task is often rendered tractable by introduction of a hierarchy to the tasks: the lower layer of hierarchy takes care of the primitives, such as locomotion tasks, grasping, etc.; while higher levels combine those to perform high-level tasks. However, if the hierarchy decomposition is designed by the engineer incorrectly, the dynamic learning capabilities of the system may be significantly restrained. Certain level of autonomy in learning hence must be assured.

Reward sparsity

One of the key principles hardwired into MDP and RL is that the reward r_t is not strictly associated with current action a_t, but rather with an indefinitely long sequence of past actions. The agent must be able to recognize this, reinforcing both current and past actions accordingly. In an extreme, yet not infrequent case, the reward is only given at the end of an episode when the agent completes (or definitively fails) the task. Such cases can include playing a board game, in which the agent keeps getting zero rewards throughout the game, and only gets ± 1 reward after winning/loosing the match.

Potential solution, used in numerous flat-RL approaches, is called *reward shaping*. This term denotes creating a rather complicated reward function, often composed of several weighted components and if-else statements. Individual components of shaped reward signal encourage certain aspects of the behavior that the engineer thinks are important, or, on the other hand, discourages the agent from unwanted actions. As an example, Popov et al. (2017) used a reward function composed of five relatively complicated terms which needed to be carefully weighted in order to train a policy for stacking a brick on top of another one.

On the other hand, Heess et al. (2017) showed that a sparse reward can produce much more stable and robust policies, as the agent is focused solely on the main task, not on artificial constrains we placed. Unexpected novel behaviors can also emerge in such conditions. Furthermore, this approach is highly domain-agnostic – creating a straightforward sparse reward is often trivial, while the reward engineering requires both RL expertise and substantial domain-specific knowledge.

Despite the advantages in robustness and simplicity, the reward sparsity comes with an additional challenge. Successful exploration is particularly difficult in sparse-reward environments. At the early stages of the learning process the agent observes no rewards, hence no learning – parameter update – is actually performed. The training can only start

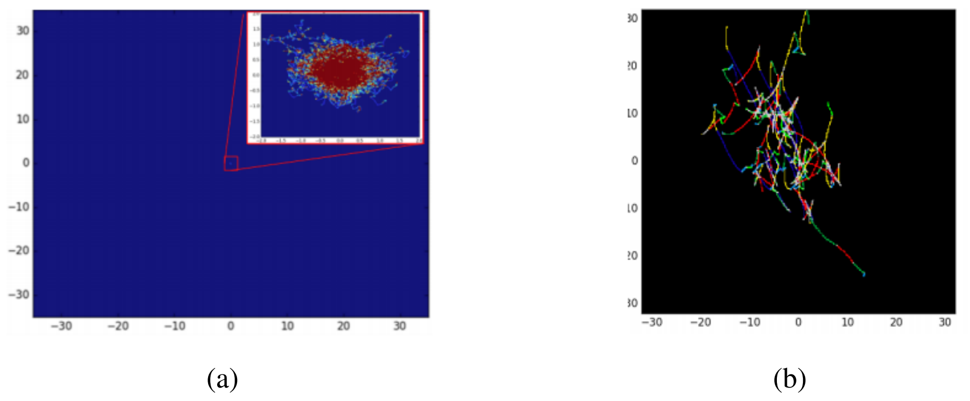


Figure 3.1: Initial exploration in flat RL and HRL. The agent is spawned in the middle and receives zero reward at each time-step. (a) Normal RL: exploration by atomic actions covers minimal area around the starting point. (b) HRL: exploration using pre-trained skills spans over significantly larger space (Florensa et al., 2017).

after the untrained policy explores some rewarded state, in pretty much haphazard manner. The probability of discovering the first rewarded state decreases exponentially with the number of steps required to get there, and so an effort should be aimed to reduce this horizon.

When the task is decomposed into a hierarchical structure, the lower-level controllers become responsible for primitive tasks, each of which is several atomic actions long. The main controller subsequently uses these skills/subtasks/options instead of the atomic actions. As a consequence, the planning horizon for discovering the first rewarded state is decreased significantly. This helps to substantially extend the explored area, as shown in figure 3.1.

Reuse of knowledge

Last but not least, hierarchical architecture in RL offers an enhanced knowledge-reuse capabilities. As many regions of state space resemble each other, it is effective to reflect such similarity in the policy itself. In the simplest flat-RL scenario, the agent has to learn each resembling region from the scratch, suffering a great loss in efficiency. On the next level, general function approximators such as neural networks offer certain degree of knowledge reusability. However, it is rather emergent, and as such it is not possible to control and difficult to explicitly exploit.

Certain HRL architectures form a next step of this ladder – using their hierarchical structure, the knowledge reuse is designed and better targeted. Specific low-level controller learned to perform certain task, e.g. walking of a humanoid robot, can work in different regions of space state, since all it needs is a plane to walk on. This can be directly exploited by the high-level controller which needs to navigate the robot through a maze.

3.2 Common features in research

Hierarchical branch of RL is still a fairly new, open field of study with a few fixed standards or baselines. Some features, however, reoccur in numerous research papers concerning HRL, either strictly specified or implicitly assumed, as previously summarized also by Barto and Mahadevan (2003), and Dillinger (2019).

Most importantly, all HRL approaches decompose the problem into fixed or adaptive hierarchy of tasks, such that higher-level tasks can invoke the lower-level ones, just as if those were their primitive actions, as depicted in figure 3.2b. The high-level task represents the original problem at hand and is solved by RL agent, typically denoted by π^H, while the lower-level controllers may be fixed, pre-trained, or solved by separate RL themselves, and they are denoted by $\pi^L_1, \ldots, \pi^L_n$. Hereinafter, we will refer to the main problem, solved by high-level controller, as a core problem or core MDP, while the tasks solved on lower levels will be called skills or sub-MDPs. Throughout the research field, other names have been given to skills, such as macros, meta-actions, options, behaviors, temporally extended actions, etc. From the vantage point of a high-level controller, the skills either augment the action space of the original atomic actions $a \in A$ of the core MDP, or replace it completely. The high-level controller π^H can then at any time t choose a certain skill π^L_i as its action: $a^H_t = \pi^L_i$.

It is important to note that as the high-level controller may no longer have access to perform atomic actions, it might not be able to converge into an optimal solution of the core

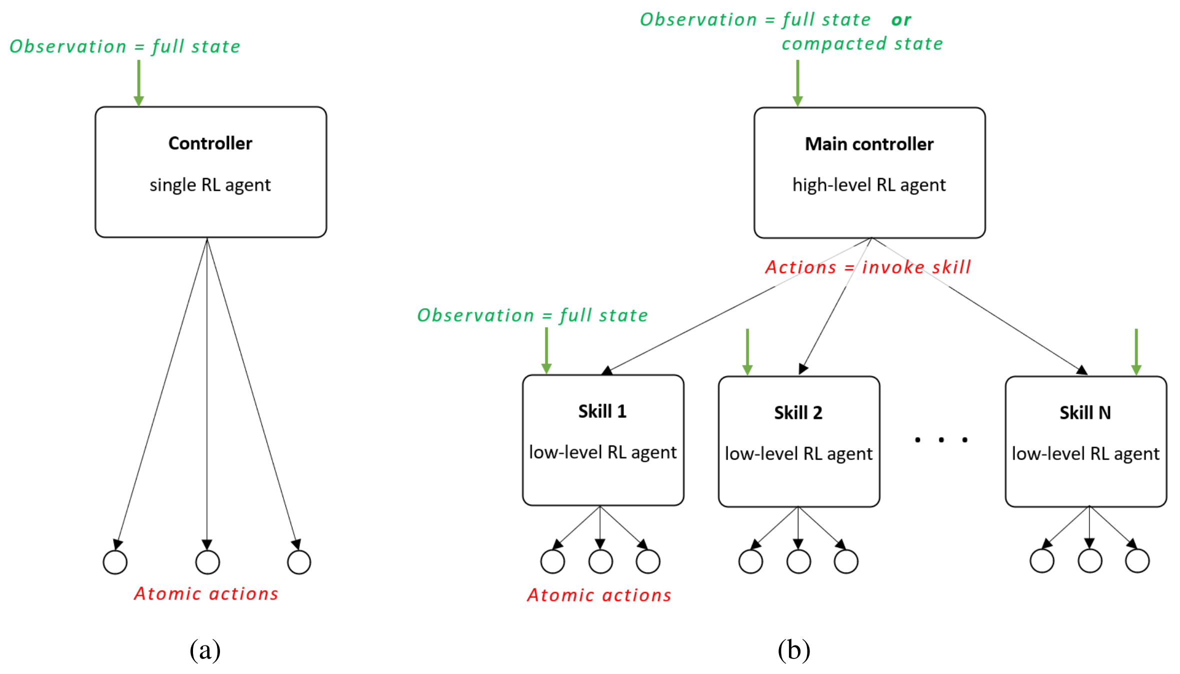


Figure 3.2: Comparison of (a) traditional flat-RL architecture, and (b) typical two-layered hierarchical RL architecture.

MDP. If skills $a^H \in A^H$ are fixed and they substitute atomic-action space A, then it is only possible to find an optimal policy for the high-level MDP $\langle S, A^H, P, p_0, R, \gamma \rangle$, which is similar, yet not identical to the original core-MDP. As an example, we can task a walking robot to navigate through the maze and provide it with two skills: walk forward and turn left, but no 'turn right' skill. The high-level controller can still learn a strategy to solve the maze with given skills, however it will be clearly suboptimal for cases where the robot should have turned right. This principle of *optimality under given hierarchy* must be accounted for when designing HRL architectures (Sutton et al., 1999).

As a reaction, some approaches involve a certain level of autonomy for choosing the skills. The decision of what behavior is the correct one for a skill is rather difficult one, and techniques used in resolving it are loosely connected with the field of *intrinsic motivation* (IM). In flat RL, intrinsic motivation is usually represented as an internal reward (in addition to the external reward from the environment), aimed to boost exploration and/or competence for a certain subtask. HRL approaches capable of autonomously identifying skills often employ similar techniques – especially specifying a task-agnostic internal reward for training the skills is widely used. In contrast with popular techniques in flat-RL, HRL skill acquisition usually falls into class of *competence-based IM*, as described in Baldassarre (2019).

Due to the usage of skills that are embedded into core-MDP's action space, or even replace it altogether, different actions can now take different time to complete. Traditional MDP does not account for such a scenario. Therefore Semi-MDP framework became a base for hierarchized RL architectures. Even though SMDP theory allows actions to last arbitrarily long time $\tau \in \mathbb{R}^+$, truly continuous time is rarely used, and rather a simplified version is used with integer durations $\tau \in \mathbb{N}$ (Sutton et al., 1999; Dietterich, 2000). As an important contribution to the field, Sutton et al. (1999) also showed that both Bellman expectation equation and Bellman optimality equation still hold true in such SMDPs, as the durations τ can be marginalized out of the sums, surrogated only by the transitional probabilities $p(s', \tau | s, a)$. This proof allows traditional RL techniques to be used on the core MDP, once the skills are fixed.

3.3 Research survey

3.3.1 Hierarchies of Abstract Machines

First attempts of reducing the MDP into more compact SMDP came in the idea of *Hierarchies of Abstract Machines* (HAMs) by Parr and Russell (1998). In their work, policies considered by the learning process were constrained by hierarchies of partially specified machines. Each machine is a stochastic finite state automaton that represents an abstraction over specific subspace of the underlying MDP. Instead of solving the original MDP, a new *HAM-induced MDP* was constructed with its action space consisting only of abstract machines, thus significantly decreasing the size of the problem.

Each machine has to be specified by the designer upfront. Formally, an abstract machine is a triple $\langle \mu, \mathcal{I}, \delta \rangle$, where μ is a finite set of machine states, $\mathcal{I}$ is a stochastic function from MDP states to machine states that determines the initial machine state, and δ is a stochastic function for choosing next machine states.[2] Each machine's state can be of one of four types: *Action* states execute an action in the environment; *Call* states execute another machine as a subroutine; *Stop* states halt the execution of the subroutine; and *Choice* states nondeterministically select a next machine state. Seen from the perspective of HRL, each machine represents a stochastic policy (skill) with extra possibilities of calling another policy or halting its execution. As these policies can take various times to complete, the final HAM-induced MDP satisfies the definition of SMDP, and can be solved by traditional algorithms.

The most significant drawback of Parr & Russel's work is the reliance on the predefined design of the machines. The HAM approach is predicated on engineers and control theorists being able to design good controllers that will realize specific lower-level behaviors. Should they fail to do so, the optimal solution of HAM-induced MDP will hit the principle of optimality under given hierarchy, hence learning a highly non-optimal final policy.

3.3.2 Options

First introduced by Sutton et al. (1999), the *Options framework* soon set a widely used trend in the HRL field for the following years. In their paper, authors extended the usual notion of action to include *options* – closed-loop policies for taking action over a period of time.

Options consist of three components: a non-deterministic policy $\pi : S \times A \to [0, 1]$, a termination condition $\beta : S \to [0, 1]$, and an initiation set $\mathcal{I} \subseteq S$. An option $\langle \mathcal{I}, \pi, \beta \rangle$ is available in state s_t if and only if $s_t \in \mathcal{I}$. If the option is taken, then actions are selected according to π until the option terminates stochastically according to β. Certain analogy can be found between options and HAMs, with policy π corresponding to the transition function δ, $\mathcal{I}$ remaining a description of initial state, and β corresponding to stop-states of HAM. On the other hand, the significant and highly important difference between these two approaches is that while HAMs *replaced* the action space, options *augment* it. Options are presented as a generalization of actions, hence all original atomic actions are just a special case of one-step option, which always terminates after the first step ($\beta(\cdot) = 1$). This union of traditional MDP actions and temporally extended SMDP options gave the name to the article "*Between MDPs and SMDPs [...]*", and is depicted in figure 3.3.

[2]Definition from Hengst (2010), as Parr & Russel did not state a formal definition in their paper.

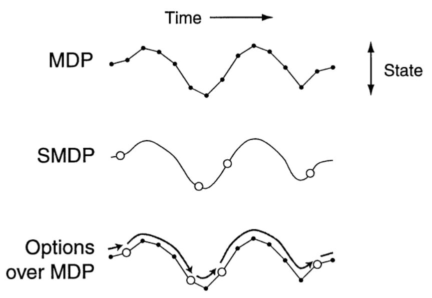


Figure 3.3: The state trajectory of an MDP is made up of small, discrete-time transitions, whereas that of an SMDP comprises larger, continuous-time transitions. Options enable an MDP trajectory to be analyzed in either way (Sutton et al., 1999).

Despite their similarities, HAMs and options put different emphasis on the approach. Options were intended to augment atomic actions, as the temporally extended actions executed by the actions yield an SMDP. As for HAMs, if the collection of machines replace atomic actions, the SMDP can be significantly reduced. There is a debate about the benefits when primitive actions are retained. Reinforcement learning may be accelerated because the value function can be backed up over greater distances in the state space and the inclusion of atomic actions guarantees convergence to globally optimal policy. On the other hand, the introduction of additional actions (options) increases the storage and exploration necessities (Hengst, 2010).

3.3.3 MAXQ value function decomposition

Even though Sutton et al. brought significant contribution to the HRL field, both in their theoretical foundations and practical algorithms, fully adaptable learning of core MDP *and* skills at the same time was still not possible. Dietterich (2000) resolved this issue by learning all involved skills simultaneously with his thorough work on *MAXQ value function decomposition*, or MAXQ for short.

In his work, Dietterich (2000) introduced yet another variation of skills, slightly different from HAMs or options, called a subtask.[3] Similar to options, subtasks are perceived as a generalization of atomic actions, thus atomic actions are just a special case of subtasks. Each subtask consists of three components: a deterministic policy π that can select actions according to the hierarchy described below, set of termination states $T \subseteq S$, and pseudo-reward function used for learning the subtask. The tasks are arranged in a predefined hierarchy, forming a directed acyclic graph where the root task represents the core MDP, and each task (policy) can only invoke its children. Example of such a hierarchy adopted from the original

[3]Despite their many similarities, we will refer to HAMs, options, and subtasks by their names proposed by original authors, to emphasize the subtle but important differences between them.

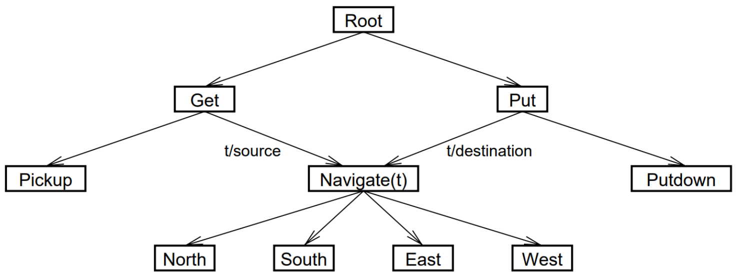


Figure 3.4: Example of task graph for MAXQ, depicting the taxi problem (Dietterich, 2000).

paper is shown in figure 3.4. The hierarchy describes a taxi problem, in which the agent has to first navigate to a passenger, pick him up, then navigate to destination and drop the passenger. Only the leaves of this graph represent atomic actions.

The decomposition of subtasks implies a decomposition of the core SMDP $\mathcal{M}$ into sub-SMDPs $\{\mathcal{M}_1, \ldots, \mathcal{M}_n\}$. For each $\mathcal{M}_i$ we train a separate policy π_i which can only choose its children as actions – an action space A_i of $\mathcal{M}_i$ consists of its child policies, i.e. $A_i = \{\pi_j | \mathcal{M}_j \text{ is a child of } \mathcal{M}_i\}$. This means that when policy π_i produces its action in given state $\pi_i(s) = a$, the *action itself* represents another policy: $a = \pi_j$. We can then ask child policy for action in given state $\pi_j(s)$ and so on, until we get to atomic actions returned by an atomic subtask.

The hierarchy described so far could be easily modeled in previous frameworks, however this precise definition allowed Dietterich to formulate his value function decomposition. The hierarchical value function $V_\pi(i, s)$ denotes a value of a state s under the assumption that policy π_i is executed until it terminates (i.e. not the whole horizon of core MDP $\mathcal{M}$, only that of $\mathcal{M}_i$). The quantity $R_i(s, a)$ represents the expected *immediate* return of action/subtask a executed from state s within SMDP $\mathcal{M}_i$. The key observation is that $R_i(s, a) = V_\pi(a, s)$, as the right-hand side corresponds to a lower-level SMDP. Expanding on this equation in combination with traditional Bellman equation, we get Dietterich's value function decomposition for hierarchical SMDPs:

$$\begin{aligned} V_\pi(i, s) &= V_\pi(\pi_i(s), s) + \sum_{s',\tau} P(s', \tau | s, \pi_i(s)) \; \gamma^\tau \; V_\pi(i, s') \\ Q_\pi(i, s, a) &= V_\pi(a, s) + \sum_{s',\tau} P(s', \tau | s, \pi_i(s)) \; \gamma^\tau \; Q_\pi(i, s', \pi_i(s')) \end{aligned} \tag{3.1}$$

This decomposition offers a recursive way to compute the value function of all sub-SMDPs, as higher-level value function on the left side is expressed by lower-level value functions on the right side. This allows for learning all value functions simultaneously, upon which a practical algorithm was presented.[4]

[4]The details of the algorithm are rather complex and beyond the scope of this review, for comprehensive description refer to Dietterich (2000).

