

Safe Hierarchical Reinforcement Learning

Hierarchical Reinforcement Learning with Safe Space Exploration

Stephen Verbist

March 6, 2017

Safe Hierarchical Reinforcement Learning

Hierarchical Reinforcement Learning with Safe Space Exploration

MASTER OF SCIENCE THESIS

For obtaining the degree of Master of Science in Aerospace Engineering
at Delft University of Technology

Stephen Verbist

March 6, 2017



Delft University of Technology

Copyright © Stephen Verbist
All rights reserved.

DELFT UNIVERSITY OF TECHNOLOGY
DEPARTMENT OF
CONTROL AND SIMULATION

The undersigned hereby certify that they have read and recommend to the Faculty of Aerospace Engineering for acceptance a thesis entitled “**Safe Hierarchical Reinforcement Learning**” by **Stephen Verbist** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: March 6, 2017

Readers:

Dr. Q.P. Chu

Dr.ir. E. van Kampen

T. Mannucci, MSc.

Dr.ir. E. Mooij

Acknowledgements

I would like to express my deep gratitude to my supervisors Dr. Ir. Erik-Jan van Kampen and Tommaso Mannucci Msc. for their invaluable help throughout this project. I am very thankful that I was allowed to freely explore the research topic and receiving feedback on my choices while brainstorming with me on the subject. The guidance of my supervisors have been a great help especially in times when I was struggling and has made sure that I did not get lost in my research.

I also wish to extend my gratitude to every other faculty member whose courses I followed during my time at the university and whose education has brought me to this point.

I would also like to thank my parents Paul and Ria for their continual encouragement throughout my education at the University. Their support has been vital to keep my focus on the big picture through the troubling times. I am also very thankful to my brother Kenneth for all his help with courses and being an inspiration to me. Additionally I'd like to thank my friends for their support and providing much needed distraction from time to time.

Finally I'd also like to thank my girlfriend Linda for always encouraging and inspiring me to keep doing my best throughout my time in pursuit of my Master of Science degree.

Acronyms

AI	Artificial Intelligence
DP	Dynamic Programming
GPS	Global Positioning System
HAM	Hierarchical Abstract Machines
HRL	Hierarchical Reinforcement Learning
MDP	Markov Decision Process
RL	Reinforcement Learning
SARSA	State-Action-Reward-State-Action
SMDP	Semi-Markov Decision Process
SRL	Safe Reinforcement Learning
TD	Temporal Difference
UAV	Unmanned Aerial Vehicle

Contents

Acronyms	vii
1 Introduction	1
1-1 Background	1
1-2 Motivation	2
1-3 Research goal	2
1-4 Report structure	3
I Article	5
II Literature study	29
2 Flat Reinforcement Learning Fundamentals	31
2-1 Markov Decision Processes	31
2-2 Agents, Environments and Actions	32
2-3 Transition Probabilities and Rewards	32
2-4 Return and Policies	33
2-5 Value Functions	33
2-6 Reinforcement Learning Methods	34
2-6-1 Dynamic Programming	35
2-6-2 Monte Carlo	35
2-6-3 Temporal Differences	37

3	Hierarchical Reinforcement Learning	39
3-1	Semi-Markov Decision Processes	40
3-2	Hierarchical Reinforcement Learning Basic Principles	40
3-3	Hierarchical Reinforcement Learning Methods	42
3-3-1	Hierarchical Abstract Machines	42
3-3-2	Options	44
3-3-3	MAXQ	46
3-3-4	A* and Reinforcement Learning	48
3-3-5	Conclusion	50
4	Safe Reinforcement Learning	51
4-1	Modifying the optimisation criterion	51
4-2	Modifying the exploration process	53
4-3	Conclusion	55
5	Preliminary Results	57
5-1	Flat Reinforcement learning	57
5-2	Hierarchical Reinforcement Learning	58
6	Research Method	63
6-1	Algorithm Design	63
6-2	Test Setup	63
6-3	Testing Parameters	64
6-4	Research Goals	65
7	Literature study conclusions	67
III	Additional results	69
8	Relative and absolute states effect on safety	71
8-1	The task	71
8-2	The agents	72
8-3	Results	72
9	Absolute and relative states in different environment sizes	77
9-1	The setup	77
9-2	Results	77
10	Preliminary environment simulations	81
10-1	The environment	81
10-2	Results	81
11	Conclusions	85
	Bibliography	87

Chapter 1

Introduction

Reinforcement Learning (RL) is an exciting field of machine learning offering many applications in the fields of robotics (Guo, Zuo, Zheng, & Xu, 2013), wind energy conversion (Wei, Zhang, Qiao, & Qu, 2016) and Unmanned Aerial Vehicle (UAV) control (Bou-Ammar, Voos, & Ertel, 2010) (Junell, van Kampen, de Visser, & Chu, 2015) among others. With RL used in physical systems that can be damaged from incorrect controlling, it becomes important to focus research on making RL safer while maintaining its benefits. This chapter presents the background and motivation behind the research conducted in Safe Reinforcement Learning (SRL).

1-1 Background

In the interest of finding solutions for complex control problems, the development of Artificial Intelligence (AI) has indeed received decades of attention (Minsky, 1961) (Negnevitsky, 2002). There are many different approaches within AI and one such an approach is RL (Sutton & Barto, 1998) (Tesauro, 1992) (Fernández, Borrajo, & Parker, 2005). Inspired by animal learning behaviour (Sutton & Barto, 1998), flat RL, that is, a learning system based purely on RL algorithms involves an agent performing (sequences of) actions based on its observed state and obtaining a positive or negative reward from the environment. It then adjusts its strategy accordingly in order to find the strategy that accumulates the highest total sum of rewards. This research will focus on finding optimal strategies for control problems with RL methods aimed towards navigational tasks that can be implemented by UAVs as an example.

Unfortunately RL suffers from what is known as the *curse of dimensionality*, a rapidly increasing demand on computation time and system memory with increasing state-action complexity. For example, a system being able to choose from n actions in a given state s , would have to consider n additional actions for each additional state. This especially becomes a problem when transitioning a learning agent from a discrete environment to a continuous one when controlling physical systems. A solution to this is using hierarchies (Barto & Mahadevan, 2003) in the learner structure, which gathers together irrelevant states into abstract states.

For example the exact position x, y in room 3 might be abstracted away to just being in room 3. Adding additional positions within this room will not change the memory requirement of the system which only needs to track the room numbers.

Another problem that a controlling agent suffers from using flat RL is that, without any prior training or external knowledge, it will not be able to tell if an action is unsafe to the system it is controlling. For example, without having experienced it before, how could an AI controlled autonomous taxi-driving agent know that driving off a cliff is dangerous? This is because flat RL agents learn only from experience and will therefore need to perform a negative action at least once in order to learn that the chosen action is negative. And even so, the taxi-driving agent driving off a cliff, receiving a negative reward might conclude that the cliff just wasn't high enough. It is unfortunately no trivial task to simply instruct the agent to never perform certain actions after the first time, as the globally optimum path might include these actions. This is a problem of exploration vs. safety and there are many ways to deal with it. A lot of approaches seem to in some way either incorporate prior knowledge of the system, or sacrifice some autonomy of the system to achieve safety (Garcia & Fernández, 2015). It is also possible to provide an agent with a filter of sorts to discover unsafe actions or states during learning (Mannucci, van Kampen, de Visser, & Chu, 2017b). In addition to dealing with the curse of dimensionality, Hierarchical Reinforcement Learning (HRL) is also being investigated as an approach to SRL (Mannucci, van Kampen, de Visser, & Chu, 2017a) (Mannucci & van Kampen, 2016).

1-2 Motivation

With computers becoming more powerful, aircraft more advanced and vehicle control shifting from human controller to AI control, it is vital that more research is conducted into machine learning methods that can keep up with the demand of the complexity of the system they are controlling as well as guaranteeing safety. Therefore the methods and algorithms in this research are developed with intelligent flight control or navigation in mind. In early development, these methods will be developed for autonomous navigational tasks of UAVs, and provide a starting point or inspiration for further research into more complicated aircraft control design. This report will present some simplified applications for flight control as proof of concept and discuss the possibilities for more complex and real-life applications .

1-3 Research goal

This research will essentially combine the concepts of HRL and SRL while attempting to keep agent autonomy as intact as possible. The goal will be to determine the benefits to the agents safety and autonomy by applying methods for ensuring safety on various hierarchical levels. Proving this concept of applying safety methods on hierarchical levels will aid in the development of online autonomous control systems where safety is critical and provide the possibility of learning agents controlling physical systems in the continuous domain. In the future, an advanced autopilot system in control of an airplane in an unknown, yet dangerous state might be able to quickly return the aircraft to a safe state without first exploring dangerous actions.

The overall goal of this research is to develop a UAV navigation control system, based on RL that is both safe and autonomous.

In order to reach the above goal, several sub-questions have been raised to guide the research into fulfilling the goal.

- What is the state-of-the-art in the field of SRL and HRL?
- How do HRL methods compare with flat RL methods in terms of safety?
- How can agent autonomy be maintained?
- How can an algorithm ensure safe domain exploration?
- How can safety algorithms be applied on various hierarchical levels?
- How does relative state observation affect safety and performance?
- How does environment size affect safety and performance?
- How can algorithms be compared to each other in terms of safety, efficiency and optimality?

1-4 Report structure

This report is separated in three parts. Part I contains a scientific article which is a stand-alone report highlighting the main research questions, results and conclusions drawn in this project. Part II contains a preliminary report or literature study performed in advance of the project that the article in Part I writes about. This literature study contains some background information regarding HRL and SRL in Chapters 3 and 4 respectively showing the state-of-the-art in these fields. Some of these approaches have been tested with preliminary results shown in Chapter 5. The methodology of the project as a whole is described in Chapter 6 which is then followed by a summary of Part II in Chapter 7. Part III shows some additional results that help answer some research questions posed above. It begins with Chapter 8 showing a grid-world environment, focused on the difference between relative and absolute state observation. The curse of dimensionality is made visible in Chapter 9 which runs simulations on the same environment of Chapter 8 but with increasing size. Chapter 10 presents supplementary results on another grid-world environment comparing flat RL with HRL agents both of which only have relative state observation. Finally this report as a whole is concluded in Chapter 11

Part I

Article

The Actor-Judge Method: safe state exploration for Hierarchical Reinforcement Learning Controllers

S. Verbist*, T. Mannucci[†] and E. van Kampen[‡]

Delft University of Technology, Delft, Zuid Holland, the Netherlands

Reinforcement Learning is a much researched topic for autonomous machine behavior and is often applied to navigation problems. In order to deal with growing environments and larger state/action spaces, Hierarchical Reinforcement Learning has been introduced. Unfortunately learning from experience, which is central to Reinforcement Learning, makes guaranteeing safety a complex problem. This paper demonstrates an approach, named the actor-judge approach, to make the exploration safer while imposing as few as possible restrictions on the agent. The approach combines ideas from the fields of Hierarchical Reinforcement Learning and Safe Reinforcement Learning to develop a Safe Hierarchical Reinforcement Learning algorithm. The algorithm is tested in a simulated environment where the agent represents an Unmanned Aerial Vehicle able to move laterally in four directions using quadridirectional range sensors to establish a relative position. Although this approach does not guarantee the agent to never explore unsafe areas of the state domain, results show the actor-judge method increases agent safety and can be used on multiple levels an HRL agent hierarchy.

Nomenclature

HAM Hierarchical Abstract Machines.

HRL Hierarchical Reinforcement Learning.

MDP Markov Decision Process.

RL Reinforcement Learning.

SARSA State-Action-Reward-State-Action.

SHERPA Safety Handling Exploration with Risk Perception Algorithm.

SHRL Safe Hierarchical Reinforcement Learning.

SMDP Semi-Markov Decision Process.

SRL Safe Reinforcement Learning.

UAV Unmanned Aerial Vehicle.

I. Introduction

Reinforcement Learning (RL) is an exciting field of machine learning offering many applications in the fields of robotics,¹ wind energy conversion² and Unmanned Aerial Vehicle (UAV) control^{3,4} among others. With used in physical systems that can be damaged from incorrect controlling, it becomes important to focus research on making safer while maintaining its benefits.

Inspired by animal learning behavior⁵ involves an agent performing (sequences of) actions based on its observed state and obtaining a positive or negative reward from the environment. It then adjusts its strategy

*MSc student, Aerospace Faculty, Control & Operations Department

[†]PhD student, Aerospace Faculty, Control & Operations Department

[‡]Assistant professor, Aerospace Faculty, Control & Operations Department

accordingly in order to find the strategy that accumulates the highest total reward. In the absence of any additional information or instructions, flat RL allows an agent to explore every state in its domain to find the optimal strategy. This includes potential unsafe states that the agent may find in its domain, however in order to autonomously explore its state domain the agent is free to find these unsafe states. Of course one option may be to simply remove these undesired states from the state domain of the agent prohibiting it to explore them, however this assumes full knowledge of the designer on every potential unsafe state and this may not always be clear.

In order to develop further understanding into balancing safety, exploration and agent autonomy, the goal of this research is to develop an algorithm that allows an agent to explore an unknown environment with as little prior knowledge about the environment as possible. During this research, an approach is formulated and tested against other methods of state exploration with and without additional safety considerations in the RL framework.

The contribution of this paper is to present the actor-judge framework, building on the knowledge in RL to create safer and better functioning autonomous robots in a world of ever increasing interest in these systems for control.

The organization of this article is as follows. This introduction is followed by a short summary in Section II of the background fundamentals in Safe Reinforcement Learning (SRL) and Hierarchical Reinforcement Learning (HRL). Section III introduces the safe learning approach and explains the experimental setup used for testing the algorithm and comparing it to other approaches. Section II shows some more detail on the environment and agent design as well as presenting the simulation variables that will be used to compare the different approaches. The results of the simulations are shown and discussed in Section V and the article is concluded in Section VI, summarizing the major findings and drawing conclusions.

II. Fundamentals

This section presents the fundamentals of RL. Since much of the working principle of RL relies on the problem being a Markov Decision Process (MDP), this section will begin by explaining this followed by the fundamentals of RL. This section concludes with some background in HRL and SRL.

A. Markov Decision Processes

A MDP consists of the quintet $\langle \mathcal{S}, \mathcal{A}, T, \mathcal{R}, \gamma \rangle$, the states, actions, transition probabilities, rewards and discount factor respectively. When there is a finite amount of states and actions, it is said to be a *finite* MDP. When an agent finds itself in state $s \in \mathcal{S}$ at time t and performs action $a \in \mathcal{A}$, which transitions the agent into state $s' \in \mathcal{S}$ with probability $T_{ss'}^a = \Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$, it receives a reward $R_{ss'}^a = E\{r_{t+1} \mid s_t = s, a_t = a\}$. The total discounted return is then $R_t = \sum_{k=1}^{\infty} \gamma r_{t+k}$ with one-step reward $r = R_{ss'}^a \in \mathcal{R}$. An important rule in an MDP is the *Markov property*, which says that each state is only the result of its immediate predecessor state and the action chosen regardless of the time-history of states and actions.⁵ This property is repeated in Equation 1.

$$\Pr(s_{t+1} \mid s_t, a_t, s_{t-1}, a_{t-1}, \dots, s_0, a_0) = \Pr(s_{t+1} \mid s_t, a_t) \quad (1)$$

This property is assumed to be valid throughout most of the research carried out. Where this property might be in violation is pointed out at the relevant section.

B. Reinforcement Learning

Reinforcement Learning is a branch of machine learning inspired by animal learning behavior.⁵ It relies on experience of past rewards given to an agent by the environment in response to its actions. The goal of the agent is usually to find the action, or sequence of actions that yield the highest total return over the course of an episode. The return is defined in Equation 2 as the sum of all discounted rewards given to the agent by the environment.

$$R_t = \sum_{k=1}^{\infty} \gamma r_{t+k} \quad (2)$$

An RL agent interacts with its environment as displayed in Figure 1. The state s is a particular state of the set of all possible states $\mathcal{S}$. Similarly, the action a of all possible actions $\mathcal{A}$ is a single particular action. The agent is tasked with finding the action $a \mid s$ that yields the highest return.

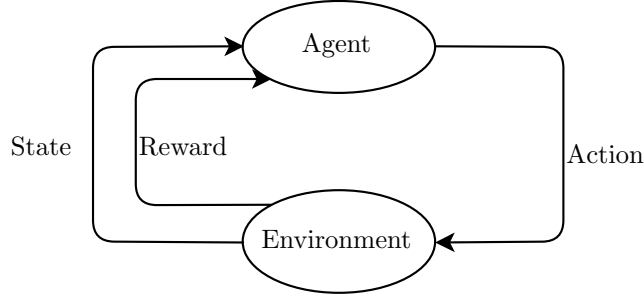


Figure 1: Agent interaction with its environment

After choosing an action, the agent receives a reward from the environment r , which is used to update the *state value function* $\mathcal{V}$ or *state-action value function* $\mathcal{Q}$. This update can be done in a variety of ways such as Q-learning,⁶ Sarsa,⁵ dynamic programming,⁷ Monte Carlo.⁸ An agent's *policy*, $\pi(s) = \mathbf{Pr}(a \mid s_t = s)$ chooses its actions based on these *value functions*. One key distinction between the various approaches is the availability of a model of the environment and the time until value update can be performed. Dynamic Programming requires a full model of the environment to work and Monte Carlo requires an entire episode be run until completion while Temporal Difference methods like Sarsa and Q-learning do not.⁵ The approach in this research is such that the agent does not have a model of the environment and performs value updates after every step and it was therefore chosen to use the State-Action-Reward-State-Action (SARSA) method.

SARSA means that this update only requires the states s_t and s_{t+1} , the actions a_t and a_{t+1} and the reward r . The value update is done as shown in Equation 3 and will be central to the updates performed in this research.

$$\mathcal{Q}(s, a)_{k+1} = \mathcal{Q}(s, a)_k + \alpha(r + \gamma * \mathcal{Q}(s_{t+1}, a_{t+1})_k - \mathcal{Q}(s, a)_k) \quad (3)$$

B.1. Hierarchical Reinforcement Learning

One area of research in this paper is the benefit to safety added by using HRL. HRL is often used as a way to deal with the *curse of dimensionality*⁹ in problems where the state- and/or action spaces $\mathcal{S}$ and $\mathcal{A}$ respectively, become too large to search through efficiently. HRL constructs a hierarchy of sub-spaces that become MDP problems by themselves under the assumption that solving each sub-problem will also solve the whole. An added benefit to this is that an agent can be given *strategies* for solving a problem. This is used in the approach developed in this research and will be explained more thoroughly in Section III.

In principle, HRL allows for an agent navigating in a house, to learn sub-problems such as *exit a room* or *open door*. The root (top-level) RL problem then becomes to use these actions of exiting a room and opening a door to navigate through the rooms without having to learn how to exit each individual room separately.

The main approaches covered in literature are Parr and Russel's *Hierarchy of Abstract Machines* (HAM),¹⁰ Sutton et. al's *Options* framework¹¹ and Dietterich's *MAXQ*.¹² The ideology behind each approach is quite similar such that they each split the main task up into smaller pieces to be solved separately under the assumption that solving them individually, will solve the original task.⁹

B.2. Safe Reinforcement Learning

SRL has only recently been coined by García and Fernández in their survey on safe reinforcement learning.¹³ Their definition of SRL is interpreted as any approach that considers safety and/or risk in RL algorithms. While pursuing safety might not have been the goal in some approaches covered in the survey, it does raise an important concern about RL agents and safety. RL is based on learning by experience and is therefore implicitly unsafe. A unknowing and inexperienced agent will not be able to tell a safe state apart from an unsafe one until it has experienced both of them at least once. The approaches covered in the survey all share in common that they augment the agent in a way that gives it more information, either by adjusting the possible actions the agent may take, or provide a-priori information of the environment. Another approach is to simply perform a (simulated) trial run before committing to a physical system.

While initially a problem solver for the curse of dimensionality, HRL is also being explored as a viable approach to SRL.¹⁴ The benefit is clear when considering the house-navigating agent from the previous section. An agent learning how to navigate one room will also learn that colliding with walls is sub-optimal. The agent can reuse this information avoiding collisions for any other similar room without having been there before which can be interpreted as SRL. Safety can therefore be noticed as an additional benefit of HRL even when it was not the goal.¹⁵

Mannucci et al. developed a type of action filter called Safety Handling Exploration with Risk Perception Algorithm (SHERPA) for RL controllers that discovers fatal states while running and provides backup actions that can steer the system back to safety.¹⁶ This algorithm is designed such that it can be used by an RL controller. The approach covered in this research paper draws inspiration from SHERPA and the various approaches covered in the survey to develop an algorithm that limits the prior knowledge given to the system while also not altering the state or action space of the agent.

III. Agent algorithm design

As explained in Section I, the contribution of this paper is to formulate an approach adding safety to exploration in the RL framework coined the *actor-judge method*. This section begins with a general explanation of the Judge followed by a description of the developed algorithm. In addition, this section describes the agent Hierarchical Abstract Machines (HAM) structure and finally, a general description of a task designed for testing the capabilities of the developed algorithm.

A. Actor-judge method

Taking inspiration from the actor-critic model,⁵ teacher advice¹³ and SHERPA,¹⁷ the 'actor-judge method' is presented. The actor-critic method has shown to improve learning rates by reducing the number of irrelevant states visited.¹⁸ The agent interacts with the judge somewhat similarly to teacher advice where the judge interacts with the agents' proposed actions. Likewise SHERPA proposes a type of 'filter' that checks the proposed action for potential unsafe behavior as a result.¹⁷

A judge has been developed in this research as a combination of these approaches. Like the teacher, it can communicate with the agent and provide limited extra knowledge (or rather, more considerations) and like SHERPA, it behaves more like a 'safety filter'. The judge is a modular addition to the RL agent as can be seen in Figure 2 where the judge has been attached to an actor-critic setup. It receives the proposed action a_p from the agent and if the judge overrules this, returns the judged action a_j . It also passes the selected action to the environment. In the actor-critic method, Figure 2a, the critic receives the reward and updates its value function while the agent chooses actions based on its policy. Note that in the judge-action method, Figure 2b, the judge does not receive the reward from the environment but uses the state information to update its own behavior.

A benefit to this simple setup is that the judge can be created as an entirely modular component of the agent-environment interaction. This is useful on a multilevel machine learning setup such as HRL where safety might be a concern on various levels. In this research, safety is primarily a concern on the primitive level i.e. on the level where actions interact with the environment, but this does not always have to be the case as is demonstrated briefly in Section V where the judge has been applied on the root level concerning itself about the agent's battery level. This modularity is shown in Figure 3.

In the setup presented, the judge is initialized with backup action $a_j \in \mathcal{A}$ where a_j is the action that the judge uses to respond to potentially dangerous situations. Note that the knowledge of these actions violates some autonomy of the algorithm as a whole and a safe action a_j might not always be known. In this research, the existence of this safe action is assumed and considered somewhat analogous to instinctual *fight-or-flight* behavior in animals. In order to maintain autonomy, one might be able to find this action using other forms of machine learning (perhaps something akin to the way animals have evolved their response to dangerous situations).

The working of the judge is shown in Algorithm 1. The judge will belay the agent actions with a probability $\mathbf{Pr}_j(a_t = a_j \mid s_t = s)$ which is initialized with j_0 for every new state. This probability changes based on the occurrence (or not) of unsafe events or states. The probability updates are shown in Equation 4 where ϕ and ρ are the *forgiveness* and *punishment* factors respectively. The forgiveness factor $0 < \phi < 1$ is multiplied with the overruling probability if nothing undesirable occurred or the punishment factor $\rho > 1$

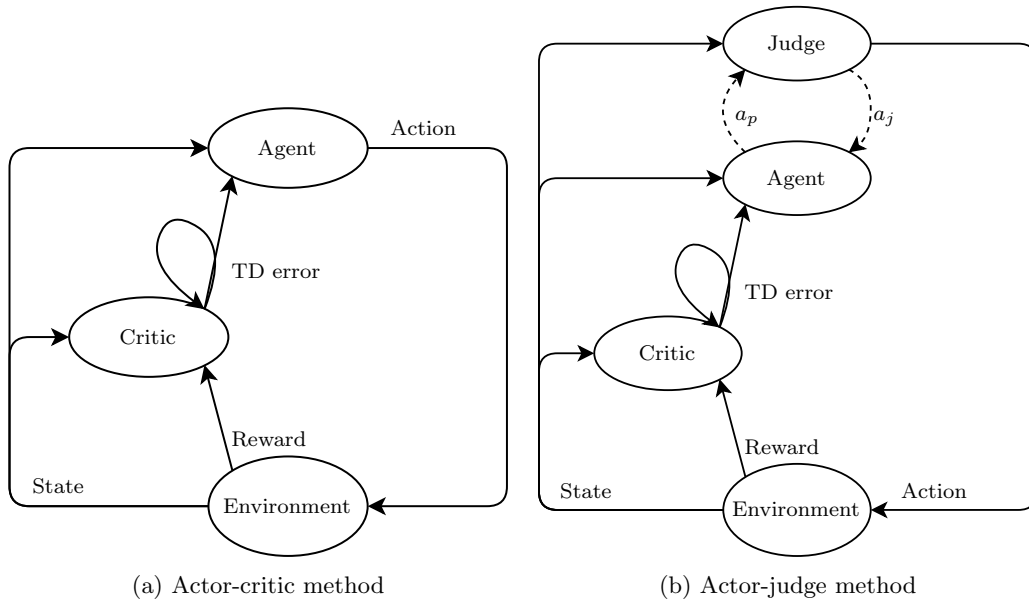


Figure 2: Diagrams of the actor-critic and actor-judge method.

