BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

USING REINFORCEMENT LEARNING AND INDUCTIVE SYNTHESIS FOR DESIGNING ROBUST CONTROLLERS IN POMDPS

VYUŽITÍ ZPĚTNOVAZEBNÉHO UČENÍ A INDUKTIVNÍ SYNTÉZY PRO KONSTRUKCI ROBUSTNÍCH KONTROLÉRŮ V POMDPS

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

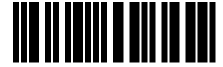
AUTHOR — **Bc. DAVID HUDÁK**
AUTOR PRÁCE

SUPERVISOR — **doc. RNDr. MILAN ČEŠKA, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2024

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment



157079

Institut:	Department of Intelligent Systems (DITS)
Student:	**Hudák David, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Machine Learning
Title:	**Using Reinforcement learning and inductive synthesis for designing robust controllers in POMDPs**
Category:	Artificial Intelligence
Academic year:	2023/24

Assignment:

1. Study the state-of-the-art planning and reinforcement learning methods for Partially Observable Markov Decision Processes (POMDPs) with the focus on model-based approaches.
2. Design an extension of the tool PAYNT allowing to combine inductive controller synthesis with reinforcement learning methods with the focus on robust controllers
3. Implement the extension with the use of existing frameworks for reinforcement learning methods.
4. Using suitable benchmarks, evaluate the performance as well as the practical usefulness of the implemented extension.

Literature:

- Kochenderfer, M.J., Wheeler, T.A., and Wray K.H, Algorithms for Decision Making, MIT Press 2021.
- Andriushchenko, R., Češka, M., Junges, S., and Katoen, J.P. Inductive synthesis of finite-state controllers for POMDPs. In UAI'22. Proceedings of Machine Learning Research.
- Andriushchenko, R., Češka, M., Junges, S., Katoen, J.P. and Stupinský, Š. PAYNT: A Tool for Inductive Synthesis of Probabilistic Programs. In *CAV 2021*. Springer.
- Antonoglou, I., Schrittwieser, J., Ozair, S., Hubert, T.K. and Silver, D. Planning in Stochastic Environments with a Learned Model. In *ICLR 2021.*
- Carr, S., Jansen, N., and Topcu, U. Task-aware verifiable RNN-based policies for partially observable markov decision processes. J. Artif. Int. Res. 72. 2022.

Requirements for the semestral defence:
Items 1, 2, and partially 3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Češka Milan, doc. RNDr., Ph.D.**
Consultant:	Andriushchenko Roman, Ing.
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	1.11.2023
Submission deadline:	17.5.2024
Approval date:	6.11.2023

Abstract

A significant challenge in sequential decision-making involves dealing with uncertainty, which arises from inaccurate sensors or only a partial knowledge of the agent's environment. This uncertainty is formally described through the framework of partially observable Markov decision processes (POMDPs). Unlike Markov decision processes (MDP), POMDPs only provide limited information about the exact state through imprecise observations. Decision-making in such settings requires estimating the current state, and generally, achieving optimal decisions is not tractable. There are two primary strategies to address this issue. The first strategy involves formal methods that concentrate on computing belief MDPs or synthesizing finite state controllers, known for their robustness and verifiability. However, these methods often struggle with scalability and require to know the underlying model. Conversely, informal methods like reinforcement learning offer scalability but lack verifiability. This thesis aims to merge these approaches by developing and implementing various techniques for interpreting and integrating the results and communication strategies between both methods. In this thesis, our experiments show that this symbiosis can improve both approaches, and we also show that our implementation overcomes other RL implementations for similar tasks.

Abstrakt

Jednou ze současných výzev při sekvenční rozhodováním je práce s neurčitostí, která je způsobena nepřesnými senzory či neúplnou informací o prostředích, ve kterých bychom chtěli dělat rozhodnutí. Tato neurčitost je formálně popsána takzvanými částečně pozorovatelnými Markovskými rozhodovacími procesy (POMDP), které oproti Markovským rozhodovacím procesům (MDP) nahrazují informaci o konkrétním stavu nepřesným pozorováním. Pro rozhodování v takových prostředích je nutno nějakým způsobem odhadovat současný stav a obecně tvorba optimálních politik v takových prostředích není rozhodnutelná. K vyrovnání se s touto výzvou existují dva zcela odlišné přístupy, kdy lze k problému přistupovat úplnými formálními metodami, a to buď s pomocí výpočtu beliefů či syntézou konečně stavových kontrolérů, nebo metodami založenými na nepřesné aproximaci současného stavu, reprezentované především hlubokým zpětnovazebným učením. Zatímco formální přístupy jsou schopné dělat verifikovatelná a robustní rozhodnutí pro malá prostředí, tak zpětnovazebné učení je schopné škálovat na reálné problémy. Tato práce se pak soustředí na spojení těchto dvou odlišných přístupů, kdy navrhuje různé metody jak pro interpretaci výsledku, tak pro vzájemné předávání nápověd. Experimenty v této práci ukazují, že z této symbiózy mohou těžit oba přístupy, ale také že zvolený přístup ke trénování agentů už sám o sobě řádově překonává současné systémy pro trénování agentů na podobných úlohách.

Keywords

Reinforcement learning, PAYNT, POMDP, interpretability, synthesis, PPO, sequential decision problems, finite state controllers, FSC, DQN, DDQN

Klíčová slova

Posilované učení, PAYNT, POMDP, interpretovatelnost, syntéza, PPO, sekvenční rozhodovací problémy, konečně stavové kontroléry, FSC, DQN, DDQN

Reference

HUDÁK, David. *Using Reinforcement learning and inductive synthesis for designing robust controllers in POMDPs.* Brno, 2024. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor doc. RNDr. Milan Češka, Ph.D.

Rozšířený abstrakt

Tato práce se zabývá sekvenčním rozhodováním v úlohách s nejistotou v modelech reprezentovaných částečně pozorovatelnými Markovskými rozhodovacími procesy. Plánování v takových prostředích je komplikované především v tom ohledu, že znalost agenta o současném stavu prostředí je limitována jeho senzory, které mu neposkytují vždy přesnou či úplnou informaci. Z tohoto důvodu vzniklo několik odlišných přístupů k řešení této nejistoty spočívající v rozlišné reprezentaci odhadu současného stavu. Formální přístupy tento odhad řeší buď s pomocí výpočtu tzv. beliefu, který reprezentuje vektor pravděpodobností výskytu v některém z možných stavů prostředí, nebo s pomocí paměťových uzlů konečně stavových kontrolérů. Právě druhý přístup je reprezentován nástrojem nazvaným PAYNT, který tvoří jeden ze dvou hlavních bloků této práce. Neformální přístupy pak k odhadu současného stavu prostředí používají jiných prostředků, například zpětnou vazbu rekurentních neuronových sítí či transformerů. Právě první jmenovaný přístup v kombinaci se současnými algoritmy posilovaného učení, jmenovitě DQN, DDQN a PPO, reprezentuje druhý hlavní blok této práce.