Using MAXQ we can resolve one of the key drawbacks of previous methods – instead of manually specifying all skills' policies, the agent is now able to learn all of them in one process. However, the hierarchy is still to be specified upfront, which may again lead to suboptimal solutions due to the principle of optimality under given hierarchy. Further problematic (or at least disputable) aspects stemmed from the analogy with Sutton's options: subtasks extend the action space, hence practically expanding the SMDP instead of compacting it.

3.3.4 Methods for automated subgoal discovery

In Sutton et al. (1999) the authors also briefly introduced the notion of a *subgoal*: "It is natural to think of options as achieving subgoals of some kind, and to adapt each option's policy to better achieve its subgoal." However, they assumed the subgoals are given and did not address the larger question of the source of the subgoals.

As adapted by later works, a subgoal is usually understood as a state, or group of states, with higher importance to the core MDP, and individual skills should be trained to achieve these subgoals. Setting subgoals is also tightly coupled with specifying the hierarchy. In the approaches mentioned earlier, the hierarchy and subgoals had to be specified a priori, which could lead to significant suboptimality if the engineer's design was incorrect. In the following sections we examine variety of methods to identify the subgoals dynamically. For all of these, some common features can be observed. First, almost all relevant work focused on discrete state-space environments with presented methods that are hardly scalable to continuous spaces. Second, none of the methods can sufficiently find optimal or near-optimal subgoals or hierarchical decomposition – they all work rather heuristically, each emphasizing a different aspect of the problem. This is not surprising, as finding the optimal subgoals is fundamentally difficult, but it has to be borne in mind that each heuristic does have its weak spot. Many of the approaches employ the gridworld environment with room-to-room task to demonstrate the results – it is easy to visualize and it stimulates the idea of skills having to learn how to exit a room. A sample multi-room environment is shown in figure 3.5.

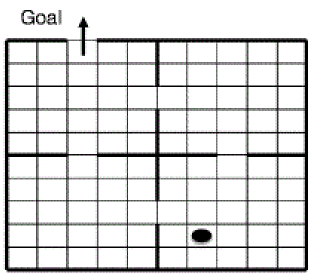


Figure 3.5: A sample four room gridworld environment, agent's location is represented by the black oval (Hengst, 2010).

An important note is also in place that not all useful skills can be characterised by subgoals. For example, in an environment with a robot collecting resources within a room, the best skill is the one correctly operating its locomotion. We can observe a subtle distinction between *subgoal-based* and *behavioral* skills. The former ones are focused solely on reaching a given subgoal state, terminating when they achieve to do so. Reaching a room or otherwise specified position is the most common example. The latter ones, on the other hand, perform a useful behavior which can be applied in any situation, and can be executed for indefinite amount of time. Such skills usually represent locomotion or sensorimotor tasks.

Region density

McGovern and Barto (2001) introduced an idea that frequently visited states might serve as useful subgoals, as shown in figure 3.6. The agent will probably repeatedly visit these states in the future, and may save time by having local policies for reaching them. As illustrated on a room-to-room navigation task, they map the visitation frequency of states during the agent's exploration to recognize regions: "If the agent uses some form of randomness to select exploratory primitive actions, it is likely to remain within the more strongly connected regions of the state space (e.g. room). An option for achieving a bottleneck region, on the other hand, will tend to connect separate strongly connected areas. [. . .] A doorway links two strongly connected regions. By adding an option to reach a doorway subgoal, the rooms become more closely connected. This allows the agent to explore its environment more uniformly." Their work laid a base ground for many successor papers, as discussed later.

Instead of using a simple frequency of visits, the authors suggest using an approach based on multiple-instance learning. Considering each trajectory as a bag of states (or observations), they split these trajectories into *positive* ones connecting two regions, and *negative* ones staying within a single region. By using the concept of diverse density, we can identify the states – bottlenecks – that occur in positive trajectories and do not in negative ones, hence find a suitable subgoal.

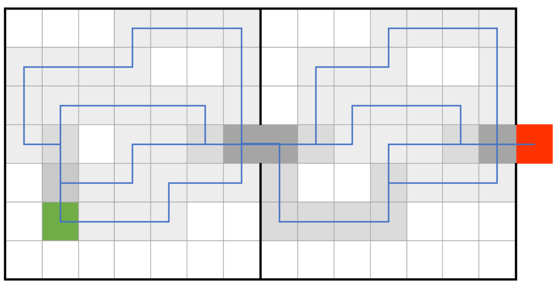


Figure 3.6: An example of visitation frequencies in gridworld. An agent starts on the green tile and aims for the red one, the shades of gray express the visitation frequency of each tile. The doorway tiles showing the highest frequency present a natural selection for a goal.

Q-Cut

Another approach was introduced by Menache et al. (2002). The authors build directly on the previous work of McGovern and Barto (2001), but they treat the whole problem from the perspective of the graph theory. They treat the whole state space as a graph, in which vertices correspond to states and edges to transitions between them. Edges are weighted according to how many times the agent performed such a transition, resulting in a weighted graph. Such a graph is subsequently fed into standard Max-cut/Min-flow algorithm, which identifies the area of minimal flow, i.e. the bottleneck. Similarly to McGovern and Barto (2001), this bottleneck is then used as a goal for new skills.

Behavioral patterns

In their following work, McGovern and Barto (2002) improved their own work of subgoal discovery via region density. The idea stemmed from expanding their framework but recognizes the skills directly, instead of using a two-step process of first discovering a subgoal and then learning a new skill to achieve it. As the agent explores a particular problem, it keeps a record of states or subsequences that occur relatively frequently in trajectories that culminate in reward. By searching for frequent patterns within successful trajectories, this approach can isolate and directly create a new skill.

Predecessor count

Altering the previous framework, Goel and Huber (2003) proposed an approach based on the number of predecessors of a given state. They define a subgoal state as one satisfying the property: “the state-space trajectories originating from a significantly larger than expected number of states lead to the subgoal state, while its successor state does not have this property”. Thus they introduce a measure $C_n(s)$ representing a number of distinct states occurring exactly n steps before entering state s, and the overall number of predecessors of a state:

$$C(s) = \sum_{i=1} C_i(s) \tag{3.2}$$

Targeting the change of C rather then its absolute values, they express the slope of C over time as $\Delta_t = C(s_t) - C(s_{t-1})$. Afterwards they follow their definition of subgoal state – a state s_t is marked as a subgoal if its Δ_t is significantly grater than Δ_{t-1}. The vague “significance” is explicitly expressed as a fixed threshold for the ratio Δ_t/Δ_{t-1}.

IFIGE skill discovery

An approach by Metzen and Kirchner (2013) also employs the graph-oriented approach to the problem, similar to Menache et al. (2002). The most notable difference in their *Incremental*

Force-based Iterative Graph Estimation (IFIGE) is the adaptation to continuous state-space environments.

The implicit one-to-one relation between states and nodes in the graph used in previous works does not stand in the case of continuous state-spaces. Hence, the authors designed an algorithm for construction and adaptation of a graph that would adequately represent a vast state space and the transitions between its regions. Building on their previous work (Metzen, 2013a), they use an iterative method for adding vertices to the graph, each of which represents a closely connected subspace of states. These vertices are moved according to the "forces" that ensure both the sample representation and graph consistency.

When such a graph is built, new skills can be identified. Instead of max-cut algorithm, the authors employ OGAHC – *Online Graph-based Agglomerative Hierarchical Clustering* (Metzen, 2013b) to create a partition of the graph into separate clusters. For each pair of clusters, a skill is trained to transition from one to another.

Similarly to previous works, IFIGE-based learning employs a developmental stage to train and fix all the skills prior to training the main task. It is also important to note that even though this approach can cope with continuous state-space environments, its graph-constructing part is not applicable to continuous action spaces.

3.3.5 Skill chaining

As most of the previous work was focusing solely on the discrete-state domains, the *Skill chaining* algorithm by Konidaris and Barto (2009) set the goal to overcome this drawback. Although they built on Sutton's options like others, certain modifications had to be made to make continuous options feasible. Most notably, the *target regions* T_i were introduced to replace single-state targets, as reaching the same state twice is virtually impossible in continuous domain. Although skill chaining works with the concept of a subgoal and its automatic identification, the subgoals themselves are treated in rather different, much less flexible way.

The core of their work is based on the idea that useful subgoals are those likely to lie on a solution path of the task the agent is facing. Therefore, their aim was to produce a sequential *chain of skills*, whose subsequent execution will lead agent to overall goal.

The cornerstone of skill chaining is teaching a skill that reaches target region T, when the agent is relatively close to it. Given the region T_i, they initialize the agent in the proximity of this region and learn policy π_i to reach the desired region. The initiation points gathered during the learning are labeled as *positive* – from where agent succeeded to reach T_i, and *negative* – from where it did not. A standard continuous-space classifier is then trained to recognize the positive initiation points, resulting in approximation of initiation set $\hat{I}_i$. As a result, a skill $\alpha_i = \langle \hat{I}_i, \pi_i, T_i \rangle$ was created given only T_i as an input.

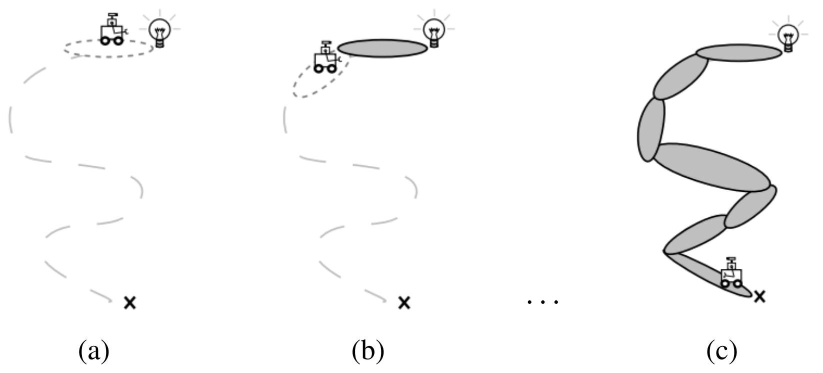
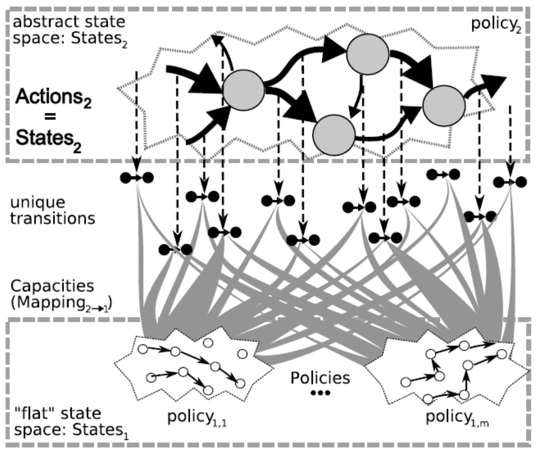


Figure 3.7: An agent creates skills using skill chaining to reach the goal (light bulb). (a) First, the agent encounters a target event and creates an option to reach it. (b) Entering the initiation set of this first option triggers the creation of a second option whose target is the initiation set of the first option. (c) Finally, after many trajectories the agent has created a chain of options to reach the original target (Konidaris and Barto, 2009).

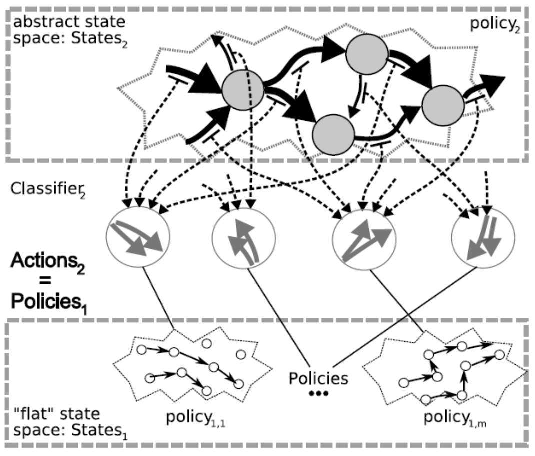
The creation of skills proceeds in a backwards-step fashion: first learn a skill α_0 to reach the overall goal, then learn a new skill α_1 to reach α_0, etc., as shown in figure 3.7. Expressed formally: first, set the target region of a first skill equal to overall target, i.e. $T_0 = T$, and learn skill $\alpha_0 = \langle \hat{I}_0, \pi_0, T_0 \rangle$. Then, for each next skill, set $T_i = \hat{I}_{i-1}$ and learn i-th skill $\alpha_i = \langle \hat{I}_i, \pi_i, T_i \rangle$. This procedure generates a chain of skills $[\alpha_0, \ldots, \alpha_n]$ which, when executed in reversed order, can successfully reach the main goal.

Even though the skills are clearly organized in a well-defined linear structure, they are subsequently fed as action set into the main controller learned by traditional RL. Thus, they may not be executed sequentially, for example if the agent learns a better policy for some parts of the chain. The authors also propose a slightly generalized version of this approach, where a tree of skills is built instead of a simple linear sequence.

The skill chaining is an interesting concept for HRL decomposition in continuous spaces, however several shortcomings had to be admitted in the process. Most importantly, the algorithm is well suitable for near-linearly organized tasks, but should fail to achieve reasonable results if the environment features a high branching factor, or if it is not bounded at all. The exploration effectively works backwards – from the target to the start – but it is not guided by the true reward signal. That means that the only "guide" for the learning process has to be the environment, organized in a friendly way with respect to this algorithm. Alternatively, one can use the skill-tree version to allow for some degree of branching, although authors place rather superficial heuristic conditions on the branching process. To sufficiently cover an unbounded state space, the branching factor would however need to be fairly high – that would lead to an exponential growth in the number of skills, effectively mitigating the benefit of reduced SMDP.


abstract state space: States$_2$
policy$_2$
Actions$_2$ = States$_2$
unique transitions
Capacities (Mapping$_{2\to1}$)
Policies
•••
"flat" state space: States$_1$
policy$_{1,1}$
policy$_{1,m}$

(a)


abstract state space: States$_2$
policy$_2$
Classifier$_2$
Actions$_2$ = Policies$_1$
Policies
•••
"flat" state space: States$_1$
policy$_{1,1}$
policy$_{1,m}$

(b)

Figure 3.8: The hierarchical structure of HASSLE and HABS: the large gray circles are the high-level states, the thick black arrows represent the high-level actions. (a) HASSLE: each subpolicy specializes in one or more of the unique transitions between high-level states (expressed as •→•). C-values are represented by the thickness of gray lines connecting subpolicies to unique transitions. (b) HABS: the actions on the high level are classified to a small set of characteristic behaviors. The classes are each associated with a particular subpolicy (Moerman, 2009).

3.3.6 HASSLE & HABS

In contrast to previous work, Bakker and Schmidhuber (2004) took a different perspective to handle continuous[5] autonomous HRL. Their main idea comes in the skill subpolicies that are initially uncommitted to any particular behavior, and they specialize their capabilities on-the-go. This gave the name to their *Hierarchical Assignment of Subgoals to Subpolicies LEarning algorithm* (HASSLE).

The HASSLE architecture is composed of two layers: high-level policy π^H and a fixed number of low-level policies $\pi^L_1, \dots, \pi^L_n$. The low-level state space S^L corresponds to core-MDP state space S, while the higher level uses an a priori constructed decomposition of S into abstract states s^H. The high-level state space S^H is therefore different, and much smaller than S, achieving compaction of the core MDP. The concept of action spaces is however the most innovative part of HASSLE: on the lower level, actions A^L again correspond to core-MDP actions A. Though, on the higher level an interesting concept is introduced, as actions A^H are identical to the *states* S^H.

At any time-step t^H of its time scale, the high-level policy π^H receives an observation of the current state s^H_{start} as its input, and its action is the selection of another high-level state

[5]Their original paper works with discrete state space, however it is directly scalable to continuous ones thanks to the use of function approximators.

s^H_{goal} as the current subgoal. Essentially, it selects the high-level state that it wants to see next. After π^H has selected the next state, aiming for transition $\langle s^H_{start}, s^H_{goal}\rangle$, a low-level policy π^L_i is selected to execute this transition – the process of subpolicy selection is described below. The job of a low-level policy is to reach the subgoal given to it: its input at each time-step t^L is the actual state s_{t^L} augmented with a tuple $\langle s^H_{start}, s^H_{goal}\rangle$, and its actions correspond to actual actions a of core-MDP. The HASSLE architecture is shown in figure 3.8a.

The process of subpolicy selection is tightly coupled with the subpolicy specialization and assignment to subgoals. Each low-level policy π^L_i stores a table of the so-called *C-values* of $\langle s^H_{start}, s^H_{goal}\rangle$ pairs, each of which represents the "*capability*" of π^L_i to execute transition from s^H_{start} to s^H_{goal}. High value of $C_i(s^H_{start}, s^H_{goal})$ denotes that policy π^L_i is highly specialized in execution of this transition. When high-level policy π^H chooses a new subgoal to be reached, the low-level execution policy is selected probabilistically according to their C_i values. This ensures that more specialized policies are selected more often (exploitation), while others are still allowed to "try" (exploration).

A rigid learning schema was developed to successfully train π^H, all of π^L_i, and values of C_i. The high-level policy uses nearly standard advantage learning (generalization of Q-learning), with an added rule to punish directly inexecutable transitions, e.g. when the agent chooses to go directly to the room on the other side of the building. Low-level policies also use advantage learning, with reward signal proportional to whether they reached the desired subgoal or not. Capability values are simply collected over time, considering also the time π^L_i needed to reach the subgoal. More details can be found in Bakker and Schmidhuber (2004), as they are beyond the scope of this report.

HASSLE introduced an effective way for simultaneous learning of all involved policies within continuous spaces. However, just like other approaches before, it did not address the issue of finding a goal – in their work, the authors use a simple vector quantization method to divide the whole state space S into subregions representing s^H states. Furthermore, as presented by Moerman (2009), this subdivision may lead to "action explosion" of a high-level policy. Given S^H, there are $\left|S^H\right|^2$ unique transitions, all of which have to be credited by C_i values for all subpolicies, as depicted in figure 3.9.