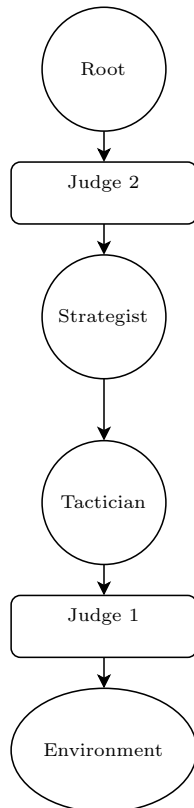


Figure 3: Diagram showing how the actor-judge method can be interjected on various levels in a HRL learner.

otherwise.

It may occur that an agent finds itself m number of times in a particular state s where $m \gg 1$ which would cause the overruling probability for this state to become $\mathbf{Pr}_j(a_t = a_j \mid s_t = s) \rightarrow \lim_{m \rightarrow \infty} \phi^m \cdot j_0 = 0$. The problem here is that when the agent eventually does reach another state, it could be an unsafe one and the agent would then again update the overruling probability as $\mathbf{Pr}_j(a_t = a_j \mid s_t = s) = \rho \phi^m j_0 \approx 0$. This means it will take n amount of unsafe transitions where $n \gg 1$ until $\mathbf{Pr}_j(a_t = a_j \mid s_t = s) = \rho^n \phi^m j_0 \approx j_0$ which is unacceptable in systems where safety is crucial. In order to prevent this, the overruling probability is given a lower bound j . This number can be configured to whatever is needed or be made adaptable based on the task. For the simulations in Section III, this value is fixed $j = 1e-3$. It should also be noted that the judge assumes that, if an unsafe state or event has occurred, it is due to the agent not having performed the backup action. The agent also receives back the judged action and can use this to update its value function internally. The agent uses a reward of $0.5 \cdot r$ to update its value function with the proposed action if this is not equal to the judged action.

```

1 Initialize backup action and unsafe state, action or event ;
2 while Performing the task do
3   Receive state and agent proposed action ;
4   Update judge overruling probability ;
5   if State unknown then
6     Initialize overruling probability  $\mathbf{Pr}_j(a_t = a_j \mid s_t = s) = j_0$ 
7   end
8   if Proposed action is not backup action then
9     Replace agent action with probability  $\mathbf{Pr}_j(a_t = a_j \mid s_t = s)$  ;
10  end
11 end

```

Algorithm 1: General algorithm of the judge

$$\mathbf{Pr}_j(a_t = a_j \mid s_t = s) \leftarrow \begin{cases} \phi \cdot \mathbf{Pr}_j(a_t = a_j \mid s_t = s) & \text{if (safe transition)} \\ \rho \cdot \mathbf{Pr}_j(a_t = a_j \mid s_t = s) & \text{if (unsafe transition)} \end{cases} \quad (4)$$

The actor-judge algorithm is illustrated with an example. Figure 4 shows an agent (circle) in a 3-cell environment with states s_1, s_2, s_3 from left to right as the cells that the agent can occupy. The filled square represents a wall whose cell the agent cannot enter. In episode 1, the agent is initialized in the middle cell s_2 at t_0 . Upon selecting an action, this gets proposed to the judge. The judge receives this action in line 3 of Algorithm 1 followed by an attempt to update the overruling probability of the previous state in line 4. However, because this is the initial state, this part has no effect. The state s_2 is unknown to the judge who assigns it an overruling probability of $\mathbf{Pr}_j(a = a_j \mid s = s_2) = j_0$ in line 6, and in line 9 the agent's action is overruled using this probability.

At time t_1 , the agent is progressed to the next cell s_3 and proposes an action to the judge. After the judge receives this action, it now updates the probability $\mathbf{Pr}_j(a = a_j \mid s = s_2) \leftarrow \phi \cdot j_0$ if this state has been reached safely. s_3 is also unknown to the judge so this is initialized $\mathbf{Pr}_j(a = a_j \mid s = s_3) = j_0$ and subsequently overruled with probability j_0 . Assuming the agent managed to try and move to the right again, this would result in an unsafe action.

Thus at $t = t_2$ the agent's next action is received and the agent's choice at t_1 is punished as $\mathbf{Pr}_j(a = a_j \mid s = s_3) \leftarrow \rho \cdot j_0$. Since the agent has been in s_3 before, line 6 of Algorithm 1 is skipped.

An important assumption for using the actor-judge algorithm is that an agent is allowed to approach unsafe states slowly. The behavior that will be exhibited by the agent will be that unknown states will be treated as unsafe and the judge will overrule the agent's proposed action. However over time, as the agent visits the same state and proposes the same action with no unsafe outcome, the agent will be allowed to perform its original action, potentially resulting in an unsafe state. If the reward function has been created such that this state results in a large negative reward, the agent will be dissuaded to perform this action again (as is usual in RL).

Although the agent has still reached a potentially harmful state, the judge can either postpone this event to the point where the agent has learned optimal behavior safely, or allow the agent to *slowly* approach

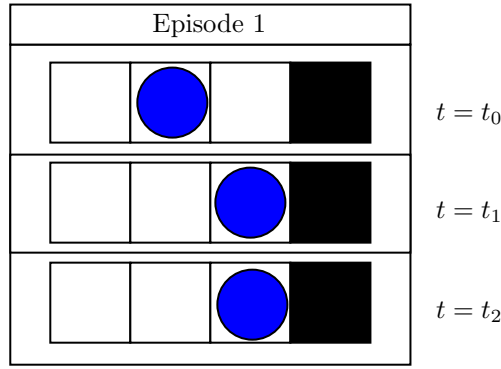


Figure 4: Three time-steps an episode to illustrate the working principle of the actor-judge method.

unsafe regions of its state space. For example a UAV may be allowed to bump into a wall with a low velocity, which is unwanted behavior, but not immediately destructive like crashing full speed instead.

Should there be unsafe actions that will instantly cause irreversible harm to the agent, the actor-judge method as shown above cannot prevent it, neither is it expected to. In this manner, the agent explores its environment while maintaining a level of autonomy that would not be had if the agent was simply forbidden from accelerating towards a wall.

B. Task description

As a means to test HRL as a means of SRL with the added benefit of a judge, a task is designed for the agent to perform. The task of the agent is quite similar to a lot of RL problems: find the quickest path to the goal. The environment in which the agent has to perform its path-finding is one governed by the laws of motion which adds some complexity to the task for the value function updates.

In addition, because safety is the primary concern of this research, the agent is given some way of sensing its environment much like a real UAV. The task therefore does not provide the agent with its absolute position within the environment but instead its position relative to its surroundings with range sensors further adding complexity to the task. Of course, in order to determine safety, it needs to be defined for the task what exactly is ‘safe’. In this navigation task, unsafe behavior is defined as colliding with walls; the higher the speed, the more unsafe it is. While colliding with the wall at any speed is considered an unsafe event, soft collisions will be considered acceptable while the agent explores its environment.

Although traditional RL should eventually learn ‘safe’ behavior with a proper reward function, the goal here is to allow collisions under some speed $\bar{v}_{safe}$ into the wall that the agent may use to learn that obstacle collision is unsafe. A distinction made here is that a traditional RL controller, upon colliding with a wall at some speed, may conclude that the reason for a negative reward was that it wasn’t approaching it fast enough. This is of course an undesirable conclusion on an expensive physical system that can break at high velocity impacts. Note that although in order to test the safety benefits of the actor-judge method it is not needed that there is a goal at all, a goal state is provided to test the impact on the learning speed as well.

C. Agent definition

The actor-judge method was developed as a means of SRL for a HRL controller but the description has been left as a general approach up to this point. In order to encourage safe behavior in an unknown environment with the agent representing a UAV with range sensors, the agent has been constructed as a HRL controller. Preliminary results have also shown the much improved behavior of HRL agents in a task with relative state awareness. This also allows for structuring the hierarchy to incorporate strategies that the agent can utilize to navigate its environment. Additionally, a flat RL agent with this limited positional awareness would violate the Markov property as the next state cannot always be known without the time history of states and actions. HRL with navigational strategies given to the agent would allow for the assumption of Semi-Markov Decision Process (SMDP) where the states for the strategies are temporally abstracted.

The agent’s hierarchy is structured according to Parr and Russel’s HAM¹⁰ and has been designed as shown in Figure 5. The root node at the top can choose strategists to realize high level goals. Each strategist in turn

can select different strategies or tacticians that are also separate machines or temporally extended actions. Finally the tactician selects the primitive actions (tactics) available to it to find the optimal (sequence of) actions to complete the sub-goal of the strategist it is enacted by. It is assumed that if the tactician finds the optimal primitive actions to complete the strategy subgoal and the strategist finds the optimal (sequence of) strategies to complete the sub-goal of the root node, then the root node has optimally completed the overall task.

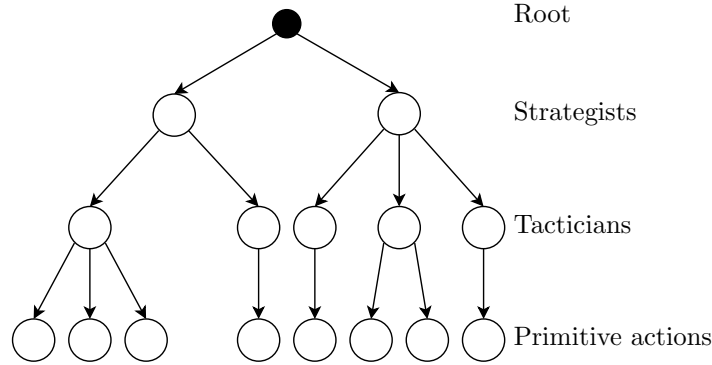


Figure 5: Structure of the HRL controller

The algorithm used to train this HRL agent is shown in Algorithm 2.

```

1 Initialize states, actions and  $Q$ -values ;
2 repeat
3   Perform  $a_{root}$  repeat
4     Perform strategy ;
5     repeat
6       Perform tactic, observe reward and next state ;
7       Select next tactic ;
8       Update tactic  $Q$ -values ;
9     until  $s_{tact}$  is terminal;
10    Observe reward and state ;
11    Select next strategy ;
12    Update strategy  $Q$ -values ;
13  until  $s_{strat}$  is terminal;
14  Observe reward and state;
15  select new root action ;
16  Update root  $Q$ -values ;
17 until  $s_{root}$  is terminal;

```

Algorithm 2: HRL pseudocode algorithm with using Q -value function updates to train an agent

IV. Simulation design and application

Before presenting simulation results in Section V, this section will provide a little more detail on the simulation framework and variables. This section begins with an explanation of the simulation framework that was developed wherein the environment for the task is created. Next is a description of the environment variables that are changed between runs to observe the effect of these variables.

A. Simulation framework

This research has been conducted entirely in a simulated environment, developed and written in the programming language Python. The simulation will consist out of an agent, representing a UAV flying in a two-dimensional environment governed by the laws of motion. The agent controls force vectors in four directions in the horizontal plane in order to move itself.

The environmental updates, i.e. changing the state vector of the agent due to dynamics, happen independently of the agent input. The agent perceives the environment and calculates an action every command interval of T_i seconds. The environment continuously updates the agent position based on the last known agent action using simple equations of motion. The agent actions are translated to force updates which in turn are twice integrated to position using dead reckoning.

The handling of the communication between the environment and agent works as follows; the environment always generates a state for the agent and adds this to a so called *Queue*, which acts like a mailman that can send messages between the agent and environment. This mailman returns every next update and if the state has been accepted, the queue is empty; if not, the state is discarded. Similarly there is a queue for the actions that the agent sends to the environment which the environment considers the agent's command until a new action is sent. It is only at a set interval that the agent sends an action to the environment, which the dynamics equations will use to alter the state of the agent. This has been chosen to more closely relate the simulation to a real world example where; instead of the environment waiting for the agent to deliver an action and processing it, the environment is continuously updating with or without agent input. The flow of the simulation is displayed in Figure 6. In this figure, the agent and environment each have a program 'thread', represented by the dotted line, which shows the processes run on each thread, represented by small red rectangles. What can be seen here is that the environment updates much more frequent than the agent and as such, the agent only uses the state of the environment at the start of the agent process to determine an action. This means that any other state update after the last agent action update and before the last state pickup, is ignored.

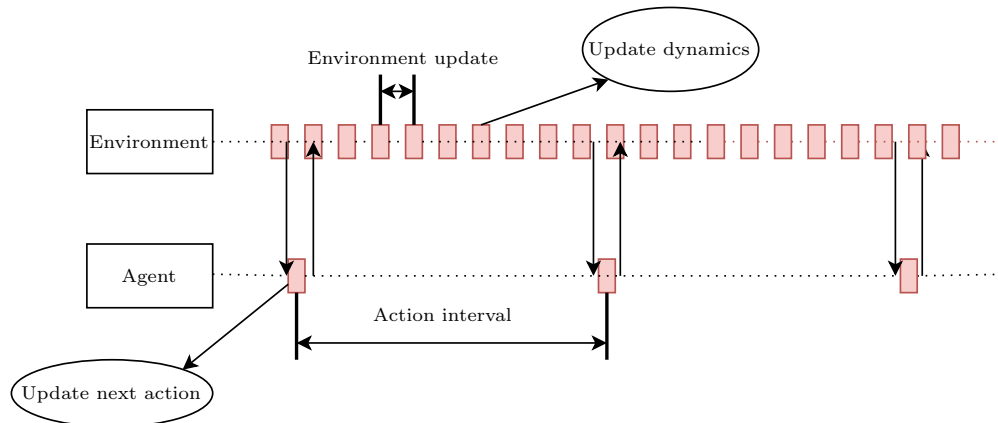


Figure 6: Flow of simulation. The environment gets updated continuously and periodically receives an action from the agent

In this environment, there is also a simple object collision and resolution mechanic that simply causes the agent to 'bounce' off of objects at a reduced speed. In order to determine the effects of safety, it is of course necessary to have a clear definition of safety. In this environment there will be two aspects that are considered 'unsafe'; bouncing into a wall and running out of battery. The goal is to develop an algorithm that will steer the decision process of an untrained agent faster towards actions that will avoid ending up in a bounce or running out of battery while minimally exposing itself to these unsafe states.

An agent performing an action that results in a bounce or finding itself with a depleted battery will yield a reward of -10. Reaching the goal will yield a reward of 0, while all other actions get a -1 reward. The environment is shown in Figure 7 with the agent represented by the circle and the goal in the lower-right as the rectangle with the triangle inside it. The total size of the environment is arbitrarily chosen at 25x25 meters (a decent-sized building interior) and the agent has a maximum speed of 3 m/s.

B. States and Action Space

This section provides more detail of the state and action space that the agent will be allowed to navigate. It is assumed that the agent is a simulated model of a UAV that only has one range sensor in four directions as shown in Figure 8. This is due to the commitment of creating a simulation that more closely resembles a real-world example where absolute position is often unknown in an unknown environment.

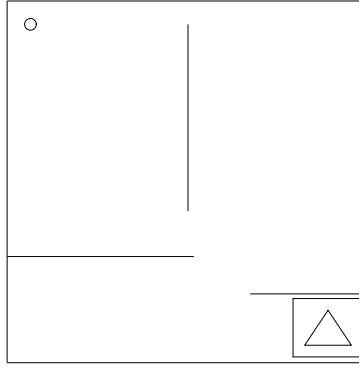


Figure 7: The environment where the agent (circle) has to navigate through. The rectangle with the triangle in the lower right is the goal.

In order to provide faster learning and avoiding the curse of dimensionality, these distances measured by the range sensors are discretized into a number of discrete ranges or sensor resolution. There will be tests with a sensor resolution of 3 where the numerical distance from the range sensor of agent is mapped to discrete values "Far", "Close" or "Critical". In addition to sensor ranges, the state also consists of the agent's velocity in the global x - and y -direction. For when it is relevant, the agent will also track its battery level and distance and direction to recharge point. Note that there will be simulated runs when the battery level is not taken into account to compare the impact on performance when battery is to be considered. When the battery life is of interest, the environment will lower the battery % over time, but maintain it at the lower bound of 10%, providing a negative reward to the agent until the agent has recharged its battery.

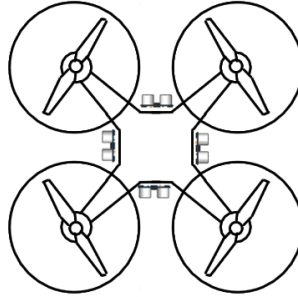


Figure 8: Drone sketch showing the location of the four range sensors

The flat RL and random-action agent will have 4 primitive actions available to them at any given state which will accelerate them in the chosen direction:

- Move up
- Move right
- Move down
- Move left

In the hierarchical setup, the root node at the top, Figure 9a, decides on exploring and learning about its environment, or exploiting a fixed fuel-finding policy by selecting strategists to realize these sub-tasks. The goal of the exploration strategist is to find the shortest path to the goal and it can select temporally extended actions, or strategies that are separate RL machines. These strategies are similarly named as the flat RL primitive actions shown in Figure 9b. However in this case, these strategies are temporally extended actions and will terminate when the agent is stationary near a wall (distance to wall "Critical"). Each strategy is

in turn enacted by a tactician which is again a separate RL machine, however the tactician chooses between primitive actions. For example, the explore strategist wants to enact the strategy 'Up'. The tactician will use the primitive actions (tactics) shown in Figure 9c to find the optimal sequence of actions to the satisfaction of the strategist machine.

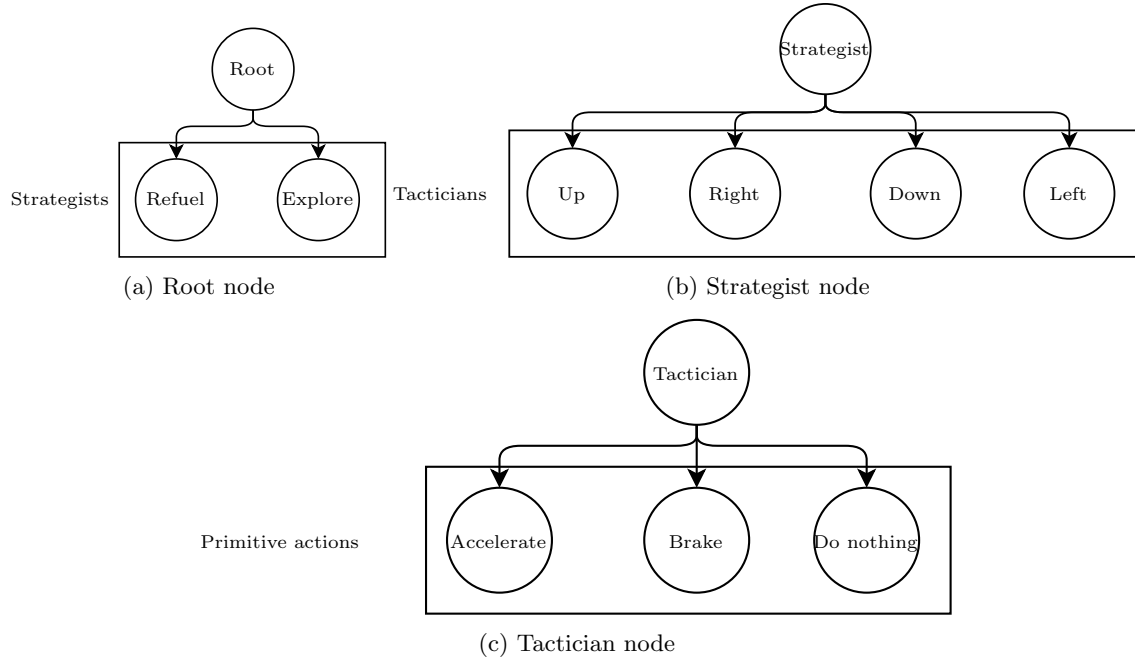


Figure 9: Structure of the HRL controller

C. Simulation Variables

Here some information is given regarding the variables that are set in the simulation. For comparison between approaches, there will be four different actors attempting the task:

- Random action agent
- Flat RL agent
- HRL agent
- HRL agent augmented with the actor-judge method

A common setting is established for all but the random action agent with fixed simulation settings as shown in Table 1. These simulation settings are specific to RL and are thus not relevant to the random action agent. Because randomness is a factor in the decision process of the agent, each simulation run will be performed three times with the same settings to reduce the luck factor. This however excludes the random action agent which will only perform the task once since there is of course no learning at all. The random action agent will also only be tested once as a baseline.

To determine their impact on the results, these settings are changed between simulations. Augmented with the actor-judge method, the Safe Hierarchical Reinforcement Learning (SHRL) agent will be tested several times with various judge settings. The goal is only to determine their effect, not the optimal settings and therefore the settings are only changed a limited amount of times.

The different simulation settings will be tested and compared with each other in the following ways:

- Number of collisions per episode
- Number of total collisions
- Collisions above v_{safe}

Table 1: Baseline simulation settings

Agent policy	$\pi(s, a)$	$\epsilon - greedy$
	ϵ	0.05
Maximum velocity	v_{max}	3 m/s
Maximum ‘safe’ collision velocity	v_{safe}	2.5 m/s
Agent mass	m	0.42 kg
Control force	f	0.25 N
Control input interval	T_i	1 s
Initial judge probability	j_0	0.95
Judge punishment factor	ρ	1.5
Judge forgiveness factor	ϕ	0.9

- Approximate episode until convergence to optimal path
- Number of actions per episode
- Judge interventions per episode

Comparing the difference in the above listed attributes will give an overview how each different approach performs and allows for conclusions to be made regarding their influence on safety and convergence. It is expected at this point, using the random agent as a baseline, that all approaches will perform better than the random agent in terms of faster convergence and more safety. Also it is expected that flat RL will be outperformed by HRL and the developed algorithm as the latter two are given additional tools that will decrease ‘same-state probability’, i.e. the probability that a subsequent state is the same as its predecessor $\Pr(s_{t+1} = s \mid s_t = s, a_t = a)$.

V. Results

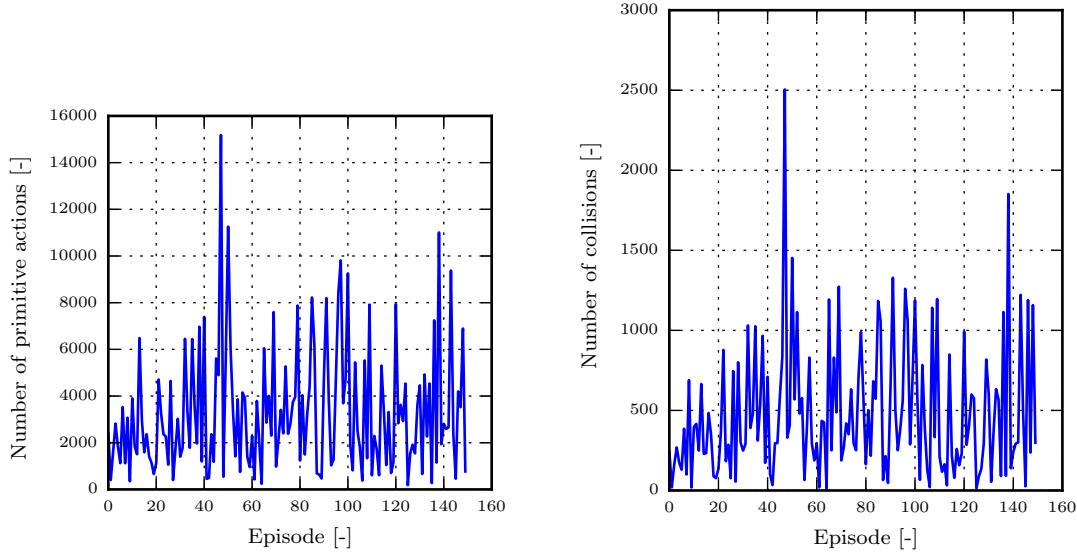
This section shows and discusses the results of the simulations performed according to the setup as described in Section IV. Beginning with an agent performing random actions, a baseline performance is established. Only the number of actions and collisions per episode is recorded here as there is no return for the agent here. Next is the flat RL agent showing a baseline performance for RL to which HRL and SHRL are compared. The section continues with the performance of the HRL agent. Following this with the SHRL agent which also presents a table summarizing the results for a clear overview of the different approach performances. After the different approaches have been presented, the actor-judge method is further investigated by changing judge settings and noting the performance changes. Finally the section concludes with a demonstration of the actor-judge method’s modularity.