V této práci jsou popsány limitace současných přístupů a to především z hlediska škálovatelnosti, robustnosti, vysvětlitelnosti, stability či výběru vhodných parametrů. Formálně založená řešení, mimo jiné PAYNT, trpí právě malou škálovatelností a nutností znát celý model prostředí. PAYNT a jemu příbuzné nástroje jsou pak obvykle limitovány velikostí na poměrně malé úlohy obsahující několik desítek tisíc různých pozorování. Na druhou stranu, posilované učení, byť jej lze škálovat na mnoho různých reálných úloh jako je řízení aut či automatizovanou průmyslovou výrobu, neumožňuje v současnosti garantovat požadované vlastnosti a mnohdy trpí silnou nestabilitou učení. Z tohoto důvodu je jedním ze současných výzkumných směrů snaha eliminovat tyto jeho limitace ať už různými přístupy pro formální verifikaci či bezpečné posilované učení za využití upravených hodnotících metrik, nebo tvorbou náhradních modelů. Právě náhradní modely jsou dnes reprezentovány celou řadou přístupů, ať už je to tvorba programatického posilovaného učení, kdy je paralelně ke klasickému posilovanému algoritmu učen ještě interpretovatelná politika ve formě kódu z předem daných primitiv. Druhým přístupem je pak tvorba prototypových sítí, kdy namísto klasických neuronů jsou využity interpretovatelné prototypy a pro práci s daty jsou využity speciálně transformační vrstvy. Třetím přístupem, kterým se do nějaké míry inspiruje i tato práce, je extrakce konečně stavových kontrolérů (FSC) přímo z naučené politiky.

Navržený přístup v této práci spočívá v kombinaci PAYNTu a posilovaného učení. Je založený na tom, že agenti trénovaní algoritmy posilovaného učení se naučí řešit zvolenou úlohu a následně je takto naučený agent interpretován navrženým algoritmem pro interpretaci politik založených na neuronových sítích. Navržený proces interpretace funguje na vyhodnocování provedených trajektorií v prostředí, kdy takto interpretovaný agent poskytuje PAYNTu orákulum ve formě doporučení, jakým způsobem prořezat exponenciální prostor potenciálních konečně stavových kontrolérů pro jednotlivá pozorování, pro která pozorování doporučuje hned ze začátku zvětšit paměť a jakým způsobem by upravoval paměť v dalších iteracích. V rámci navrženého řešení existuje pak i opačná cesta, a to ve formě extrakce konkrétních konečně stavových kontrolérů z PAYNTu, které jsou pak využívány pro vylepšení explorační politiky posilovaného učení. Toto vylepšení může probíhat buď tím způsobem, že extrahované FSC zvyšuje pravděpodobnosti akcí, které by měl agent dle FSC hrát, nebo přímo sám takové akce vybírá. Z takto zahraných akcí (trajektorií) se pak učí algoritmus posilovaného učení, což mu mnohdy umožňuje snadněji najít alespoň sub-optimální řešení v prostředích s řídkou odměnou.

Součástí navrženého řešení je kompletně vlastní implementace učící smyčky vycházející z nejnovějších přístupů v rámci posilovaného učení s použitím implementace učících algoritmů z knihovny TensorFlow Agents. Tuto implementaci rozšiřuje nový přístup k práci s kódováním pozorování založený na valuacích z nástroje Storm, který v této práci implementuje základní simulátor prostředí. Dále je v této práci diskutován problém s dynamickým akčním prostorem, kdy je navržen nový přístup pro maskování nelegálních akcí s pomocí obalování původních politik, které takové filtrování neumožňují. Pro vylepšení počáteční stability učících algoritmů byla také navržena jednoduchá metoda spočívající v restartování iniciálního nastavení vah politik reprezentovaných neuronovými sítěmi, kdy pro počáteční nastavení agenta se využívá nastavení s nejlepším počátečním výkonem.

Experimentální část této práce pak ukazuje potenciální síly i slabiny navrženého řešení. Ukazuje se, že pokud byl natrénován kvalitní agent, který umí rozumně řešit danou úlohu, navržená interpretace je schopna předat syntéze konečně stavových kontrolérů v PAYNTu významnou informaci pro její vylepšení. V některých úlohách tak dochází i ke zvednutí výkonu z neschopnosti jakkoli vyřešit danou úlohu na syntézu rozumného kontorléru. Nicméně, hlavní limitací navrženého řešení je současná úroveň posilovaného učení pro tento typ úloh, kdy současné přístupy mnohdy nejsou zcela stabilní a mnohé již existující implementace posilovaného učení mnohdy prohrávají i s vyváženou kostkou. V této práci jsou původní přístupy významným způsobem překonávány a díky tomu na mnohých úlohách lze vyextrahovat kvalitní nápovědu. Přesto stále existují limitace, které nebyly překonány, například v případě modelů s tzv. řídkou odměnou, kdy je problém v daném prostředí dosáhnout odměny pro motivaci agentů.

Dílčími přínosy této práce jsou pak alespoň částečné prozkoumání a vyhodnocení velmi široké oblasti posilovaného učení, navržení a implementace mnoha užitečných přístupů pro jeho vylepšení, interpretaci a následné zpracování získaných dat v rámci PAYNTu a také bylo implementováno zprostředkování nápověd v rámci posilovaného učení.

Using Reinforcement learning and inductive synthesis for designing robust controllers in POMDPs

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Mr. doc. RNDr. Milan Češka, Ph.D. The supplementary information was provided by Ing. Roman Andriushchenko, Ing. Filip Macák and Bc. Daniel Karásek. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
David Hudák
May 14, 2024

Acknowledgements

I would like to express my gratitude to my supervisor, Milan Češka, and my consultants, Roman Andriushchenko and Filip Macák, who provided me with help and guidance throughout this thesis. I am also thankful to my dear friends Honza, Michal, Petr, Radim, Štefan, Martin, Daniel, and Tomáš, whose unwavering support was invaluable during my studies. Additionally, I extend my appreciation to all the teachers at Gymnázium Brno, Vídeňská, whose guidance sparked my interest in further studies. Last but not least, I am deeply grateful to my family for their unconditional support from the very beginning.

Contents

1 Introduction **4**
1.1 Thesis Approach 5
1.2 Thesis Structure 6

2 Preliminaries **7**
2.1 Partially Observable Markov Decision Processes 7
2.1.1 Examples of POMDPs 8
2.2 Model-Based Approaches for POMDPs 9
2.2.1 Belief-State MDPs 9
2.2.2 Finite State Controllers 10
2.2.3 PAYNT 11
2.2.4 Approximate Model-Based Approaches 12
2.3 Reinforcement Learning 12
2.3.1 Machine Learning 12
2.3.2 Reinforcement Learning 14
2.3.3 Discount Factor 15
2.3.4 Exploration versus Exploitation 15
2.4 Neural Networks 16
2.4.1 Feedforward Neural Networks 16
2.4.2 Convolutional Neural Networks 17
2.4.3 Recurrent Neural Networks 18
2.4.4 Activation Functions 20
2.4.5 Backpropagation Algorithm 20
2.5 Reinforcement Learning Algorithms 21
2.5.1 Taxonomy of Reinforcement Learning Algorithms 21
2.5.2 Value Based Learning 22
2.5.3 Policy-Based Algorithms 23
2.5.4 Actor, Critic and Actor-Critic Agents 24
2.5.5 Reinforcement Learning for POMDPs 25
2.5.6 DQN and DDQN 26
2.5.7 Proximal Policy Optimization 27

3 Challenges **29**
3.1 Explainability 29
3.1.1 Goal 30
3.1.2 Current Approaches 30
3.1.3 Difference in Belief Implementation of RL and FSC 31
3.1.4 Explaining LSTMs 32

3.2 Stability and Parameter Selection . . . 32
3.3 Scalability . . . 34
3.4 Robustness . . . 34
3.5 Sparse Reward . . . 35