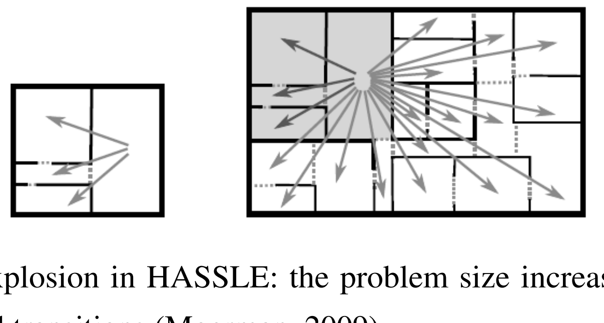


Figure 3.9: Action explosion in HASSLE: the problem size increases, and therefore the number of higher-level transitions (Moerman, 2009).

HABS

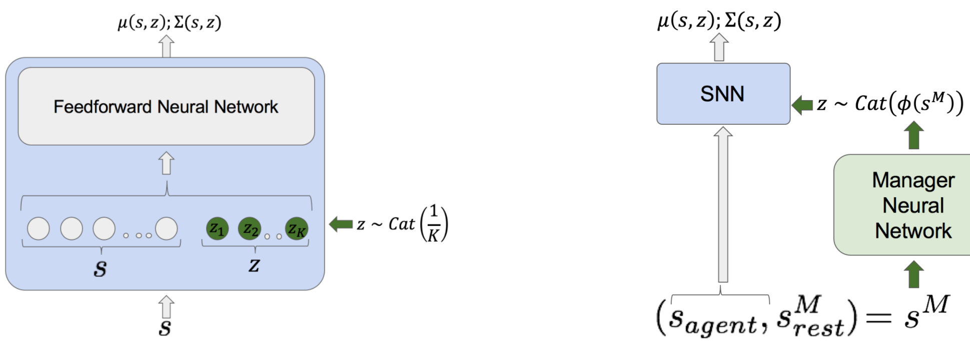
As a precaution before HASSLE action explosion, Moerman (2009) proposed an alternative version called *Hierarchical Assignment of Behaviors by Self-organizing* (HABS). His aim was to develop skills that are defined relative to the abstract state space, analogous to the primitive actions in lower state space. As summarized by the author: "A classification algorithm is used to map the transitions between high level states to characteristic skills. These skills are then added directly to the high-level policy as high-level actions. By this 'short circuiting' of the HASSLE algorithm, both the capacities and the use of states (subgoals) as actions can be avoided. The Q-values of the high-level policy now directly determine which subpolicy is suited for which transition, because the Q-values give the value for a high level action (skill) in a high level state."

HABS algorithm reverses the chain that links transitions between subgoals and subpolicies. In HASSLE, this linkage was determined by the transitions: each policy was specialized to one or more transitions. Moreover, several subpolicies were allowed to specialize in the same transition, resulting in the *many-to-many* relationship. Conversely, HABS does not use the unique transitions themselves, but rather the *direction* they represent in the high-level state space. The transitions are then classified into several groups of similarly-heading transitions, while each of these groups corresponds to a specific subpolicy, as shown in figure 3.8b. This way, the linkage is logically determined by the subpolicies, and a simpler *many-to-one* relationship is achieved as well.

3.3.7 Stochastic NN skill training

Florensa et al. (2017) proposed a new framework for HRL learning, with quite an unintuitive name of *Stochastic Neural Networks for HRL*, which we will refer to as SNN training. The usage of stochastic networks may be seen merely as a tool, while the approach has much more to offer from the perspective of HRL, or even general RL. Namely, SNN learning utilizes the architecture to provide highly increased sample efficiency and offers a new perspective on skill sharing among different tasks.

At the core sits a relatively simple, two-layered architecture with a high-level policy (referred to as the manager) and the fixed number of skill policies. However, the authors consider not only one, but a set of different core MDPs $\{\mathcal{M}_1, \mathcal{M}_2, \ldots\}$. The manager policy is trained separately for each of these problems, but skill policies are shared among all of them. To the best of our knowledge, such level of skill sharing between different core tasks has not been considered before. As an example, we can introduce a legged robot with different high-level tasks, such as solving a simple maze or gathering food through the space, while the same skills can be used in all of them – individual locomotion policies enabling the robot



(a) (b)

Figure 3.10: The architecture of SNN training: (a) Integration of latent code z into input of skill-learning network. (b) Flow of information in the SNN model. (Florensa et al., 2017)

to walk in the first place. The skills are pre-trained in a pre-training environment, where the agent can learn a span of skills useful for all downstream tasks. Such an environment could be an unbounded plane for a robot to learn to walk at.

A factorization of a state space is used as proposed by Konidaris and Barto (2007): for each MDP $\mathcal{M}_i$, it is assumed that the state space $S^{\mathcal{M}_i}$ can be factored into two components, S_{agent} and $S^{\mathcal{M}_i}_{rest}$, which only weakly interact with each other. The S_{agent} should be the same for all considered core-MDPs. Intuitively, considering our example of robot who faces a collection of tasks, the dynamics of the robot are shared across tasks and are covered by S_{agent}, but there may be other components in a task-specific state space such as obstacles or coins, which will be denoted by $S^{\mathcal{M}_i}_{rest}$.

The skill pre-learning process represents the main contribution of SNN learning. We first need to note that the skills are learned in a simplified pre-training environment mentioned above, one that features only the state space S_{agent}. The reward signal is not set individually for each skill, as that would require precise specification about what each skill should entail. Instead, a general *proxy reward* is used to guide learning of all skills. As the authors explain, "the design of the proxy reward should encourage the existence of locally optimal solutions, which will correspond to different skills the agent should learn. For a mobile robot, this reward can be as simple as proportional to the magnitude of the speed of the robot, without constraining the direction of movement." In addition to using a single reward signal, SNN learning also leverages the recent advantages in neural networks, and uses a single network to train all skills simultaneously. This opposes previous methods based on NN, which always used separate network for each skill's policy. Single-network architecture dramatically reduces the samples needed to train the skills – a near-constant number of samples instead of linear scaling.

The single skill-learning network receives an input that consists of two integrated components: the S_{agent}, and one-hot encoded latent variable z denoting which skill is being

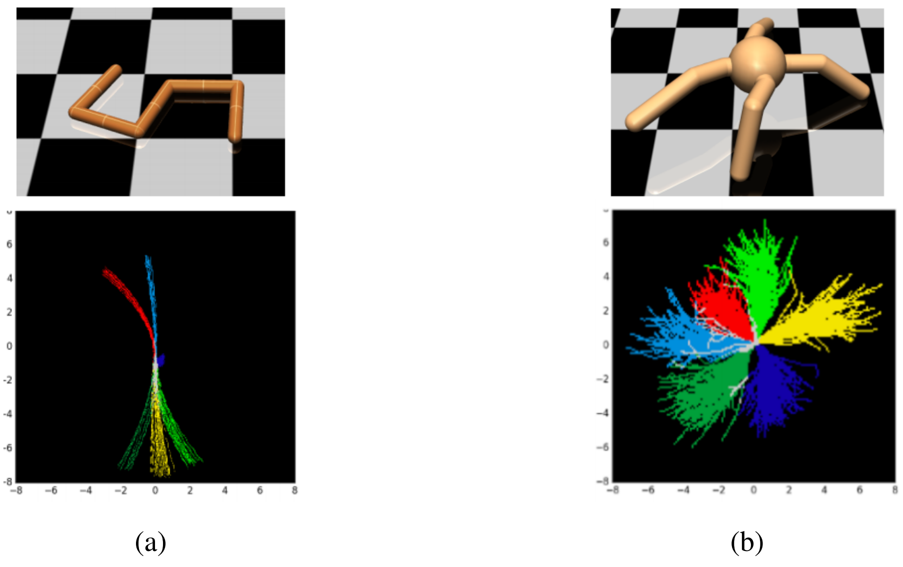


Figure 3.11: The model of a snake (a) and an ant (b), and their learned skills. Each color represents a region covered by different skill. (Florensa et al., 2017)

learned/used. Two integration schemata have been proposed – concatenation (shown in figure 3.10a) and bilinear[6]. The high-level manager network is responsible for selection of skills in specific core task: it receives full state $s \in S_{agent} \times S_{rest}^{\mathcal{M}}$, and outputs the latent code z. The complete architecture is described in figure 3.10b.

To ensure that acquired skills do not converge into one, SNN approach introduces an additional component of skill-learning reward signal, which effectively encourages the individual skills to be distinct. The idea is based on the mutual information (MI) theory: if a skill is sufficiently distinctive from others, it covers its own region of a state space. Thus, if we can uniquely identify the skill being used given the region we are in, it means the distinction is successful and the skill is rewarded more. Formally, maximizing MI is equivalent to minimizing the conditional entropy $H(Z|S)$. This yields the final shape of reward signal to be:

$$r_t \leftarrow r_t + \alpha_H . \log p(z|s_t) \tag{3.3}$$

where α_H is a hyperparameter denoting how much we encourage the MI reward component. For the estimation of $p(z|s_t)$ authors used a count-based method in the state space, which had to be discretized to enable greater-than-one visitation counts.

As seen in figure 3.11, MI reward component successfully encourages learning of different skills. The movement skills of a snake model are all highly concentrated around forward-backward motion – this directly results from the physiology of the robot. Several different core tasks were subsequently learned using fixed set of these pre-trained skills – for more details please refer to Florensa et al. (2017).

[6]The bilinear integration essentially trains a separate *first* layer for each skill, while all further layers are shared.

3.3.8 HIRO

In previous section we saw a model with one shared network for all skills. The *Hierarchical reinforcement learning with off-policy correction* (HIRO) by Nachum et al. (2018) takes it one step further by having a single skill policy which is parametrised by a goal state.

HIRO is a method based on UMDP (see section 1.2.4) with a two-layered hierarchical architecture. The high-level policy π^H operates on a state-space S from the original core-MDP, and its actions represent goals for the low-level policy: $a_t^H = g_t^L$. There is a single low-level policy π^L whose behavior is parametrised by the given goal. It operates on a state-space $S^L = S \times G$ and produces atomic actions from the core-MDP's action-space A. The behavior of such UMDP-based hierarchical architecture, though adapted from a different paper, is shown in figure 3.12.

When the agent is performing a rollout, the high-level policy π^H observes the current state s_t and produces a goal g_t^L. The low-level policy, observing both s_t and desired g_t^L, then produces an atomic action a_t, trying to reach the goal. The goal stays fixed for c time-steps, during which π^L continues to receive observations $\langle s_{t+i}, g_t^L \rangle$ and producing actions a_{t+i} to reach g_t^L. After those c time-steps, π^H chooses a new goal g_{t+c}^L, and the cycle repeats. The high-level reward signal r_t^H is equal to the reward from core-MDP, while the low-level reward r_{t+i}^L is inversely proportional to the distance of the current state s_{t+i} from the desired goal g_t^L.[7]

This novel architecture produces a hierarchy in which the low-level policy encapsulates the complicated task of robot's locomotion, while the high-level policy can focus on the core-MDP task. Both policies can learn at the same time, enabling better sample-efficiency of the method. However, the simultaneous training of both policies introduces an instability to the training of π^H, as its transitions are based on ever-changing π^L.

To overcome this drawback, the authors employ an off-policy corrections to the training of a high-level policy π^H. The high-level transitions, collected during the rollout, are represented by $e_t^H = \left\langle s_t, g_t^L, \sum r_{t+i-1}, s_{t+c} \right\rangle$. However, the transitions obtained using an older version of a low-level policy do not accurately reflect the actions (and therefore the resulting state s_{t+c}) that would occur if the same goal g_t^L was used with the current low-level policy. Thus, the goal g_t^L from e_t^H has to be changed in order to agree with the changed behavior of π^L.

In HIRO, the transitions in the experience replay buffer of π^H are periodically updated to contain a modified goal (i.e. high-level action) $\tilde{g}_t^L$ instead of original g_t^L. The $\tilde{g}_t^L$ is chosen to maximise the probability of observed low-level actions $\pi^L\left(a_{t+i-1} \middle| \langle s_{t+i-1}, \tilde{g}_t^L \rangle\right)$ for $i \in \{1, \dots, c\}$. This probability can be computed exactly, but c evaluations of π^L must be executed to do so.

The remaining question is how to choose the $\tilde{g}_t^L$ that maximises over the given probability

[7]We abstracted from two aspects of the original approach: first, the goal g_t^L is not an absolute position, but is relative to the current state, i.e. π^L is trying to achieve $s_{t+c-1} = s_t + g_t^L$. Second, π^H produces a goal in *each* intermediate time-step $[t, \dots, t + c - 1]$, but it is constructed in a way that $s_t + g_t^L = s_{t+i} + g_{t+i}^L$.

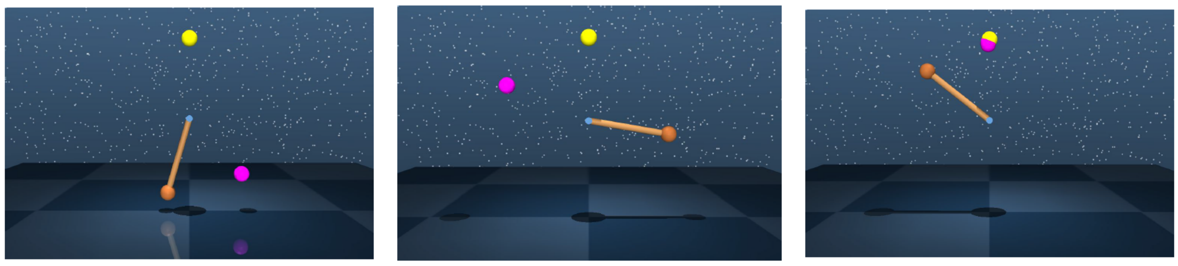


Figure 3.12: Hierarchical architecture of HIRO (Nachum et al., 2018) and HAC (Levy et al., 2018) demonstrated on the inverted-pendulum task. The goal of core-UMDP is to reach the yellow dot. The high-level policy π^H generates a subgoal g_t^L represented by pink dot. Low-level policy π^L then tries to match the current state s_{t+i} with goal g_t^L. (Levy et al., 2018)

distribution. The authors solve this by sampling 10 candidates for $\tilde{g}_t^L$, computing their probabilities, and choosing the best one. Resulting $\tilde{g}_t^L$ is then placed instead of original g_t^L in the transition e_t^H.

Using this elaborate scheme, HIRO is able to overcome the problem of training with unstable distributions. Though, computational sacrifices had to be done to achieve this – $10c$ evaluations of π^L has to be executed for *each* modified transition e_t^H on regular basis.

3.3.9 HRL with Hindsight

The *Hindsight Experience Replay* (HER) by Andrychowicz et al. (2017) leverages the idea of experience replay buffer and modifies it to achieve faster convergence in sparse-reward environments – so far only for flat-RL architecture. Later, Levy et al. (2018) adapted this approach also for HRL. The key idea of both papers is to insert fabricated experience transitions into the replay buffer, such that they always terminate with a non-zero reward. Similarly to HIRO, both hindsight methods rely on being used in UMDP.

Within sparse-reward environments, the agent often struggles to start the learning process. Until it reaches a rewarded state for the first time, it randomly explores the states, always receiving zero reward, hence not updating its parameters at all. As the authors explained by an example: “Imagine that you are learning how to play hockey and are trying to shoot a puck into a net. You hit the puck but it misses the net on the right side. The conclusion drawn by a standard RL algorithm in such a situation would be that the performed sequence of actions does not lead to a successful shot, and little (if anything) would be learned. It is, however, possible to draw another conclusion, namely that this sequence of actions would be successful if the net had been placed further to the right”. To tackle this problem, HER method introduces *hindsight* transitions that are inserted into the replay buffer. These transitions are fabricated, but consistent with the learning process, and they always form a trajectory with a non-zero reward. In the transitions, they modify the *goal* to match the actual state the agent ended up in – i.e. in hindsight, we move the net to the right, which results in us scoring a goal and getting a reward.

During the training, the agent collects transitions[8] $e_t = \langle\langle s_t, g\rangle, a_t, r_t, \langle s_{t+1}, g\rangle\rangle$, where reward is computed by traditional reward function $r_t = R(\langle s_t, g\rangle, a_t)$. In addition to these real transitions, the hindsight transitions are also generated: $\tilde{e}_t = \langle\langle s_t, \tilde{g}\rangle, a_t, \tilde{r}_t, \langle s_{t+1}, \tilde{g}\rangle\rangle$, where $\tilde{g}_t$ is a *hindsight goal* and $\tilde{r}_t = R(\langle s_t, \tilde{g}\rangle, a_t)$. The hindsight goal $\tilde{g}_t$ is always chosen in a way so that $\tilde{r}_t > 0$, in the simplest case we use final state of the trajectory: $\tilde{g}_t = s_T$. Both kinds of transitions are then added to the replay buffer.

The addition of hindsight transitions ensures that the agent always has data with non-zero reward in its replay buffer, and so the parameters can always be modified and learning progresses.

Hierarchic Actor-Critic

Levy et al. (2018) took the idea of HER and implemented it into a HIRO-inspired architecture, creating a method named *Hierarchic Actor-Critic* (HAC), also depicted in figure 3.12. It is based on a multi-level hierarchy, with one UMDP agent at each level, but for the sake of simplicity we will describe a two-layered version with a high-level policy π^H and a low-level policy π^L.

To clarify the explanation, we first revise the nomenclature and inputs/outputs of both controllers. On the higher level, policy π^H receives a state-goal tuple $\langle s_t, g^H\rangle$, in which the goal g^H stays fixed during whole episode. It then produces an action, which will serve as a goal for a low-level policy: $a^H_t = g^L_t$. This goal is fixed for c time-steps, during which π^L operates, and the next input for π^H comes at time $t + c$. During the execution of the low-level policy π^L, at time $t + i$ it receives a state-goal tuple $\langle s_{t+i}, g^L_t\rangle$, and produces an atomic action $a^L_{t+i} \in A$. Similarly to HIRO, the high-level reward signal is a sum of rewards from core-MDP: $r^H_t = \sum_i^c r_{t+i-1}$. The low-level reward r^L_{t+i} is again inversely proportional to the distance of the current state s_{t+i} from the desired goal g^L_t.

The Hierarchic Actor-Critic employs two types of hindsight transitions. **Hindsight goal transitions** are a direct implementation of HER approach. The goal in each transition is replaced by the state the agent actually ended up in. This happens on both levels, i.e. the following transitions are generated:

$$\begin{aligned} \tilde{e}^H_t &= \Big\langle\langle s_t, s_T\rangle, g^L_t, \tilde{r}^H_t, \langle s_{t+c}, s_T\rangle\Big\rangle \\ \tilde{e}^L_{t+i} &= \Big\langle\langle s_{t+i}, s_{t+c}\rangle, a_t, \tilde{r}^L_{t+i}, \langle s_{t+i+1}, s_{t+c}\rangle\Big\rangle \end{aligned} \tag{3.4}$$

[8]Note that since HER is operating in flat-RL, the goal g stays fixed during whole episode.