A. Random action agent

The task is first attempted by an agent that performs only random primitive actions. This serves only as a minimal benchmark that any algorithm must surely beat and to give an indication also how much better a given algorithm performs. The agent chooses every action interval an action from the primitive action space $\mathcal{A} = \{UP, RIGHT, DOWN, LEFT\}$. This agent collides with the obstacles a total of 71602 times in the entire trial of 150 episodes where an episode lasts from initialization until the goal state has been reached. Additionally it takes the agent an average of ≈ 3278 primitive actions to complete an episode. This trial is of course not expected to show improvement over time but the plots in Figure 10 provide a good starting point to compare the shape of the primitive actions plot (Figure 10a) and collisions per episode plot (Figure 10b) with subsequent RL algorithm performances.

B. Flat RL

Figure 11 shows the result of 3 independent runs of a flat RL agent performing the task. The reward, number of actions and collisions per episode has been averaged and is shown in Figures 11a, 11b and 11c respectively.



(a) Number of actions of the random action agent (b) Collisions per episode of the random action agent

Figure 10: Performance of agent choosing random actions

Additionally, the run with the least amount of collisions is shown in Figure 11d. Each dot on Figure 11d shows both the episode in which the collision occurred and the velocity at which it happened.

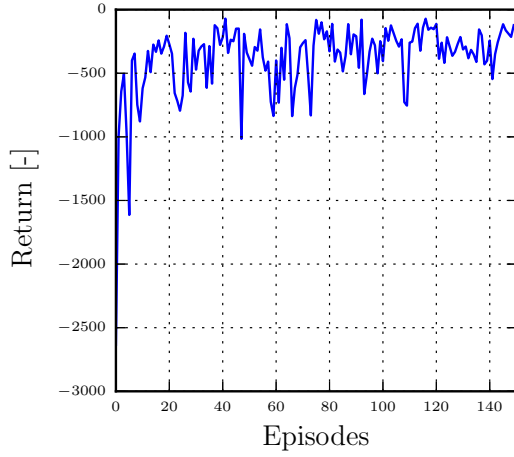
The learner does appear to show signs of converging to an optimal policy as the reward function starts deeply negative but rises towards 0. Also the 'collision density' appears to be decreasing over time. This behavior of the average return per episode is fairly consistent with the behavior of other RL tasks performed in various other literature which confirms the proper implementation of the algorithm and additionally already proves that RL can be applied on tasks where an agent only has state information relative to its surroundings without sacrificing learning.

Shown in Figure 11c, safety is not at all guaranteed for this learner which is not surprising as there was nothing in place to stop this agent from colliding with walls other than a reward function. The agent starts the first few episodes with no regard for its own safety and collides with walls almost on every approach to a wall. This behavior seems to improve over time as around episode 40 and 80, the collision density is lower although near episodes 60 and 110, the agent has reached peaks of collision density. Relapsing back into unsafe behavior such as this may be caused by a reward for collision that is too high; the agent performs a lot of actions that may not result in another state e.g. the agent may be 'far' away from a wall, perform accelerate and still be 'far' the next update. If the agent finds itself in these situations often enough, the reward given to all other alternative actions may become less desirable to the agent than attempting a collide with a wall. Lowering the reward for collision even further will likely help in deterring the agent from selecting actions that result in collision. It does seem that eventually the agent learns from its mistakes and around episode 150, the collision density appears lowest.

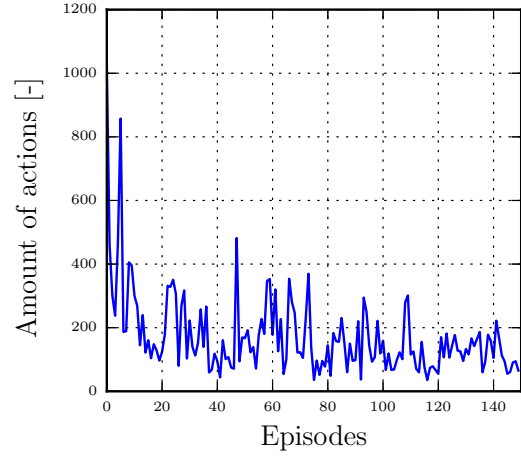
Counting the amount of collisions in Figure 11d will show that the best trial has 3956 collisions of which 724 occurred at speeds between 2.5 m/s and 3 m/s which could represent high enough speeds to cause physical damage. Amongst the three trials of this agent, there was an average of 4380 total collisions per trial. Regardless of its performance, this gives a good baseline to compare other approaches to.

C. HRL controller

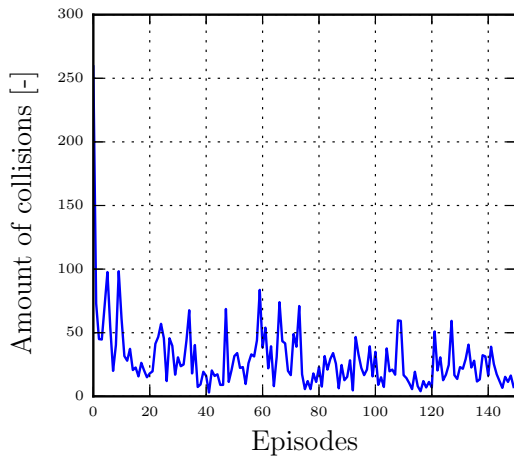
Figure 12 shows the performance of the HRL controller on the task without any additional safety considerations. The first difference compared to the flat RL controller is on Figure 12d showing a lot less density of collisions. However there are still a number of collisions at the maximum velocity which would still be undesirable in a physical system.



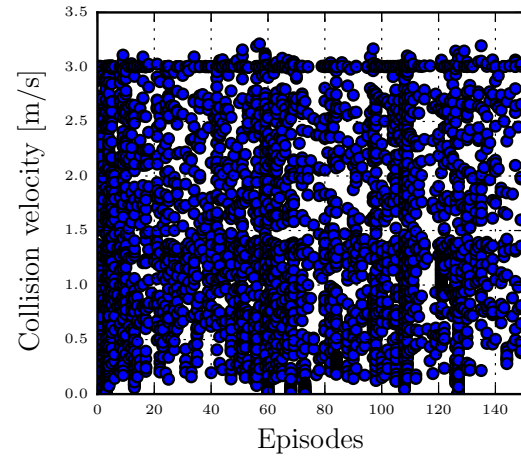
(a) Average reward



(b) Average number of actions



(c) Average number of collisions



(d) Run with the least amount of collisions, showing the velocity of each collision

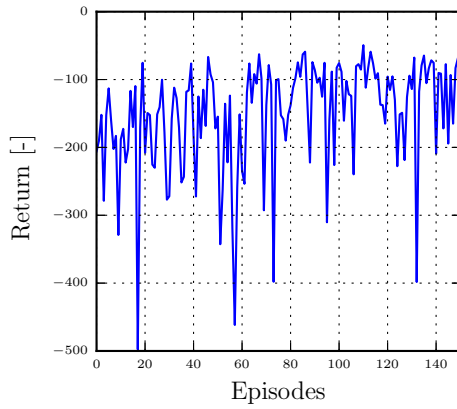
Figure 11: Performance over 3 runs of the flat RL agent performing the task

The average return here is the sum of the reward given at the primitive action level. This gives a good impression of its performance to compare it to the flat RL agent. Looking at Figure 12a, there is significant fluctuations in performance but overall appears to have an increasing trend. Comparing it to the return of the flat RL agent's case, the HRL agent seems to have learned faster, reaching the average return of -200 already within the first few episodes while the flat learner reached this point only after episode 40.

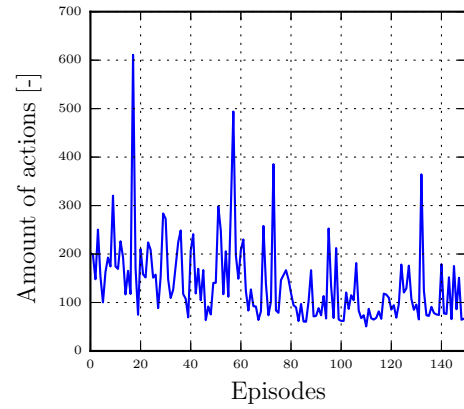
What is more of interest to this research is the comparison between the two in terms of safety. Comparing Figures 11c and 12c with each other, it is clear the HRL learner collides with its environment a significant amount of time less. Tallying the amount of collisions for the HRL learner over the three trials it shows to have reduced the average total amount of collisions from 4380 to 348; a reduction of over 92 %.

Note that these results are averaged over 3 trials each so the possibility of this simply being a 'good' run for the HRL is reduced. This seems to indicate that HRL itself is already quite a promising approach to SRL. Strengthening this argument is Figure 12d, which shows an exceptionally large decrease in collisions. Also seen on this figure is the speed at which these occur and note the decrease of collisions near maximum speed $v \geq v_{safe}$. Between episodes 80 and 150, the amount of collisions at maximum speed has decreased compared to between episodes 1 and 80.

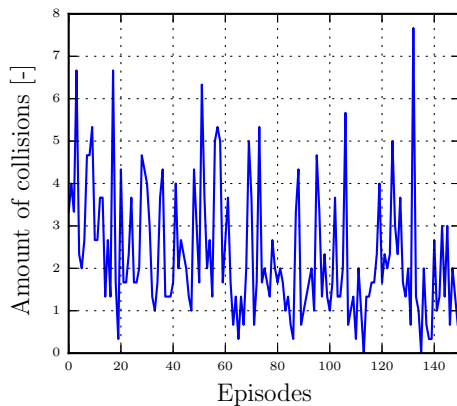
If a navigational control system should be designed for a UAV, it could already be suggested to use this learner as an option and pre-learn the agent in simulations before adding it to a physical system. This of course if few and/or soft collisions are allowed and not immediately destructive to the controlled system. But compared to a flat RL learner it is already a significant improvement.



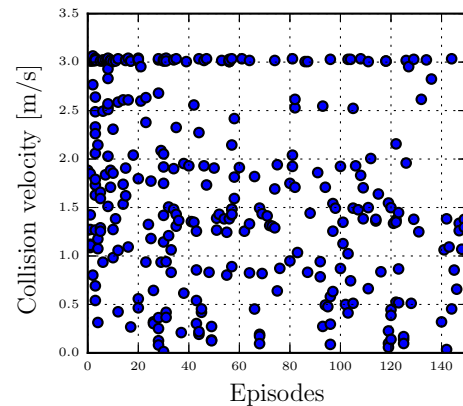
(a) Average reward



(b) Average number of actions



(c) Average number of collisions



(d) Run with the least amount of collisions, showing the velocity of each collision

Figure 12: Performance over 3 runs of the HRL agent performing the task

D. Actor-judge method

The HRL learner is now augmented with the actor-judge method on the lowest level to judge primitive actions. Figure 13 shows the overall performance of this SHRL learner. Comparing the average return over the three trials with that of the HRL learner without this augmentation shows that performance in terms of optimality is not much affected. Learning appears to occur in a similar speed as HRL and learning faster than the RL learner albeit including the same fluctuations in return as with the HRL case.

One significant difference however is shown in the collision density plot in Figure 13c where it can be seen that the density is again much lower than flat RL, and lower still than the HRL. Even more noticeable is the lower amount of collisions near maximum speed $v_{safe} \leq v \leq v_{max}$. The agent is not withheld from exploring its state space or given prior instruction on how to behave in certain conditions, maintaining some level of autonomy but was instead stopped by the judge to explore unknown states too fast. This gives the RL algorithm time to learn the more likely safe actions, while still allowing it to slowly approach a dangerous situation (a wall collision). Once a collision occurred, the agent has already learned more about its safer alternative actions to employ them instead the next time even though it could still try these unsafe actions.

Comparing the total amount of collisions with the HRL agent, the actor-judge method further decreases the average amount of collisions over the three trials to 250 from 348 which is almost a 30 % reduction.

While the addition of this system has not withheld the agent entirely from crashing full-speed into walls, it did reduce the occurrence which indicates a viable approach for safe navigation based on RL. The judge in this case is initialized with starting conditions $j_0 = 0.95$, $\phi = 0.95$ and $\rho = 1.5$ which may have more impact on the safety of the system. Of course a judge with starting conditions $j_0 = 0$ is no different from a non augmented HRL learner so some more experiments are carried out varying these starting conditions to learn their effect on the safety and/or performance.

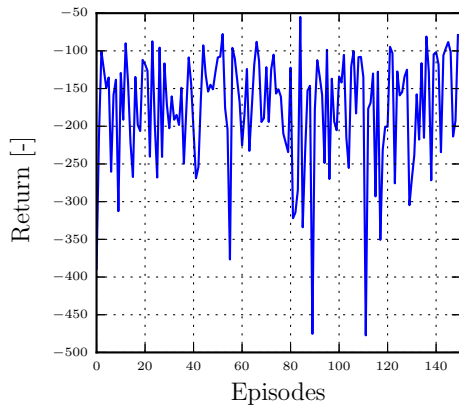
E. Summary of the results

Before looking at different settings for the judge, the results of the previous experiments is summarized in Table 2 to provide a clear overview of the performance. As described in the previous sections, the average amount of collisions is highest for the flat RL agent at an average of 4380 collisions per episode. This drops for the HRL agent to about 348 and further decreases to about 250 for the SHRL agent. What is also seen is that the average return is nearest 0 for the HRL agent however both the HRL and SHRL agents outperform the flat RL agent.

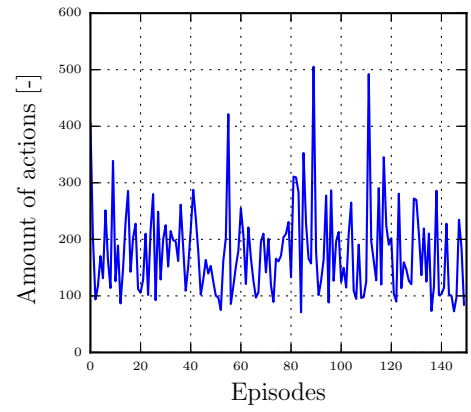
Another point of interest is the amount of collisions where $v \geq v_{safe} = 2.5\text{m/s}$ where v_{safe} is considered a velocity near maximum where collisions become a danger to the integrity of the UAV. Here the absolute number of collisions above this velocity are compared which shows the flat RL agent performing worst and SHRL best. Another way of looking at this is the average ratio of the number of collisions where $v \geq v_{safe}$ over the number of total collisions. Note that this is **not** $\frac{\text{average number of collisions where } v \geq v_{safe}}{\text{average number of total collisions}}$. It is instead the ratio calculated per episode and averaged over the episodes and then averaged over the trials. Doing this calculation gives a better understanding of how many collisions happen at $v \geq v_{safe}$ per collision. For flat RL, this value is 0.22 and drops to 0.13 and 0.11 for the HRL and SHRL agents respectively. What this indicates is that these agents become less inclined to collide with obstacles at higher velocities and thus are both better performing agents in terms of safety than the RL approach. SHRL performs about 15% better on this averaged result than the HRL.

Table 2: Summary of the experiment results for the different agents.

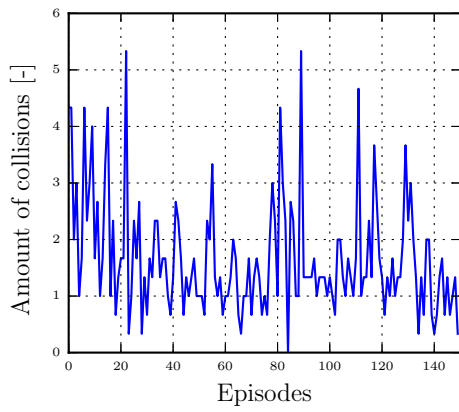
Agent	Flat RL	HRL	SHRL
Average return	-379.2	-151.5	-176.3
Average amount of collisions	4380	348.3	250.3
Average amount of collisions $v \geq v_{safe}$	845.3	70	66.7
Average ratio	0.22	0.13	0.11



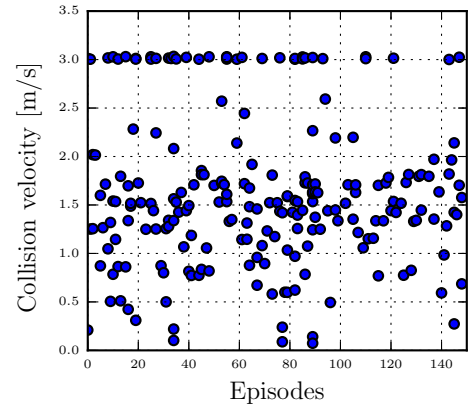
(a) Average reward



(b) Average number of actions



(c) Average number of collisions



(d) Run with the least amount of collisions, showing the velocity of each collision

Figure 13: Performance over 3 runs of the HRL agent performing the task augmented with the actor-judge method

F. Varying judge initial conditions

This section shows the results of various initial judge conditions. The judge settings as they are shown in position 1 of Table 3 are the settings used for the actor-judge agent in previous section. This section will compare other settings to this baseline to quantify the effect of the different judge settings of overruling probability j_0 , the forgiveness factor ϕ and punishment factor ρ . Again three trials are run on the same settings in an attempt to lessen the randomness of performance and the results are averaged in the table.

Table 3: The averaged results over three trials of various different settings of the judge initial conditions

No.	Initial conditions	Collisions	Collided with $v > 2.5$ m/s	Return	Interjections
1	$j_0 = 0.95, \phi = 0.95, \rho = 1.5$	250.33	156	-176.27	43.67
2	$j_0 = 0.95, \phi = 0.45, \rho = 1.5$	247.33	184.67	-152.83	54
3	$j_0 = 0.95, \phi = 0.05, \rho = 1.5$	191	113.3	-127.2	25
4	$j_0 = 0.25, \phi = 0.95, \rho = 1.5$	222.67	140.3	-173.67	138.67
5	$j_0 = 0.95, \phi = 0.95, \rho = 2$	177.3	38.3	-144.9	163.7
6	$j_0 = 0.95, \phi = 0.95, \rho = 4$	192.33	61	-163.83	221.33
7	$j_0 = 0.95, \phi = 0.05, \rho = 4$	348.33	238.33	-151.55	31.33
8	$j_0 = 0.95, \phi = 0.99, \rho = 8$	76.33	10.7	-224.88	606

The first value changed is the forgiveness factor ϕ . Lowering this from 0.95 to 0.45 shows an average of 30 more collisions at high speed, but a better average return. It could be that a lower ϕ lets the agent approach the behavior of flat RL which might explain the higher number of high speed collisions. However when looking at position 3 in the table, which lowers ϕ even further, a pattern only seems to emerge in the amount of total collisions which decreases and in the average return which approaches 0. Settings of position 3 would also explain the low amount of interjections as this agent performs more like a flat RL agent with most states having a very low overruling probability in only a few visits.

Position 4 looks at the difference of changing the initial probability. Comparing it to position 1, the return is hardly affected though the amount of interjections is higher. It is likely that a lower initial probability results in more collisions when approaching an obstacle the first time. This would cause the judge to quickly increase the overruling probability with the punishment factor and thus interjecting more.

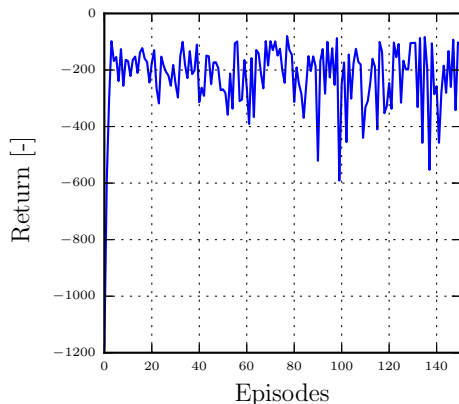
Next the punishment factor ρ is increased to 2 in position 5. The effect this has compared to position 1 seems to be a reduction in amount of collisions. The total collisions went from 250 to 177 while the percentage of high speed collisions to total went from roughly 60% to about 20%. This factor seems to have a strong effect on the safety of the agent. In addition the average return seems to also have improved so the agent has performed better in both safety and in its search for the optimal performance. Unfortunately the benefits of increasing ρ cannot always be relied upon as shown in position 6 where $\rho = 4$. Although the ratio of high speed to total collisions is still better than in the baseline ($\sim 30\%$ compared to $\sim 60\%$), it is worse than in position 4.

Positions 7 and 8 experiment with combinations of ρ and ϕ . Again lowering ϕ shows an increase in collisions and a low amount of interjections and ρ might not be high enough to counter this. After all, a state is ‘forgiven’ whenever nothing negative happens and in the experiment performed, this occurs a lot more than the amount of collisions which result in a state ‘punishment’. Maintaining a ϕ close to 1 and a high ρ results in the safest performance of the tests. Only 10.7 collisions at high speed for the judge settings as in position 8 which is roughly 14% of the total. However the performance in terms of average return is not as good as that of position 1, but this is an understandable outcome considering the high amount of interjections, resulting in more actions being needed to reach the goal and thus a lower return.

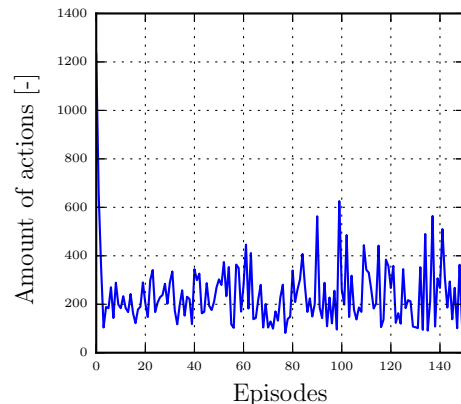
Although not every aspect of this experiment shows a consistent pattern, some conclusions can still be made, namely that the performance depends on both ϕ and ρ dependently. Having high values for both these factors results in less collisions albeit less optimal behavior but this becomes then a trade off between safety and optimality. The settings of these numbers is also quite dependent on the task as the occurrence of unsafe states becomes a factor in how fast or slow the overruling probability changes. For a given task, a conservative approach might be to start with a j_0 and ϕ close (less and not equal) to 1 and a high ρ .

As a demonstration of these settings, the best trial is shown in Figure 14 to compare with Figure 12. In this trial, there have been only 19 collisions of which 5 occurred at speeds over 2.5 m/s. Compared to the

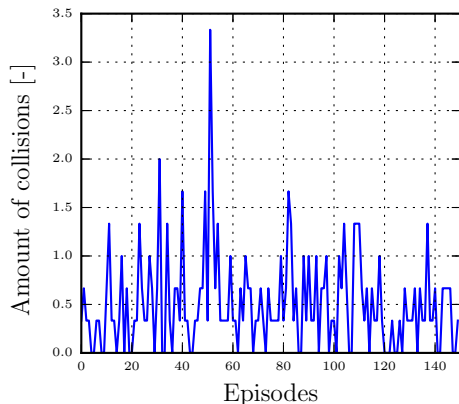
flat RL learner, the amount of collisions have therefore been reduced by 99.5 % in this particular run. Of course this is the best trial of the three, but even compared to the average of 76 collisions, the total collisions have been decreased by over 98% showing clear improvement.