4 Proposed Approach 36
4.1 Overall Schema . . . 36
4.2 Reinforcement Learning Design . . . 37
4.2.1 Environment Wrapper and Agents . . . 37
4.2.2 Replay Buffer and Collector Drivers . . . 38
4.2.3 Weight Restarting . . . 40
4.2.4 Algorithms Optimization . . . 40
4.3 Communication Between PAYNT and RL toolkit . . . 41
4.3.1 Conversion of the Action Mapping . . . 41
4.3.2 Format of Communication . . . 42
4.3.3 File Communication . . . 42
4.3.4 Integration of RL to PAYNT . . . 42
4.4 Usage of RL Oracle in PAYNT . . . 42
4.4.1 Reduction of the Design Space . . . 43
4.4.2 Selection of the Initial Memory . . . 44
4.4.3 Memory Update Prioritizing . . . 44
4.5 Usage of FSCs in RL Approach . . . 44
4.5.1 FSC as Hard Collector Policy . . . 45
4.5.2 Soft FSC with PPO Policy . . . 46

5 Technical Details 47
5.1 Environment . . . 47
5.1.1 Template and Properties . . . 47
5.1.2 Storm . . . 48
5.1.3 Stormpy Model and Simulator . . . 48
5.1.4 TensorFlow Wrapper . . . 49
5.1.5 Reward Shaping . . . 49
5.2 Observation Representation . . . 50
5.2.1 Stormpy Valuations . . . 51
5.3 Dynamic Action Space . . . 51
5.3.1 Masking . . . 53
5.3.2 Environment Action Filtering . . . 54
5.3.3 Policy Wrapping . . . 55
5.4 Interpretation . . . 55
5.4.1 RNN Feedback Approximation . . . 55
5.4.2 Model-Free Approximation . . . 56
5.4.3 Tracing Interpret . . . 57

6 Experiments 60
6.1 Experimental Setting and Benchmark Selection . . . 60
6.2 Research Questions and Answers . . . 61
6.3 Summary . . . 72

7 Conclusion 73

7.1 Future Research 73

Bibliography **75**

A Experimental Setting **82**
A.1 Agents Arguments 82
A.2 Environment Setting 82
A.3 PAYNT Setting 84
A.4 Used Models 84
A.5 Interpretation Arguments 84
A.6 PAYNT-RL Loop Arguments 84

Chapter 1

Introduction

Today, reinforcement learning is widely used in various domains. Its notable characteristic of learning without requiring an exact model or an existing solution has greatly facilitated the development of agents in various disciplines. Numerous applications can be found, including the advancement of sophisticated natural language processing (NLP) tools such as large language models (LLMs) such as ChatGPT [55, 53], Gemini [31, 28], and GitHub Copilot [29, 39], as well as professional translation tools [41] or algorithms for social networks used in advertising recommendation systems [8]. Reinforcement learning also serves as one of the main strategies for planning in sequential decision-making scenarios, encompassing applications such as autonomous vehicles [68, 9], robot control [19], management of aircraft equipped with airborne collision avoidance systems [38, 49] (ACAS), or playing video games [76, 65, 20].

The formal basis for sequential decision making is established by the Markov decision process [43] (MDP), which outlines states, actions, and the probabilities of transitioning to future states. The goal in this type of decision making is to determine the best actions within states according to some reward model, aiming to maximize the rewards received. Often, the exact state cannot be observed in real-world situations, due to imperfect or limited sensors, or the inability to maintain all relevant system data. This requires the use of the partially observable Markov decision process framework [42] (POMDP), which builds on the basic MDP by incorporating uncertain observations of current states. This introduces a challenge in estimating the current state, typically by incorporating some form of memory. However, creating an optimal controller for environments modeled by POMDPs is generally considered to be intractable [48, 12, 3, 73].

Today, we can employ various approaches for sequential decision-making under uncertainty. There exist model-based approaches, which are based on the computation of belief MDPs [12], where each state represents a set of probabilities of each state, or on the synthesis of finite-state controllers [3, 48, 1, 2] (FSC), which uses memory nodes to differentiate suitable actions given some observations. These approaches are usually complete, robust, and we can verify them, but often lack scalability, and in general need to know the entire underlying model, or at least its approximation, if we consider the model-based Monte Carlo tree search [4] (MCTS).

On the other hand, if we do not know the model, we can use some model-free methods based on the reinforcement learning framework [66, 5, 33, 61]. These methods usually scale well and can achieve or overcome controllers that could develop humans. However, the main drawback of modern reinforcement learning approaches is their lack of interpretability and robustness guarantees. These methods usually rely on deep learning technologies, such as

the deep neural network (DNN) [30], and it is challenging to understand their inner workings and ensure their performance and safety [78, 17, 18]. An approach to ensure safety may be formal verification of models based on neural networks; however, even state-of-the-art approaches [77, 35, 25] do not scale enough to verify commonly used models.

The drawback of using formal approaches is that they require a complete model and are not easily scalable [18]. As a result, numerous experiments have been conducted to develop more scalable but still verifiable solutions. One such approach [18] involves using RNN-based (recurrent neural network) RL algorithms, where an agent is trained using an RL algorithm, and then a much smaller FSC (finite state controller) is extracted from this large and unmanageable policy. In cases where the tool cannot extract an FSC that satisfies certain formal properties (specified in LTL) within a given condition (size), the original agent is enhanced with additional data. If the requirements are met, the learned policy is considered correct and the original agent or the extracted FSC can be used. A different approach could involve the use of shielding [17] as a strategy. In this technique, the toolkit utilizes the knowledge of the model to calculate the winning regions for a given POMDP before executing the RL algorithm. The agent is then restricted to selecting only actions that lead to these winning regions.

Another possible alternative approach is to address the challenge of eXplainable AI [22, 6] (XAI) by interpreting complex neural network models using explainable models. The authors in [70] suggest a method called programmatic reinforcement learning (PIRL), which uses human-readable properties to facilitate learning and generate interpretable policies in the form of code. In this approach, a pre-trained neural „oracle“ is employed to guide the learning process. Another proposal, as mentioned in [54], extends this solution by incorporating simultaneous training instead of relying on a pre-trained oracle. Additionally, a similar approach involves the use of a wrapper that is trained simultaneously, called the prototype wrapper network [40]. This model mimics the behavior of the network through state encoders, transformations, and prototypes (complex functions) that humans can understand.

An alternative approach is PCMRPP [21] (POMDP file Creator for Mobile Robot Path Planning), which does not focus on modifying the agent, but instead on altering the environment through the creation of sparser representations. This involves the use of POMDPs with sparser matrices or with lower granularity after discretizing the continuous space. As a result, this approach enables more scalable solutions, since the task becomes easier to solve.

However, despite the implementation of various strategies to address the challenges of scalability and verifiability in policy generation for partially observable Markov decision processes (POMDPs), the results remain inadequate for practical applications such as autonomous driving [9] or aircraft safety [49]. Furthermore, the current surge in artificial intelligence (AI) has led to efforts to regulate AI[1], leading to a greater need for security assurance.

1.1 Thesis Approach

The fact that we often desire both interpretable and verifiable policies, as well as a more complex policy providing ability to scale, such as a neural network, poses a challenge. We

[1] For example, see https://www.europarl.europa.eu/news/en/press-room/20231206IPR15699/artificial-intelligence-act-deal-on-comprehensive-rules-for-trustworthy-ai

consider the neural network as a black box and aim to facilitate the exchange of information between these models. Although various methods have been proposed to address this problem [18, 70, 40, 6, 22], no universally effective approaches have been implemented.