As in other HRL architectures that train both levels at the same time, HAC also needs to tackle the problem of training π^H over ever-changing distribution caused by modifying π^L. While HIRO employed a computationally-expensive correction to modify g_t^L to simulate *current state* of π^L, HAC takes this idea further by simulating the *optimal* π^L using a second type of hindsight transitions.

The purpose of **hindsight action transitions** is to simulate a transition function that uses an optimal lower level policy hierarchy. It does so by replacing the high-level action g_t^L in e_t^H. We know that the optimal π^L would take us from s_t to s_{t+c} if it had been given s_{t+c} as a goal. And so, to simulate this optimal behavior, $\tilde{e}_t^H$ should contain s_{t+c} as the goal chosen for a lower level, i.e. the high-level action. This further changes the $\tilde{e}_t^H$ from equation 3.4 to:

$$\tilde{e}_t^H = \Big\langle \langle s_t, s_T \rangle, s_{t+c}, \tilde{r}_t^H, \langle s_{t+c}, s_T \rangle \Big\rangle \tag{3.5}$$

Similarly to HER, both the original transitions and *hindsight goal transitions* are placed into the replay buffer. However, the *hindsight action transition* – i.e. substitution of g_t^L for s_{t+c} is always performed, and no transitions with original action are saved in the high-level replay buffer.

The elaborate architecture of HAC enabled it to significantly speed up the training in multiple robotic environments. However, due to its reliance on UMDP, it is mostly useful for tasks that require the agent to reach a specific position – generalising to broader range of environments might not be feasible. Moreover, a rather complicated machinery had to be implemented to overcome the problem when unreachable goals g_t^L are selected by the high-level agent, which is beyond the scope of this survey. Levy et al. (2018) provides comprehensive explanation.

3.3.10 Hierarchical Proximal Policy Optimization

By combining several previous approaches and enriching them by own theoretical findings, Li et al. (2019) implemented an algorithm called *Hierarchical Proximal Policy Optimization* (HiPPO). Their key contribution lays in formulating the policy gradient jointly for both high-level *and* low-level policies at the same time, and demonstrating that even approximate version of such a gradient yields an effective training method.

The architecture of HiPPO stems from the one of SNN (Florensa et al., 2017): a high-level policy $\pi^H(z_t|s_t)$ chooses a latent variable z_t every c time-steps. The integer-valued z_t essentially denotes which skill π^H chose to use, where $z_t \in Z = \{1, \ldots, n\}$. For the next c time-steps, the low-level network performs actions according to chosen z_t, i.e. $\pi^L(a_{t+i}|s_{t+i}, z_t)$. This cycle repeats until the end of a rollout at time T is reached. In HiPPO architecture, the authors chose to employ a random length c of skill execution, acting as an external variable not accessible to either policy. Note that this hierarchical policy is more restrictive than others like the Options framework, where the time-commitment is also decided by the policy.

From the view of underlying core-MDP, we can observe a whole trajectory of states and actions, denoted by $\tau = \langle s_0, a_0, \dots, s_T, a_T \rangle$. For the traditional flat-RL scenario, the gradient of the objective function $J(\theta)$, which we use to optimize all policy parameters θ, can be expressed as:

$$\begin{gathered}\nabla_\theta J(\theta) = E_\tau \left[\log p(\tau)\ G(\tau)\right] \\ \text{where } p(\tau) = p(s_0) \prod_{t=0}^{T} p(s_{t+1}|s_t, a_t) \prod_{t=0}^{T} \pi_\theta(a_t|s_t)\end{gathered} \tag{3.6}$$

The probability $p(\tau)$ of observing a trajectory τ is a key term in evaluating the gradient. We can extract the product inside logarithm into sum outside of the logarithm, and sample the state probabilities $p(s_0)$ and $p(s_{t+1}|s_t, a_t)$ from the environment. This yields a familiar term:

$$\nabla_\theta J(\theta) \approx \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)\ G(\tau) \tag{3.7}$$

Moving from flat-RL to HRL, the situation gets more complicated, as $p(\tau)$ now depends on both π^H and π^L. With π^H choosing a latent action at time-steps $t = kc, k \in \{0, \dots, T/c\}$, and π^L choosing an atomic action at each time-step $t \in \{0, \dots, T\}$, the overall probability of observing a trajectory τ can be expressed as:

$$p(\tau) = p(s_0) \prod_{t=0}^{T} p(s_{t+1}|s_t, a_t) \prod_{k=0}^{T/c} \left(\sum_{z \in Z} \pi^H(z|s_{kc}) \prod_{t=kc}^{(k+1)c-1} \pi^L(a_t|s_t, z) \right) \tag{3.8}$$

The mixture action distribution, which presents itself as an additional summation over skills, prevents additive factorization when taking the logarithm, as we performed from equation 3.6 to 3.7. However, as a key contribution of HiPPO, we can sufficiently approximate this probability, using only a mildly restrictive assumption.

In a well-trained HRL architecture, we can expect skills to be well specialised. That means that in every state s_t, we can expect $\pi^H(\cdot|s_t)$ to have a high value for one specific z_t chosen by the policy, while all other $z \in Z \setminus \{z_t\}$ will have low values. These low values imply that contribution of all $z \in Z \setminus \{z_t\}$ to the sum in equation 3.8 will be minimal, and we can marginalize them out. With the summation out of the equation for $p(\tau)$, we can now easily formulate a hierarchical equivalent for equation 3.7:

$$\nabla_\theta J(\theta) \approx \left(\sum_{k=0}^{T/c} \nabla_\theta \log \pi_\theta^H(z_{kc}|s_{kc}) \ + \ \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta^L(a_t|s_t, z_{kc}) \right) G(\tau) \tag{3.9}$$

The authors proved that this approximation generates an error $O\left(|Z|T\epsilon^{c-1}\right)$, where ϵ is the upper bound for non-chosen skills: $\forall z \in Z \setminus \{z_t\} : \pi^H(z|s_t) < \epsilon$. This low error was further backed by empirical data from the experiments.

With the gradient of policy parameters expressed jointly for both high- and low-level policy, HiPPO can train both levels at the same time. To do so, the authors incorporated their gradient rule into *Proximal Policy Optimization* (Schulman et al., 2017) - an approximate, more compute-efficient alternative of TRPO. Additionally, separate baselines for each level were used, which further stabilised the training process. With a complete algorithm, authors proved the abilities of HiPPO on numerous tasks, surpassing previous approaches such as SNN (Florensa et al., 2017) or HIRO (Levy et al., 2018).

Chapter 4

Adaptive Skill Acquisition Framework

One of the most difficult tasks in hierarchical reinforcement learning is undoubtedly constructing a skill set that would be helpful in solving the core task. However, finding the optimal skills is practically impossible. Designing the skills by hand introduces great architectural bias, while automated skill training in pre-training phase may not reflect the task's peculiarities. These flaws can, in turn, limit the agent's desired abilities. We would like to contribute to solution of this uneasy task by introducing an *Adaptive Skill Acquisition* framework, or ASA for short.

The principle goal of ASA is to discover imperfections within the hierarchy of policies, and address them by training a new skill. The novelty of this approach lies in additional augmentation of existing pre-trained skills according to the real needs of the agent, all in the midst of training the core task. ASA can observe the high-level controller during its training and identify skills that it lacks to successfully learn the task. These missing skills are subsequently trained and integrated into the hierarchy, enabling better performance of the overall architecture.

4.1 Motivation

Throughout the research field, we can observe several common streams for building up the HRL hierarchy. In the simplest scenario, Parr and Russell (1998) or Sutton et al. (1999) created the set of skills by hand-crafting their behaviors, and training them manually as a series of independent RL agents. The agents were then placed into a hierarchical structure (tree or directed graph), which was also fully specified by the engineer.

When skills are identified autonomously, majority of previous approaches employ a pre-training phase (McGovern and Barto, 2001; Menache et al., 2002; McGovern and Barto, 2002; Goel and Huber, 2003; Konidaris and Barto, 2009; Florensa et al., 2017, and others). During this phase, the agent typically explores the environment without a reward or using a surrogate one, learning only the skills. Once the pre-training is done, the hierarchy of skill policies is fixed, and the top-level agent is trained.

A class of algorithms based on HASSLE trains the skills based on abstract state space (Bakker and Schmidhuber, 2004; Moerman, 2009; Dillinger, 2019). Although the skills are trained jointly with the core task, their usage is limited by the suitability of the high-level abstraction, created by the engineer. The crucial part of specifying this abstraction is again performed upfront.

We can observe a common structure in all of these implementations: *first* create a hierarchy of useful, yet not necessarily optimal skills, *then* train the high-level controller atop of fixed hierarchy. This pipeline, however, can hit problems stemming from the principle of *optimality under given hierarchy* (Sutton et al., 1999). It was shown that having the skill set fixed before training the higher level of a hierarchy can considerably limit the final performance (Levy et al., 2018).

As an example, we can imagine a walking robot tasked to navigate through the maze, and provide it with two skills: 'walk forward' and 'turn left', but no 'turn right' skill. The high-level controller can still learn a strategy to solve the maze with given skills, yet it will be clearly suboptimal for cases when the robot should have turned right.

We can distinguish between two main imperfections that can occur within the skill set. Firstly, a skill can be malformed – either through specification of wrong behavior, or due to failed training of the skill. Amending the aforementioned example, we can provide the robot with a 'sit down' skill, which is clearly useless for the maze-solving task. Malformed skill does not help the high-level agent to reach the goal. On the other hand, it does not prevent from doing so either. As the training of high-level agent progresses, it can easily learn to ignore the useless skills, as we also show in section 5.4. Presence of such a skill hence does not impact the overall performance too much.

Secondly, a skill can be missing from the hierarchy altogether. Such situations are common if the pre-training phase was terminated too early, or if it failed to explore a certain subspace. This problem cannot be overcome by traditional RL methods and predisposes the agent to find a suboptimal solution. After the pre-training phase has ended, leaving a skill out, majority of HRL approaches can do little to nothing to resolve this situation.

To address this problem, our Adaptive Skill Acquisition framework aims to identify whether a useful skill was left out from the initial hierarchical composition. If so, ASA is able to specify the needed behavior, train a policy to perform it, and integrate the new skill into the existing hierarchy. This process occurs even after the pre-training phase has finished, aiming for the challenges that the high-level agent currently struggles with.

4.2 High-level description

As previously mentioned, the Adaptive Skill Acquisition is a method for *augmenting* an existing hierarchy with a new skill. As such, its goal is not to train the complete hierarchy of agents. Thus, ASA is not a closed, self-contained architecture, but rather a universal *pluggable component* that can be used on top of almost any HRL algorithm, enriching it by new functionality. It has been designed to support a wide variety of existing algorithms, or those yet to come.

4.2.1 Architectural capabilities

The set of hierarchical architectures ASA can be deployed on spans those from almost all relevant research – a HRL system consisting of two layers.[1] The high-level policy π^H can operate over the original state space S, or compacted $\hat{S}$. Its action space consists of passing the control to one of the skills, i.e. $A^H = \{\pi_1^L, \dots, \pi_n^L\}$. The lower level contains n pre-trained skills which operate on the original state and action spaces S and A. When the high-level controller chooses its action $a_t^H = \pi_i^L$, the chosen skill policy π_i^L is executed until a termination criterion is met.

In order to maximise compatibility, ASA supports a generous class of stopping criteria by accepting a *skill-stopping function* in the following form:

$$f_i^{\text{stop}}\left(s_{t-c}, a_{t-c}^L, \dots, s_t, a_t^L\right) \in \{true, false\} \tag{4.1}$$

By accepting the whole sequence of states and actions from the skill's execution, f_i^{stop} can be specialised to almost any function – from fixed- or random-length skills, through state-based probabilistic functions mimicking skill's target regions, to surrogate-reward based conditions used in some algorithms. A separate skill-stopping function f_i^{stop} can be specified for each skill π_i^L, which further broadens the applicability. These loose architectural constraints enable the usage of ASA in a wide variety of algorithms.

4.2.2 Main logical components

The method by which ASA adds a new skill consists of several subsequent processes. These can be separated into three key logical components:

1) Identification of a missing skill

During the learning of the core task, the agent must be able to recognize the need for another skill. By means of self-observation, the agent will try to identify potentially sub-optimal

[1]ASA can be deployed on multi-layered architectures as well, as discussed in chapter 6.

sequences of high-level actions (skill invocations) that tend to occur significantly more often. Such sequences hint at a regularity in the core-MDP that was not discovered by the original skill building process, and is only modelled using the reoccurring sequence of pre-defined skills. This sequence will serve as a candidate for training a new skill, capable of solving the subtask in a more efficient way.

2) Training of the new skill

The reinforcement learning process is guided solely by the reward signal. Therefore, if we want to teach a new skill to perform certain task, we need to create a reward function R' that will lead it. Using the newly constructed reward signal, we formulate a sub-MDP that represents the identified skill. After the complete MDP specification, the problem simplifies into a standard task of flat reinforcement learning.

3) Integration of the new skill

After the new skill is ready, we can integrate it into the existing HRL architecture. This step is the only one that is inherently approach-specific, i.e. it might need to be adjusted when ASA will be used in different architectures. Many algorithms, however, share the usage of a neural-network based policy at the top level of hierarchy. Integration of the new skill thus represents adding a new output unit to a partially trained policy network.

These three key logical steps are described in details in sections 4.3 – 4.5.

4.2.3 Properties of ASA

As the research in HRL progresses, individual approaches tend to focus on more difficult variants of MDPs. With the broader diversity of these difficult problems, some methods focus only on a single aspect to be solved, e.g. continuity of state space. Contrary to such cases, we made an effort to support almost any subclass of MDPs. The individual components of ASA were designed always with more general use-case in mind, so that the limitations in applicability would be shrunk to minimal.

Continuous spaces

First and foremost comes the differentiation between discrete and continuous spaces. Majority of research prior to 2009 was focused solely on the discrete MDPs, being easier to control and solve. Even afterwards, some algorithms considered the continuous *state* space, but required a discrete *action* space in order to work properly (Metzen and Kirchner, 2013).

Recent approaches eventually moved towards supporting continuity in both state- and action-spaces. We continue in this trend, with ASA being capable of operating atop either discrete or continuous environments, with no implementation or configuration changes needed.

Reward sparsity

As discussed in section 3.1, reward sparsity is one of the key motivations for hierarchization in RL. We also thoroughly focus on working with sparse-reward environments which, though being much harder to solve, offer the greater research potential. Specifically, ASA considers this aspect on two levels: Firstly, the core task that agent is trying to solve is assumed to have sparse rewards, as is common in HRL. Secondly, ASA employs sparse rewards internally when training a new skill. By doing so, we are able to automatically construct a robust surrogate reward signal which is agnostic to the core task.

Observation considerations

Partial observation of the state space is a common concept, especially in robotic-based environments. In such cases, the agent only ingests the data from a camera or other sensors, which typically do not reflect the state of the entire space. Since this became a new standard, ASA is also applicable in partially observable MDPs.

Other approaches, on the other hand, enrich the state space even further by adding a desired goal state. This results in an UMDP-based methods such as Nachum et al. (2018) or Levy et al. (2018). Our approach does not rely on any kind of additional information such as these. Theoretically, ASA can be deployed even on UMDP environments without further modifications. However, ASA *does* rely on policy π^H choosing from n discrete skills. The aforementioned methods changed this mechanism to π^H choosing a goal vector for a single UMDP-based skill, hence they represent one of the few exceptions ASA cannot be deployed on.

Model usage

The model-based RL methods assume that a complete knowledge of underlying MDP is known to the agent, especially the transition distribution and the reward function. It can subsequently be used to compute the optimal solution. ASA falls into the more general category of model-free algorithms, which do not access these properties directly, and rather do so by sampling the environment.

A single limiting factor that ASA needs is the option to initialize the agent in a given state. This can usually be easily implemented in most of the frequently used environments. In case it cannot be done, a learned approximation such as in Gu et al. (2016) can be used.

Types of trained skills

The majority of the current HRL methods use their own approaches to train the skills. As outlined in section 3.3.4, these approaches fall into two categories. The *goal-based* skills are made to reach a specific state, which was identified as one with greater importance, such as doorway. This type of skills is undoubtedly more popular (McGovern and Barto, 2001; Goel and Huber, 2003; Bakker and Schmidhuber, 2004; Konidaris and Barto, 2009; Metzen and Kirchner, 2013; Nachum et al., 2018; Levy et al., 2018).

On the other hand, *behavioral* skills are crafted to perform a specific behavior which can be useful in almost any state of the environment. These skills typically do not aim to move to a specific state, but rather to move in a specific direction within the state space. A typical behavioral skill is a walking of a legged robot (Florensa et al., 2017; Li et al., 2019).

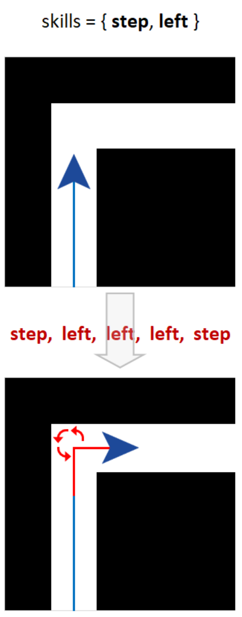
To the best of our knowledge, there has not yet been a published algorithm that would be able to train *both* goal-based and behavioral skills. The distinctive feature of ASA is the ability to train either one of them. Furthermore, it can do so without the need of changing any parameters, or even specifying the type upfront. This ability is demonstrated in section 5.3, when trained on two fundamentally different tasks.

4.3 Identification of a missing skill

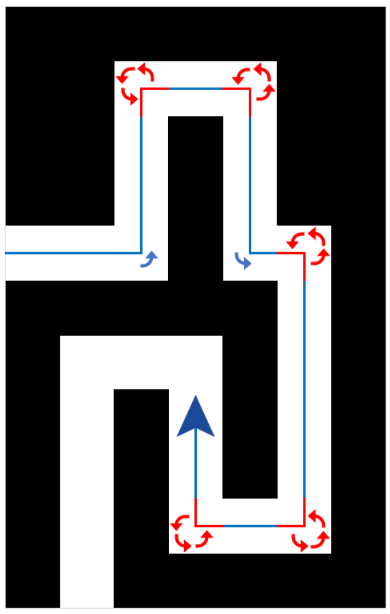
The first step in the pipeline of Adaptive Skill Acquisition is to recognize that a useful skill is missing, and to identify what such a skill should do. To achieve this, we use a technique based on self-observation of the high-level agent.

In order to detect a missing skill behavior, we try to identify potentially sub-optimal sequences of high-level actions (skill invocations) that tend to occur significantly more often. Such sequences hint at a regularity in the core-MDP that was not discovered by the original skill building process, and is only modelled using the reoccurring sequence of pre-defined skills. Training a skill specialised to handle such regularity may result in a more efficient behavior to solve it, which, in turn, helps the overall performance of the system.