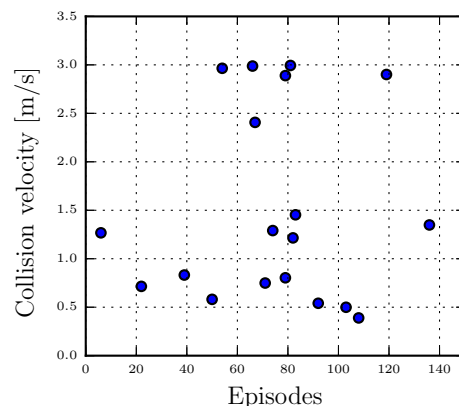
(a) Average reward



(b) Average number of actions



(c) Average number of collisions



(d) Run with the least amount of collisions, showing the velocity of each collision

Figure 14: Performance over 3 runs of the HRL agent performing the task augmented with the judge No. 8 of table 3

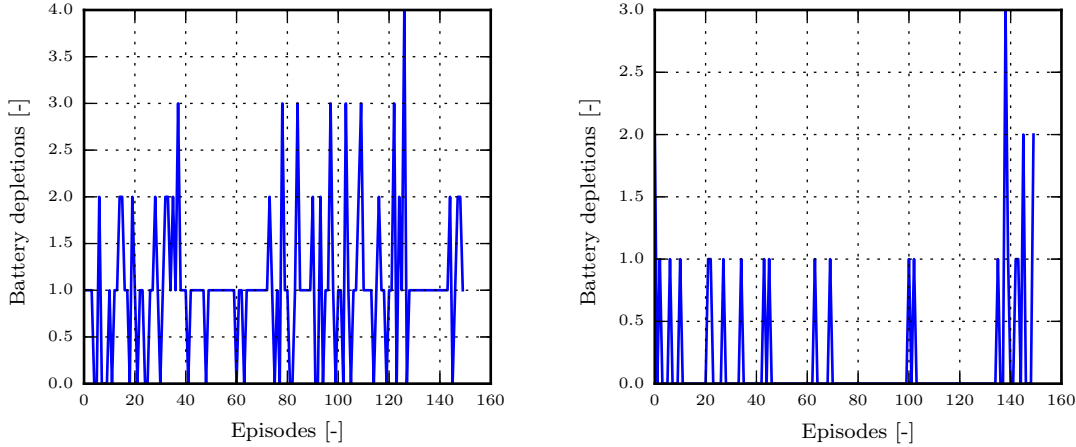
G. Modular implementation of the actor-judge method

In the interest of demonstrating the modularity of the actor-judge method, the SHRL agent is further augmented with another judge at the root level concerned with either recharging the battery or exploring the state space. An unsafe state in this level in the hierarchy is being low on battery. The HRL agent at the root level must decide to either use a fixed policy to recharge its battery or explore its state domain. Choosing the explore strategist, the strategist will select a sequence of up to 5 tacticians to either find the goal or not and hand control back to the root in case of the latter. The judge in this case will overrule the choice of the root with recharging the battery as in this case it's the safe choice.

It is outside the scope to find the optimal settings for this particular judge as the sensitivity of the judge initial settings and subsequent contribution to safety has already been explored. It is assumed at this point that the same kind of sensitivity can be found at different levels.

The results shown here in Figure 15 serve to demonstrate the possibility and give a brief insight on the effect a judge can have when implemented in a HRL learning machine. Comparing Figure 15b with Figure 15a, it is shown that the amount of battery depletions has decreased when the judge can overrule the agent.

Note that an episode is only ended when the goal is reached, a battery is considered ‘depleted’ when it reaches a lower bound of 10%; the agent will at this point be allowed to recharge and continue the episode which explains multiple depletions in a single episode. What the results here show is that the actor-judge method can be applied on a HRL agent and it will have a beneficial effect.



(a) Amount of battery depletions without a judge intervening (b) Amount of battery depletions with a judge intervening

Figure 15: Comparison between judge intervening or not on a higher level in the hierarchy

VI. Conclusion

This research has a multitude of areas to investigate but mainly, it has been agent-safety oriented. Reinforcement Learning (RL) inherently has no additional method in place to avoid states dangerous to agent integrity such a Unmanned Aerial Vehicle (UAV) avoiding to collision against obstacles. This is largely due to RL approaches needing to explore its state space (including unsafe areas) in order to learn what is optimal and safety can largely be considered a side effect.

In this research, safety is the top priority and the effect of hierarchies to RL on safety is investigated using a path-finding task in a simulated environment where wall collisions are to be avoided. This research focuses on a simulated UAV exploring the unknown environment and is given range sensor information and a speedometer to determine its state relative to its surroundings.

A Hierarchical Reinforcement Learning (HRL) agent has a smaller state space to explore than a flat RL agent and therefore this increases the safety of an agent. This is due to previously learned behavior that can be reapplied in other areas of the environment. Results show that HRL has a clear benefit over flat RL in terms of safety which seems to indicate that HRL is already a valid approach to Safe Reinforcement Learning (SRL) even without additional safety backups.

In addition to investigating the benefit of HRL, an algorithm is developed that brings additional safety considerations to the HRL agent. This algorithm has is named the *actor-judge method* as it consists of the RL agent and a so-called judge which acts like a filter on the actions proposed by the agent. The judge is initialized as a system that has a probability to override the agent’s actions. This probability to override is decreased or increased depending on the occurrence of negative events in the next state using a *forgiveness factor* and *punishment factor* respectively.

Although the actor-judge method does not fully eliminate the occurrence of what could be destructive states in physical systems (a UAV colliding full speed into a wall), it can significantly reduce the amount of times this may happen. Both the total amount of collisions are decreased as well as the number of high-speed collisions.

The actor-judge method is not developed as a means to immediately reduce the number of fatal state occurrences to 0 because, without supplying the agent with additional information i.e. policies or outside

information, it is unlikely to expect a model-free, autonomous, from-experience-learning agent to always entirely avoid unsafe states. However it is possible that this simple and modular approach can be rather easily be combined with other forms of safety-concerned approaches to reduce the number of unsafe actions even further.

References

- ¹Guo, Q., Zuo, L., Zheng, R., and Xu, X., *A Hierarchical Path Planning Approach Based on Reinforcement Learning for Mobile Robots*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 393–400.
- ²Wei, C., Zhang, Z., Qiao, W., and Qu, L., “An Adaptive Network-Based Reinforcement Learning Method for MPPT Control of PMSG Wind Energy Conversion Systems,” *IEEE Transactions on Power Electronics*, Vol. 31, No. 11, Nov 2016, pp. 7837–7848.
- ³Bou-Ammar, H., Voos, H., and Ertel, W., “Controller design for quadrotor UAVs using reinforcement learning,” *2010 IEEE International Conference on Control Applications*, Sept 2010, pp. 2130–2135.
- ⁴Junell, J., van Kampen, E., de Visser, C., and Chu, Q., “Reinforcement Learning Applied to a Quadrotor Guidance Law in Autonomous Flight,” *AIAA Guidance, Navigation, and Control Conference*, 2015.
- ⁵Sutton, R. S. and Barto, A. G., *Reinforcement Learning: An Introduction*, MIT Press, 1998.
- ⁶Watkins, C. J. and Dayan, P., “Q-Learning,” *Machine Learning*, Vol. 8, 1992, pp. 279–292.
- ⁷Minsky, M., “Steps toward Artificial Intelligence,” *Proceedings of the IRE*, Vol. 49, 1961, pp. 8–30.
- ⁸Michie, D. and Chambers, R. A., “BOXES: An Experiment in Adaptive Control,” *Machine Intelligence*, edited by E. Dale and D. Michie, Oliver and Boyd, Edinburgh, UK, 1968.
- ⁹Barto, A. G. and Mahadevan, S., “Recent Advances in Hierarchical Reinforcement Learning,” *Discrete Event Dynamic Systems: Theory and Applications*, Vol. 13, 2003, pp. 41–77.
- ¹⁰Parr, R. and Russell, S., “Reinforcement Learning with Hierarchies of Machines,” *Proceedings of the 1997 Conference on Advances in Neural Information Processing Systems 10*, MIT Press, 1998, pp. 1043–1049.
- ¹¹Sutton, R. S., Precup, D., and Singh, S., “Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning,” *Artificial Intelligence*, Vol. 112, 1999, pp. 181 – 211.
- ¹²Dietterich, T. G., “The MAXQ Method for Hierarchical Reinforcement Learning,” *In Proceedings of the Fifteenth International Conference on Machine Learning*, Morgan Kaufmann, 1998, pp. 118–126.
- ¹³García, J. and Fernández, F., “A Comprehensive Survey on Safe Reinforcement Learning,” *Journal of Machine Learning Research*, Vol. 16, 2015, pp. 1437–1480.
- ¹⁴Mannucci, T., van Kampen, E., de Visser, C., and Chu, Q., “Hierarchically Structured Controllers for Safe UAV Reinforcement Learning Applications,” *AIAA Information Systems-AIAA Infotech@ Aerospace*, 2017.
- ¹⁵Mannucci, T. and van Kampen, E., “A hierarchical maze navigation algorithm with Reinforcement Learning and mapping,” *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, Dec 2016, pp. 1–8.
- ¹⁶Mannucci, T., van Kampen, E., de Visser, C., and Chu, Q., “Safe Exploration Algorithms for Reinforcement Learning Controllers,” *IEEE Transactions on Neural Networks and Learning Systems*, Vol. PP, No. 99, 2017, pp. 1–13.
- ¹⁷Mannucci, T., van Kampen, E., de Visser, C., and Chu, Q., “SHERPA: a safe exploration algorithm for Reinforcement Learning controllers,” *AIAA Guidance, Navigation, and Control Conference*, 2015.
- ¹⁸Cetina, V. U., “Autonomous Agent Learning Using an Actor-critic Algorithm and Behavior Models,” *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 3*, AAMAS ’08, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 2008, pp. 1353–1356.

Part II

Literature study

Flat Reinforcement Learning Fundamentals

RL is the basis that will be used in developing algorithms for this research; a thorough understanding is therefore necessary. This chapter is focused on reviewing RL literature and showing the fundamental theory behind flat RL and some common methods of application. In addition, this chapter presents the terminology that will be used throughout this report. The chapter begins with presenting the framework used by most RL systems, the Markov Decision Process (MDP), in Section 2-1. Next, the terminology used to describe MDPs and RL is shown in Sections 2-2, 2-3 and 2-4. In order for a system to measure how positive or negative its current state is, it uses *value functions* which are described in Section 2-5. Lastly, the chapter concludes with an explanation of the working principles of solving an MDP with RL.

2-1 Markov Decision Processes

RL is a method to solve problems through an agent interacting with an environment, and from experience, converge on an optimal agent-environment interaction process. A common mathematical framework to do this in is the MDP. This process consists of the quintet $\langle S, A, T, R, \gamma \rangle$, the states, actions, transition probabilities, rewards and discount factor respectively. These values will be explained in more detail in the following sections. An MDP containing a finite amount of states and actions is referred to as a *finite* MDP. Another feature of an MDP is the *Markov property*, which states that each state is only a result of the predecessor state and the action chosen (Sutton & Barto, 1998). This property is shown mathematically in Equation 2-1.

$$\mathbf{Pr}(s_{t+1} \mid s_t, a_t, s_{t-1}, a_{t-1}, \dots, s_0, a_0) = \mathbf{Pr}(s_{t+1} \mid s_t, a_t) \quad (2-1)$$

What the above equation essentially states is that the entire state and action history $(s_{t-1}, a_{t-1}, s_{t-2}, a_{t-2}, \dots, s_0, a_0)$ of the agent is irrelevant for transitioning to successor state

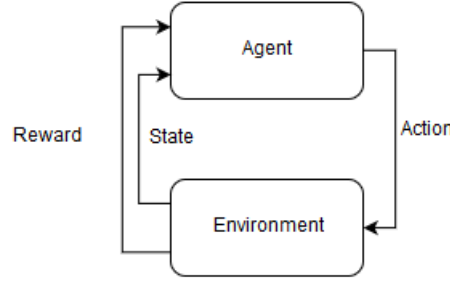


Figure 2-1: Basic Reinforcement Learning setup

s_{t+1} from state s . In this research, only *finite* MDP problems satisfying the Markov property will be considered.

2-2 Agents, Environments and Actions

In this report, the theory of RL will be defined using the terms *agent* for the controllable entity. What exactly the agent is, depends on the problem definition. It could be a simple on-off controller for the lights in a room or a wheeled robot mowing the lawn. Everything that is not the agent, is called the *environment* $\mathcal{S}$, and the state $s \in \mathcal{S}$ is the environment as perceived by the agent. This could be information that the agent receives from its given sensors; often, an agent does not perceive its entire environment and merely chooses actions based on its current observed state. An agent can interact with the environment through performing an *action* $a \in \mathcal{A}$ where $\mathcal{A}$ is the set of all possible actions available to the agent. As an example, a navigation agent could change its position in an environment by performing a *move* action. In flat RL, an agent perceives the observable environment, its state, performs an action and receives a reward from the environment. The basic relationship is shown in Figure 2-1.

2-3 Transition Probabilities and Rewards

When an agent performs an action a_t at time t , its state s_t changes to another state s_{t+1} with *transition probability* $\mathcal{T}_{ss'}^a = \mathbf{Pr}(s_{t+1} = s' \mid s_t = s, a_t = a)$. It is important to note that since the state is only a subset of the environment, the environment may still change even though the observable state does not.

After an agent performs an action at time t , arrives in a new state or after a time step depending on the problem definition, it receives a *reward* r_{t+1} from the environment. The reward is defined as $\mathcal{R}_{ss'}^a = E\{r_{t+1} \mid s_t = s, a_t = a\}$, the expected reward given a state s_t and action a_t . The agent uses this reward to adjust its strategy for solving the problem.

The transition probability can be seen as the system dynamics. If one would simulate a race car controlled by a RL agent, the transition probabilities would be where equations of motion can be used to populate the probability space.

The *reward function* can be chosen arbitrarily and depends on the problem definition. In essence it assigns numbers to certain state-action pairs. An agent given the task to find the highest cumulative reward will want to find the highest sum of reward.

Together, the transition probabilities and reward function are called a *model* which may or may not be available to the agent. If it is available to the agent, it may be used to *plan* actions before committing. Alternatively, a model can be used to train the agent offline in a simulated environment.

2-4 Return and Policies

An agent performs a 1-step action in each state and receives an immediate reward from the environment before advancing to the next state until reaching a terminal state. The total time steps from initialization until the terminal state is called an episode. The sum of rewards over the episode given to an agent is called the *return* and is given by Equation 2-2 where k is the episode length and $0 < \gamma \leq 1$ is called the *discount factor*. This discount factor controls the value of future rewards received by the agent.

$$R_t = \sum_{k=1}^{\infty} \gamma r_{t+k} \quad (2-2)$$

The behavior of an agent is given by its *policy* $\pi(s, a) \rightarrow [0, 1]$, which essentially controls the action chosen by the agent in a given state. This policy may be stochastic or deterministic, assigning probability $\pi(s, a)$ to performing action a while in state s . For a stochastic policy, this probability may be any random value between 0 and 1, whereas in a deterministic policy, the probability of performing action a in state s is exactly 0 or 1.

The goal for a RL learner is to find the optimal strategy π^* that will lead to the optimal return, or maximum of Equation 2-2:

$$\pi^*(s, a) = \arg \max_{\pi(s, a)} \sum_{k=0}^{\infty} \gamma \mathcal{R}_{ss'}^{\pi(s, a)}$$

2-5 Value Functions

In order for the agent to determine whether or not its current state is desirable or which action yields the best outcome, it uses *value functions*. There are two common value functions that calculate the expected return from a given state; the *state-value function* and *action-value function*. The state-value function, given in Equation 2-3, calculates the expected return an agent will receive starting from state s following policy $\pi(s, a)$. The second method is called the *action-value function*, shown in Equation 2-4. Similarly to the state-value function, it calculates the expected return from a given state, but also takes into account the action.

$$V^{\pi}(s) = E\{R_t \mid s_t = s, \pi\} \quad (2-3)$$

$$Q^\pi(s, a) = E\{R_t \mid s_t = s, a_t = a, \pi\} \quad (2-4)$$

The equations for the value functions can be rewritten to obtain the so called *Bellman equations* (Sutton & Barto, 1998) shown in Equations 2-5 and 2-6. These equations calculate, for policy π , the value for each state or state-action pair, given the transition probabilities and reward function for each state.

$$V^\pi(s) = \sum_a \pi(s, a) \sum_{s'} \mathcal{P}_{ss'}^a [\mathcal{R}_{ss'}^a + \gamma V^\pi(s')] \quad (2-5)$$

$$Q^\pi(s, a) = \sum_{s'} \mathcal{R}_{ss'}^a \mathcal{P}_{ss'}^a + \gamma \sum_{s'} \sum_{a'} Q^\pi(s', a') \pi(s', a') \mathcal{P}_{ss'}^a \quad (2-6)$$

The goal of the learner agent is to find the optimal policy π^* such that these value functions become $V^*(s)$ and $Q^*(s, a)$ for the optimal state-value and action-value functions respectively as shown in Equations 2-7 and 2-8.

$$V^*(s) = \max_a V^\pi(s) \quad (2-7)$$

$$Q^*(s, a) = \max_a Q^\pi(s, a) \quad (2-8)$$

Rewriting these equations for the optimal value functions gives Equations 2-9 and 2-10 (Sutton & Barto, 1998). These equations are called the *Bellman optimality equations* for the state-value and action-value functions respectively. These equations calculate the optimal state or state-action values following optimal policy π^* .

$$V^*(s) = \max_a \sum_{s'} \mathcal{P}_{ss'}^a [\mathcal{R}_{ss'}^a + \gamma V^*(s')] \quad (2-9)$$

$$Q^*(s, a) = \sum_{s'} \mathcal{P}_{ss'}^a \left[\mathcal{R}_{ss'}^a + \gamma \max_{a'} Q^*(s', a') \right] \quad (2-10)$$

2-6 Reinforcement Learning Methods

There are a number of methods for solving the Bellman equations, Equations 2-9 and 2-10, of which the more common ones are Dynamic Programming (DP) (Sutton & Barto, 1998), Monte Carlo methods (Sutton & Barto, 1998) and Temporal Difference (TD) (Sutton & Barto, 1998). TD methods are commonly viewed as a combination between Monte Carlo and DP and are used quite often in RL, notably Q-learning (Watkins & Dayan, 1992) and Sarsa (Barto & Mahadevan, 2003).

2-6-1 Dynamic Programming

DP starts an agent in state s at iteration k and considers for each action the possible successor states the agent can advance to. The algorithm calculates the immediate reward gained in every state choosing actions according to its policy and assigns these rewards to the state or state-action value function for their state or state-action respectively. It advances to iteration $k+1$ and does this update repeatedly until the value functions converge to optimality. An update of a single value from $V_k(s)$ to $V_{k+1}(s)$ is called a *backup* because it updates $V_k(s)$ with information of the obtained reward of all subsequent actions. The update is done using Equation 2-11 for the state-value function and Equation 2-12 for the action-value function (Sutton & Barto, 1998). Backing up all the values of the state is called a *sweep*.

$$V_{k+1}(s) = \max_a \left[\mathcal{R}_{ss'}^a + \gamma \sum_{s'} \mathcal{P}_{ss'}^a V_k(s') \right] \quad (2-11)$$

$$Q_{k+1}(s, a) = \mathcal{R}_{ss'}^a + \gamma \sum_{s'} \mathcal{P}_{ss'}^a \max_{a'} Q_k(s', a') \quad (2-12)$$

However the setup of DP does require a full model of the environment for the agent to converge to an optimal strategy. This model of the environment, namely the transition probabilities and reward functions are often not available or incomplete, especially in an unknown environment. Furthermore, because DP algorithms consider every all states and subsequent states for each possible action until reaching a goal state, it is possible for a lot of computing time to be spent performing even a single sweep. It is easy to see in Figure 2-2 how the computing time can quickly explode with even only 2 actions per state due to the fast increasing amount of states to backup.

Because the focus of this research lies with safe online learning, it is more fitting to consider alternative methods that rely on experience rather than considering all possible outcomes. This of course because all possible outcomes include unsafe situations which need to be avoided when learning online.

2-6-2 Monte Carlo

This brings attention to the Monte Carlo method, which relies solely on experience to estimate the value functions and find the optimal policy π^* . The Monte Carlo method uses the averaging of sample episodes to estimate the value functions. Each episode of $n+1$ time steps consists of an agent following a policy π , performing actions $(a_t, a_{t+1}, \dots, a_n)$ until reaching a terminal state. It then stores the reward gained in state s on each backup and after simulating a number of episodes, the algorithm averages the total reward gained for each state over the episodes to converge to the state or state-action values. This process is shown in Figure 2-3 where three episodes are shown starting in some first state, performing a number of steps and each ending in a goal state. Upon reaching a goal state, the learner stores the return given to state S and then averages this return over the three states. This shows the basic working principle of the Monte Carlo method.

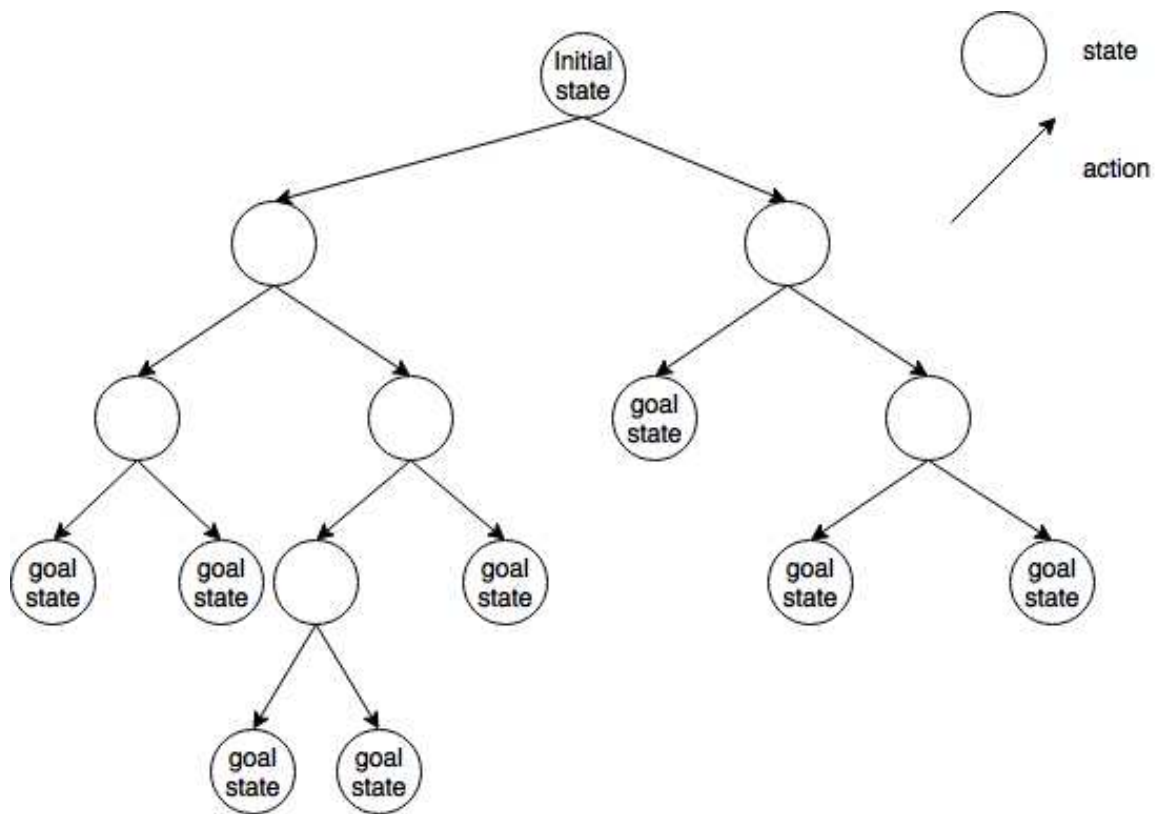


Figure 2-2: Dynamic programming considers every path from the initial state to the goal state to find the optimal strategy

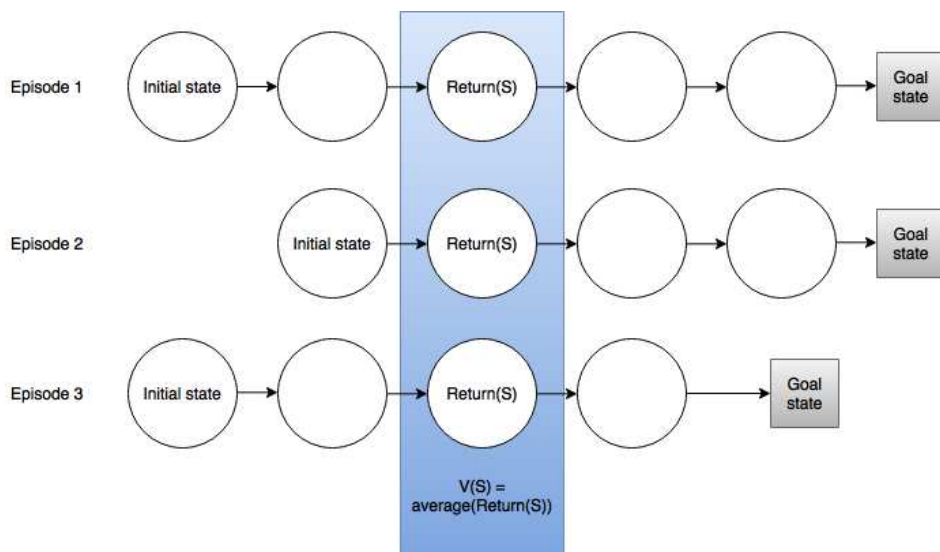


Figure 2-3: Basic principle of Monte Carlo method to estimate state value for state S using the average return of 3 episodes for state S

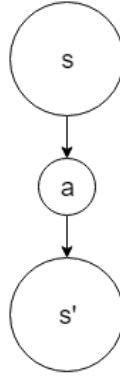


Figure 2-4: Temporal-Difference can update a value for state s using only the information available at t and $t + 1$, namely s, s', a and r

It can be seen that this requires no model of the system and still it can converge to the optimal policies and true value functions if given enough episodes or sample data (Sutton & Barto, 1998). However, a downside of this method is that the system only applies a learning update after completion of an episode. It could therefore be an issue that the system might not explore unknown states enough to learn whether or not they are better alternatives.