This thesis mainly explores the integration of PAYNT's formal and precise approach [2, 3, 1, 48] with reinforcement learning, which is inspired by the approaches [17] and [18]. We propose a novel interpretation approach to obtain hints for the FSC synthesis performed by PAYNT. The approach is based on the interpretation of the tracing of policies, which are represented by neural networks using LSTMs for state estimation. In this thesis, we also designed and developed a toolkit for reinforcement learning based on TensorFlow Agents [67] and we also improved current reinforcement learning approaches with semantic learning, the novel method to solve the dynamic action space called policy wrapping, and the availability to use FSCs for behavior and critic policies. Our experiments show that our approach overcomes other similar approaches, which work with similar models, primarily because our novel semantic learning significantly improves the stability and performance of our agents. Furthermore, our experimental results demonstrate that with adequate training and interpretation of well-trained policies, it is feasible to generate effective policies for larger models. Moreover, our findings indicate that PAYNT can improve RL agents by optimizing the initial search for episodes that lead to goal states in models with sparsely distributed rewards.

1.2 Thesis Structure

Before we start with the main issues related to this thesis approach, we outline essential preliminaries to understand the overall concept of decision making under uncertainty, where we describe various approaches to learning policies, starting from the concept of formal model-based approaches, and ending with complex reinforcement learning algorithms based on neural networks. Then we describe the main challenges that pose decision-making in environments with uncertainty and some related approaches on how to face them. In the following chapter, we outline the main idea and concepts of our approaches, followed by a chapter describing some important technical details, including the observation encoding, environment wrapping, dealing with dynamic action space, or our approach for interpretation. Then we outline experiments and benchmarks for our implementation in terms of comparison with other related approaches.

Chapter 2

Preliminaries

This chapter introduces the fundamental principles involved in the development of controllers for partially observable Markov decision processes (POMDPs). We begin by detailing the problem, formally presenting the model, and addressing the primary challenge of making sequential decisions under uncertainty. The subsequent section examines established model-based strategies for crafting robust controllers with a known model. Next, we explore the basic principles of an alternative strategy, reinforcement learning. This is followed by a brief overview of a crucial component of reinforcement learning algorithms, namely neural networks, which work as a useful functional approximator. Finally, we discuss particular reinforcement learning algorithms, specifically for designing controllers under uncertainty.

2.1 Partially Observable Markov Decision Processes

In this thesis, we focus on solving sequential decision problems [43], where we try to construct optimal and robust decision systems (controller) on general framework called a Markov decision process.

Definition 2.1.1 (Markov Decision Process) *A Markov Decision Process (MDP) is a tuple $M = (S, A, T, R)$ consisting of a finite set of states S, a finite set of actions A, a transmission probability function T that maps $S \times A$ to a distribution over S, and a reward R for transitioning from state $s \in S$ to $s' \in S$ by taking some action $a \in A$ [18, 66].*

MDPs possess the Markov property, which states that the probability of a transition depends only on the current state and not on the history of states. This can be formally expressed as $p(x_n|x_1, x_2, \ldots, x_{n-1}) = p(x_n|x_{n-1})$[11]. Our goal is then to create some controllers that, given some state $s \in S$, select optimal action given some specification, for example, maximize the reward obtained in environment and reach the final state. Another important concept given the MDP framework is the sink states, which correspond to states, and the transition property is $P(s|s, a) = 1$ for all actions $a \in A$.

However, in reality, we usually cannot obtain an exact MDP states, as we have imprecise sensors, we cannot predict behavior of our opponent and, in general, we cannot observe whole information from whole environment represented by MDP. For example, we may have a camera [65], but we can observe only things that are in front of it. This leads us to the concept of a partially observable Markov decision process [42].

Definition 2.1.2 (POMDP) *A Partially Observable Markov Decision Process (POMDP is a tuple $\mathcal{M} = (S, A, T, R, Z, O)$. It extends the Markov Decision Process $M = (S, A, T, R)$ with a finite set of observations Z and an observation function $O(s) = z$, which returns observations $z \in Z$ for each state $s \in S$. If there is only a single state s that leads to observation z, we call z trivial [2, 18, 42].*

It has been previously mentioned that MDPs (Markov Decision Processes) possess the Markov property, indicating that the transition probability $P(s_{t+1}|s_t, a_t)$ is independent of prior states [42]. In the case of a POMDP, this Markovian transition characteristic persists. However, given that only partial observations z are accessible and the precise state s remains unknown, our strategies to address POMDPs need to incorporate a mechanism to approximate the current state s. Consequently, decision-making in POMDPs is not based merely on a single observation z_t, but on the sequence of observations $z_{1:t}$, which allows us to deduce the current state to some extent.

However, since the infinite history of observations $z_{1:t}$ may be intractable, we may use various approaches to deal with state uncertainty. These include belief MDPs [12], finite state controllers [3], alpha vectors [42, 43], or approaches based on deep learning such as recurrent neural networks [33] or transformers [23]. In the following subsection, we discuss some examples of POMDPs.

2.1.1 Examples of POMDPs

Even if the general framework of reinforcement learning is Markov decision process (MDP), in this subsection we show that partially observable Markov decision process (POMDP) are actually really common situation.

Crying Baby Problem

The Crying Baby Problem is one of the simplest problems described by POMDPs. It involves two states, three actions, and two observations. The baby is sated or not, but we cannot observe this directly. Instead, we can only observe whether the baby is crying or not. We can feed the baby, sing to it, or ignore it. If the baby is not fed, we receive a high negative reward regardless of the action taken. Feeding and singing result in smaller negative rewards. The goal is to minimize the time spent in the unsated state [43].

Grid Problems

One type of benchmark problem involves an agent moving in a discrete grid (or maze). It is typically only able to observe certain information from the environment. For example, in the **Maze** task, the agent can only observe the neighboring states (whether there is a wall or not). In the standard **Grid** problem, the agent is uniformly placed in some state on the grid and can only observe whether it is in the goal state or not. The objectives of these tasks can vary; the agent may only need to reach the goal state, collect rewards placed in the grid, or avoid certain obstacles, etc. One of the benefits of these problems is that we can adjust the size of the grid [17, 18].

Video Games

Despite having a complete simulator at our disposal in this case (the video game), we typically use POMDPs to the incomplete information available about the system. Taking

Starcraft [71] as an example, our knowledge is limited to the area where we are currently situated, some details of the map, and the rest of information is hidden either outside the current frame or by fog of war. Consequently, to make accurate decisions, it is necessary to remember the enemy's location or our previous actions, as we cannot gather all the information in a single frame.

Healthcare

Healthcare is another prevalent field where POMDP is often utilized. Typically, we can collect certain data, such as body temperature, blood pressure, conduct various tests such as blood chemistry tests, laparoscopy, and so on. However, the overall condition of the subject under observation (patient) remains unknown, and we are limited to dealing with certain latent states (observation). Furthermore, there is usually a delay in the response to medical treatment from the time it begins to work, requiring us to rely on statistical assumptions regarding the outcome and possibly conduct additional tests. For example, a study focused on POMDPs attempted to model and examine decision making in POMDPs for patients with sepsis [69].

Financial Markets

One of the other areas in which we consider POMDP to model system behavior is the financial markets, where we usually have some information from stock exchanges, history of transactions, company, bank or country financial reports, statements in the media or insider gossip. However, exact state of the market remains unknown, and thus we have to work with state uncertainty and make predictions given available information and speculations. There are multiple approaches, how to model markets [14] and make predictions with decisions and usually work with some history-based approaches based on the POMDP framework.