An example of this principle is displayed in figure 4.1, in which we continue with our example of a robot in a maze with ‘step forward’ and ‘turn left’ skills, but no ‘turn right’. We can see in subfigure (a) that the agent is able to train a behavior that would effectively turn it rightwards – i.e. performing a sequence of skills $[step, left, left, left, step]$. If this behavior is used extensively, we should be able to detect that this sequence is executed significantly more often then any other 5-step sequence (subfigure b). Using this detected inefficient sequence, we can target it and train a dedicated ‘turn right’ skill, one that would perform the desired behavior much more efficiently.



(a)



(b)

Figure 4.1: HRL agent in a maze with insufficient skill set: (a) A sequence of skills is trained to perform a right turn; (b) Inefficient sequence is repeated many times, significantly more than any other.

Having stated the descriptive definition, we can now define the problem more formally. Let us denote δ to be an arbitrarily long sequence of high-level actions: $\delta = [a_t^H, a_{t+1}^H, \ldots, a_{t+m-1}^H]$, where m represents its length. Hereinafter, we consider t to denote the timeframe of the high-level agent, in order to abstract the potentially uneven durations of individual skills' executions. We are given a set of trajectories $\mathcal{T} = \{\tau_1, \tau_2, \ldots\}$, and our goal is to find a sequence δ^* which is the most frequent one, i.e. which has the highest probability of occurrence within the trajectories $\mathcal{T}$:

$$\delta^* = \arg\max_{\delta} \left(p(\delta \mid \mathcal{T})\right) \tag{4.2}$$

4.3.1 Storing frequent sequences

In order to identify the frequent sequences, we first need a way to store their counts, and periodically update these values. We implemented a custom data structure *path-trie* based on lexicographical trees that is able to incrementally add all sequences, and update counts of those that are already stored. Moreover, our path-trie is both faster and more space-efficient in comparison with traditional lookup hash-table that is usually used for this use-case.

The path-trie stores data for sequences with a variable length, limiting this length to a reasonable threshold d. Its core is a lexicographical tree over an alphabet $\Sigma = \{1, \ldots, n\}$, where n is the number of current skills. Each node of the tree hence has (at most) n child nodes – one for each skill. The path from a root to a specific node represents a unique

Algorithm 1: Inserting new skill sequences into path-trie. Each path-trie node contains children array, counter, and a set of additional data.

```
Data: partial trajectory of high-level agent $\tau_{\text{part}} = \left[\ldots, a^H_{t-1}, a^H_t\right]$
Result: all sequences $\delta$ ending with $a^H_t$ are stored in the path trie

procedure insert_new_sequences
    input: current high-level time $t$
begin
    node ← root of path-trie;
    for $i \leftarrow 0$ to $d-1$ do
        node ← node.children [$a^H_{t-i}$];
        node.counter ++;
        node.data ← data for sequence $\delta = \left[a^H_{t-i}, \ldots a^H_t\right]$;
        if $i + 1 \geq t$ then
            return;
        end
    end
end
```

sequence of skills. The difference between traditional lexicographical tree is that path-trie stores the sequences reversed. The root node hence represents the *end* of all sequences within the trie, while individual inner nodes denote their starts. This allows for a faster insertion to the structure.

The data is populated into the path-trie in a following way. Each time the high-level agent executes a new action a^H_t, the partial trajectory $\tau_{\text{part}} = [\ldots, a^H_{t-1}, a^H_t]$ is used to store all new skill sequences of variable lengths that end with a^H_t. Iterating over τ_{part} from the end, we traverse the path-trie, updating the counters of all nodes during the way – as shown in algorithm 1. The resulting path-trie with the counts in each node can be seen in figure 4.2 (only 5 levels of the tree are shown for clarity reasons).

Since we also need to collect additional information that will be used in subsequent steps of ASA, each node stores the following data about the sequence:

- the sequence itself[2]: $\delta = \left[a^H_1, \ldots, a^H_m\right]$
- number of occurrences: $C(\delta)$
- list of *start-states* in which each occurrence of δ started: $S_{\text{start}}(\delta) = \left[s^{(1)}_{\text{start}}, \ldots, s^{(C(\delta))}_{\text{start}}\right]$
- list of *end-states* in which each occurrence of δ ended up: $S_{\text{end}}(\delta) = \left[s^{(1)}_{\text{end}}, \ldots, s^{(C(\delta))}_{\text{end}}\right]$

[2]Note that we overload the notation and reindex the actions to start from a^H_1 rather than a^H_t, since each occurrence of δ starts at a different time t.

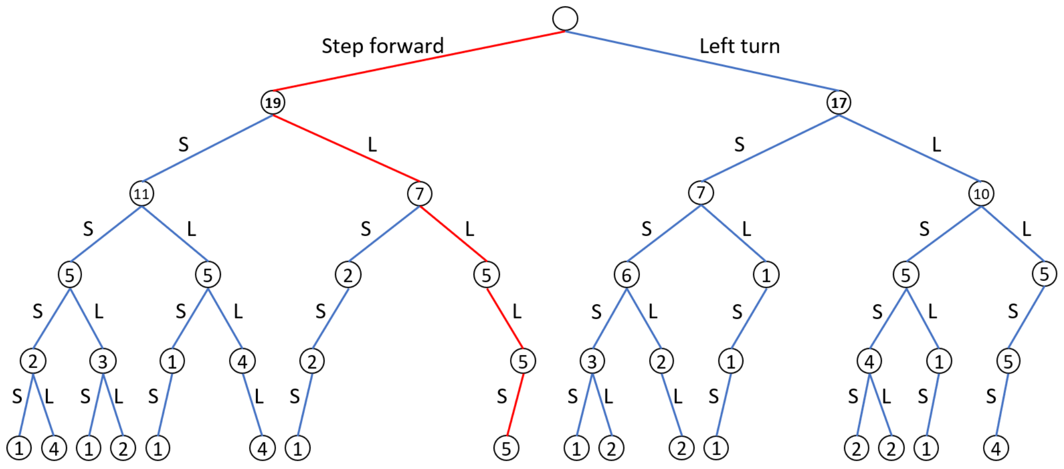


Figure 4.2: A path-trie constructed for a trajectory of the agent from figure 4.1b. Number in each node represent the counter – how many times have such sequence occurred. Highlighted is the frequently used sequence of skills with the highest count within its level.

Each start-state represents the state before the first action was taken, i.e. $s^{(i)}_{\text{start}} = s^{(i)}_1$. Analogously, the end-state is the state *after* last action was executed, i.e. $s^{(i)}_{\text{end}} = s^{(i)}_{m+1}$.

Using this algorithm for populating the path-trie, we are able to speed up the process after each high-level action to $O(d)$ time. If a simple hash-table was used, a higher $O(d^2)$ time would be required for the same operation, as the $O(d)$-long hash function would have to be computed for each sequence. The space consumption, though being less crucial, was also shrunk – the path-trie tops up at $O(n^d)$, whereas hash-table containing the same data would take $O(dn^d)$ space.

4.3.2 Choosing the best sequence

Once we have collected the desired information about all executed sequences of skills, we now have to choose the best one. The most frequently used sequence is the most likely one to execute a recurring, non-optimal behavior. Such a sequence will serve as a candidate for a new skill, and will be processed in further steps.

The direct naïve approach to pick the most frequent skill would be to rank the sequences using their counters from the path-trie. The top sequence of this list is the most common one. However, this approach comes with a caveat – the shorter sequences tend to naturally occur more often. As an example, we can take an untrained random discrete policy over 3 skills. When this policy is executed for 1000 steps, any two-steps-long sequence is expected to appear 111 times, three-steps-long sequence would be seen approximately 37 times, and four-steps-long sequence only 12 times. We can observe this behavior even in real data in figure 4.2 – the nodes in first or second levels (representing short sequences) have relatively high counts, while nodes deeper within the trie contain significantly lower values.

In our approach we account for this problem. Instead of asking 'how many times has sequence δ occurred' we ask a question '*how much more often then expected* do we see δ'. To answer this question, we introduce a *null-hypothesis count* $C_H(\delta)$ for any sequence δ. Given the set of trajectories $\mathcal{T} = \{\tau_1, \tau_2, \dots\}$ that was used to build the path-trie, we compute $C_H(\delta)$ as follows:

$$C_H(\delta) = \left(\sum_{\tau \in \mathcal{T}} |\tau| \ - \ |\mathcal{T}||\delta| \ + \ |\mathcal{T}| \right) \prod_{i=1}^{|\delta|} p(\delta_i) \tag{4.3}$$

This quantity represents the expected number of times δ would occur if a random 'null' policy was used. However, since we can compute the value of $C_H(\delta)$ analytically, no trajectories of the null policy have to be actually executed, and thus almost no computational complexity is added.

After acquiring $C_H(\delta)$, we use it to normalize the actual count of sequence $C(\delta)$ which is stored in the path-trie. The overall (frequency) score is defined as:

$$f(\delta) = \frac{C(\delta)}{C_H(\delta)} \tag{4.4}$$

Contrarily to the count $C(\delta)$, the f-score is no longer prone to prioritize short sequences. Every sequence with $f(\delta) > 1$ occurs more often than it would do under the null policy. Hence, the higher f-score the sequence gains, the more likely it is to execute the repeated non-optimal steps. By choosing the sequence with greatest f-score, we can finally identify possible candidate for a new skill.

4.4 Training of the new skill

In the previous section we described a way to identify a sequence of skills that serves as a candidate for a new skill to be trained. Now we need to use the collected information to construct a formal definition of the identified problem, i.e. create a sub-MDP that represents the skill we want to train. With the MDP created, we can employ standard RL methods to train a policy that solves it.

The desired sub-MDP is a tuple of six elements $\mathcal{M} = \langle S, A, P, p_0, R, \gamma \rangle$ which have to be specified. This process, however, needs to have two key properties. Firstly, it needs to be automated, so that the engineer does not need to interrupt the overall training, tweak the individual components of sub-MDP, and then initiate new skill training. Secondly, even more importantly, it needs to be task-agnostic. We cannot pose any assumptions about the nature of the skill, the behavior it is supposed to learn, or the core-MDP in which our system operates.

To satisfy these conditions, we build the sub-MDP which trains the agent to *perform the same transition from start-state to end-state as δ did*, but to perform it more efficiently. By using the atomic actions instead of a sequence of otherwise-focused skills, the newly trained agent can take a significantly more directed way towards reaching the desired end-state.

The easiest components of sub-MDP to be specified are S and A. Since all skills operate on the state and action spaces of the core-MDP, we can directly use its original S and A. Next, the discount factor γ, which in sparse-reward environments effectively regulates how eager the agent is to find a shortest path. The lower value γ has, the shorter paths the agent tends to choose, but on the other hand, it is also more inclined to failure due to lack of exploration. Since the engineer already specified γ value suitable for the environment, accounting for this trade-off, we reuse this value from the core-MDP as well. This default behavior can be overwritten by specifying a separate γ^L to be used for new skills, if desired.

Moving on to the dynamics of the system – initial-state probability distribution $p_0(s)$ and transition probability distribution $P(s'|s,a)$. As stated before, our goal is to train the agent to move from start-states of δ to its end-states. Thus, the most natural initial states of the new sub-MDP are the collected start-states. During the training of the new skill, we randomly choose a state $s^{(i)}_{\text{start}}$ from $S_{\text{start}}(\delta)$ and initialize the agent in it. Note that a state can repeat multiple times in $S_{\text{start}}(\delta)$, which makes it more probable to be chosen. The constructed probability p'_0 thus can be evaluated as:

$$p'_0(s) = \frac{1}{C(\delta)} \sum_{s^{(i)}_{\text{start}} \in S_{\text{start}}(\delta)} \left[s = s^{(i)}_{\text{start}} \right]_1 \tag{4.5}$$

where $[\_]_1$ denotes the indicator function. Finally, since the transition dynamics in which the agent is trained must be equal to the one it will be used in, we reuse the P from the original core-MDP.

The last, but certainly the most important element of MDP, is the reward function $R(s,a)$. We again directly follow our goal of reaching the δ's end-states. In each episode of the skill-training process we remember the specific start-state $s^{(i)}_{\text{start}}$ it was initialised in. The agent is then rewarded only upon reaching the corresponding end-state $s^{(i)}_{\text{end}}$. Note that the agent is not rewarded if it reaches any *other* end-state, as this would cause the agent to be falsely rewarded if the start-state of one instance of δ is close (or identical) to end-state of different one. We intentionally train a skill using such a sparse reward, as we want to avoid any engineered bias caused by more complicated reward shaping.

Reaching the exact end-state $s^{(i)}_{\text{end}}$, of course, is not possible in case of continuous state spaces, and an approximate criterion has to be applied. We were loosely inspired by the idea of *end-regions*[3] by Konidaris and Barto (2009), adjusting it to our needs. The end-region is a small area within a continuous state space, into which the agent aims to get – as an abstraction of end-state from discrete-space environments. While Konidaris and Barto (2009) had a whole set of positive and negative examples of states for an end-region, from which they created a classifier, we only have one example – the end-state $s^{(i)}_{\text{end}}$ corresponding to start-state $s^{(i)}_{\text{start}}$.

[3]Originally named 'target regions' – we will use 'end-regions' to provide clear analogy to our end-states.

To remedy this issue, we use a combination of a normalization scheme and distance-based condition.

First, we adapt a technique from deep learning called batch normalization (Ioffe and Szegedy, 2015). This technique normalizes each dimension across the samples in a minibatch to have zero mean and unit variance. In neural networks, it is used to minimize covariance shift during training. We use it to flatten the difference between relative distances within each dimension of the state space. After the normalization, we know that the distance between any two data points (states) averages to ≈ 1 in each dimension.

When we know the approximate scale of each state dimension, we construct the end-region as a hypercube centered around the end-state $s_{\text{end}}^{(i)}$, with a side of length 2ϵ. When the agent reaches this region, it is rewarded and episode is terminated. This method yields a reward function:

$$R'(s,a) = \left[\max_{d \in \{1..\dim(S)\}} \left| s_{\text{end}}^{(i)}[d] - s[d] \right| < \epsilon \right]_1 \tag{4.6}$$

where $\dim(S)$ is the dimensionality of the state space, and $s[d]$ represents d-th dimension of the vector s. This formulation in practice means that the agent is rewarded if each dimension of current state s is no more than ϵ away from the desired value. Since we already know that all dimensions have the same scale due to normalization, we can share a single ϵ value among all of them. Moreover, since we know this scale is based on unit variance, we can easily set the ϵ to a reasonable value. With the average distance being 1, setting $\epsilon = 0.1$ means that roughly 90% match between state s and desired $s_{\text{end}}^{(i)}$ in each dimension is needed for agent's success.

We deliberately chose this hypercube based criterion instead of a simpler Manhattan- or Euclidean-distance based – i.e. defining an end-region using $||s_{\text{end}}^{(i)} - s||_2 < \epsilon$. The reason is that such simpler solutions would face a problem in certain scenarios: Let us consider a high-dimensional state space which contains a binary dimension with significant importance. As an example, we can imagine a robot for delivering payload, with 20 sensory dimensions, and the binary dimension being a flag 'I am holding a payload'. If ϵ was set too low, the end-region might get too strict, as the ϵ-sized error has to distribute over a high number of dimensions. E.g. for $\epsilon = 0.1$, states with difference 0.02 in each dimension would be still considered as failure. On the other hand, if ϵ was too high, then we might accept even an error in the important binary flag. Considering the robotic example – if the agent perfected a desired body position, it would ignore whether it is still carrying the cargo, or had dropped it to the ground. However, these situations will not happen when using the hypercube based method. As each dimension is compared separately, neither gets too strict or too lax, since partial match is required in each of them. Also, the binary dimensions automatically fallback to exact comparison.

To summarize the sub-MDP creation, let us revise its individual components. The state and action spaces S and A, along with transitional probabilities P, are taken from the original core-MDP. The discount factor γ can be taken as well, or specified separately if desired. The initial-state probability p'_0 is realised by initializing the agent in one of start-states $S_{\text{start}}(\delta)$. Finally, the reward function R' is constructed to reward the agent upon reaching the corresponding end-region. The sub-MDP for the new skill is now complete, and is specified by $\mathcal{M}_{\text{new}} = \langle S, A, P, p'_0, R', \gamma \rangle$.

Now, as the MDP to be solved is fully specified, the problem of training the new skill simplifies to traditional flat RL. We train a new agent using standard techniques to solve $\mathcal{M}_{\text{new}}$, and thus perform the behavior of the identified missing skill. We employed the Trust-region policy optimization (Schulman et al., 2015) for this task, resulting in a policy $\pi^L_{n+1} = \pi^L_{\text{new}}$ for the new $(n + 1)$-th skill.

It can happen, however, that the new agent is not trained successfully. While the presence of malformed skill does not impact the long-term performance of the HRL system, as already discussed in section 4.1, we can ease up the high-level agent's life by not including such poorly trained skill into the skill set. Hence, if a skill is trained with less than 75% success rate (tunable parameter), the new skill policy is tossed away. Otherwise, it is passed down to the next step in ASA pipeline.

4.5 Integration of the new skill

At this point in the Adaptive Skill Acquisition process we have already identified what should a missing skill do, and trained a policy to perform it. The last step is to integrate the new skill into the overall HRL system. The high-level agent has to be modified so that it can choose the new skill as well, and sufficient exploration of this skill should be accomplished. Since the modification of the high-level agent is needed, this step is the only one which is inherently approach-specific, i.e. it might need to be adjusted when ASA will be used in different architectures. Nevertheless, we focused on the most common implementation and created several options that can be directly employed.

Most of the recent HRL approaches utilize one of two paradigms on the top level of hierarchy. The first branch uses algorithms based on generalised Q-learning, such as DQN, while the second one utilizes policy-optimization schemes stemming from the actor-critic architecture. In both cases, the policy or actor, responsible for choosing the actions, is typically implemented as a neural network. The architecture of this network differs widely, as it is also dependant on the specific environment and use-case – anything from a simple multi-layer perceptron to convolutional neural network or recurrent models such as LSTM. However, its output always directly represents the action to be executed, either as a real-valued vector of actions (in continuous-action environments), or one-hot-encoded index of an action (i.e. classification in discrete-action environments).

The high-level policy in most HRL architectures chooses from a discrete set of n actions (skills), forming a neural classifier. Therefore, adding a new $(n + 1)$-th skill to the architecture essentially means extending the output vector of the policy network by one extra output unit. As the output layer is almost exclusively a simple, fully-connected one, we aimed our focus on it.

The output layer of high-level policy network π^H can be viewed as a series of n weights vectors w_i and n bias values[4] b_i, one for each current skill π^L_i. Thus, to add a new skill policy π^L_{n+1} to the architecture, we have to construct the new weights vector w_{n+1} and bias b_{n+1}. Although these weights will get adjusted as the high-level agent progresses in its training, careful initialisation might considerably help.