2-6-3 Temporal Differences

The most common method to solving value functions and finding optimal functions is TD which combines the ideas of DP and Monte Carlo (Sutton & Barto, 1998). A well-known application is Tesauro's TD-Gammon, which, using Temporal Differences managed to develop a program capable of near expert level play in the game of Backgammon (Tesauro, 1995). TD, like Monte Carlo, relies on direct experience of agent-environment interaction to learn and, like DP, can bootstrap. This means that it is a model-free learning method that can update its value function estimations with other learned function estimates. Whereas Monte Carlo requires episodes to run from an initial state to a goal state in order to update the Q-values, TD methods can already update the value for s while only requiring to be in the next state s' . This can be seen in Equation 2-13.

Common algorithms within TD are *SARSA* (Rummery & Niranjan, 1994) and *Q-Learning* (Watkins & Dayan, 1992). *SARSA* refers to the set s, a, r, s', a' that is required to perform an update as can be seen in Equation 2-13. The simple setup for this method makes it very attractive for implementing in systems. *Q-Learning* looks quite similar to *SARSA* and has enjoyed many forms of implementation (Duguleana & Mogan, 2016) (Wei et al., 2016) and is proven to converge quite rapidly (Watkins & Dayan, 1992). It is shown in Figure 2-4 that TD methods don't require a terminal or goal state to initiate learning.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \quad (2-13)$$

Hierarchical Reinforcement Learning

RL suffers from what is known as the *curse of dimensionality*. Because a MDP consists of all possible states and actions available to the agent, there are $\mathcal{S} \times \mathcal{A}$ combinations of state-action pairs. When using the action-value function for example, an agent would need a memory large enough to accommodate the value of each state-action pair. Should the state or action space increase, the total memory needed to store all the values for each state-action pair will quickly explode as can be seen in the following example.

Consider an MDP consisting of $\mathcal{S} = \{s1, s2\}$ and $\mathcal{A} = \{a1, a2, a3\}$, where the total amount of state-action pairs is $\mathcal{S} \times \mathcal{A} = \{s1 \cdot a1, s1 \cdot a2, s1 \cdot a3, s2 \cdot a1, s2 \cdot a2, s2 \cdot a3\}$. A learner agent using Q-Learning would need to keep track of the Q-values for all six of these combinations. If just one additional state, $s3$, is added, then the agent would have to keep track of 9 Q-values. In systems where there are much more states and actions, this rapid increase could potentially cause the system to run out of memory.

To lessen the memory requirement for larger state or action spaces, HRL methods have been developed as a solution (Barto & Mahadevan, 2003). For example the *MAXQ* (Dietterich, 1998), *Hierarchy of Abstract Machines* (Parr & Russell, 1998) and *options* (Sutton, Precup, & Singh, 1999) methods are among the often used (Barto & Mahadevan, 2003) (Hengst, 2011). This chapter is dedicated to reviewing the working principles of the state-of-the-art HRL methods and applications.

In this report, a HRL problem is defined as a control task that is decomposed into a hierarchy of subtasks where RL algorithms are integrated in a multilayer machine learning routine. This includes tasks where the highest-level control task is posed as a RL problem, such as UAV being tasked with finding a particular person on a field, but the lower-level controlling of performing primitive actions is performed by an artificial neural network. Alternatively it could be where each individual subtask is a RL problem or any combination which includes as the principle learning system RL methods. The control hierarchy is often imposed on the system by the designer or programmer.

3-1 Semi-Markov Decision Processes

Much of the theory covering HRL problems requires an extension of the definition of a MDP (Barto & Mahadevan, 2003). In a MDP, an agent would at time t and in state s perform an action a and subsequently arrive in state s' at time $t + 1$ with probability $\mathcal{T}_{ss'}^a$. The entire state and action history $(s_{t-1}, a_{t-1}, s_{t-2}, a_{t-2}, \dots, s_0, a_0)$ before time t is not relevant to the process due to the Markov property.

Now the process is extended in that an action may require a finite positive non-zero stochastic amount of time steps. In other words, an agent performs temporally extended action a at time t and arrives in state s' at time $t + 1 + \tau$ where $\tau > 0$. Under these conditions, the MDP becomes known as a *Semi-Markov Decision Process (SMDP)*. Using this extension, the Bellman equations given in Equations 2-9 and 2-10 can be the same as Equations 2-11 and 2-12 (Barto & Mahadevan, 2003) repeated below.

$$V_{k+1}(s) = \max_a \left[\mathcal{R}_{ss'}^a + \gamma^\tau \sum_{s'} \mathcal{P}_{ss'}^a V_k(s') \right]$$

$$Q_{k+1}(s, a) = \mathcal{R}_{ss'}^a + \gamma \sum_{s'} \mathcal{P}_{ss'}^a \max_{a'} Q_k(s', a')$$

3-2 Hierarchical Reinforcement Learning Basic Principles

There are various approaches to solving HRL problems (Barto & Mahadevan, 2003). It is possible for a RL task to be subdivided into multiple tasks where a higher level controller needs to achieve an abstracted state or goal (e.g. exiting a house) where lower level controllers control the non abstract primitive actions of a system (e.g. move 1 step forward). Although technically moving the system 1 step in a direction can also be seen as the abstract action of powering actuators on the physical system to achieve the movement, for the sake of this research, a primitive action is the lowest level control that the controller can perform. In a simulated discrete environment, it can be assumed that if each primitive action performed is an instantaneous transition from s to s' and this transition is only dependent on s , then this is an MDP subject to the Markov property.

However for the higher level controller, the Markov property may be violated such that it no longer is a MDP but instead a SMDP. Should a higher level controller try to move a mobile agent out of a room using temporally abstracted action **exit room** in state s_t , it may require multiple primitive actions to accomplish this as can be seen in Figure 3-2. The agent in this example is shown in Figure 3-1 along with its hierarchy and action space $\mathcal{A}$. In this example, the agent (blue circle) can perform temporally extended action **exit room** to exit the room to the right, however at least two primitive actions **move right** must be performed before the agent receives the reward for having performed the **exit room** action. Such that when the agent has eventually left the room, the update to the action-value function for the action **exit room** is no longer dependent on only the next state, but on all states following s_t until exiting the room. A solution is to treat the system as an SMDP.

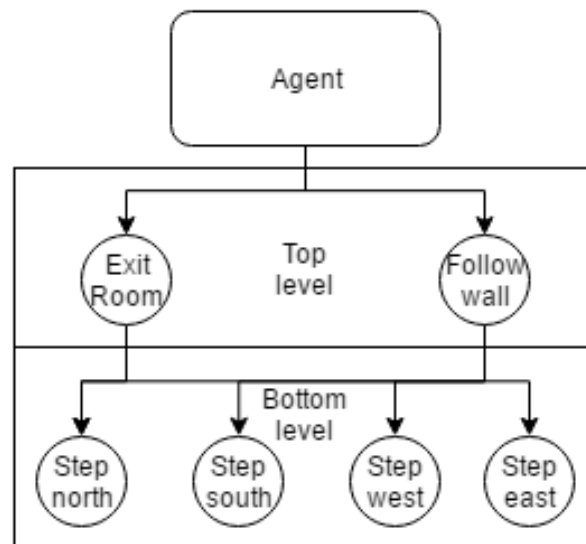


Figure 3-1: Diagram showing an example of a basic HRL system. A top level controller can invoke a lower level controller that in turn invokes primitive actions.

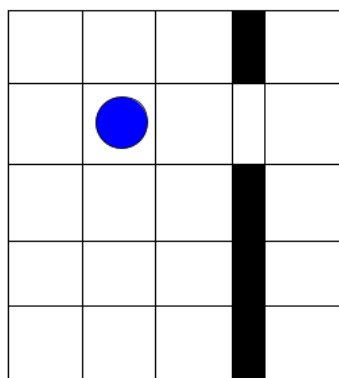


Figure 3-2: The agent (blue circle) is tasked going from the left room to the right room, separated by the thick black line.

Treating a task as a SMDP allows lower as well as higher level controllers to follow RL methods for finding optimal solutions. Examples that use this method are *options*(Sutton et al., 1999), *Hierarchical Abstract Machines (HAM)*(Parr & Russell, 1998) and *MAXQ*(Dietterich, 1998).

In this research, also control tasks combining other forms of machine learning are considered. For example it is entirely possible for a lower level controller considering primitive actions to be a RL agent while the higher level controllers considering temporally abstract actions come from other forms of machine learning. Examples of this are a hierarchical system incorporating both supervised and unsupervised neural networks with RL (Hu, Sasakawa, Hirasawa, & Zheng, 2007) and combining the A* search algorithm with RL for mobile robots (Guo et al., 2013).

3-3 Hierarchical Reinforcement Learning Methods

Reformulating a RL as a HRL can be done in a couple different ways. The seemingly furthest developed methods are called *HAM*(Parr & Russell, 1998), *options*(Sutton et al., 1999), *MAXQ*(Dietterich, 2000). Some literature review is also done towards finding other forms of machine learning combined with RL methods which automatically makes it structured as HRL.

3-3-1 Hierarchical Abstract Machines

A HAM is a type of macro controller that simply maps machine states to subsequent states according to its policy. The controller can be in one of four states; a *choice* state, an *action* state, a *call* state or a *stop* state (Parr & Russell, 1998). These *choice* state lets a HAM program choose between the next machine states. This allows it to select another HAM to select actions according to its policy. Selecting another HAM is done through the *call* state. An invoked child machine can in turn be in an *action* state which relates its state to primitive actions. Once a machine has finished its task, the HAM finds itself in the *stop* state and hands back control to the parent program.

The possibility to structure hierarchies is clear, although this alone does not solve the curse of dimensionality. For this, prior knowledge of the environment is programmed into the system in the form of bounding the action space for each HAM. As an example, in a room-navigation problem, the designer of the system could program a HAM to exit a room and only allowing this machine to find exits using two compass directional movements. The designer could then let another HAM to exit a room using the remaining two compass directions. Thus, each HAM has a reduced search space to learn the optimal policy and their parent HAM has a reduced search space also as it would only need to choose between two machine states.

Having subdivided a single machine learner into a hierarchy of abstract machines, RL can be introduced in the form of *HAMQ*-learning. In this form of learning, the principles of Q-learning are combined with the theory of HAM. A learning agent keeps track of its current observed state s , current machine state m , the next observed state s' and machine state m' , the action a taken at state s , the reward given at s' and discount factor γ . Using this information, the Q update can be structured as in Equation 3-1 (Parr & Russell, 1998). It can be proved that this will converge to the optimal with probability 1 (Parr & Russell, 1998).

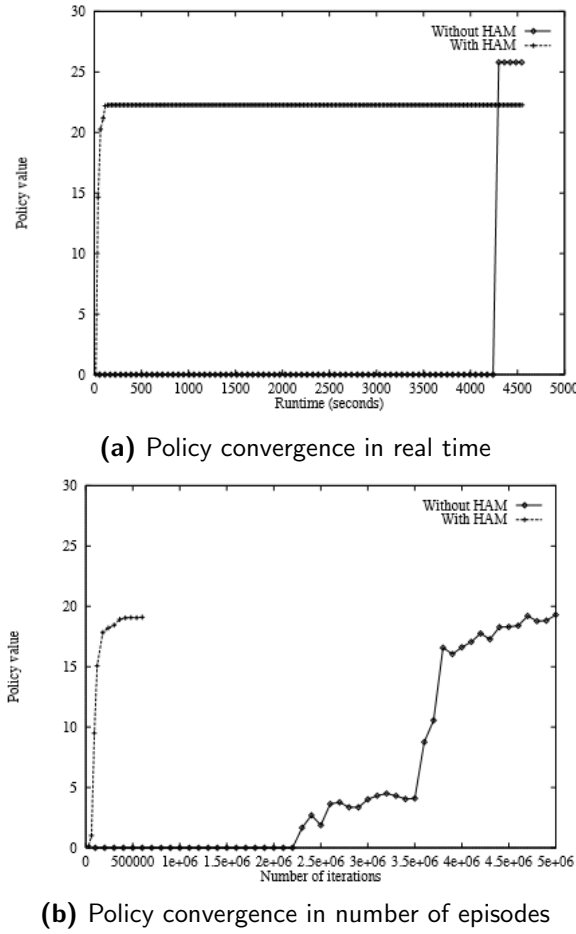


Figure 3-3: Results of Parr and Russel's Hierarchy of Abstract Machines (Parr & Russell, 1998)

$$Q([s, m], a) \leftarrow Q([s, m], a) + \alpha[r + \gamma * V([s', m']) - Q([s, m], a)] \quad (3-1)$$

Parr and Russel tested this technique on their maze, shown in Figure 3-4; a grid world with roughly 3600 states. Constraining the system to choose between the temporally extended actions **follow wall** and **back off**, which in turn enact the primitive movements in four compass directions, allowed it to find the goal state much faster than it would have had it been a flat RL system as can be seen in Figure 3-3 (Parr & Russell, 1998).

It can be seen that HAMs can indeed lessen the curse of dimensionality, however it comes at the cost of sacrificing some autonomy of the system. As more knowledge is provided at initialization, the less the system 'learns' on its own, hence reducing its level of autonomy. This type of HRL can be relevant for research into integrating safety with RL if HAM machines are trained for danger avoidance offline. In combination with the fact that the hierarchy of the abstract machines is structured beforehand, an integration of the two seems logical.

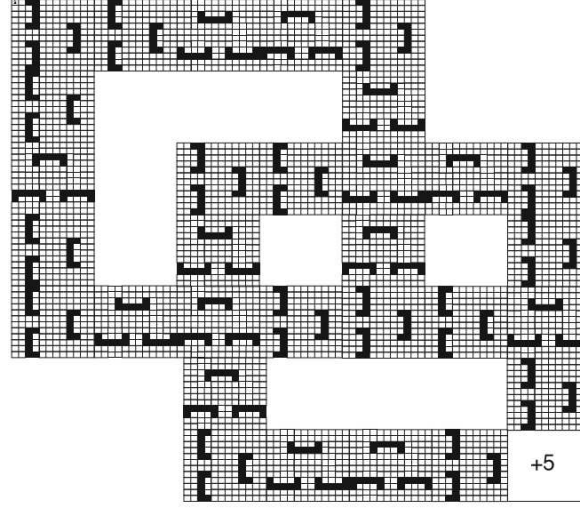


Figure 3-4: Parr and Russel's maze for a hierarchy of machines to navigate from the initial state in the top left to the goal state in bottom right (Parr & Russell, 1998)

3-3-2 Options

The options framework seems a natural extension to RL. Where a flat RL agent can only choose primitive actions, the action space of a learning system following the options framework is generalized and extended to include temporally extended actions which aim to realize more abstract goals. An option consists of the triplet $\langle I, \pi, \beta \rangle$ where $I \subseteq \mathcal{S}$. The option is only available in state s if $s \in I$ and chooses actions according to policy π until the option is terminated according to the termination probability $\beta : \mathcal{S} \rightarrow [0, 1]$ (Sutton et al., 1999).

Q-learning can be used to find the optimal policy to follow. The action-value function is slightly altered to allow for the admittance of temporally extended actions as well as primitive ones. The definition for the *option-value function* is shown in Equation 3-2 (Sutton et al., 1999). In this equation, $Q^\mu(s, o)$ is the option-value of choosing option o in state s and thereafter following policy μ . $E(o\mu, s, t)$ denotes the execution of option o until it terminates and thereafter following policy μ at time t in state s (Stolle & Precup, 2002).

$$Q^\mu(s, o) = E \{ r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots \mid E(o\mu, s, t) \} \quad (3-2)$$

The method has been implemented on a room-navigation learner system in which the task was to reach a goal state in as little steps as possible (Sutton et al., 1999). The environment is shown in Figure 3-5 with two tested goal state locations. It can be seen in this figure that the primitive action space $\mathcal{A}$ consists of the one-step actions moving in four compass directions, while there are also eight options (two for each room because an agent can leave each room in two ways) $h_i \in \langle$ where $i = \{1, 2, \dots, 8\}$ with $\langle$ as the room-leaving option space. A typical room leaving option policy is shown in Figure 3-6.

Two different goals were tested, one being in the hallway between two rooms and the other within a room. A comparison is made between allowing the agent to choose only primitive actions or only room-leaving options for both goals. Looking at Figure 3-7, the increased learning speed becomes clear to reach goal G_1 . For goal G_2 however, the agent cannot

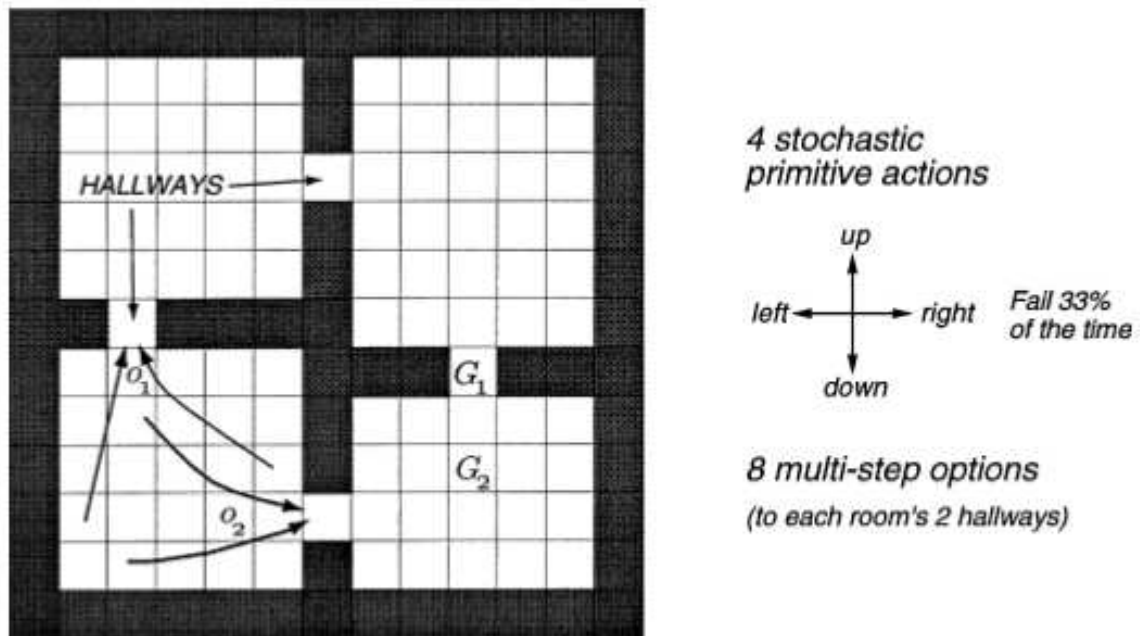


Figure 3-5: Learner environment for the options method where G_1 and G_2 are the two tested goal states and two possible room-leaving options are depicted as o_1 and o_2 (Sutton et al., 1999)

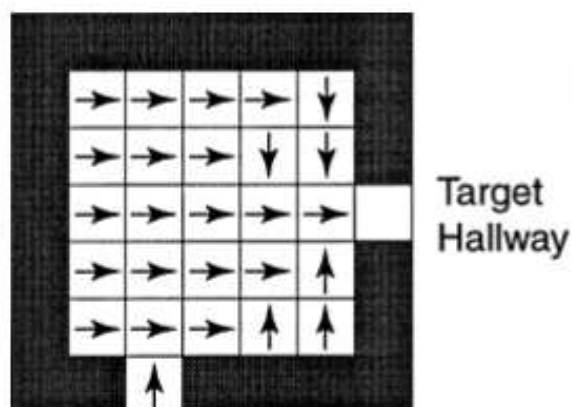


Figure 3-6: Typical room leaving option policy (Sutton et al., 1999)

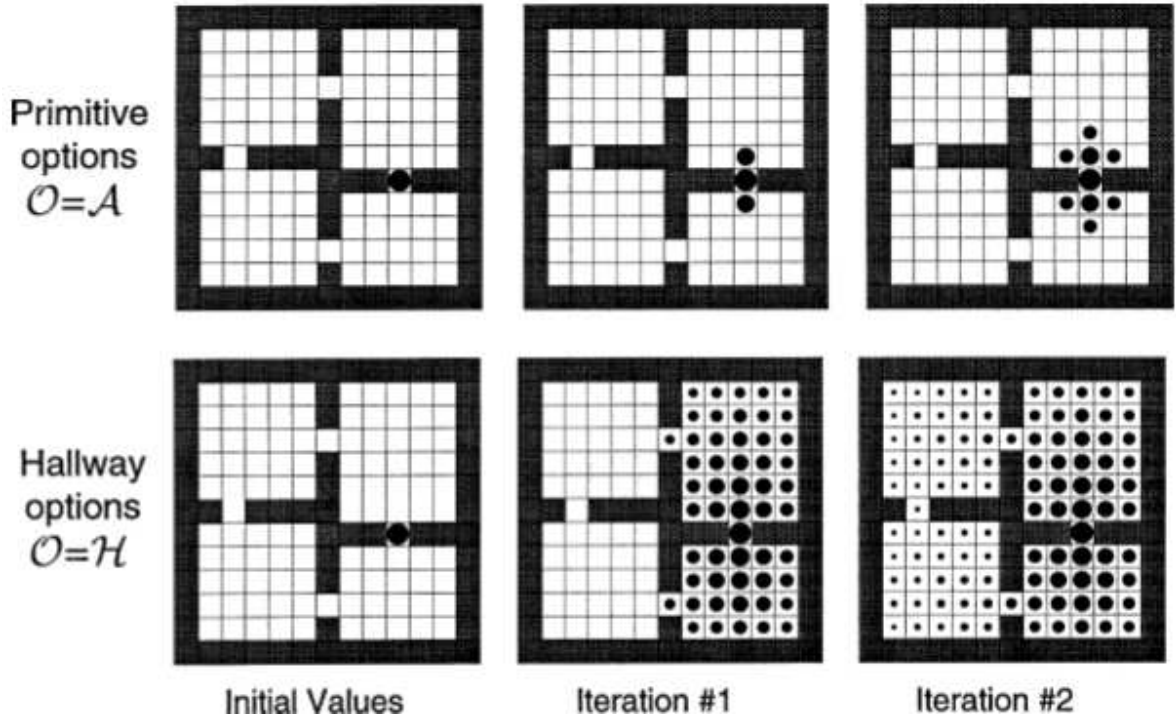


Figure 3-7: Comparing choosing primitive actions with choosing room-leaving options (Sutton et al., 1999)

converge to the optimal policy using only room-leaving options. This is because the goal is not in a hallway and none of the options allow for finding the optimal sub-goal within a room. In order to find the solution, the option space $\mathcal{O}$ is chosen to consist of both of the primitive action space $\mathcal{A}$ and room-leaving options $\mathcal{H}$. Doing this lets the learner converge like shown in Figure 3-8. The results of this method are shown in Figure 3-9 which clearly shows that in the case of G_1 , the room leaving option space $\mathcal{H}$ converges the fastest while the combination option space $\mathcal{O} = \mathcal{A} \cup \mathcal{H}$ converges the fastest to goal G_2 .

This method is quite similar to HAM in that an option, like an abstract machine can initiate a temporally extended action. Once an option is initiated, it follows the assigned policy by initiating further options or primitive actions until completion. Once an option is terminated, control is passed back up the hierarchy. Also like the HAM method, knowledge of the environment can be programmed into the learning system by the designer to structure the subtasks such as option 'open door'. However, unlike HAM, the exact hierarchy of the system can be left stochastic in the sense that all options are available to the system and can be activated as long as $s \in I$. In HAM it is up to the system designer to define what choice states are available in a machine state at a given time.