2.2 Model-Based Approaches for POMDPs

We begin our discussion on decision-making in POMDPs with model-based approaches. These algorithms utilize established or learned models and confront the primary challenge in POMDP decision-making: history. We explore three distinct strategies: first, the construction of belief MDPs that compute beliefs to estimate historical data, and second, finite-state controllers (FSCs) that use a finite-state automata for internal memory representation. We also outline one of the most important aspects of this thesis, PAYNT, which is based on the synthesis of FSCs. Alternatively, we describe the third approach based on a model-based reinforcement learning strategy that employs an approximate model with latent states. The following subsections will outline the core principles, benefits, and limitations of these methodologies.

2.2.1 Belief-State MDPs

Since we have manageable solutions for MDPs, one of the decision-making strategies in POMDPs involves transforming them into MDPs. Nevertheless, given that our information is based solely on observations, it becomes necessary to formulate belief-state MDPs. These are identical to MDPs, but each state is associated with a belief b, symbolizing a vector of probabilities $b \in R^n$ for being in states $s \in S$, where n represents the total number of states in POMDP. To execute this method, a complete model of POMDP is required.

In order to calculate the probability of the succeeding state b_{t+1}, it is necessary to determine the transition function $\tau(b_{t+1}b_t, a_t) = P(b_{t+1}b_t, a_t)$, which is in form of:

$$P(b_{t+1}|b_t, a_t) = \sum_o P(b_{t+1}|b_t, a_t, o_t) \sum_{s_{t+1}} P(o|s_{t+1}) \sum_s T(s_{t+1}|s, a)b(s)$$

where $P(b_{t+1}|b_t, a_t, o_t)$ is Krockner delta function [42]. The Krockner delta function may be computed exactly if the original state space is discrete. If we want to calculate an immediate reward for taking action a in belief b, we use formula $R(b, a) = \sum_s R(s, a)b(s)$.

The main drawback of using belief-state MDPs is the fact that the set of possible beliefs β is infinite even for simple and small POMDPs. For example, as we can see in Figure 2.1, we obtain an infinite sequence of probabilities of being in the state s_2 or s_3 for the blue observation if we always take action α.

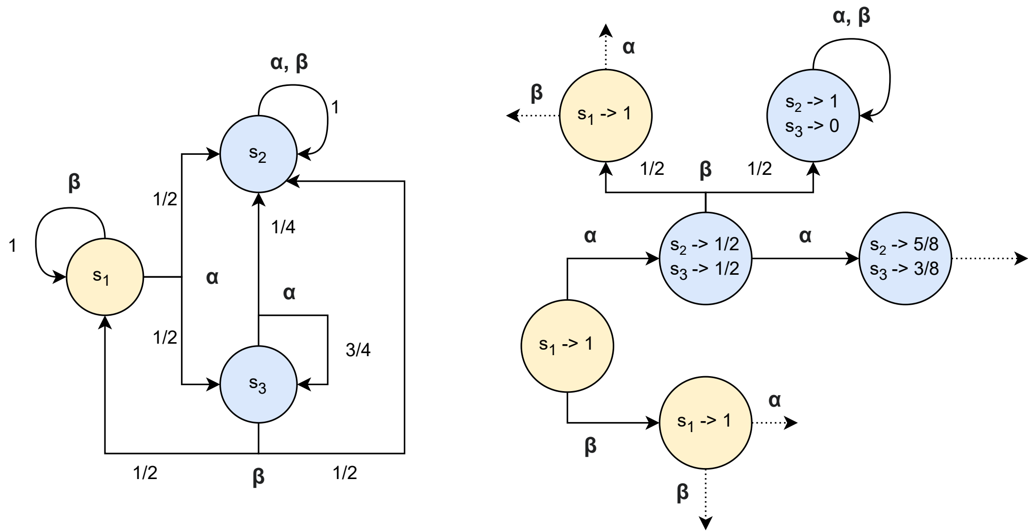


Figure 2.1: Belief state MDP (right) for a given POMDP (left), inspired from [12].

Many advanced methods address the problem of potential state explosion by employing finite approximations of belief MDPs [36], basic cut-off techniques, or more complex belief clipping strategies [12] to handle the potential infinite nature of belief MDPs. Moreover, many reinforcement learning solutions to decision-making algorithms use some algorithms to compute the approximate belief with neural networks [33, 23]. Some of the other solutions then use pre-computed belief in hybrid approaches [5], where the input of the neural network is the current observation and the pre-computed belief.

2.2.2 Finite State Controllers

The second traditional approach to decision making in POMDPs is the usage of finite-state controllers (FSCs).

Definition 2.2.1 *A k-FSC for a POMDP $\mathcal{M}$ is a tuple $\mathcal{A} = (N, n_I, \alpha, \delta)$, where N is a finite state of k memory nodes, n_I is the initial memory node, α is the action mapping $N \times Z \to Distr(A)$, and δ is the memory update function $N \times Z \times A \to Distr(N)$ [18, 64].*

FSCs are usually considered compact representations of policies and are generally described as Mealy machines [3]. The application of FSCs to POMDP then leads to an induced Markov chain $\mathcal{M}^{\mathcal{A}} = (S^{\mathcal{A}}, (s_0, n_I), P^{\mathcal{A}})$.

Compared to the previous approach, belief-state MDPs, FSC does not lead to infinite state space, and overall leads to more compact solutions. However, one of the main issues with FSCs is the necessity of obtaining them. Approaches to inducing FSCs are usually based on an exponentially large design space that contains all possible k-FSCs. We can see some examples from the previously mentioned POMDP in Figure 2.2.

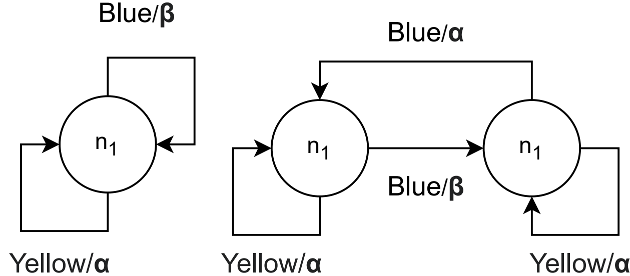


Figure 2.2: Example of 1-FSC (left) and 2-FSC (right) for POMDP from Figure 2.1.

The developed toolkits [48, 3] then try to choose a sufficient k and prune the design space to select the most optimal FSC for a given POMDP. Moreover, the pruning of the design space and the selection of the optimal k are the main task of this thesis when we try to estimate these parameters from the RL oracle. Similar but slightly different is the approach [18], where they try to compute the optimal FSC from the trained policy based on neural networks. These toolkits can verify FSCs its application on POMDPs and inducing MC (Markov chains).

2.2.3 PAYNT

Since we discussed the algorithms for complete solutions of POMDPs, we should describe the main toolkit used in this thesis, which is based primarily on the second-mentioned approach, the inductive synthesis of deterministic FSCs.

PAYNT (Probabilistic progrAm sYNThesizer) is a tool for the automated synthesis of probabilistic programs [2]. The purpose of this toolkit is to synthesize FSC given some program with holes (sketch) and PCTL specification, which adjust the synthesis goals. It is based on the principle of oracle-guided reasoning through an exponentially large design space. The current symbiotic approach combines the search for policies with approximate solutions to explore beliefs [1]. It works with various input sketches based on PRISM, Cassandra, JANI, or DRN, combined with the PCTL specification, which describes the goals for synthesis performed by PAYNT. Its default implementation uses Z3 SAT solver, which enables to reduce design space by detecting unsatisfiable solutions and enables verification of required properties.