We introduce six different integration schemes which create the initial values of w_{n+1} and b_{n+1} – two *uninformed* and four *informed* ones. These schemes were constructed with two key goals in mind: Firstly, we want to enhance the usage of the new skill, in order to support its successful exploration. As the policy π^H is already partially trained, it could happen that the new skill would not be used after it was created, and the high-level agent would have no chance to recognize its benefits. Adequate exploration of the new skill is hence crucial. Secondly, we want to gently 'nudge' the high-level agent into using the new skill in situations it was made for. Since the skill was trained based on a specific sequence δ, it is mostly probable to help in situations when δ had been previously used. It is important to note that both of these principles concern only the *initial* weights of the new skill – they are meant to kick-start the adaptation of the new skill, but the rest is then left upon the RL algorithm. Hence, in cases that the skill is not useful after all, the high-level agent is not forced into using it, but rather learns to ignore it.

4.5.1 Uninformed schemes

We grouped the first two integration schemes under a term 'uninformed', as they work solely with the output layer of π^H policy network, and they do not access any additional information.

Random initialization

As the name suggests, this simple scheme initializes the new skill's weights randomly. It serves mainly as a baseline, to which we will compare the other schemes. Contrary to all others, the random integration scheme is the only one which does not enhance the exploration of the new skill.

[4] All of the following methods work even if bias is not used, except for bias-boosted random initialization.

We used normal distribution for both w_{n+1} and b_{n+1}. In order to initialize the weights to reasonable values, we draw the new weights from a distribution with same per-dimension mean and variance as the old weights has.

$$\begin{aligned} w_{n+1} &\sim \mathcal{N}\left(\text{mean}([w_i]_{i=1}^n),\ \text{std}([w_i]_{i=1}^n)^2\right) \\ b_{n+1} &\sim \mathcal{N}\left(\text{mean}([b_i]_{i=1}^n),\ \text{std}([b_i]_{i=1}^n)^2\right) \end{aligned} \tag{4.7}$$

Bias-boosted random initialization

Similarly to the previous one, the core of this scheme is also a random initialization of the weights w_{n+1}. However, the bias b_{n+1} was made significantly greater in order to intentionally increase the activation of $(n+1)$-th unit. This should encourage natural exploration of new skill by the high-level RL agent.

Since the biases of the original skills can have negative values, we cannot get b_{n+1} by simply multiplying the highest b_i, as that could result in even smaller activation. Instead, we add the range of all old b_i to their maximum. This ensures that the new value of b_{n+1} will be the highest, yet still within a reasonable range compared to original bias values. The weight vector is generated the same way as previously:

$$\begin{aligned} w_{n+1} &\sim \mathcal{N}\left(\text{mean}([w_i]_{i=1}^n),\ \text{std}([w_i]_{i=1}^n)^2\right) \\ b_{n+1} &= \max_{i\in\{1..n\}} b_i + \left(\max_{i\in\{1..n\}} b_i - \min_{i\in\{1..n\}} b_i\right) \end{aligned} \tag{4.8}$$

4.5.2 Informed schemes

The more sophisticated informed integration schemes take into account also the additional information, which we gathered during the skill-identification step. For the new skill π^L_{new} we consider the sequence δ it was based on, its start-states $S_{\text{start}}(\delta)$ and end-states $S_{\text{end}}(\delta)$.

In these schemes, more effort is given to using the new skill in correct situations. As mentioned earlier, the skill should be used in situations when δ had been executed before, as it was created specifically for these scenarios. To achieve it, we combine the old skills' weight vectors $w_1 \ldots, w_n$ to get the new weights w_{n+1}, instead of randomized creation. The main aspect, in which these schemes differ, is which skills' weights are combined, and in what manner.

Start-states' skills

The idea of this scheme is to execute π^L_{new} in those states where δ was starting. To achieve a high activation of $(n+1)$-th output for a specific state s, we mimic other output units which have high activation for the same input state s.

The start-states are already gathered in $S_{\text{start}}(\delta)$. For each $s \in S_{\text{start}}(\delta)$, we can ask the high-level policy π^H which skills it prefers. If the policy strongly prefers to choose one or two skills, we combine their weights into w_{n+1}, and it will prefer the new skill as well.

We hence construct the new weight vector w_{n+1} as a weighted average of old weight vectors w_1 to w_n (equation 4.9). In this weighted average, the coefficient c_i of weight vector w_i is proportional to the probability that high-level policy π^H chooses skill π^L_i in the start-states $S_{\text{start}}(\delta)$ (equation 4.10).

$$w_{n+1} = \sum_{i=1}^{n} c_i w_i \quad \text{where} \quad \sum_{i=1}^{n} c_i = 1 \tag{4.9}$$

$$c_i \propto \sum_{s \in S_{\text{start}}(\delta)} \pi^H\left(\pi^L_i \middle| s\right) \tag{4.10}$$

To compute the coefficients c_i, we need to obtain the probabilities $\pi^H(\pi^L_i|s)$. Given that the policy is realised as a neural classifier, we assume that the output of the last layer passes through soft-max activation, yielding this probability for each high-level action π^L_i. To get all c_i values, we hence need $|S_{\text{start}}(\delta)|$ invocations of the policy network.

First skill from δ

The next integration scheme shares similar ideology with the previous one – execute π^L_{new} in those situations when δ had been executed. However, we take a bit more direct approach – we try to 'replace' the start of δ with the new skill. We know that each execution of δ starts with its first action a^H_1 being chosen by the policy π^H. To integrate the new skill, we now want the π^L_{new} to be chosen in such cases instead. Hence, to achieve high activation of $(n + 1)$-th output in the situations when a^H_1 was being chosen, we simply copy its weights into w_{n+1}:

$$w_{n+1} = w_{a^H_1} + \varepsilon \tag{4.11}$$

A small white noise ε is added to the new weight vector in order to prevent potential unwanted effects of two output units having the exact same weights.

Note that this initialization scheme can be considered as a special case of the *Start–states* scheme, but computationally much simpler. If π^H chooses only a^H_1 in all states $S_{\text{start}}(\delta)$, then these two schemes become equal. Hence we assume that they will yield similar results, despite the large gap in their computational complexity.

All skills from δ

In the previous scheme we were trying to 'catch' the beginning of the δ, and launch π^L_{new} at that exact time. However, if we miss the beginning, i.e. if the policy π^H chooses a^H_1 and embark on the trajectory of δ, we will no longer enhance the execution of the new skill. However, the new skill might be useful even if it starts a bit later, i.e. when the agent is in the midst of execution δ's actions. Hence, it might make sense to mimic not only a^H_1, but all actions $[a^H_1, \ldots, a^H_m]$ from δ.

In order to account for all actions a^H_k from δ, we will again copy and average their weights using equation 4.9. The coefficients c_i will now be proportional to usage of skill π^L_i in sequence δ. Note that an action can repeat several times within δ, in which case it will have greater coefficient withing the weighted average:

$$c_i \propto \sum_{k=1}^{m} \left[a^H_k = \pi^L_i \right]_1 \tag{4.12}$$

Smoothed skills from δ

In the *First–skill* scheme we aimed to start the new skill at the very beginning of δ, ignoring the rest. On the other hand, the *All–skills* scheme tried to do so at any time during δ's execution, even at its very end – when the desired behavior might have been already finished. To find a middle ground between these two approaches, we also introduce a *Smoothed–skills* integration scheme.

Its idea and mechanism is nearly identical to *All–skills*, but it strongly prefers the first few skills. We do so by computing an exponentially smoothed average of old skills' weights, instead of normal average in *All–skills*. Having $\lambda < 1$, we multiply the coefficient of a^H_1 by λ^0, keeping it high, while the coefficient of last action a^H_m is multiplied by λ^{m-1}, lowering its effect:

$$c_i \propto \sum_{k=1}^{m} \lambda^{k-1} \left[a^H_k = \pi^L_i \right]_1 \tag{4.13}$$

The λ hyperparameter can be set to a fixed value, but such setting could have negative impact on different-sized sequences. If λ was low enough, it would perform desirably on shorter sequences, but on longer ones it would essentially diminish the effect of later skills to zero. On the other hand, setting λ high enough would work well for longer sequence, but on shorter ones it would barely change coefficients of all skills, reverting back to *All–skills* scheme.

Since we cannot know the length m of a sequence upfront, we remedy this issue by setting a value only for λ_{last}. It represents the effect of the last action from δ, i.e. a^H_m. From this quantity, we then compute the actual value of λ as:

$$\lambda = (\lambda_{\text{last}})^{\frac{1}{m-1}} \tag{4.14}$$

4.5.3 Finalizing the integration

In the previous sections we described how we can create a new weight vector w_{n+1} that will help establish the new skill π^L_{new} within the hierarchy. To finalize the skill integration process, we have to perform the last few technical steps.

As stated in section 4.2.1, each skill within our HRL architecture has a dedicated skill-stopping function which handles the termination of the skill's execution. The stopping function of the new skill will directly follow what the skill was trained for – reaching the end-regions $S_{\text{end}}(\delta)$. When the policy achieves to do that, it is terminated. This behavior is hence identical to terminating the episodes during the skill-training phase, when the end of an episode was determined by the surrogate reward function R'. By plugging the equation 4.6 of R' into the skill-stopping function signature from equation 4.1, we obtain the new skill-stopping function f^{stop}_{n+1} for the policy π^L_{new}:

$$f^{\text{stop}}_{n+1}(\ldots, s_t) = \left(\exists s^{(i)}_{\text{end}} \in S_{\text{end}}(\delta) \ : \max_{d \in \{1..\dim(S)\}} \left|s^{(i)}_{\text{end}}[d] - s_t[d]\right| < \epsilon\right) \vee \left(c > t_{\max}\right) \tag{4.15}$$

We also included a condition $c > t_{\max}$ to limit the maximal execution time of the skill, in case it fails to reach the desired end-region.

Now, with all segments prepared, we can put all pieces together and resume the training of the high-level agent:

- Include the new skill into high-level agent's actions:
  $A^H \leftarrow A^H \cup \left\{\pi^L_{\text{new}}\right\}$
- Update the output layer of π^H policy network, so that it contains new weight vector:
  $W_{out} \leftarrow [w_1, \ldots, w_n, w_{n+1}]$
- Include the new skill's stopping function f^{stop}_{n+1} into the hierarchy.
- Reset the path-trie, so that it can collect new data about sequences.
- Resume the training of high-level agent π^H, which can now fully use the newly acquired skill π^L_{new}.

Chapter 5

Experiments and Results

In order to measure the effectiveness of Adaptive Skill Acquisition, we conducted series of experiments focused on various aspects of our work. Since the principle goal of ASA is to enhance the performance of overall HRL system, our primary tool is the comparison between scenarios in which ASA *was* or *was not* used, keeping all other aspects identical. Thus in each experiment, we ran a simulation of a 'Base run' without deploying ASA, and the same simulation but with engaged ASA mechanism.
The individual experiments were based on three principal questions:

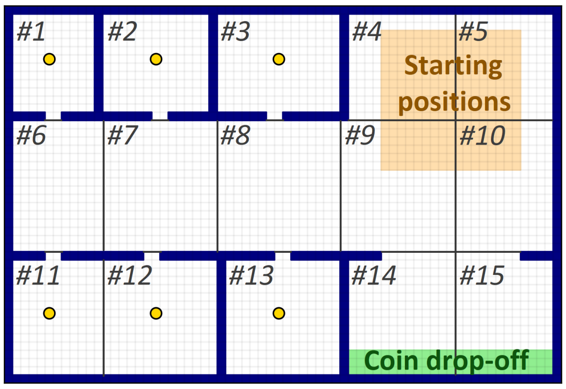
- How much does adding a new skill help in training an overall task?
- How well does ASA identify and formulate a new skill?
- How do different skill integration schemes affect the efficiency in an overall task?

These three questions are addressed in sections 5.3 – 5.5.

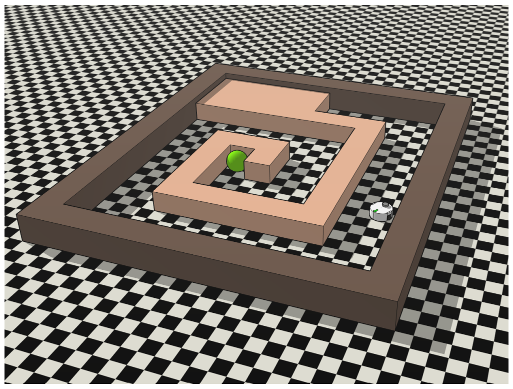
5.1 Environments

For our experiments we chose two environments which on purpose differ in numerous aspects, such as continuity, observability, skill types, etc. The reason for choosing these environments was to demonstrate that ASA can be used universally, not only in a single type of tasks. The only common feature of both environments is their reward sparsity. This aspect made the tasks sufficiently difficult to solve, such that we were not able to successfully train these tasks by traditional flat-RL (using Schulman et al., 2015). Hence, they represent a suitable challenge for a HRL agent.

A HRL architecture has been designed for both environments. In order to leverage the functionality of ASA, we intentionally built these architectures with an incomplete skill set – leaving one skill out, and observed if ASA can identify the missing skill. However,



(a)



(b)

Figure 5.1: Environments used for experiments: (a) *Coin-gatherer* map – agent is initiated in orange area, has to collect coins (yellow dots) and bring them to green drop-off area. The numbers represent 15 ideal skill regions. (b) *Maze-bot* – one of six possible mazes that agent has to solve. The green sphere represents the goal position.

we also considered a scenario of using ASA atop of optimal HRL architecture. Even if it was used and would (falsely) identify a missing skill, we show in section 5.4 that adding an unsuitable skill does not negatively impact the performance of the overall HRL system.

5.1.1 Coin-gatherer

The first used environment is a relatively simple gridworld-based one. This conforms with numerous previous research, when various forms of gridworlds were used to evaluate HRL algorithms (Sutton et al., 1999; Dietterich, 2000; Menache et al., 2002; Goel and Huber, 2003; Bakker and Schmidhuber, 2004; Moerman, 2009; Dillinger, 2019, and others). The map we use is an adaptation of the largest map from Moerman (2009), which is 68×46 tiles large, as shown in figure 5.1a. In this task the agent has to gather all six coins, depicted as yellow dots, and deliver them to the green drop-off area. However, it can carry only one coin at a time.

The agent can observe the whole state of the environment: its position, whether it is carrying a coin, and positions of all coins. This makes for a discrete state space with a total of 4×10^5 possible states. The original action space of the environment consists of four actions: move *north*, *south*, *east*, or *west*. The only reward agent receives is when it manages to successfully deliver a coin to the drop-off area. When all coins have been delivered, the episode is terminated.

The HRL decomposition we use also resembles various previous approaches (Bakker and Schmidhuber, 2004; Moerman, 2009; Metzen and Kirchner, 2013; Dillinger, 2019). The map is divided into non-overlapping regions (marked #1 – #15 in figure 5.1a), and a separate skill is trained to reach each region. These skills are used to augment the original actions (N, S, E, W), which are still available to the high-level agent (as so-called 'atomic skills'). This resembles the original notion of temporally extended actions (Sutton et al., 1999). To introduce an imperfection into the HRL architecture, we left out the skills for regions #14 and #15. The skill set is thus composed of thirteen goal-based skills and four atomic skills.

As a result, the task that agent faces is a discrete, deterministic, fully observable one, with the HRL architecture using a high number of 17 skills, which are mainly goal-based.

5.1.2 Maze-bot

The second environment aligns with newer approaches by employing a small simulated robot. The agent is a vacuum-cleaner-like robot and its task is to navigate through a simple maze towards its goal position. We constructed a set of six different mazes, altering both structure and difficulty of the maps – one of them is shown in figure 5.1b. In each episode a map is chosen at random and the agent is placed at its start. It then has to navigate through the map and reach the goal position depicted by green sphere.

The agent has a LIDAR-like sensor informing it about obstacles in its proximity, the range of the sensor is roughly 6 times the robots size. However, the agent does *not* have any information about its orientation (compass), or which maze it has been placed into. Hence, it needs to learn e.g. the 'follow wall' strategy instead of blindly memorizing an action sequence for each map. The atomic action is a vector of torques for the wheels, clipped to $[-1, 1]$ interval. The agent receives a sparse positive reward of $+1$ only when it reaches the desired goal position. All other actions are uniformly penalised with -0.05 reward per step, giving the agent no information about its progress.

The hierarchical architecture is realised by supplying two locomotion skills: one for efficiently moving forward a larger distance, and the other one to turn the robot left by 90°. The skill for turning right is not provided for the agent, representing a missing skill.

In contrast to *Coin-gatherer*, the *Maze-bot* environment is continuous in both states and actions, randomised, and partially observable, with the HRL architecture using low number of two skills, which are behaviorally focused.

5.2 Implementation and experiments setup

In this section we cover the details about the technical setup of the experiments, agents, and the training process. The implementation of ASA, which is also available online[1], was greatly accelerated by *RL-lab* (Duan et al., 2016), and after its discontinuation by the *Garage* library (Garage contributors, 2019).

[1]The most up-to-date version of ASA is published at https://github.com/holasjuraj/asa

The core of the top layer of our agents was a stochastic categorical policy using a simple multi-layered perceptron. On the lower level, including the new skill trained by ASA, we used the same policy architecture for *Coin-gatherer* environment, and a continuous Gaussian MLP policy for *Maze-bot*. All policy networks featured two hidden layers with 32 (low-level) or 64 (high-level) units using hyperbolic tangent activation. The agents were originally trained by Natural Policy Gradient algorithm (Kakade, 2002), which was later consistently surpassed by TRPO (Schulman et al., 2015), so we used it for all agents (both high-level and low-level). In terms of neural networks training, TRPO employs a conjugate gradient optimizer to adjust the networks' weights. We used *Garage* implementations for all these algorithms.

All *Maze-bot* agents were trained for 80 iterations, *Coin-gatherer* and all skills were trained for 300 iterations.[2] A batch-training method was used for the training, in which the policies were updated after a batch of 5000 steps. The maximal length of an episode was set to 100 high-level steps for *Maze-bot* and 50 for *Coin-gatherer* – if this limit exceeded, the episode was terminated as an unsuccessful run. This choice of the limit is particularly interesting in *Coin-gatherer*, since the optimal strategy is 40 steps-long, so the agent had to follow a near-optimal strategy in order to deliver all coins in time. Finally, the discount factors were set to 0.99 (*Coin-gatherer*) or 0.9 (*Maze-bot*).

The evaluation metric we chose for all models is the *average discounted return*:

$$G(\pi) = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t)\right] \tag{5.1}$$

In our case, the general notion of policy π is represented by the whole HRL system, i.e. $\pi = \{\pi^H, \pi^L_1, \ldots, \pi^L_n\}$. We consider it to be the most suitable metric for evaluation, since the agent(s) were trained by optimising the same criterion.

All experiments were executed 8 times, changing only seed of the random numbers generator. The results were then averaged over these trials to get a mean performance of each model. In the figures later in this chapter, we also use a lighter shade to display the 25–75 percentile interval for *Coin-gatherer* environment, and 5–95 percentile interval for *Maze-bot* (since its training was more stable and consistent).