3-3-3 MAXQ

The *MAXQ* HRL framework works rather different than the options or HAM methods in that the MDP is separated into a hierarchy of smaller (*semi*)MDPs. The overall MDP M is subdivided into smaller MDPs $\{M_0, M_1, M_2, \dots, M_i, \dots, M_n\}$ from a root MDP M_0 attempting

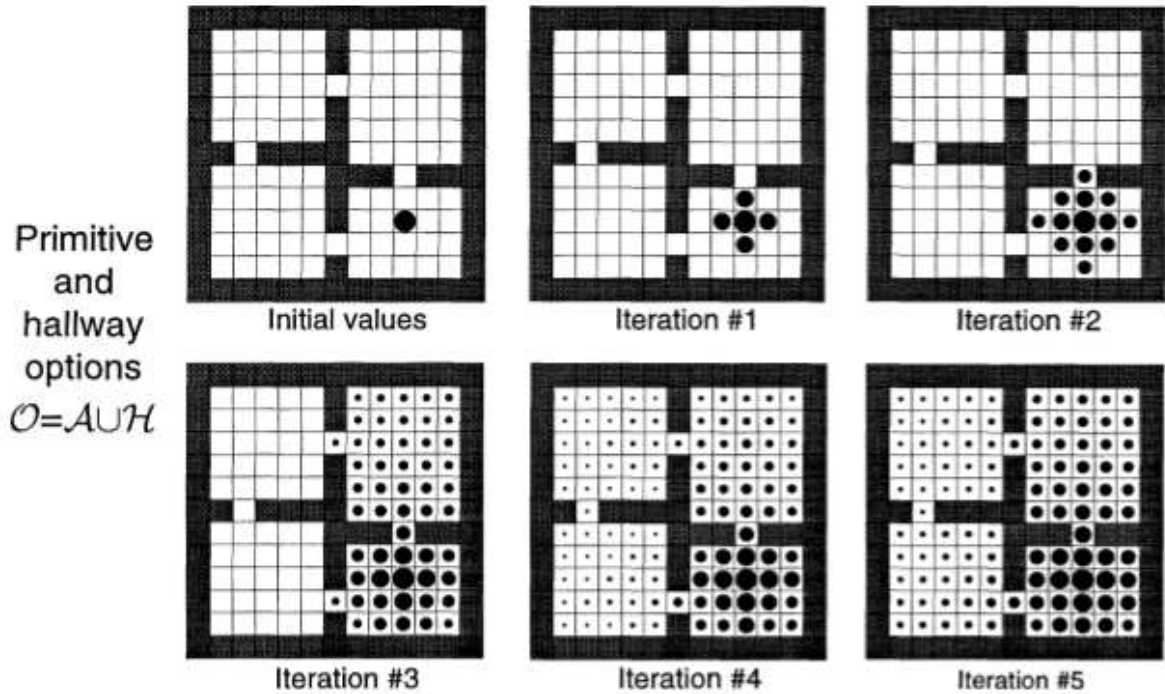


Figure 3-8: Reaching goal G_2 by combining the room-leaving policies with primitive actions space (Sutton et al., 1999)

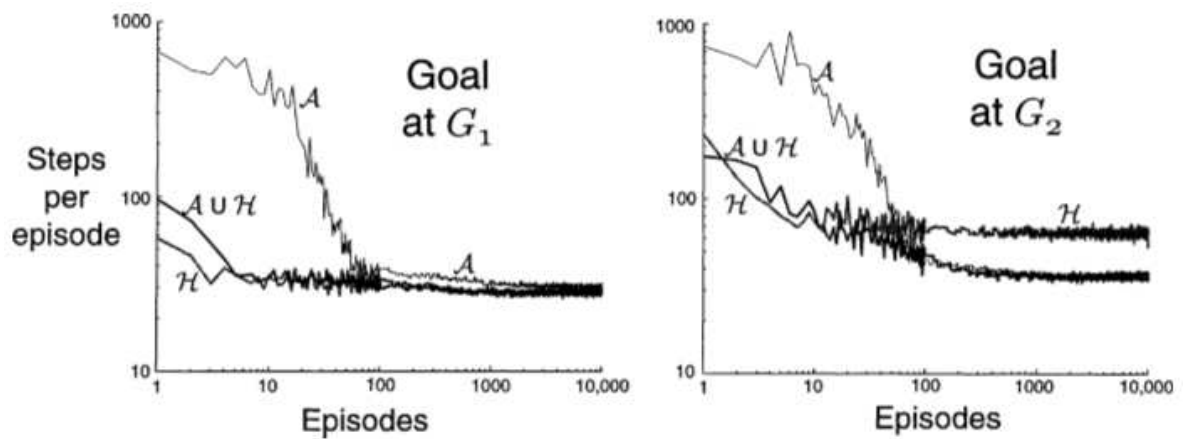


Figure 3-9: Results showing the various techniques reaching the goal states (Sutton et al., 1999)

to achieve an abstract goal (e.g. make breakfast) down to the lower MDPs M_i performing primitive actions. Similarly to options and HAM, the abstract goals are achieved by activating temporally extended actions which requires the system to be viewed as a SMDP. The bigger difference is that, since every subtask is a MDP or SMDP, they each have their own value functions and a more isolated learning process. A large benefit is that learned policies can be reused by similar subtasks which greatly reduces learning time (Dietterich, 1998).

Like options and HAM, Q-learning can be used to solve the value functions and find an optimal hierarchical policy $\pi = \{\pi_0, \pi_1, \pi_2 \dots \pi_n\}$, that is, a set of policies for each subtask. The equation for the action-value function is however, formulated slightly different than before. For the MAXQ framework, Equation 3-3 is used. The *completion function* $C_i^\pi(s, a)$ is given in Equation 3-5 and is the cumulative reward received for completing the subtask belonging to MDP M_i .

$$Q_i^\pi(s, a) = V_a^\pi(s) + C_i^\pi(s, a) \quad (3-3)$$

where

$$V_a^\pi(s) = \begin{cases} Q_i^\pi(s, \pi_i(s)) & \text{i composite} \\ \sum_s' P(s' | s, i) R(s' | s, i) & \text{i primitive} \end{cases} \quad (3-4)$$

$$C_i^\pi(s, a) = \sum_{s'} P(s' | s, a) V_j^\pi(s') \quad (3-5)$$

MAXQ has been developed by Dietterich and demonstrated its effectiveness by testing it on the taxi problem shown in Figure 3-10 (Dietterich, 1998). In this problem an agent has to transport a passenger from the pickup location to the dropoff location. A passenger originates at one of four locations, depicted as cells R, B, G or Y and needs to be dropped off at either one of these same positions (pickup and dropoff at the same location is allowed). The agent has a certain amount of fuel, which basically limits the amount of steps it can make before terminating the episode and the agent can refuel in cell F allowing it to perform more steps. The hierarchy is defined by letting the agent perform temporally abstract actions *Get*, *Put* or *Refuel*, which in turn navigate to a sub-goal using primitive actions.

Figure 3-11 shows the results of this method compared with flat RL which clearly shows the increased convergence speed.

3-3-4 A* and Reinforcement Learning

The A* search algorithm is an accepted method for optimal path finding and planning (Yao, Lin, Xie, Wang, & Hung, 2010). Guo et al. combine this algorithm with RL to create a hierarchical path planning algorithm for mobile robots (Guo et al., 2013). In their design, the A* search algorithm acts as the top layer for navigation, planning a path from an initial state to a goal state. In order for the mobile robot to reach the goal state, it is tasked to navigate to a series of midpoints along the route to find the optimal path to the goal state. However unlike the HRL methods described in the previous section, these midpoints are found by the A* search pattern rather than selected by the human designer. The RL part of the system in turn finds the optimal path to these midpoints until it reaches the end.

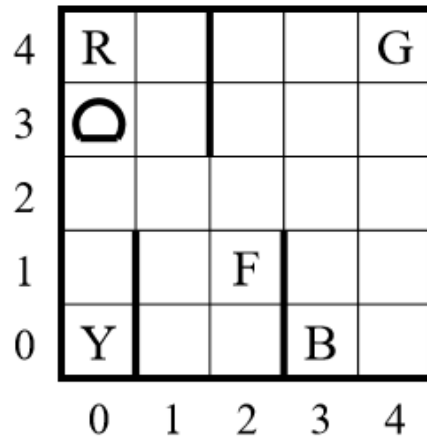


Figure 3-10: Dietterich's taxi problem with the agent depicted by the oval shape and sub-tasks being the transportation of a passenger to a goal state. For a given episode, a passenger and a goal state originate at either the cell R, Y, B or G (Dietterich, 1998)

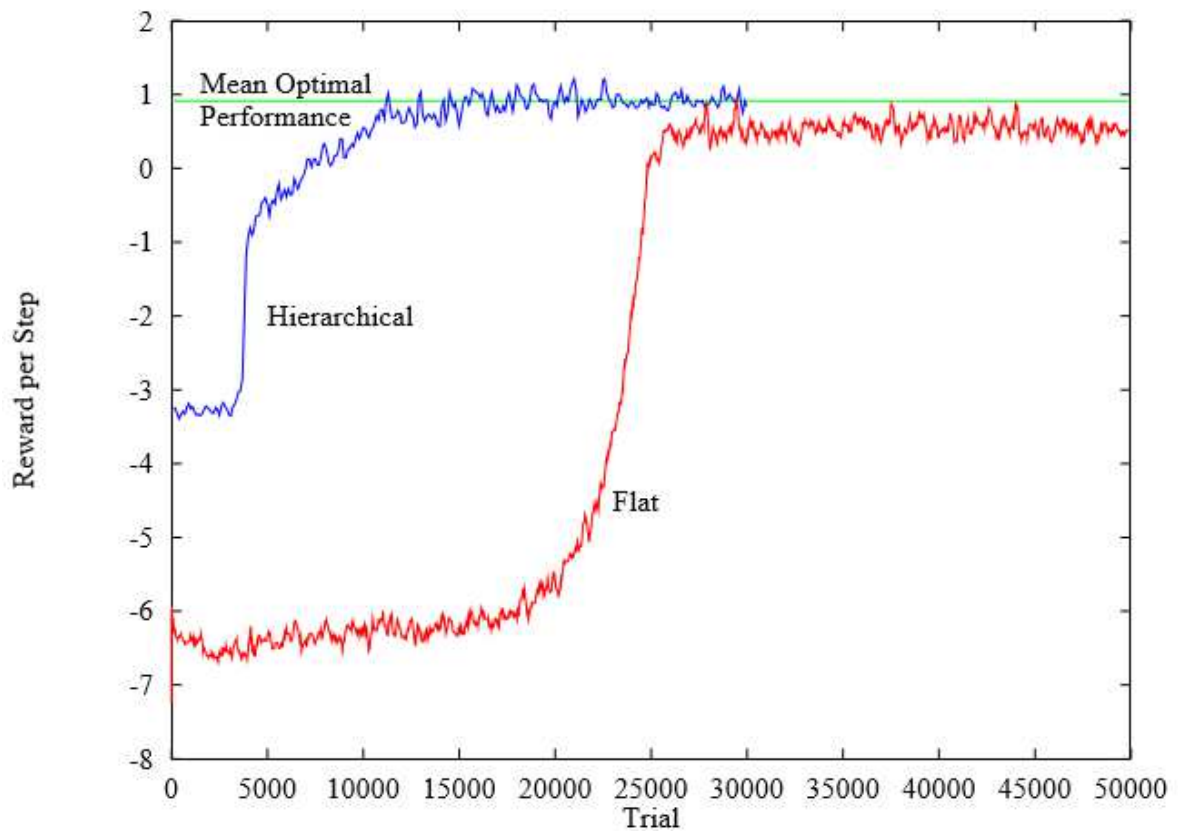


Figure 3-11: Results of the MAXQ method vs. flat Reinforcement Learning (Dietterich, 1998)

However, this method does require prior knowledge of the environment for the A^* algorithm to find these sub-goals. It would be an interesting development if this system can be adapted in such a way that a learner with no external knowledge could first explore the environment with a random policy and during learning, it tries to identify frequently traveled states as midpoints. But as a concept of combining RL with another type of machine learning, it has been proven to be effective and some further research into A^* could be beneficial to this research (Guo et al., 2013).

3-3-5 Conclusion

It can be said that each of these techniques has a good potential to be implemented in a Safe Hierarchical Reinforcement Learning algorithm. The ideology behind each method is quite similar also. They require that the task is split in smaller sub-task and rely on the assumption that completing each sub-task optimally leads to the global optimum of the original task. Where they differ is the actual update function used to converge as well as the exact method of implementation of these various sub-tasks is different.

Safe Reinforcement Learning

Often, during online operation of a system, it is vital that unsafe states are avoided. However, the conflicting goals of autonomous behaviour and state exploration make this a hard task. Safety inhibits exploration and exploration can cause unsafe states. The combination of safety in RL is still a rather new concept and therefore lacks an established definition. Building on the definition of García and Fernández, SRL is defined here as a machine learning system which incorporates, or is based on RL methodology, i.e. improving a policy to maximise or optimise expected return, while attempting to guarantee a degree of safety for online learning systems and simultaneously maintaining agent autonomy as much as possible (Garcia & Fernández, 2015). The inclusion of other machine learning techniques to provide the safety aspect is inspired by biology. In nature, avoiding dangerous situations is often not taught through negative reinforcement but the instinct to avoid is already there at birth (Bolles, 1970).

A system is deemed *safe* when it is able to avoid fatal states during online implementation. A fatal state is defined as a terminal state, that when reached, disallows any further learning or actions by the agent. It is clear that safety through RL alone can only be learned autonomously through offline simulations rather than online. While this is perfectly acceptable, it is in the interest of further research to look for systems that require as little as possible offline training to achieve a degree of safety. Purely online learning to find safe strategies cannot be included in this definition because it is during the online period that a system should be kept safe from harm and a system cannot learn to avoid a fatal state without first experiencing it. The work of García and Fernández categorises two methods of SRL; modifying the optimisation criterion and modifying the exploration process (Garcia & Fernández, 2015). This chapter will present and summarise these methods used to combine safety with RL.

4-1 Modifying the optimisation criterion

A RL agent learns by optimising its control policy, that is, find the function that maps states to actions which yield the best return. The first method of improving safety for a RL

agent is to provide a modified optimisation criterion that not only considers the return, but also includes a consideration for risk. This can be done using the *worst-case criterion*, the *risk-sensitive criterion* or the *constrained criterion* (Garcia & Fernández, 2015).

In the worst-case criterion, the expected cumulative reward considers lowest possible return of a policy, regardless of how rare of an occurrence it may be. The agent in this case converges to the policy with the best worst-case scenario. This means that in its search for the policy with the highest cumulative reward, it may disregard policies that have a possibility of selecting a bad action even though it is a rare occurrence and instead opt for policies which may have lower total cumulative reward, but have a lower or zero probability of selecting bad actions. This difference in criterion can be seen by comparing Equation 4-1 and Equation 4-2. Here $E(R_t)$ is the expected return at time step t and w is the trajectory followed by following policy π . The main difference in these equations is that Equation 4-1 does not care about the trajectory taken and is only concerned about the optimal return whereas Equation 4-2 considers that the trajectory follows the worst possible action for each state along the trajectory according to policy π no matter how rare (Garcia & Fernández, 2015).

$$\max_{\pi} E(R_t) = \max_{\pi} \sum_{k=0}^{\infty} \gamma r_{t+k+1} \quad (4-1)$$

$$\max_{\pi} \min_w E(R_t) = \max_{\pi} \min_w \sum_{k=0}^{\infty} \gamma r_{t+k+1} \quad (4-2)$$

This method causes the policy that is eventually chosen to be the safest, however it does not stop the exploration process from including these situations where the 1-step reward is bad or where the agent enters a fatal terminal state. It only guarantees that after learning and once converged, the agent will stay away from potentially harmful states. It does nothing for preventing an agent to enter an unsafe state during exploration.

Using the risk-sensitive criterion involves changing the optimisation criterion to include a factor $\beta \in (-1, 1)$ where $\beta = -1$ is risk-seeking behaviour and $\beta = 1$ is risk-averse behaviour. An example is given in Equation 4-3 (Garcia & Fernández, 2015). Unfortunately there is not a lot of work done on researching this type of optimisation criterion modification and the research that uses Equation 4-3 could not include a model-free task. In addition, this method like the worst-case criterion, does not influence the exploration process and only guarantees safe behaviour after converging to an optimal policy.

$$\max_{\pi} \beta^{-1} \log E_{\pi}(e^{\beta R}) = \max_{\pi} \beta^{-1} \log E_{\pi}(e^{\beta \sum_{t=0}^{\infty} \gamma^t r_t}) \quad (4-3)$$

Lastly, in modifying the optimisation criterion, it is also possible to use a constrained criterion function. The goal of an agent is then to find the optimal control policy with the highest expected return, subject to a number of constraints. The general form of this constraint can be seen in Equation 4-4 where c_i is the i th constraint in C and h_i is a function related to the return with a threshold value of α (Garcia & Fernández, 2015).

$$\max_{\pi} E_{\pi}(R) \text{ subject to } c_i \in C. c_i = \{h_i \leq \alpha_i\} \quad (4-4)$$

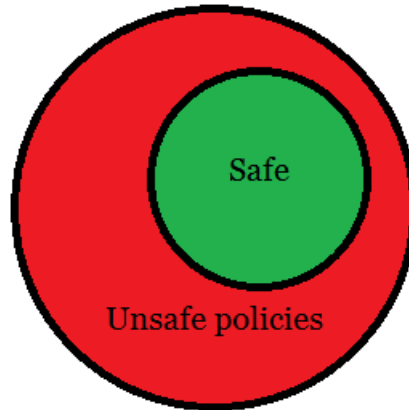


Figure 4-1: Total policy space constrained to produce a subset of the policy space deemed safe

Using this approach, the total policy space available to the agent is reduced by the threshold values α . A system designer can with this, control the level of safety in an agent's chosen actions. However it would appear that this is mostly useful on a system where a model of the environment is available to the agent or prior knowledge is instilled upon the agent by the designer. A model is required by the agent due to the otherwise inability of the untrained agent to predict an unsafe choice as identified by the constraints. Alternatively, an untrained learner would also require prior knowledge of the designer in the form of proper settings of these threshold values because to the agent, it is not known what subset of the policy space is considered safe.

Figure 4-1 shows how the total policy space is reduced by these constraints and this makes it clear that prior knowledge is needed to set this border between safe and unsafe policy. An untrained pure RL learner would not be able to set this border by itself and thus makes this approach unsuitable for autonomous safe online RL.

It can be seen that the above methods of using optimisation criterion modification are hard to apply to a model-free RL agent or one which is expected to autonomously explore an environment in a safe manner. None of the above methods will avoid unsafe states entirely without any prior knowledge by the designer.

4-2 Modifying the exploration process

The second method for SRL is to directly modify the way that an agent explores its environment. This can be done by incorporating external knowledge or through the use of risk-directed exploration (Garcia & Fernández, 2015).

External knowledge can be implemented in a variety of ways to influence the exploration process of a learner agent. The most simple example is initialising the policy space with some non-random definition. To illustrate this, consider a navigation control agent depicted as the blue circle in Figure 4-2 with each cell being a state and the red cell being an unsafe state. The agent can move up, down, left or right. To an untrained agent, each action is equally viable as the agent has no information to the safety of each action. However if the policy

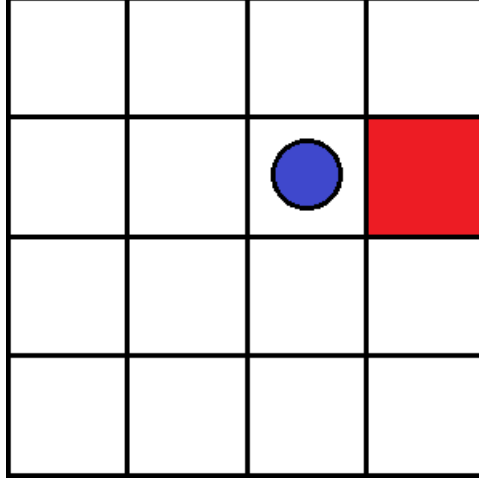


Figure 4-2: A simple navigation task for the agent in blue with an unsafe state in red

Table 4-1: State-action space for states 0, 1, 2, 3, 4, 5 with demonstrated action choices

State	Action	Demonstrations
S0	A0, A1, A2	A0, A1
S1	A0, A3, A5	A3, A5
S2	A2, A4, A5	A2
S3	A3	A3
S4	A1, A4	A4
S5	A4	A4

space is initialised (bootstrapped) with the knowledge that a step to the right is unsafe, the agent will not explore this option and it can be said that safety is guaranteed.

An alternative method is to optimise a policy function using examples derived of demonstrations. This is basically limiting the total action space for each state to those actions that are present in the examples. The examples can be created by a human designer or from other forms of learning algorithms. To illustrate the principle, consider the state-action space shown in Table 4-1. This example shows the actions available to an agent in states S0 through S5 as well as actions chosen by a demonstrator in those same states. This shows that in S0, the demonstrator did not choose action A2 as this may be an unsafe action. Now a RL agent may be tasked to find the optimal solution where the action space is limited to the demonstrated actions. It can be said that this would result in a safer system if it can be guaranteed that the demonstrated actions are chosen based on their accepted safety.

The third method of using external knowledge to optimize a system is to use a teacher. A teacher can be defined as an external policy agent, which may be human or another machine learning system, that provides an action for a given state. The basic relationship is modified as shown in Figure 4-3. In this architecture, a teacher also receives the state from the environment and uses its own policies to decide on an action. However this action is only performed when the teacher expects the agent to perform an unsafe action and the agent then performs the action of the teacher instead. Alternatively, the agent can request a correct action to perform if the policy of the agent does not give a clear or safe action to perform.

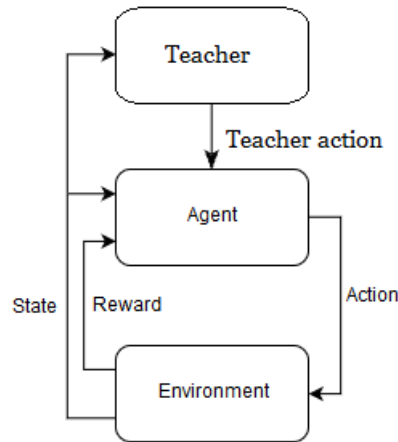


Figure 4-3: Teacher-agent reinforcement learning basic architecture

4-3 Conclusion

The task of making a reinforcement learner safer is a difficult one. From the methods shown in this chapter, it becomes clear that it is unavoidable that some prior knowledge has to be implemented in the system at one point or another. It is important to draw the line for calling the safe learner an autonomous one because under some definitions, it could be possible to define a flat RL as an SRL simply by letting it learn a task for a number of episodes, saving its learned policies and then restarting its learning. It should be clear that doing this with a flat RL learner would result in one performing wrong actions less frequently after it has been restarted although being able to call this safe (or a useful contribution to SRL research) is debatable. This analogy is made to highlight that the origin of the pre-learning information should be a factor in determining the effectiveness of SRL techniques. Whether the agent is preloaded by a teacher or demonstrations (human or otherwise) with a biased policy to improve, or simply learns for 10 episodes to initialize a biased policy after which it is called safe can be seen as the same principle. For example, in the case of learning from demonstrations, which may be considered safe, the origin of these demonstrations are unquestioned even though they might as well come from the same learning agent that uses them.

This realization makes it clear that one has to accept that an agent cannot learn safety fully autonomously without some form of prior knowledge implemented in the system. It becomes then important for comparing SRL learners with non-safety focused learners to account for this in terms of convergence speed since the safe learner will have started with more information and subsequently will not explore potential unsafe actions reducing the convergence speed.

Chapter 5

Preliminary Results

The first step towards an algorithm design is to implement a selection of algorithms from literature and validate their working principles. This will ensure that there is proper understanding of the topics discussed and to test their usability in a combined algorithms. In this chapter, the setup and results of an implemented flat RL algorithm is shown in Section 5-1 followed by a HRL implementation in Section 5-2.

5-1 Flat Reinforcement learning

Flat RL has been tested on a simple navigation control case where an agent is tasked to find the shortest path from its origin to the goal state. The task for the navigating agent is displayed in Figure 5-1 with the agent represented as a blue circle, starting in the cell marked by *start*. The goal for the agent is to reach the goal state, the green cell, in as little steps as possible. This is done through applying a 1-step *Sarsa* algorithm with eligibility (Sutton & Barto, 1998) traces to backup all the Q-values for each visited state.

The environment is an 18 by 18 cell grid world, meaning there are $S \rightarrow 18 \cdot 18 = 324$ states. The agent's action set consists of moving 1 step in each of the 4 compass directions *North*, *South*, *West*, *East*. A reward of -1 is given to the agent for each step that does not end in the goal state, which rewards the agent with 0. Steps that would place the agent inside a wall do not move the agent, however still reward -1. This means that if the agent finds the path that rewards the highest, or least negative total return, it has subsequently found the shortest path from its origin to the goal state.

The means that the Q table contains $S \times \mathcal{A} = 324 \cdot 4 = 1296$ values. With the start location and goal location fixed, the environment is static such that this problem can readily be solved with RL. The algorithm used to update the learner agent is displayed in Algorithm 1 where E contains the eligibility trace value for each state.