Currently, PAYNT incorporates a variety of techniques that aim to minimize design space and accurately estimate the minimal necessary memory requirements [3]. The first challenge is tackled through several strategies that identify optimal actions for each observation, employing counter-example inductive methods, deductive abstract refinement, or a combination of multiple strategies. The second issue is managed through precise memory

injections, designed to augment memory solely for observations that require additional capacity, thereby decreasing the size of the resulting FSCs. Both of these issues are addressed within this thesis.

2.2.4 Approximate Model-Based Approaches

In previous three subsections, we described the formal and robust approaches, which produce safe and verifiable policies for POMDPs. However, their main limitation is that we have to give them exact model and, moreover, the size of the model highly affects the performance of outlined approaches. We can increase the performance of these approaches by using some form of approximation, as we can use the reinforcement learning algorithm to approximate the model (if the model is unknown) or to approximate the current state by the so-called latent states.

Within these approaches, we may, for example, include stochastic MuZero [4], which is based on the estimation of the model by recurrent neural networks or vector-quantized variational autoencoders combined with Monte Carlo tree search, which uses the learned approximation to search the model and create an online policy. It solves many issues of previous algorithms, which are, for example, the inability to plan within stochastic models by introducing so-called after-states and stochastic search. This algorithm works with approximating states of the system using the so-called latent states, which are representations of state estimation given some history of observation $o_{1:t}$ formed by deep learning. Moreover, these algorithms must estimate the values of these states and the transition function $T(s' \mid s, a)$, where s corresponds to the latent state in this case, and also the after-state dynamics and predictions.

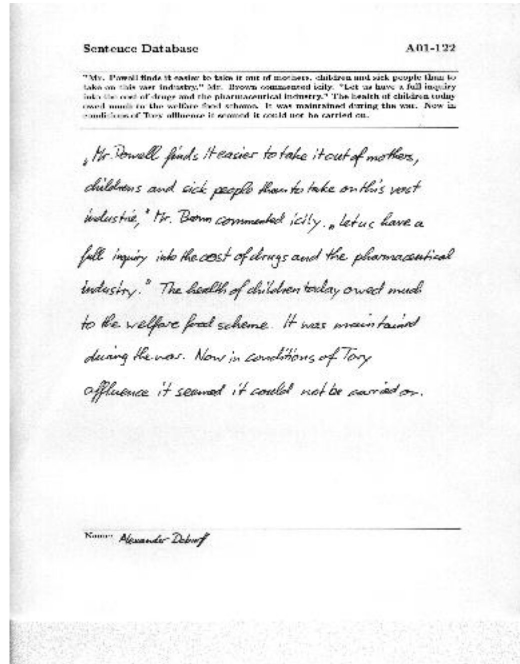
This approach provides robust state-of-the-art results for many different tasks like Backgammon, 2048 or Go, but are still limited by the quality of model estimation. However, the disadvantages of this approach is the high reliability of the quality of the approximated model, where small differences between reality and approximated model can lead to wrong results. Moreover, these approaches usually do not provide that high scalability as model-free algorithms.

2.3 Reinforcement Learning

In this section, we focus on the fundamental concepts of machine learning, with a particular emphasis on reinforcement learning, which is necessary to better understand the topics discussed in this thesis.

2.3.1 Machine Learning

Before we begin to discuss reinforcement learning, let us first consider the broader context of machine learning. It is a type of artificial intelligence that provides computers with the ability to learn from data without being explicitly programmed. They are able to identify patterns and trends in the data and use them to make predictions and decisions [30]. Many books, college courses, and articles mention the definition from [51], which says *"A computer program is said to learn from experience **E** with respect to some class of tasks **T** and performance measure **P**, if its performance at tasks in **T**, as measured by **P**, improves with experience **E**."*



Sentence Database A01-122

"Mr. Powell finds it easier to take it out of mothers, children and sick people than to take on this vast industry." Mr. Brown commented icily. "Let us have a full inquiry into the cost of drugs and the pharmaceutical industry." The health of children today owed much to the welfare food scheme. It was maintained during the war. Now in conditions of Tory affluence it seemed it could not be carried on.

„Mr. Powell finds it easier to take it out of mothers,
childrens and sick people than to take on this vast
industrie," Mr. Brown commented icily. „Let us have a
full inquiry into the cost of drugs and the pharmaceutical
industry." The health of children today owed much
to the welfare food scheme. It was maintained
during the war. Now in conditions of Tory
affluence it seemed it could not be carried on.

Name: Alexander Dobrof

Figure 2.3: Example of image from IAM dataset from https://fki.tic.heia-fr.ch.

Task T refers to the problem that we are trying to solve using a machine learning algorithm. Generally, these tasks are issues that cannot be resolved with a straightforward and logical approach by writing some computer code. Examples of activities involving machine learning include operating a vehicle or robot, facial recognition technology, machine translations, anomaly detection, or suggestions of content on social networks [30, 66, 11, 8].

In most cases, we do not have an exact algorithm, but we have a great amount of data. For example, one of the approaches for optical character recognition (OCR) that uses a complex structure of neural networks known as transformers, TrOCR [47], was pre-trained on hundreds of millions of unlabeled data. The objective of this approach is to transform documents from a visual format into a digital form that can be utilized, for instance, in administrative tasks. An example of input data is shown in Figure 2.3.

An important part of machine learning is the usage of something that tells us how well our solution (model) works and what results we can expect when it is deployed in the real world. The task at hand usually determines the metric used. There are a variety of metrics that can be used, such as cost functions, error functions, precision, and cumulative reward (return) [66, 30]. For example, when we work with classification tasks, such as image recognition, we use the *cross-entropy* error function:

$$H(X) = -\sum_{n=1}^{N} \{t_n * \ln(y_n) + (1 - t_n) * \ln(1 - y_n)\}$$

where t_n is correct class, y_n is predicted class by the model and N is the number of examples. Another example of a common error function is the mean square error defined as:

$$MSE = \frac{1}{N} \sum_{i}^{N} (\widehat{y}_i - y_i)^2$$

where $\widehat{y_i}$ is estimation of value produced by some ML algorithm for some input and y_i is the real value. Or in the case of K-means clustering, we use *the distortion measure*:

$$J = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} ||\boldsymbol{x_n} - \boldsymbol{\mu_k}||^2$$

where N is number of examples, K number of clusters, $\boldsymbol{\mu_k}$ is cluster vector, $\boldsymbol{x_n}$ is data point, $r_{nk} = 1$ if data point belongs to cluster and $r_{nk} = 0$ otherwise [11].

Usually, we use different approaches to measure performance; we can also talk about different datasets to measure performance, training for model learning, and testing dataset for real task evaluation. Another approach is to modify the error functions using a fault tolerance coefficient[1] [11]. We will talk more about measuring the performance of models in the following sections and chapters, as it is a key factor for implementing learning algorithms.

Different machine learning algorithms acquire experience in different ways. Generally, there are three main categories: **supervised** learning, **unsupervised** learning, and **reinforcement** learning. Supervised learning requires that all data be labeled, whereas unsupervised learning does not require any labeling and is usually used for exploration of the provided dataset. Reinforcement learning, on the other hand, obtains data by selecting actions and interacting with the environment and therefore cannot be included in the categories mentioned previously [66, 11]. Additionally, there are some combinations of these approaches, such as semi-supervised or self-supervised learning, that are not the focus of this thesis.