5.3 Overall performance

In Experiment 1, we focused on the most important question to be answered: how much does adding a new skill help in training an overall task. We ran a Base run simulation, training the HRL architecture with the pre-trained skill set which was incomplete, as described in section 5.1. Another simulation was then run on identical setup, but with ASA turned on – which resulted in ASA identifying the missing skills and adding them into the system.

[2]Although much fewer iterations would be sufficient for the skills, since they converged relatively fast.

Additionally, as a comparative study, we used the same environments to train agents using HiPPO algorithm (Li et al., 2019). We chose this approach as it is one of few models capable of skill training even *during* high-level training, as we do in our approach. It trains the whole HRL system – i.e. the high-level agent and the skills at the same time, and has been shown

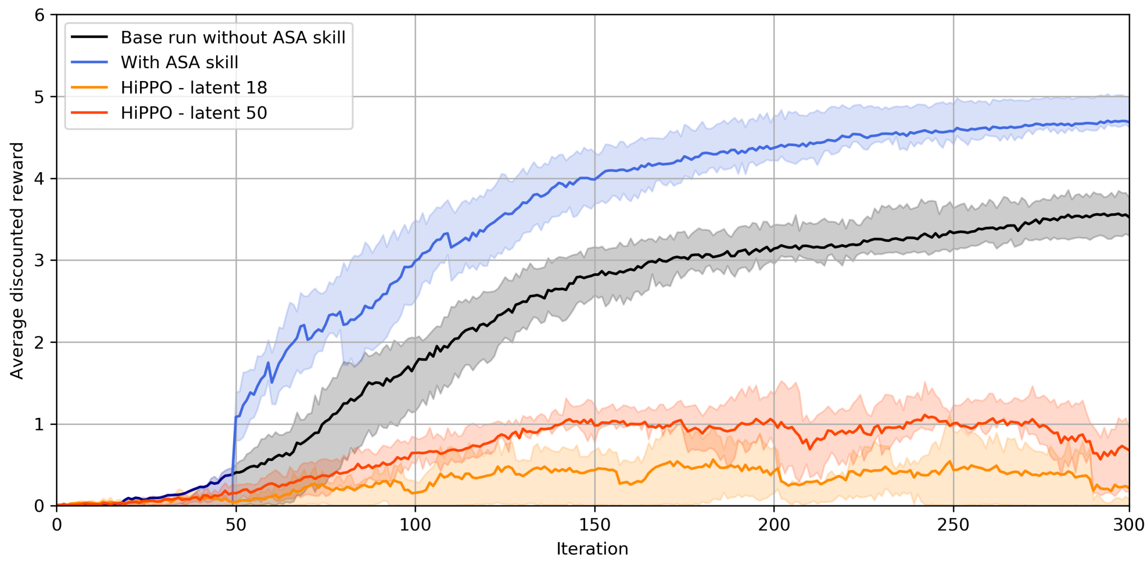


Figure 5.2: Overall performance of ASA in *Coin-gatherer* environment. The increase gained by adding the ASA skill resulted in collecting ~1.2 more coins. The equivalent HiPPO model was not able to train a reasonable behavior – raising number of skills to 50 helped, but still yielded subpar performance.

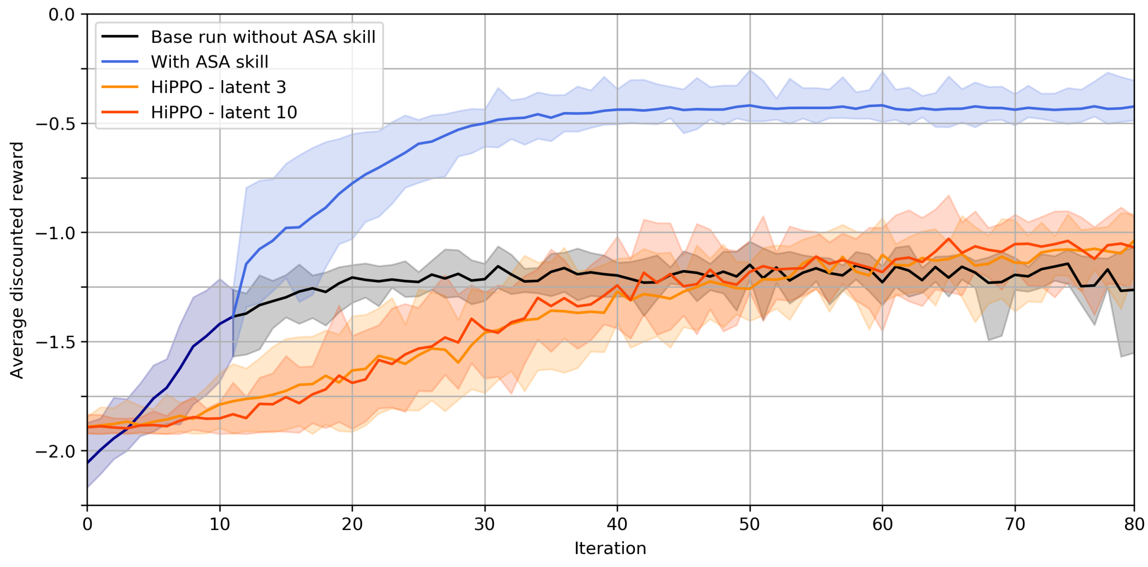


Figure 5.3: Overall performance of ASA in *Maze-bot* environment. The increase gained by adding the ASA skill resulted in shortening the path by ~16.2 high-level steps. The HiPPO model again performed worse, as it has probably identified only some of the useful skills.

to outperform previous similarly oriented approaches. We also tried modifying the latent dimension $\dim(Z) = n$, which affects how many skills are to be trained. We tried two options for each environment: setting n to the same number of skills as our ASA agent had (18 / 3 skills), or making the skill set roughly tree times as large (50 / 10 skills). For further details on HiPPO algorithm, see section 3.3.10.

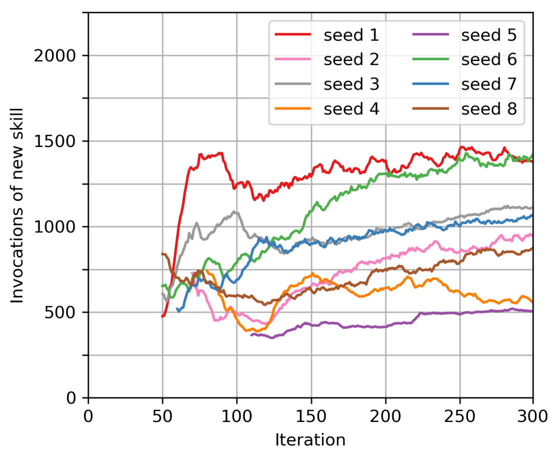
Figures 5.2 and 5.3 display the overall performance of ASA-enabled agents, in comparison to the Base runs without ASA, for both testing environments. We can observe in both cases that adding a new skill identified by ASA consistently and significantly increases the performance of the agent. In case of *Coin-gatherer*, this means an increase from an average of 4.4 to 5.6 delivered coins, out of 6 possible. The *Maze-bot*'s increase by 0.81 in average discounted return translates to paths shorter by 16.2 high-level steps, on average.

As mentioned before, the two environments use fundamentally different types of skills – goal-based in *Coin-gatherer* vs. behavioral in *Maze-bot*. Since ASA was successful in both cases, it suggests that it is able to train both goal-based and behavioral types of skills. To the best of our knowledge, there has not yet been a published model that would be demonstrate such behavior.

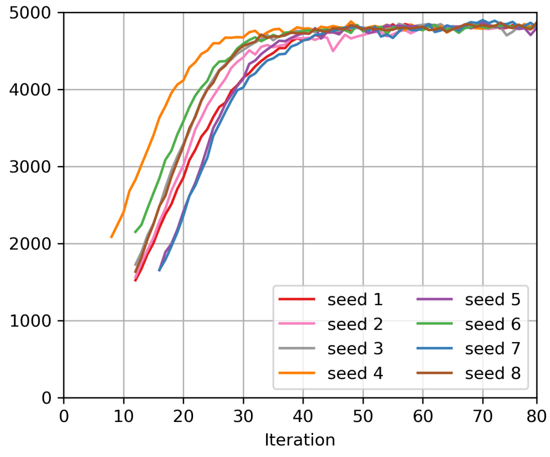
We can also observe that the performance of HiPPO lacks greatly behind the performance of our models. The slower start of the training can be in large part explained by the fact that HiPPO starts with untrained skills, in comparison to the pre-trained skill set in the Base runs. However, even after sufficient time and near convergence, HiPPO still had lower overall performance than ASA. This suggests that it was not able to successfully identify all useful skills, and hence the resulting skill set was still incomplete.

The nature of HiPPO can also explain its subpar performance in *Coin-gatherer* environment. This algorithm was primarily designed for behavioral skills, while this environment strongly benefits from goal-based ones. On the other hand, we can see that *Maze-bot* environment, which was focused on behavioral skills, enabled the HiPPO agent to score reasonably well.

Figure 5.4 displays the usage of the new skill which was added by ASA. In both environments and all seeds the new skill is used reasonably often by the high-level agent, which hints that the skill was indeed useful. The nearly unchanged values in case of *Coin-gatherer* also suggest that the skill integration performed well, and set the new skill to be used in the correct situations. On the other hand, the rising trend in case of *Maze-bot* shows that the integration was not ideal, and weights for the new skill needed to be significantly adjusted by the training of the high-level agent.



(a) *Coin-gatherer* environment



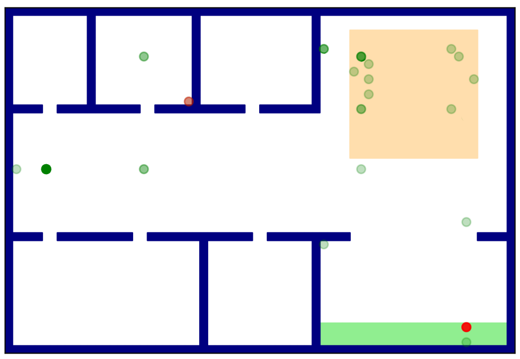
(b) *Maze-bot* environment

Figure 5.4: Usage of the new skill by the high-level agent. (a) The constant trend is a sign of good skill integration. (b) The large change suggests that strong adaptation was needed, since the integration was not ideal.

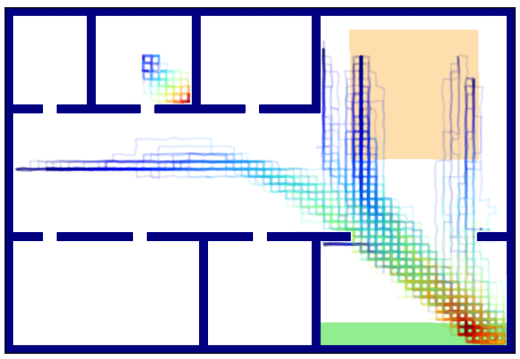
5.4 Quality of the new skill

Since the new skill is the key component of ASA method, in Experiment 2 we aimed to evaluate its quality and abilities.

First, as a form of empirical example, figure 5.5 shows a sample skill from the *Coin-gatherer* environment. When the ASA identified the best sequence δ to be the candidate for the new skill, we extracted the set of its start-states – scattered through the map, and end-states – focused mainly in the coin drop-off area, as shown in subfigure (a). A new skill was then trained using these two sets of states, and its behavior is displayed in subfigure (b).



(a)



(b)

Figure 5.5: Example of a skill from *Coin-gatherer* environment. (a) The set of start-states (green dots) and end-states (red dots) of the identified sequence δ. (b) Paths taken by the skill trained from δ – each path starts in in blue and ends in red color.

In order to rigorously evaluate the quality of the new skill in general, we constructed the upper and lower bounds for the skill performance. These bounds come in form of an *ideal skill* as the upper bound, and a *bad skill* as the lower bound.

The *ideal* skills were manually constructed in order to optimally enrich the skill set. In the case of *Coin-gatherer*, it was a skill for reaching the regions #14 and #15 (from figure 5.1a). For *Maze-bot*, it was the skill to turn right. It is important to note that by constructing these ideal skills manually, we might have introduced an engineered bias to them. Hence, they might not be the *optimal* skills per se, but they are as good as we could think of.

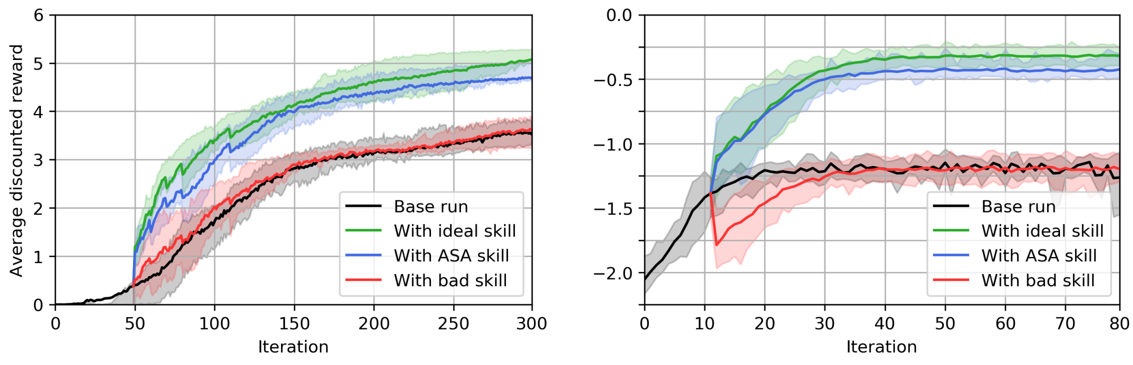
On the other side of the spectrum, the *bad* skills that form a lower bound are constructed as simple random policies for both environments. Such skills have zero contribution to the overall system, which is as low as we can get before actively hurting the HRL agent.

Having set the bounds for usefulness of the skill, we can now evaluate how high the ASA-trained skill stands between them. In the first trial, we kept all submodules of ASA active (decision, integration, ...), except for the skill training step, which was substituted by manually constructed ideal or bad skills. The final results for such setup are shown in figure 5.6. We can observe that ASA-trained skill performs very well, falling short only slightly behind the ideal skill. If we consider the ideal and bad skills as a range bounding the fitness of the skill, we can express that ASA-trained skill scored solid 73.2% of the possible performance in *Coin-gatherer*, and excellent 88.7% in *Maze-bot*.

As a form of ablation study, we further disabled the decision element and triggered the addition of new skills manually. The ASA mechanism then took the imperfect data it had collected up to the trigger time, and was forced to formulate, train, and integrate a new skill from these data. In case of manually adding the ideal or bad skills, the skill creation was also done outside ASA, and we only let ASA integrate it into agent's hierarchy.

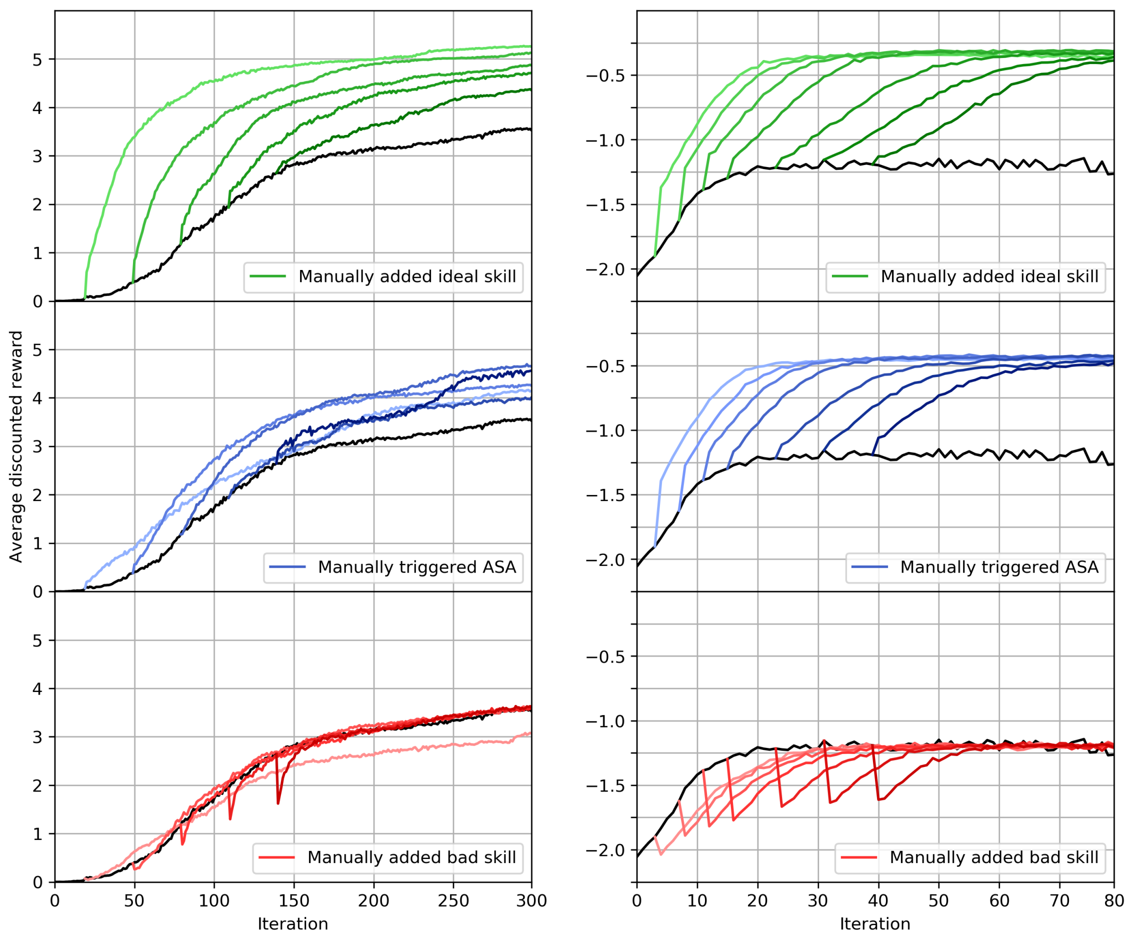
Figure 5.7 shows the results for these manually triggered trials – for ideal, ASA-train, as well as bad skills. We can draw several conclusions from this series of plots. First and foremost, we can conclude that ASA is able to formulate a reasonably good skill regardless of the trigger time. All of ASA-trained skills improved the overall performance significantly, with the majority of them keeping pace with the ideal skills. The exception to this can be seen only if ASA was triggered at the very beginning of *Coin-gatherer* training (middle left plot). While the new skill was somewhat useful, its speed of convergence was significantly slower in comparison to the ideal skill (upper left plot). We attribute this effect to the fact that the high-level agent had not yet learned almost anything up to this point (black Base run line is nearly zero), and hence the sequences δ consisted mainly of exploration paths.