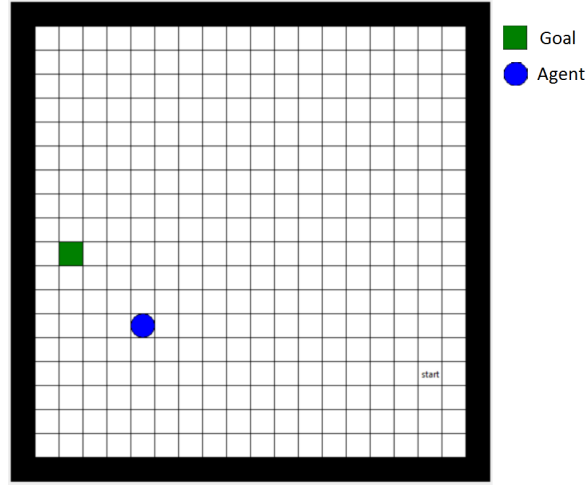


Figure 5-1: Flat reinforcement learning agent implementation to navigate an 18 by 18 grid world

```

1 Initialize  $Q(s, a) \forall s \in \mathcal{S}, a \in \mathcal{A}$  ;
2 repeat
3    $e(s, a) = 0 \forall s \in \mathcal{S}, a \in \mathcal{A}$  ;
4   Initialize s, a ;
5   repeat
6     Take action a, observe r, s' ;
7     Choose a' from s' using policy derived from Q ;
8      $\delta \leftarrow r + \gamma Q(s', a') - Q(s, a)$ ;
9      $e(s, a) \leftarrow e(s, a) + 1$ ;
10    forall  $s \in \mathcal{S}, a \in \mathcal{A}$  do
11       $Q(s, a) \leftarrow Q(s, a) + \alpha \delta e(s, a)$  ;
12       $e(s, a) \leftarrow \gamma \lambda e(s, a)$  ;
13    end
14     $s \leftarrow s', a \leftarrow a'$  ;
15  until s is terminal;
16 until end;

```

Algorithm 1: Sarsa algorithm with elibility traces used to train a flat RL agent

The optimal path as learned by the agent is shown in Figure 5-2. It can be seen that there are many global optima in this scenario and running the learning process multiple times will likely result in the agent converging to another optimum path. A graph has been plotted by the system to show the convergence speed, seen in Figure 5-3.

5-2 Hierarchical Reinforcement Learning

After having tested the flat RL, the next step was to implement a HRL algorithm and compare it with flat RL. This was done using a modified setup of the RL implementation of the previous section. The grid world has been split into four rooms with a door between the rooms and the goal placed in the doorway of one of the rooms. Figure 5-4 shows this modified grid world

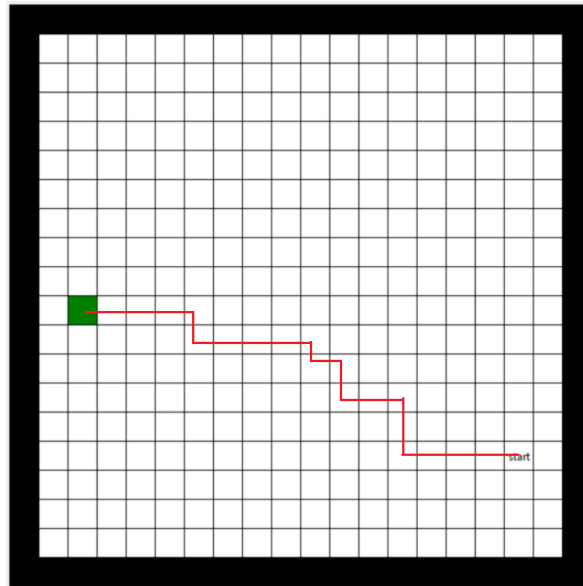


Figure 5-2: Optimal path shown in red traced by learner agent after converging

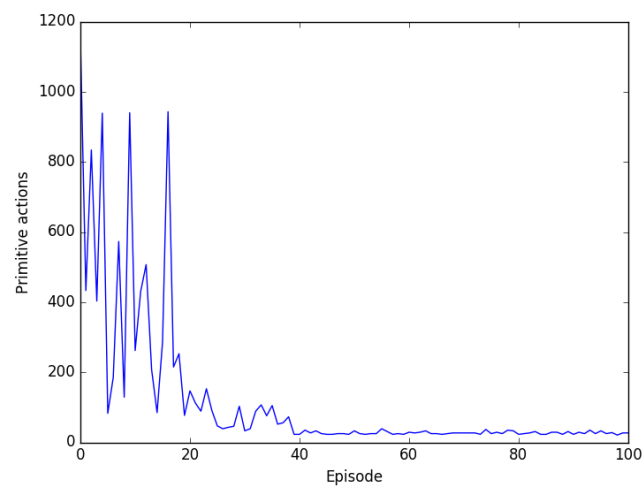


Figure 5-3: Graph showing the convergence speed of the learner agent.

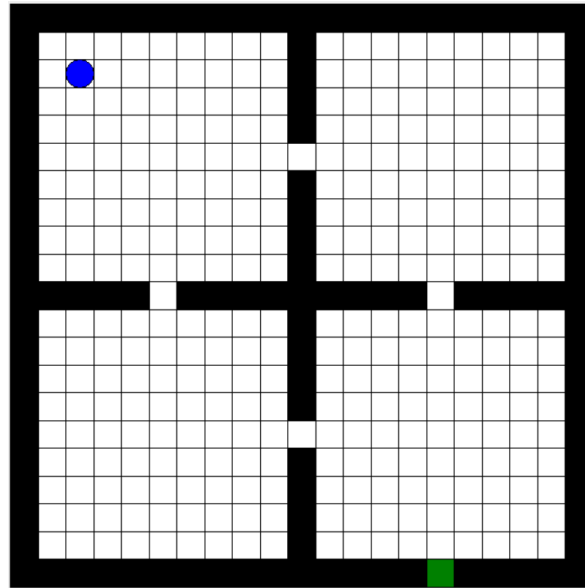


Figure 5-4: Grid world for the HRL learner to find the shortest path from origin to goal

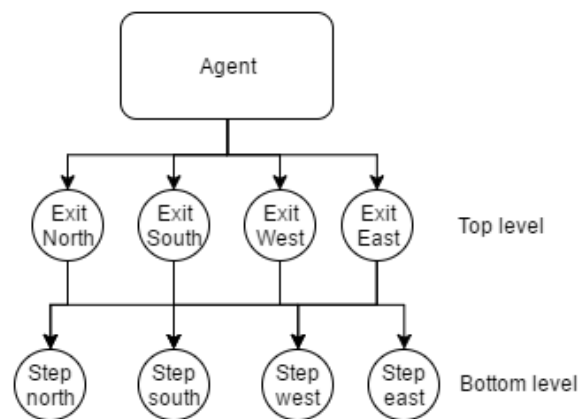


Figure 5-5: Diagram showing the hierarchy structure for the room navigating agent

with the agent being the blue circle and the goal being the green southern doorway in the bottom-right room.

The goal of the agent was again to find the shortest path from the agent's origin to the goal. This is done by giving the agent a reward of -1 for every primitive action that is not the goal state, similarly to the flat RL implementation. However, in this case, the agent structure has been modified into a hierarchy where the top level chooses the direction to exit the room it currently inhabits as can be seen in Figure 5-5.

Upon initialization, the agent chooses a direction to exit its current inhabited room. This temporally extended action then causes the agent to pick and choose primitive actions to achieve the sub-goal of exiting the room towards the chosen direction. It performs these primitive actions until the room has been left in the correct direction, rewarding the temporally extended action with the cumulative reward gained for the primitive actions. Leaving the room in the incorrect manner, that is, leaving through the north when the selected exit

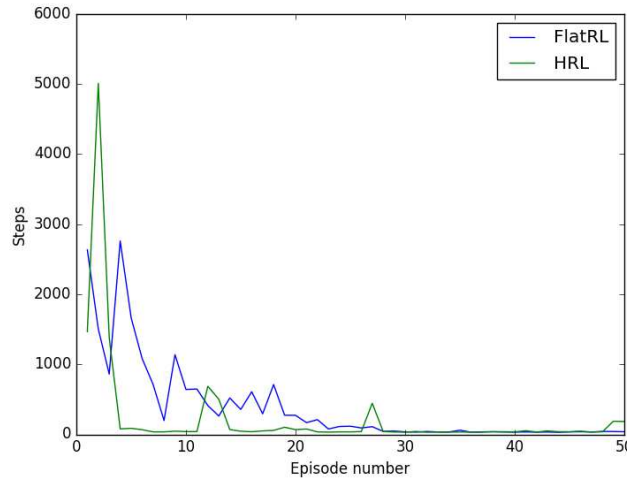


Figure 5-6: Plot showing the amount of primitive steps taken per episode comparing flat RL with HRL

is south is not allowed and will be processed similar to hitting a wall; no step taken, but negative reward still given.

To compare the effectiveness of the HRL method, this grid world has also been navigated by a flat RL algorithm. The amount of primitive steps of both algorithms are compared together in a plot to show the difference. This plot shown in 5-6, shows that the HRL method converges quite rapidly compared the the flat RL method.

It should be noted that for this particular case, there is no computing memory benefit in terms of less state-action values to compute. This is due to the fact that there are only 4 rooms, each being a 9 by 9 cell area with the addition of 5 doorway states, totaling to 329 states. For the flat RL system, it can choose 4 primitive actions in each cell that is not the goal state meaning $\mathcal{S} \times \mathcal{A} = (329 - 1) \cdot 4 = 1312$ Q values. The HRL system first has 4 room states, from which it can choose to exit from 4 directions meaning $\mathcal{S}_l \times \mathcal{A}_l = 16$ where $(\mathcal{S}_l, \mathcal{A}_l)$ are the states and actions available in the top level respectively. For each room, the lower level has 4 actions, resulting in an additional $9 \cdot 9 \cdot 4 = 324$ Q values *for each top level action*. This means that the HRL method actually has $4 \cdot 324 + 16 = 1312$ Q values to learn as well. However, it becomes interesting when more rooms are added and then the difference becomes more apparent. For each additional room added, the flat RL system will have to keep track of an additional 324 Q values, whereas the HRL method need only keep track of an additional 4 Q values; the top level just needs to learn in which direction to exit this new room and the bottom level will already know how to.

Note that the doorways have been omitted from these calculations because the HRL is programmed in such a way that exiting a room through a doorway lets the system take the additional step to land within the next room; it cannot choose an action inside a doorway. While this may seem an unfair comparison, it is not significant enough to warrant further investigation as the purpose of this test was more to become accustomed to HRL implementation.

Chapter 6

Research Method

In order to determine the benefits that come from combining methods for safe space exploration with HRL, it is important to get acquainted to flat RL, HRL and SRL separately and develop algorithms combining them. This chapter discusses the methodology to be used to effectively explore the possibilities of combining the knowledge gained in the literature review.

6-1 Algorithm Design

In the initial stage of the project, it is important to develop a task for the learning algorithm to optimize. This task should fulfill a number of requirements in order to truly test the capabilities of the developed Safe Hierarchical Reinforcement Learning algorithm. These requirements include obstacles to determine safety behavior and a relatively large state or action space to demonstrate the ability to counter the curse of dimensionality.

Initially it is possible to develop an algorithm for an all-knowing agent meaning, it knows all the relevant state information such as the exact position within a room. Upon successful development, the agent will only be given partial or uncertain information regarding its state. This in order to simulate how a physical system would only know information that it receives from its sensors. Gradually more actions and states can be changed from instantaneous and complete to dynamic and uncertainty. Doing this will allow the system to be better implemented in physical systems in the future.

6-2 Test Setup

Once the algorithm has been designed, it will be tested on a task and compared with a selection of other established approaches. Since the research is about safety and a hierarchical reinforcement learning approach, the following will need to be contained in the task definition.

The task should contain obstacles. Obstacles are broadly defined as fatal terminal states. This means that when these states are reached, the learner ceases to perform actions and also

ceases to learn. For this particular learner, the exercise is over and will need to be restarted from initial conditions again for another trial. These obstacles may be walls in a navigation controller, which would be like a UAV crashing into walls in the physical world.

In addition the task should also contain ways to temporally abstract states and/or actions. This could be like the exact positions in a room to be abstracted away into simply the room. Alternatively primitive actions in an aircraft elevator controller such as deflecting 1 degree up or down may be abstracted away into temporally abstract action *level aircraft*.

The focus of this research will be on finite episode tasks. This means that the task should contain at least one terminal goal state. This is a state where, if reached by the learner, the episode ends, the agent is reset to initial conditions but maintaining the knowledge of the environment learned up to this point.

6-3 Testing Parameters

From the preliminary testing of algorithms, it has become clearer what kind of parameters can be viewed to compare efficiencies of different algorithms with each other. For this design, a selection of parameters are chosen to determine the effectiveness of a given algorithm and to give an objective measure to compare various approaches. These parameters are shown in the list below, followed by a more detailed explanation.

- Number of primitive actions to convergence
- Real time until convergence
- Number of episodes to convergence
- Number of unsafe actions performed
- Policy optimality

The first parameter is the total amount of primitive actions performed until convergence. A flat RL's action space consists of only primitive actions, thus counting the total amount of primitive actions until convergence can give an indication how the algorithm compares to other methods. This is especially convenient because although HRL can perform temporally abstract actions on higher levels, the lowest levels will still perform primitive actions, allowing it to compare with flat RL methods.

Another parameter can be the total real time until convergence. Having a system converge to a solution with a low amount of primitive actions means little if the total time is very long. So during testing, the total time will be tracked to give accurate real time to convergence to compare different methods. However the validity of this parameter may run into problems when methods for safety cause events such as halting the learning algorithm or restarting it and this must be accounted for.

The third parameter is the total episodes to convergence. When using finite episode lengths, keeping track of the total amount of episodes can show the speed of convergence. Keeping

track of both the total amount of primitive actions and the total episodes or the amount of primitive actions per episode gives the speed of convergence from episode to episode.

Since safety is an important feature in the research, the amount of unsafe actions performed needs to be tracked as well. If a learner ceases to learn and the training exercise finishes after reaching a fatal terminal state, this should be noted. Alternatively a safe-guard can be employed in the algorithm that will stop the learner from performing an unsafe action, but still count the amount of times this happens. For a safe system, it should be the goal to make this value as low as possible.

Finally, the final policy trained by the algorithm needs to be compared. In a large state-space domain, it is hard to determine how close the policy is to the global optimum, so this would be more of subjective or comparative parameter. When a navigation trainer achieves the goal in a certain amount of time steps, it can be compared with another algorithm to see which has the better step count. If the global optimum is known, the final policy can be compared with this instead.

6-4 Research Goals

The goal of the research is to answer the research question and meet the various sub-goals that were shown in Chapter 1. Using the test setup and test parameters as presented in Sections 6-2 and 6-3, a fair comparison between various RL methods can be made. This is important to meet the sub-goals of determining the effectiveness of the various algorithms and compare their level of safety, efficiency and optimality. When these sub-goals are met, the comparisons will allow for the research question to be answered by discussing the possible benefits the developed algorithm has over other forms of RL.

Literature study conclusions

In the interest of intelligent flight control systems or navigation, machine learning such as RL could be a potentially effective method for autonomous optimal navigation or flight control. However it is vital in aerospace engineering applications that a controller is safe and does not compromise the system that it's controlling. This research attempts to find methods to ensure system safety during the learning phase.

In order to be effective in a continuous domain or an otherwise fairly sized state space, the curse of dimensionality needs to be addressed. HRL is an effective method to counter the curse of dimensionality by reducing state-action space by abstraction. A UAV navigating through a house could abstract away exact position (x, y) in a room to simply being in room z . This research looks into various methods of HRL and applies a selection of them to the research algorithm to find a safe intelligent flight controller.

The research goal is to find out what the benefits are at applying methods for safe state exploration on a HRL system. That is, methods that attempt to ensure safety will be applied on some, or all levels of a hierarchical RL system to create an autonomous learner that, with as little external knowledge or intervention as possible, can explore a state domain.

A preliminary research was carried out to find the state-of-the-art in the fields of HRL and SRL and the various methods have been discussed to determine their approach for use in a combined algorithm. A quick implementation of flat RL and HRL have been done to test their usability and for getting acquainted to designing and programming the algorithms.

A methodology and project plan has been presented as well, showing further research plans and project time-line to determine the benefits of combining safety methods in a HRL system.

Part III

Additional results

Relative and absolute states effect on safety

A simple test has been performed to compare the performances of an agent with relative state awareness and absolute state awareness. This chapter presents the findings of this test and discusses the results.

8-1 The task

The task is a rather simple one; navigate from a starting location to the goal in the least amount of steps. In order to compare the difference between an agent with a global state awareness and a relative one, this task is attempted by two different agents. Both agents are learning using flat RL with state-action value function Q updates to find the optimal path. These updates are performed using State-Action-Reward-State-Action (SARSA) value function updates as shown in Equation 10-1 with learning rate $\alpha = 0.25$ and discount factor $\gamma = 1$

$$Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma Q(s', a') - Q(s, a)) \quad (8-1)$$

A grid world, shown in Figure 8-1, is constructed with an agent starting location (circle) and goal state (triangle). The agent cannot go through the outside borders so aside from the outside walls, there are no obstacles. For this experiment, the size of the environment is a 15 by 15 grid world where the outer cells are walls. For every action performed, the agent receives a -1 reward for normal actions, -10 for attempting to go through a wall or 0 for entering the goal state.

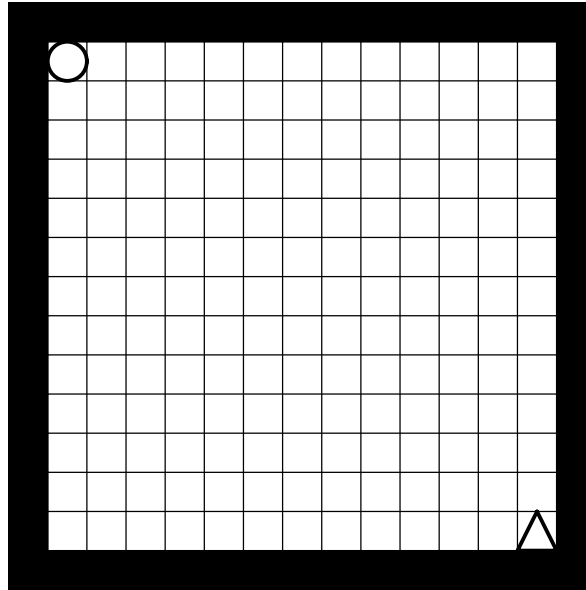


Figure 8-1: Grid world for testing effect on RL agents with relative or absolute state awareness

8-2 The agents

The agents are flat RL learners with the only difference between the two being their perceived states. The absolute state awareness agent knows its exact position in the world i.e. which cell of the grid world it is in. This state is presented to the agent as [row, column].

On the other hand the relative state awareness agent knows only what is above, below, to the left and right of it. Its state is presented to the agent as $[\uparrow, \rightarrow, \downarrow, \leftarrow]$ where the arrows represent whether or not there is a wall in that direction.

The actions available to both agents is to move one step in the directions up, right, down or left.

8-3 Results

This section presents the results of the two agents performing the task described in the previous section. For ease of understanding, the agent with absolute state awareness will be referred to as the *absolute agent* and the agent with relative state awareness as the *relative agent*.

Figure 8-2 shows the overall agent performance with the amount of wall collisions, return and actions per episode shown in Figures 8-2a, 8-2c and 8-2e respectively for the absolute agent in the left column. The same information for the relative agent is shown in the right column of Figure 8-2.

It can clearly be seen that the absolute agent performance improves over time and the number of wall collisions decreases although is still a regular occurrence. This is expected behavior as there are no measures in place to prevent the agent from colliding any walls and every cell

Table 8-1: Results showing the average number of collisions per episode *after the first episode*, the total number of collisions and the amount of collisions in the first episode.

	Reward given for obstacle collision								
	-10			-100			-1000		
Agent	Avg	Total	Episode 1	Avg	Total	Episode 1	Avg	Total	Episode 1
Absolute	1.19	179	61	-	-	-	-	-	-
Relative	0.28	371	343	1.16	148	33	0.84	95	12

in the grid world is an entirely new state for the agent to explore every action. The agent has also been initialized with an ϵ -greedy policy and optimistic i.e. $Q(s, a) = 0 \forall s \in \mathcal{S}, a \in \mathcal{A}$ which results in the occasional sub-optimal state being explored.

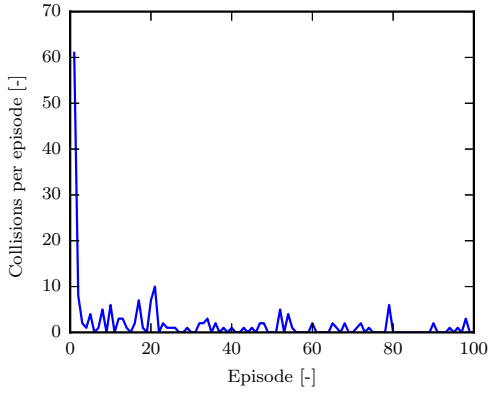
The state available to the relative agent is only the knowledge of what is in the cell immediately next to the agent. Figure 8-2f shows the amount of actions taken from start to finish and clearly demonstrates the agent approaches and often even reaches optimal behavior. This in contrast to the absolute agent which although it was approaching optimality, it had not done it yet within the 100 episodes as can be seen in Figure 8-2e. In general comparing Figures 8-2c and 8-2e with Figures 8-2d and 8-2c it can be seen that in terms of return and actions, the relative agent outperforms the absolute agent.

The difference in safety performance between the two agents is not so clear from the figures. Comparing Figure 8-2b with Figure 8-2a, one might argue the absolute agent performs slightly worse. However, counting the total amount of collisions reveals that the absolute agent collided 179 times while the relative agent did so 371 times. The relative agent actually collided 343 times with a wall in the first episode. It performed much better after the first episode, colliding an average of 0.28 times per episode compared to 1.19 times for the absolute agent.

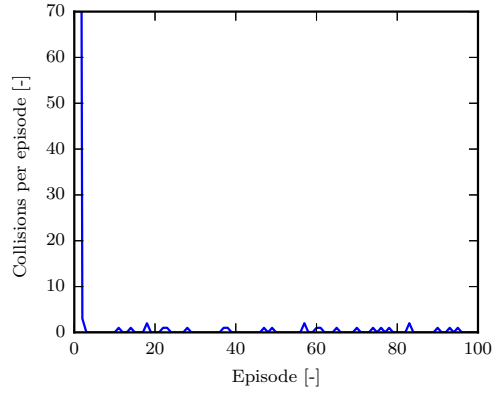
The reason for this seems quite simple; as the reward given to the agent is -1 for every action and -10 for a collision, every 10 steps taken by the agent equals 1 collision. This becomes important to understand for the relative agent because the agent might take an action and find its state unchanged (although it has moved in the world, it does not necessarily notice this). Colliding with the wall once, the agent will probably not try it again until the reward it expects to receive from other actions become equal or less to that of collision. Once the goal has been found, the agent will begin favoring certain states over others and learning happens much faster. The validity of the effect of reward on collisions is easily checked by changing the reward for collision to a much more negative number.

The test has been performed again for the relative agent with the reward of a collision changed from -10 to -1000. All other things remain equal. From this simulation, the amount of times the new relative agent has collided was only 95 times in total with an average of 0.84 times per episode after the first. It collided 12 times in episode 1 and 83 times afterwards. These results and that of the previous agents is summed up in Table 8-1. The experiment has only been repeated for the relative agent because the goal was to find the impact of the different settings on this particular agent. The results of the absolute agent are added as a reference.