2.3.2 Reinforcement Learning

Reinforcement learning represents a branch of machine learning in which an agent learns through feedback from an environment. This framework employs a trial-and-error strategy, trying various actions and noting the results or rewards [66], in order to maximize these rewards. Similarly to learning processes in living organisms, beneficial actions yield rewards (survival), and detrimental actions result in penalties (death). A significant benefit of reinforcement learning over supervised learning is its ability to surpass the potential teacher and devise superior solutions.

In case of reinforcement learning, our goal is to create a decision-making process, policy, which maximizes following optimization criterion:

$$\max_{\pi \in \Pi} \mathbb{E}_{\pi}(R) = \max_{\pi \in \Pi} \mathbb{E}_{\pi}(\sum_{t=0}^{\infty} \gamma^t r_t) \tag{2.1}$$

Here, R is the expected cumulative future discounted reward (return), $\gamma \in [0, 1]$ is the discount factor, r_t is the reward at the time step t based on the selection of the action of the policy π, Π is the set of all possible policies, and $E_{\pi}(\cdot)$ is the expectation with respect to policy π [66]. There also exist some modified optimization criterion, where we try to minimize taking dangerous action even with consideration of decreasing overall reward. Some approaches are mentioned in [27]. The general framework with which reinforcement learning works is the Markov decision process (MDP), where the MDP represents the environment the agent explores.

[1] For example we can penalize less machine doctor if he cures healthy person and penalize more for not curing ill person.

In reinforcement learning, we typically discuss the **value** and **q-value** functions, which calculate the (Q) values for each state of the environment. These values represent the expected cumulative reward from each state; in the case of a value, we calculate the following $V^{\pi}(s)$ as:

$$V^{\pi}(s_t) = \sum_{t'=t}^{T} \mathbb{E}_{\pi}[\gamma^{t'-t} r(s_{t'}, a_{t'}) \mid s_t]$$

where π is the policy we follow, s_t state in time t, T number of last time (may be ∞), γ discount factor, r reward function and a action selected by policy π. This means that we want to predict the value of state s according to some policy π. The Q value is similar, but in this case we talk about $Q^{\pi}(s, a)$. We try to find the value of the pair of states s and action a in this case. Ideally, we want to find the optimal $V^*(s)$ or $Q^*(s, a)$ according to the optimal policy π^* [43, 66, 42].

2.3.3 Discount Factor

One of the important parameters of any reinforcement learning algorithm is the discount factor, which adjusts how far we want to consider future rewards. As we can see in the previous subsection, we multiply each reward by the powered discount by the number of time steps. This changes how much each future reward value affects in some state (observation) and decreases the overall variance during learning, as it reduces the impact of outliers. When we use lower values of discount, we force algorithm to be more local – we compute value of each state only from close states. For value 0, we consider only the reward from a single transition, as the other rewards are nullified. Higher discount values (closer to 1) then prioritize long-term future rewards, and if we use discount 1, we consider rewards for each state even from infinity steps later [43, 66].

Moreover, we may use the discount for endless environments (with infinite episodes) to limit the distance we consider for future rewards [66]. For example, PAYNT [2, 3] uses the discount factor to adjust a threshold for the relevance of future rewards to evaluate FSCs.

2.3.4 Exploration versus Exploitation

A major challenge in reinforcement learning is the tension between two goals: exploration and exploitation. Exploration involves exploring the environment as much as possible, while exploitation involves taking the most successful actions in a given state, which can lead to overlooking potential better actions that the agent has not yet tried. If we focus on exploitation, the most direct approach is to quickly find a satisfactory solution. However, this may mean missing out on better options [66].

For example, an autonomous driving agent could go straight ahead, as it is closer to the destination, but if it turns right, it could find a highway where it could drive much faster and reach the destination earlier. Excessive exploration can cause an agent to make illogical decisions and to spend less time on potentially rewarding ones [66]. Exploitation is usually geared towards achieving quick gains, whereas exploration is more concerned with long-term improvements.

One of the common approaches for dealing with exploration and exploitation trade-off is the ϵ-greedy [43, 66] strategy, which adjusts the policy π to select the most beneficial action or a random action with a pre-determined probability. This approach enhances our capacity to learn during the agent's lifetime; however, its application in real-life scenarios

can result in taking really dangerous decisions. For example, think of an autonomous car that decides to take a really risky action in a certain situation.

In practice, we usually differentiate between two policies, the behavioral (collector) and the target. The behavioral one is focused on the exploration of the environment and usually selects action with some mechanism for generating stochastic decisions. In case of **off-policy** algorithms, these two policies are different [33, 32, 52] (e.g., with different weights), while in the case of **on-policy** algorithms, these policies are the same [61, 43].

2.4 Neural Networks

Neural networks are one of the most popular state-of-the-art machine learning frameworks that represent highly non-linear parametric functions. The universal approximation theorem states that, with enough neurons and some non-linear "squashing" function, they can approximate any function [30]. This is important, as in reinforcement learning, we usually try to optimize some value or policy, which may not be for large environments tractable, and we may use neural networks for function approximation.

Generally, a neural network is a differentiable function $f_{\boldsymbol{\theta}}(\mathbf{X}) = \mathbf{Y}$ that maps an input $\mathbf{X}$ (scalar, vector, matrix, sequence, etc.) to an output $\mathbf{Y}$. This function is characterized by a large $\boldsymbol{\theta}$, which represents the parameters of the neural network: Modern networks usually have millions or billions of parameters. Neural networks are used to process high-resolution images, videos, voice recordings, magnetic resonance images, long texts, etc., often resulting in very large datasets $\mathbf{D}$. The main purpose of these networks is to identify patterns in these large datasets. We can train them to do this by optimizing parameters $\boldsymbol{\theta}$ with following formula:

$$\arg\min_{\boldsymbol{\theta}} \sum_{(\mathbf{x},\mathbf{y})\in\mathbf{D}} l(f_{\boldsymbol{\theta}}(\mathbf{x}), \mathbf{y})$$

where l is some differentiable loss function [43].

Neural networks are not just a random collection of neurons that take input data and produce output; they are usually a carefully designed sequence of layers with well-described learning algorithms. In the following subsections, we will look at three common neural network architectures, explore various neural network learning algorithms, describe important activation functions, and outline the elementary principle of backpropagation algorithm.

2.4.1 Feedforward Neural Networks

The architecture of feedforward neural networks (FFNN) is one of the fundamental structures of neural networks. These networks transmit their input through multiple sequentially ordered layers that are completely connected. The propagation through the fully connected layers is determined by the equation:

$$\mathbf{y_n} = f(\mathbf{W} * \mathbf{y_{n-1}} + \mathbf{b})$$

where $\mathbf{y_n}$ is output of layer n, f is an activation function, $\mathbf{W}$ is weight matrix between two layers – each neuron of previous layer has connection with weight to every neuron in the following layer, and $\mathbf{b}$ is bias vector which adds weighted bias to each neuron[2] [11, 15, 30].

[2]Sometimes we consider extended weight matrix W with weight of bias and extended output of previous layer with constant value 1

Fully connected layers are usually used as part of more complex networks. Usually we find them as the last layers after some convolutional or recurrent layers, where they count some classification or regression from extracted features by previous layers. We train these networks with a backpropagation algorithm.