Secondly, we are able to falsify the hypothesis that *any* added skill may cause an improvement. We can see from the bottom-row plots that the performance with a bad skill initially dropped as the high-level agent tried to explore it, and eventually leveled out with the original



(a) *Coin-gatherer* environment (b) *Maze-bot* environment

Figure 5.6: Comparison of ASA-trained skill with a manually constructed ideal skill (upper bound), and a useless bad skill (lower bound).



(a) *Coin-gatherer* environment (b) *Maze-bot* environment

Figure 5.7: Addition of the new skill was triggered manually, and we forced ASA to integrate an ideal skill (top), ASA-trained skill (middle), or a bad skill (bottom).

performance when the agent learned to ignore it. However, no significant improvement was observed in any run. This proves that the success of ASA is not only coincidental.

Finally, the bottom-row plots also show that adding a useless skill does not hurt the model performance in the long-term view. This information means that even if ASA produced a malformed skill, its addition to the HRL system would not hurt its final performance. Hence, ASA can be deployed on any HRL architecture without worries – if the architecture is suboptimal, ASA will try to fix it, but if it is already optimal, ASA will not break it.

5.5 Integration schemes

Experiment 3 was focused to evaluate the efficiency of individual skill integration schemes described in section 4.5. We introduced two uninformed schemes – one of which was purely random, serving as a baseline for comparison. We also constructed four informed schemes, which leveraged an additional information to better integrate the new skill. Our hypothesis was to observe substantially better performance in cases when the informed schemes are used.

We executed multiple simulations in which a new skill was created at different times during the high-level training. The resulting performance of all integration schemes is displayed in figure 5.8. We can conclude that usage of different integration schemes before (subfigure a) or at time when ASA decided to add new skill (subfigure b) had negligible effect on the reward gain or learning speed, disproving our initial hypothesis, since all integrators exhibited comparable performance to the baseline *Random* integrator in these situations.

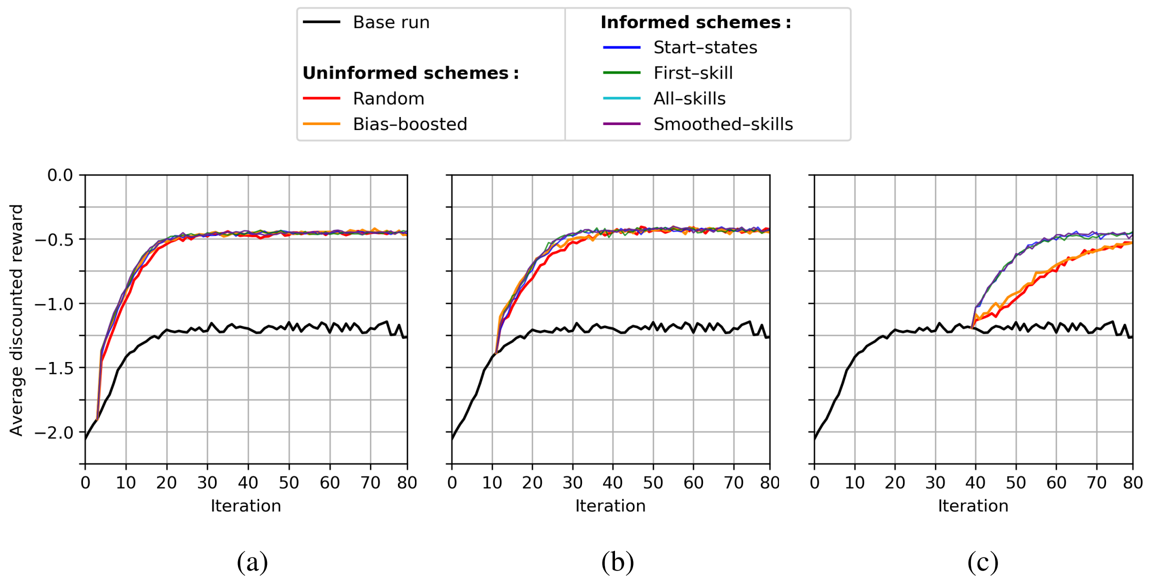


Figure 5.8: Effect of using different integration schemes during the new skill's integration in *Maze-bot* environment: (a) at 4th iteration, (b) at 12th iteration (ASA decision), (c) at 40th iteration.

The only, yet still not very significant difference in learning speed can be observed between uninformed and informed schemes if we add a new skill much later, i.e. after the training of the imperfect hierarchy converged to a stable solution (subfigure c). By observing this phenomenon we found that the later we add a new skill, the more notable this difference is. We attribute this effect to the naturally decreased exploration in later stages of the high-level training, which is not supported by *Random* scheme in contrast to the informed ones. However, the simple exploration enhancement of the uninformed *Bias-boosted* integrator did not work even in this scenario. These findings suggest that more sophisticated directed exploration of informed schemes does indeed work, but its effects are too week to manifest if the high-level agent performs a significant exploration by itself.

Chapter 6

Discussion and Future Work

In the previous chapter we described in detail the results of individual experiments in our work. Now we can combine the conclusions from individual experiments, and summarize the functionality of individual components of Adaptive Skill Acquisition. Here we will also discuss further topics on the applicability of ASA, as well as improvements that can be implemented in the future.

As described in section 4.2.1, the ASA framework consists of three conceptual steps, which we will try to evaluate: identification, training, and integration of the new skill. Starting with the skill identification process, we feel confident to conclude that it works exceptionally well. This claim can be backed mainly by the results of Experiment 2 (section 5.4) – even though this experiment tested jointly both skill identification *and* its training. However, the skills performed well despite the fact that the trained policy was not always 100% accurate to the identified behavior, which hints that the skill identification was indeed robust enough. Although the implementation composed of the null-hypothesis count $C_H(\delta)$ (eq. 4.3) and the f-score (eq. 4.4) can be seen as a rather simple one, the underlying idea based on identifying frequent sequences proved to be a viable road to success. In fact, the simplicity of the approach can be even recognized as its advantage, considering the principle of Occam's razor.

Moving on to the skill training phase of the ASA process, we can again conclude that it performed sufficiently well, though not flawlessly. The MDP constructed in section 4.4, and especially its reward function R', well represents the desired behavior of the identified skill. This can again be seen in the results from Experiment 2, and was also empirically observed when looking at the behavior of the trained skills. However, having intercepted the training at various times and observed the partial results, we sometimes noticed that some of the trained policies failed to achieve the desired behavior. Implementing the filter for untrained skills helped to prevent integration of such skills, but further steps could be taken to increase the skill success rate.

Finally, the skill integration process did successfully incorporate the new skill into the existing HRL architecture, although its efficiency was far from ideal. As the results from Experiment 3 (section 5.5) showed, none of the proposed integration schemes was able to consistently surpass the baseline *Random* integrator. As a result, substantial adjustment of the new skill's weight vectors had to be sometimes performed, as also seen from figure 5.4b. Therefore, we consider the skill integration to be the weakest point of ASA at this time, and it would benefit from further improvements.

Although all aforementioned methods described the usage of ASA within a two-layered architecture, it is worth discussing also the case of multi-layered hierarchies. The vast majority of research field still focuses on strictly two-level hierarchies (Sutton et al., 1999; McGovern and Barto, 2001; Menache et al., 2002; Bakker and Schmidhuber, 2004; Moerman, 2009; Metzen and Kirchner, 2013; Florensa et al., 2017; Li et al., 2019, and others). Nonetheless, some authors did rather choose a more dynamical multilevel hierarchies (Dietterich, 2000; Levy et al., 2018). In such cases, the hierarchy can be represented by a tree in which an inner node acts as a skill of its parent, but is a separate HRL system of itself. Alternatively, the policies within the same level can be shared, essentially creating a directed acyclic graph of policies.

Despite having tested ASA only on two levels, we designed it in a way that it could be in principle deployed on any level of a general multi-level hierarchy. Considering the tree-like structure of such a hierarchy, ASA can be directly used on any inner node, to which it can add new skills forming new leaves in the hierarchy tree. Alternatively, the skill-training step might be changed from a simple flat-RL training to construction of the whole HRL system – in such a case the ASA would create a whole subtree of policies.

The future work that can be conducted on ASA focuses mainly on its identified weak point – the skill integration process. New integration schemes can be developed in order to better enhance the exploration and adaptation of the new skill. We could employ a forced exploration of the new skill – modify the high-level agent's choices to force the execution of the new skill in selected situations. This effect would, of course, decrease over time, eventually leaving the full control back to the high-level policy.

We also consider using a pseudo-rehearsal method (Robins, 1995) to create the weight vector for the new skill. Thus, it would be properly pre-trained, instead of static initialisation used in current version. Since we use a batch-training method, we already gather all training samples, and no further simulations would be needed to obtain them.

Conclusion

In the presented thesis we addressed the issue of missing skills within the architecture of Hierarchical Reinforcement Learning models. As our main contribution, we introduced the *Adaptive Skill Acquisition* framework, which is able to dynamically identify that a skill is missing, and initiates a process to fix such malformed architecture. ASA autonomously formulates what a new skill should do, trains a new agent that implements this behavior, and integrates the skill agent into the overall HRL system.

ASA was designed as a pluggable component, and can be used on top of almost any HRL architecture. Its applicability is enhanced by the low implementation effort that is needed to incorporate it into a new system. Furthermore, all components of ASA were designed so that it could be used in wide variety of tasks too. Being capable of dealing with continuous state- and action-spaces, sparse rewards, partial observations, or model free environments, ASA is enabled to be work in any use-case.

The capabilities of our approach were demonstrated in two fundamentally different environments. The *Coin-gatherer* represents an ideologically simpler, discrete gridworld task, while *Maze-bot* employs a simulated robot in a continuous environment. In both cases, ASA was able to increase the overall performance of the HRL agent, proving its interoperability among different tasks. Moreover, given the different nature of skills used in these environments, the results suggest that ASA is capable of training both goal-based and behaviorally-oriented skills. To the best of our knowledge, there has not yet been a model that would demonstrate such ability.

Our experiments proved that employing ASA into an imperfect HRL architecture provides clear advantage to the model. This can be best seen in comparison to the Base runs without ASA. The results showed that adding a new skill identified by ASA consistently and significantly increases the performance of the agent in both environments. Further ablation study revealed that the skill identification component is exceptionally robust, and able to formulate useful behaviors even from imperfect data.

We also compared our approach to HiPPO algorithm (Li et al., 2019), which, similarly to us, adjusts the skills even during the high-level training. This relatively new model aims to identify and train all useful skills. However, the comparative study showed that ASA outperforms this approach by large margin. While the slower start of HiPPO can be easily explained by its nature (training *all* skills at once), its premature convergence into suboptimal solution clearly shows that it is not able to identify all useful skills. These, however, could be identified by ASA, which in turn resulted in improved performance.

To conclude, we believe we have provided an interesting piece to the mosaic of hierarchical RL approaches that can be expected in the future to contribute to the development of intelligent autonomous learning adaptive systems that will serve to the benefit of the whole society in the technology-rich 21st century.

Bibliography

Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong, R., Welinder, P., McGrew, B., Tobin, J., Abbeel, P., and Zaremba, W. (2017). Hindsight experience replay. In *Advances in Neural Information Processing Systems*, pages 5055–5065.

Bakker, B. and Schmidhuber, J. (2004). Hierarchical reinforcement learning based on subgoal discovery and subpolicy specialization. In *Proceedings of the 8-th Conference on Intelligent Autonomous Systems*, pages 438–445.

Baldassarre, G. (2019). Intrinsic motivations and open-ended learning. *arXiv preprint: 1912.13263*.

Banach, S. (1922). Sur les opérations dans les ensembles abstraits et leur application aux équations intégrales. *Fundamenta Mathematicae*, 3(1):133–181.

Barto, A. G. and Mahadevan, S. (2003). Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13(4):341–379.

Bellman, R. (1957a). *Dynamic programming*. Princeton University Press.

Bellman, R. (1957b). A markovian decision process. *Journal of Mathematics and Mechanics*, pages 679–684.

Dayan, P. and Hinton, G. E. (1993). Feudal reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 271–278.

Deisenroth, M. P., Neumann, G., Peters, J., et al. (2013). A survey on policy search for robotics. *Foundations and Trends in Robotics*, 2(1–2):1–142.

Dietterich, T. G. (2000). Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of Artificial Intelligence Research*, 13(1):227–303.

Dillinger, V. (2019). *Abstract state space construction in hierarchical reinforcement learning*. PhD thesis, Comenius University in Bratislava, Faculty of Mathematics, Physics and Informatics.

Duan, Y., Chen, X., Houthooft, R., Schulman, J., and Abbeel, P. (2016). Benchmarking deep reinforcement learning for continuous control. In *International Conference on Machine Learning*, pages 1329–1338. PMLR.

Florensa, C., Duan, Y., and Abbeel, P. (2017). Stochastic neural networks for hierarchical reinforcement learning. *arXiv preprint: 1704.03012*.

Garage contributors (2019). Garage: A toolkit for reproducible reinforcement learning research. `https://github.com/rlworkgroup/garage`.

Gaskett, C., Wettergreen, D., and Zelinsky, A. (1999). Q-learning in continuous state and action spaces. In *Australasian Joint Conference on AI*, pages 417–428. Springer.

Goel, S. and Huber, M. (2003). Subgoal discovery for hierarchical reinforcement learning using learned policies. In *Proceedings of FLAIRS – Florida AI Research Society Conference*, pages 346–350.

Gu, S., Lillicrap, T., Sutskever, I., and Levine, S. (2016). Continuous deep q-learning with model-based acceleration. In *International Conference on Machine Learning*, pages 2829–2838.

Hasselt, H. V. (2012). *Reinforcement learning in continuous state and action spaces*, volume 12 of *Adaptation, Learning, and Optimization*, pages 207–251. Springer.

Heess, N., Sriram, S., Lemmon, J., Merel, J., Wayne, G., Tassa, Y., Erez, T., Wang, Z., Eslami, A., Riedmiller, M., et al. (2017). Emergence of locomotion behaviours in rich environments. *arXiv preprint: 1707.02286*.

Hengst, B. (2010). Hierarchical reinforcement learning. In *Encyclopedia of Machine Learning*, pages 495–502, Boston, MA. Springer US.

Howard, R. A. (1964). *Dynamic programming and Markov processes*. Wiley for The Massachusetts Institute of Technology.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pages 448–456.

Kakade, S. and Langford, J. (2002). Approximately optimal approximate reinforcement learning. In *International Conference on Machine Learning*, volume 2, pages 267–274.

Kakade, S. M. (2002). A natural policy gradient. In *Advances in Neural Information Processing Systems*, pages 1531–1538.

Kober, J. and Peters, J. (2012). Reinforcement learning in robotics: A survey. In *Reinforcement Learning*, pages 579–610. Springer.

Konda, V. R. and Tsitsiklis, J. N. (2000). Actor-critic algorithms. In *Advances in Neural Information Processing Systems*, pages 1008–1014.

Konidaris, G. and Barto, A. G. (2007). Building portable options: Skill transfer in reinforcement learning. In *International Joint Conference on Artificial Intelligence*, volume 7, pages 895–900.

Konidaris, G. and Barto, A. G. (2009). Skill discovery in continuous reinforcement learning domains using skill chaining. In *Advances in Neural Information Processing Systems*, pages 1015–1023.

Kovács, P. (2017). Learning of object grasping in a robotic system. Master's thesis, Comenius University in Bratislava, Faculty of Mathematics, Physics and Informatics.

Levy, A., Konidaris, G., Platt, R., and Saenko, K. (2018). Learning multi-level hierarchies with hindsight. In *International Conference on Learning Representations*.

Li, A., Florensa, C., Clavera, I., and Abbeel, P. (2019). Sub-policy adaptation for hierarchical reinforcement learning. In *International Conference on Learning Representations*.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2016). Continuous control with deep reinforcement learning. In *International Conference on Learning Representations*.

McGovern, A. and Barto, A. G. (2001). Automatic discovery of subgoals in reinforcement learning using diverse density. In *International Conference on Machine Learning*, volume 1, pages 361–368.

McGovern, E. A. and Barto, A. G. (2002). *Autonomous discovery of temporal abstractions from interaction with an environment*. PhD thesis, University of Massachusetts at Amherst.

Menache, I., Mannor, S., and Shimkin, N. (2002). Q-cut—dynamic discovery of sub-goals in reinforcement learning. In *European Conference on Machine Learning*, pages 295–306. Springer.

Metzen, J. H. (2013a). Learning graph-based representations for continuous reinforcement learning domains. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 81–96. Springer.

Metzen, J. H. (2013b). Online skill discovery using graph-based clustering. In *European Workshop on Reinforcement Learning*, pages 77–88. PMLR.

Metzen, J. H. and Kirchner, F. (2013). Incremental learning of skill collections based on intrinsic motivation. *Frontiers in Neurorobotics*, 7:11.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529.

Moerman, W. (2009). *Hierarchical reinforcement learning: Assignment of behaviours to subpolicies by self-organization*. PhD thesis, Cognitive Artificial Intelligence, Utrecht University.

Nachum, O., Gu, S., Lee, H., and Levine, S. (2018). Data-efficient hierarchical reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 3307–3317.

Parr, R. and Russell, S. J. (1998). Reinforcement learning with hierarchies of machines. In *Advances in Neural Information Processing Systems*, pages 1043–1049.

Polyak, B. T. and Juditsky, A. B. (1992). Acceleration of stochastic approximation by averaging. *SIAM Journal on Control and Optimization*, 30(4):838–855.

Popov, I., Heess, N., Lillicrap, T., Hafner, R., Barth-Maron, G., Vecerik, M., Lampe, T., Tassa, Y., Erez, T., and Riedmiller, M. (2017). Data-efficient deep reinforcement learning for dexterous manipulation. *arXiv preprint: 1704.03073*.

Robins, A. (1995). Catastrophic forgetting, rehearsal and pseudorehearsal. *Connection Science*, 7(2):123–146.

Rummery, G. A. and Niranjan, M. (1994). *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering.

Schaul, T., Quan, J., Antonoglou, I., and Silver, D. (2016). Prioritized experience replay. In *International Conference on Learning Representations*.

Schulman, J., Levine, S., Abbeel, P., Jordan, M., and Moritz, P. (2015). Trust region policy optimization. In *International Conference on Machine Learning*, pages 1889–1897.

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

Singh, S., Jaakkola, T., Littman, M. L., and Szepesvári, C. (2000). Convergence results for single-step on-policy reinforcement-learning algorithms. *Machine Learning*, 38(3):287–308.

Sutton, R. S. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the 7th International Conference on Machine Learning*, pages 216–224. Elsevier.

Sutton, R. S., McAllester, D. A., Singh, S. P., and Mansour, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems*, pages 1057–1063.

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

Watkins, C. and Hellaby, J. C. (1989). *Learning from delayed rewards*. PhD thesis, King's College, Cambridge.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. In *Reinforcement Learning*, pages 5–32. Springer.