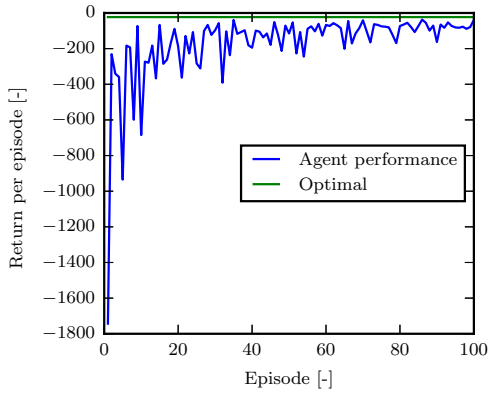
From the results shown in table 8-1, it can be seen that the reward given to the agent indeed has a large impact on the collisions amount. With a relatively small negative reward, the agent will attempt a movement towards a wall often while this becomes significantly less with



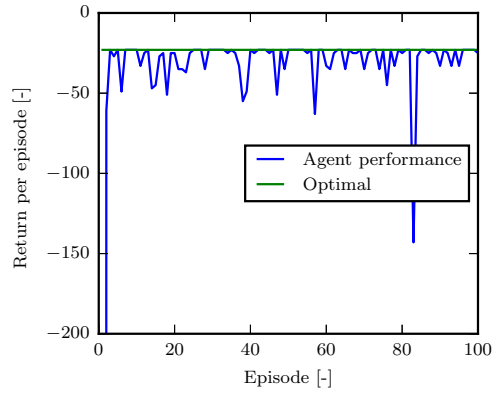
(a) Amount of bumps per episode of flat reinforcement learning agent on task with absolute state observation



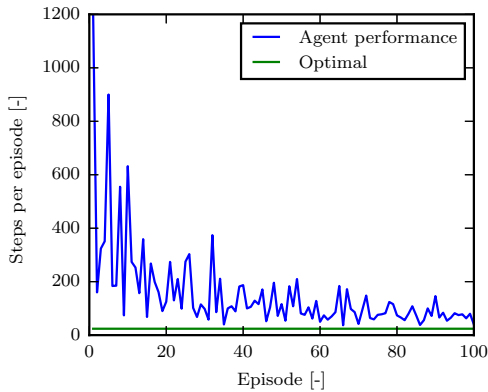
(b) Amount of bumps per episode of flat reinforcement learning agent on task with relative state observation



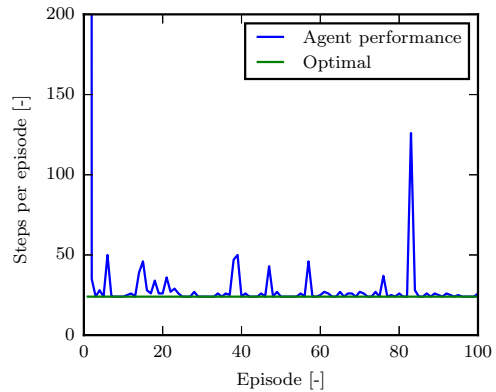
(c) Return per episode of flat reinforcement learning agent on task with absolute state observation



(d) Return per episode of flat reinforcement learning agent on task with relative state observation



(e) Amount of actions per episode of flat reinforcement learning agent on task with absolute state observation



(f) Amount of actions per episode of flat reinforcement learning agent on task with relative state observation

Figure 8-2: Performance of the flat RL agent with absolute state observation and absolute state observation performing the task

lower rewards. The agent consistently tries less collisions with the walls with lower rewards, however comparing the cases of a -10 and -1000 reward, it can be seen that in the first case, the agent collides only 0.28 times per episode on average *after* the first episode. With 343 collisions out of 371 in the first episode, it seems the agent collides a large amount until it has found the goal after which it starts to favor one state over the other. The case of -1000 reward only needs to collide a very small amount the first episode to understand the large negative feedback it will receive, but it still collides almost once per episode.

The results on this section indicate strongly the effect of a reward function on an agent with relative state information on safety. It seems evident at this point already tweaking the reward function is an approach to SRL in a relative state information task.

Absolute and relative states in different environment sizes

While running the simulations that provided the results shown in Chapter 8, it became clear that simulation time is greatly impacted by the size of the environment. This is caused by the *curse of dimensionality* and this chapter presents some results from simulations aimed at getting a closer look at this effect the environment size has on the simulation time and the safety.

9-1 The setup

For this experiment the grid world ranges from a 5 by 5 grid shown in Figure 9-1a to a 19 by 19 grid shown in Figure 9-1b where again the outer cells are walls and the agent must find the goal from its starting location.

The agent always starts in the top left empty cell and must find the goal in the bottom right empty cell. In a similar fashion to testing the impact for the collision reward of Chapter 8, the task is repeated for a relative agent with different reward given for colliding with a wall. Every environment size has been performed 100 times.

9-2 Results

The results of the task described in the previous section is shown in Figure 9-2. It can clearly be seen in Figure 9-2a that the increasing environment size has an exponential relation with the simulation time for the absolute agent. This exposes the *curse of dimensionality* and indicates that even for this relatively simple task, already simulation times can become very large very fast. Even running 100 episodes on a 19 by 19 grid world with no other obstacles than the outside walls already takes almost half an hour to complete and by this time the optimal path has not even been found yet.

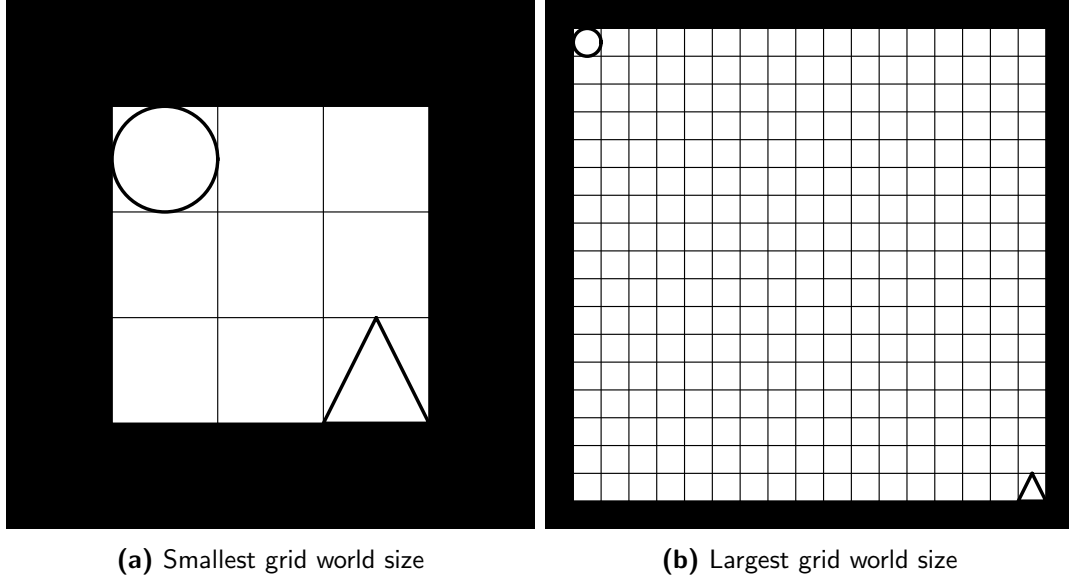
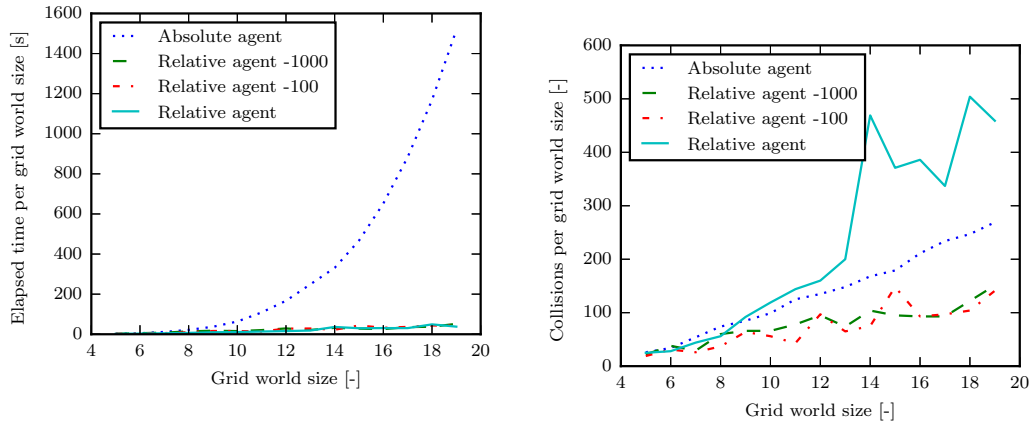


Figure 9-1: The smallest and largest grid-worlds tested to show the difference in impact of the environment size on safety and simulation time between absolute and relative agents

On the other hand, the relative agent seems unaffected in comparison. Zooming in on the performance of the relative agents shown in Figure 9-3, it shows a rather linear relation between environment size and simulation time. Coupled reaching optimality faster than the absolute agent as shown in the previous chapter, it would indicate that a relative agent is a good solution for scaling up environment sizes in RL tasks.

This research is not merely after better performance however and the safety is another concern. Figure 9-2b shows the impact the environment size has on safety i.e. wall collisions. It is shown here that although the absolute agent performs better than the relative agent with a -10 reward for wall collisions, both are outperformed by the relative agents receiving -100 and -1000 reward for collisions. In fact there seems little difference between -100 and -1000 which might be due to the size of the negative reward compared to the grid size. The larger the environment, the longer an agent might be stuck switching back and forth between its states until it encounters the goal state. And the longer it is stuck between states, the more the expected reward of collision starts resembling that of any other action. The lack of difference between -100 and -1000 would indicate that -100 is a sufficiently negative reward for finding the goal state before resorting to wall collisions.

It is concluded that for RL tasks, relative state awareness is a good solution in terms of scalability of the environment. However the safety of a relative state is largely dependent on a well thought-out reward function in the absence of other safety measures.



(a) Plot showing the effect of grid size on simulation time

(b) Plot showing the effect of grid size on safety

Figure 9-2: Results showing effect of various grid-world sizes on performance

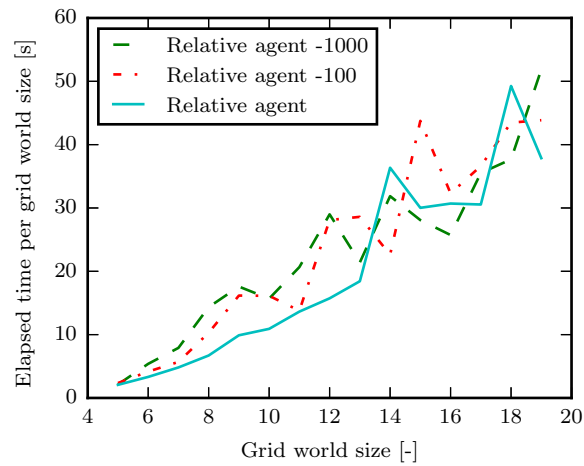


Figure 9-3: Plot zooming in on the time per grid world size of the relative agents

Preliminary environment simulations

Before creating the environment shown in Part I which is more focused on safety, another environment was created to test performance differences between relative and absolute state awareness. This chapter discusses this environment and the tests that were performed.

10-1 The environment

For this preliminary testing, an grid-world environment was created in which the agent can move one cell in one of 4 directions each time-step. The environment, shown in Figure 10-1, would instantly move the agent to the chosen location. There is also only a single processing thread, meaning the environment waits patiently for the agent to decide each action and does not update while the agent is calculating its next move.

Since it is desired to move to a relative state observation approach in order to more closely resemble a real UAV (without Global Positioning System (GPS) available) performing indoor navigation, some testing was conducted to check the feasibility of this transition. The environment shown above starts the agent in the top-left room and it must navigate to the goal state in the bottom-right with as little primitive actions as possible. This task was performed by a flat RL agent with relative state awareness, a HRL agent with absolute awareness and also with relative state awareness. The goal in these tests were to determine if and how well RL learners could perform with relative state awareness as well as gain some understanding on how to set up RL with relative states.

The state available to the agent in the relative state awareness tasks is only whether or not the cells above, below, to the left and right of the agent contain an obstacle. In case of absolute state awareness, the agent is aware what room it is in as well as its position within that room.

10-2 Results

This section shows the results of the different attempts at the task described in the previous section. In each of the three attempts, the environment initializes the agent in a certain

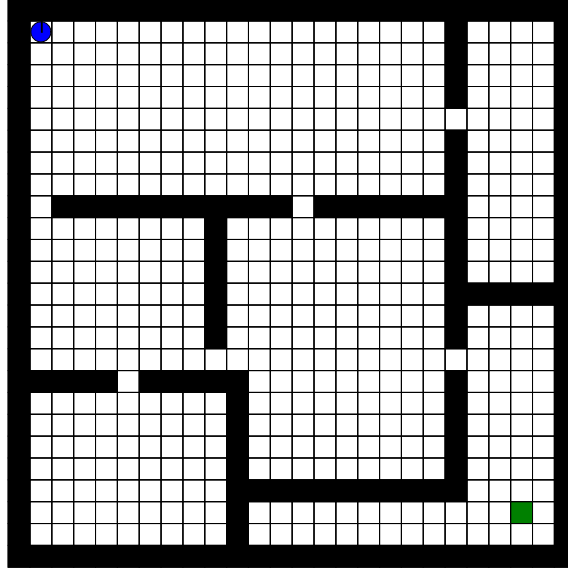


Figure 10-1: The environment in the preliminary simulations. The agent (circle) must find the goal state (rectangle bottom-right)

condition and every update waits for the agent to provide an action. This process is shown in Algorithm 8. The agents in all cases have an ϵ -greedy policy $\pi(s)$ that chooses an action based on their state-action value function $Q(s, a)$. This value function is updated using the SARSA equation shown in Equation 10-1 with the learning rate $\alpha = 0.25$ and discount factor $\gamma = 1$.

```

1 Initialize grid world ;
2 Initialize initial state  $s$  ;
3 while Goal not reached do
4   Agent generates an action  $a$  ;
5   Update environment ;
6   Generate reward  $r$  and give to the agent ;
7   Agent updates value functions ;
8 end

```

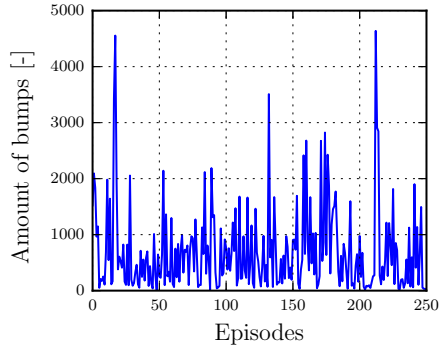
$$Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma Q(s', a') - Q(s, a)) \quad (10-1)$$

Figure 10-2 shows the results for the flat RL learner with relative state awareness in the left column as well as the HRL learner in the right column. The results tracked are the amount of wall collisions per episode in Figures 10-2a and 10-2b, return per episode in Figures 10-2c and 10-2d and primitive actions per episode in Figures 10-2e and 10-2f.

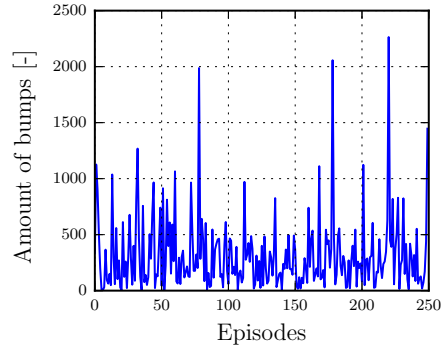
It can be seen that for the flat learner the results seem rather sporadic and no clear sign of learning (improvement) can be seen i.e. return approaching 0. However it is more or less a consistent performance and during testing it was also observed that the agent does tend to move towards the goal. But due to the rooms in the bottom-left and top-right of the

environment, the agent often finds itself trapped for extended amounts of time, leading it to unlearn good behavior (such as moving down and towards the right). It would seem that these rooms are a reason why the agent does not show clear signs of learning.

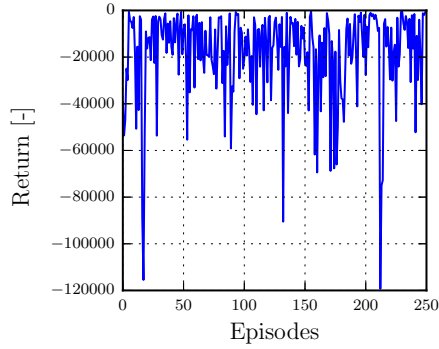
Similarly to the flat RL performance, there is no clear sign of learning however it is clearly performing better than the RL agent. This improvement is due to the structure of the hierarchical learner. It allows the agent more flexibility in selecting actions because they are based on the subtask employed by a higher level agent i.e. leaving a room in a certain manner based on some sensory information. This causes the agent not to decide an action solely on the perceived state, but also on the sub-goal at hand. Using this additional information, the primitive actions are selected and learned to realize this goal. The HRL approach also collides significantly less with its environment than the non-hierarchical approach and the episodes are also much shorter. This already shows the effect of HRL as an approach to SRL; hierarchy can have a significant positive effect on safety.



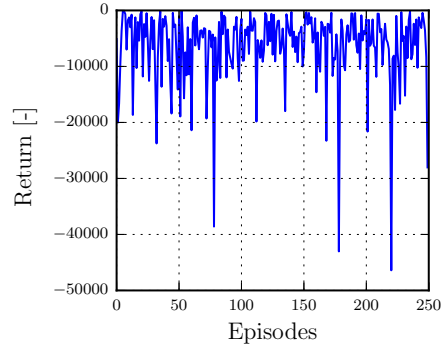
(a) Amount of collisions per episode of flat reinforcement learning agent on task with relative state observation



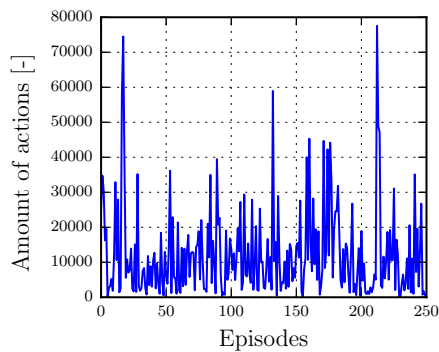
(b) Amount of collisions per episode of hierarchical reinforcement learning agent on task with relative state observation



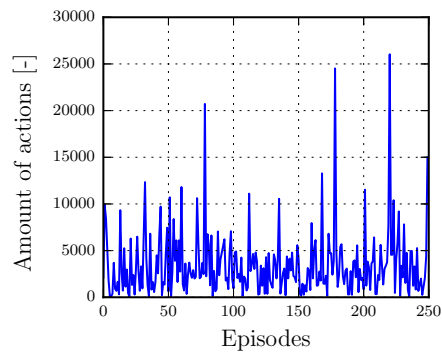
(c) Return per episode of flat reinforcement learning agent on task with relative state observation



(d) Return per episode of hierarchical reinforcement learning agent on task with relative state observation



(e) Amount of actions per episode of flat reinforcement learning agent on task with relative state observation



(f) Amount of actions per episode of hierarchical reinforcement learning agent on task with relative state observation

Figure 10-2: Performance of the flat RL agent and HRL agent with relative state observation performing the task

Chapter 11

Conclusions

Reinforcement Learning (RL) is a field of machine learning that learns from experience which is a very natural method of learning to us. It requires no model of the environment or the system it controls which makes it very attractive to control engineers. However RL suffers from the *curse of dimensionality* which means it does not scale well with increasing state or action spaces and safety is not inherently a concern of a RL agent. This project performed research into ways of making RL more safe while an agent explores its environment. The extra challenge added to the task is to maintain agent autonomy i.e., avoiding to simply disallow certain actions from being performed. Hierarchical Reinforcement Learning (HRL) was investigated as an approach for Safe Reinforcement Learning (SRL) while building on the knowledge of other approaches in the field of SRL.

This project investigates the benefits of HRL as an approach to SRL while attempting to keep agent autonomy as intact as possible. The *actor-judge method* was developed as an approach to enabling safe state space exploration in a RL agent which can be applied on various hierarchical levels such as in a HRL machine. In Part I, this method is presented and it shows its improvement over flat RL and HRL with no safety precautions. The actor-judge method has been developed such that it requires no model of the environment. The only information that needs to be provided is what is considered an unsafe state transition hereby maintaining a level of agent autonomy. Also shown in Part I is the performance of RL compared to HRL in terms of safety clearly demonstrating the added safety of HRL systems.

A brief experiment has been shown to demonstrate the ability of the actor-judge method to be applied on various levels in the agent's hierarchy. This shows that this approach is not required to be applied on the lowest level but can also function where safety is not due to the primitive actions an agent chooses, but rather on a higher level of abstraction such as the state of a Unmanned Aerial Vehicle (UAV) battery charge.

Part II shows a literature study showing the state-of-the-art in RL, SRL and HRL that were used as inspiration and knowledge to build this project on.

Lastly Part III shows additional supplementary results highlighting the difference between flat RL and HRL as approaches to SRL. These results show that an agent with HRL quickly

outperforms a flat RL agent in terms of safety in a simple task while also converging faster to an optimal result. Additionally, the curse of dimensionality is made visible and relative state awareness is shown as a means to reduce the time required to learn a behavior while also increasing agent safety in larger environments. Shown here are some results demonstrating the effect of absolute state versus relative state awareness on agent safety and performance. Relative state awareness and HRL are both promising solutions to the *curse of dimensionality*, however relative state awareness alone is not always a clear approach to get more safety. A proper reward function shows a greater impact on safety with larger state spaces.

Bibliography

- Barto, A. G., & Mahadevan, S. (2003). Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13(4), 341–379.
- Bolles, R. C. (1970). Species-specific defense reactions and avoidance learning. *Psychological Review*, 77, 32-48.
- Bou-Ammar, H., Voos, H., & Ertel, W. (2010). Controller design for quadrotor uavs using reinforcement learning. In *IEEE multi-conference on systems and control* (pp. 2130–2135).
- Dietterich, T. G. (1998). The MAXQ method for hierarchical reinforcement learning. In *Proceedings of the fifteenth international conference on machine learning* (p. 118-126).
- Dietterich, T. G. (2000). Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, 13, 227-303.
- Duguleana, M., & Mogan, G. (2016). Neural networks based reinforcement learning for mobile robots obstacle avoidance. *Expert Systems with Applications*, 62, 104 - 115.
- Fernández, F., Borrajo, D., & Parker, L. E. (2005). A reinforcement learning algorithm in cooperative multi-robot domains. *Journal of Intelligent and Robotic Systems*, 43, 161-174.
- Garcia, J., & Fernández, F. (2015). A comprehensive survey on safe reinforcement learning. *Journal of Machine Learning Research*, 16(1), 1437–1480.
- Guo, Q., Zuo, L., Zheng, R., & Xu, X. (2013). A hierarchical path planning approach based on reinforcement learning for mobile robots. In *International conference on intelligent science and big data engineering* (pp. 393–400).
- Hengst, B. (2011). Hierarchical reinforcement learning. In *Encyclopedia of machine learning* (pp. 495–502). Springer US.
- Hu, J., Sasakawa, T., Hirasawa, K., & Zheng, H. (2007). A hierarchical learning system incorporating with supervised, unsupervised and reinforcement learning. In *Advances in neural networks – 4th international symposium on neural networks* (pp. 403–412). Springer Berlin Heidelberg.
- Junell, J., van Kampen, E., de Visser, C., & Chu, Q. (2015). Reinforcement learning applied to a quadrotor guidance law in autonomous flight. In *AIAA guidance, navigation, and control conference*.

- Mannucci, T., & van Kampen, E. (2016). A hierarchical maze navigation algorithm with reinforcement learning and mapping. In *2016 IEEE symposium series on computational intelligence (SSCI)* (p. 1-8).
- Mannucci, T., van Kampen, E., de Visser, C., & Chu, Q. (2017a). Hierarchically structured controllers for safe uav reinforcement learning applications. In *AIAA information systems-AIAA infotech@ aerospace*.
- Mannucci, T., van Kampen, E., de Visser, C., & Chu, Q. (2017b). Safe exploration algorithms for reinforcement learning controllers. *IEEE Transactions on Neural Networks and Learning Systems*, *PP*(99), 1-13.
- Minsky, M. (1961). Steps toward artificial intelligence. *Proceedings of the IRE*, *49*(1), 8–30.
- Negnevitsky, M. (2002). *Artificial intelligence: A guide to intelligent systems*. Addison Wesley.
- Parr, R., & Russell, S. (1998). Reinforcement learning with hierarchies of machines. *Advances in neural information processing systems*, 1043–1049.
- Rummery, G. A., & Niranjan, M. (1994). *On-line Q-learning using connectionist systems*. University of Cambridge, Department of Engineering.
- Stolle, M., & Precup, D. (2002). Learning options in reinforcement learning. In *Abstraction, reformulation, and approximation: 5th international symposium, SARA* (p. 212-223). Springer Berlin Heidelberg.
- Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning: An introduction*. MIT press Cambridge.
- 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.
- Tesauro, G. (1992). Practical issues in temporal difference learning. *Machine Learning*, *8*, 257-277.
- Tesauro, G. (1995). Temporal difference learning and TD-gammon. *Communications of the ACM*, *38*(3), 58-68.
- Watkins, C. J., & Dayan, P. (1992). Q-learning. *Machine learning*, *8*(3-4), 279–292.
- Wei, C., Zhang, Z., Qiao, W., & Qu, L. (2016). An adaptive network-based reinforcement learning method for MPPT control of PMSG wind energy conversion systems. *IEEE Transactions on Power Electronics*, *31*(11), 7837–7848.
- Yao, J., Lin, C., Xie, X., Wang, A. J., & Hung, C. C. (2010). Path planning for virtual human motion using improved A* star algorithm. In *Information technology: New generations (ITNG), 2010 seventh international conference* (p. 1154-1158).