2.4.2 Convolutional Neural Networks

A major issue with feedforward neural networks is the large number of parameters (weights) required even for small inputs. For example, when processing a low resolution image[3] from the MNIST dataset with a single small layer of 20 neurons, the number of weights between two layers is $m_n * (m_{n-1} + 1)$, which in this case is $20 * (784 + 1) = 15700$ weights and biases[4]. Furthermore, feedforward neural networks are limited to the position of inputs specified in the training data set, whereas in real-world applications, the observed pattern can be shifted, rotated, or changed position [11, 43, 30].

Convolutional neural networks (CNNs), in the modern form introduced by Yann Le Cunn [45], address the mentioned issues by utilizing the operation of convolution. This operation is formally defined as:

$$(f * g)[n] = \sum_{k=-\infty}^{\infty} f[k] \cdot g[n-k]$$

where f is typically an input image, g is a convolution kernel, and n is the position in the resulting space. This definition is for the 1D input and the 1D kernel, but in CNNs, we usually work with the 2D input and 2D kernels, which are usually much smaller than the input image (usually 3×3 or 5×5).

Architecture of Typical CNN

In Figure 2.4, we can observe the structure of typical convolutional neural networks. The process begins with an input image that is processed through convolution, which utilizes multiple convolutional kernels to transform smaller parts of the image into feature maps. Generally, we combine convolution with some activation function, such as ReLU. Subsequently, subsampling is used to reduce the size of the feature maps. Pooling functions such as average pooling or max pooling are usually used, where feature maps are divided into small segments (e.g., 2×2) and each segment is converted into a single value (average or maximum of the segment). Older architectures only use different stride values, which alters the movement of the convolutional kernel across the input image to make the following feature maps smaller[5].

Generally, we use more layers with convolutional kernels and layers with pooling functions, which generate feature maps from low-level features to feature maps with complex shapes and information. These operations are typically followed by the flatten function, which converts the 3D set of feature maps to a 1D vector, which we can use in one or more subsequent fully connected layers, producing output for classification, regression, or whatever task was [30, 11, 43].

[3] MNIST images have resolution $28 \times 28 = 784$ pixels.

[4] Neural networks typically use floating point numbers (floats or doubles) as weights. If one considers the size of a single small layer when dealing with modern 4K images, it is easy to imagine how large it can be.

[5] Modern architectures still use various values of stride combined with padding (adding zeroes around input image) and pooling functions.

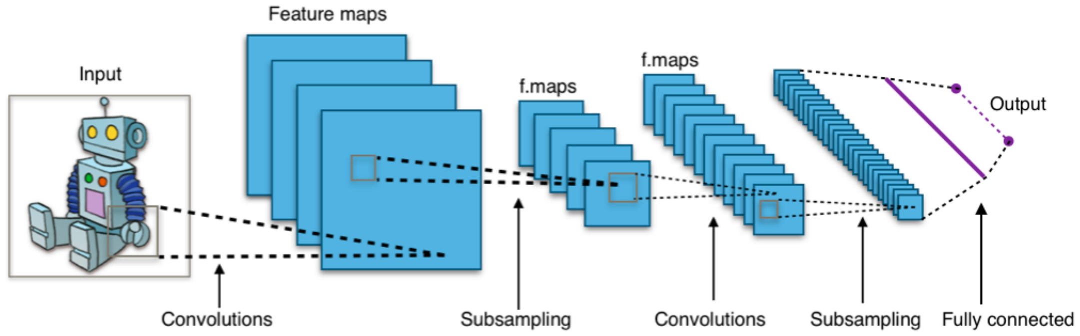


Figure 2.4: Architecture of a typical convolutional neural network. Image adopted from https://en.wikipedia.org/wiki/Convolutional_neural_network.

For training convolutional neural networks, we employ a backpropagation algorithm similar to that used for FFNNs, with slight modifications due to shared weights [11]. CNNs are widely used for tasks such as image or video processing. They can also be incorporated into more complex models, for instance, for feature extraction in language processing (followed by Hidden Markov Chains) or for processing the environment to create more useful approximations for reinforcement learning agents [66, 43, 30].

2.4.3 Recurrent Neural Networks

Recurrent neural networks (RNNs) can address a challenge that neither feedforward nor convolutional neural networks (CNNs) can easily manage: processing sequences of input and output of varying lengths. Think of an audio recording of a school lecture, a large language model (LLM), or a sequence of actions taken by an agent in an environment; these data can have different lengths, but we still want to process them with the same single model. We can solve this "puzzle" using recurrent neural networks (or layers).

The main idea is that the layer of the vanilla recurrent neural network works with hidden states, which we can use as outputs, or we can use them for computation of the following hidden state in combination with the following input from the input sequence. Formally:

$$\mathbf{h}_t = f(\mathbf{W}_{in} * \mathbf{x_t} + \mathbf{W}_{hid} * \mathbf{h}_{t-1})$$

where $\mathbf{h}$ is output of state t (time of sequence $\mathbf{T}$), $\mathbf{W}_{in}$ weight matrix for input values, $\mathbf{W}_{hid}$ weight matrix for previous hidden state and $\mathbf{x_t}$ is input of state t (input in time t). When we want to train these models, we use the so-called unfolding process [30, 43]. Visualization of this process is as we can see in Figure 2.5.

As we mentioned earlier, RNNs are usually used in areas where there is no fixed input length. We can find them in applications of natural speech processing,

LSTM

Although recurrent neural networks have certain advantages over feedforward neural networks and convolutional neural networks, a major issue arises when the input length is too long. The chain rule is used in the gradient computation process, which allows the derivation of more complex functions by multiplying the derivatives of the inner and outer functions. For example, if an agent has taken 10,000 actions and we want to use the sequence as training data, the gradient calculation would look like $\Delta w = \delta * h_1 * h_2 * \ldots * h_{10000}$,

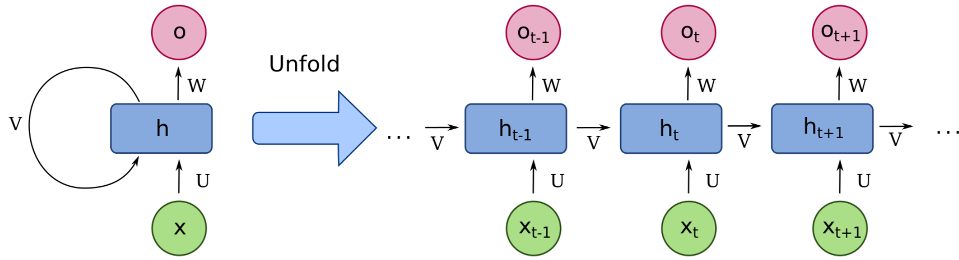


Figure 2.5: Example of unfolding process for vanilla recurrent neural network from https://en.wikipedia.org/wiki/Recurrent_neural_network.

where δ is the derivation of the network and h_i is a propagated value of recurrence. If the values of $\mathbf{h}$ are less than 1, the derivative will be close to 0 (no learning). If the values are greater than 1, the derivative will approach the upper limit of the floating point number (infinity) [43, 63].

Solutions to these problems are provided by long-short-term memory (LSTM) neural networks and their variations, such as gated recurrent unit (GRU) and bidirectional long-short-term memory (BLSTM). These networks are designed to avoid the effects of vanishing and exploding gradients using a combination of sigmoids, tanhs, and side channels [30]. As illustrated in Figure 2.6, the hidden state of the RNNs is not used as a direct input to subsequent layers. Instead, long-term memory (top) and short-term memory (bottom) are used.

Long-term memory helps us recall and control events from the distant past, whereas short-term memory has an immediate effect on the following layer (which is equivalent to the hidden state value). We can see that the value in long-term memory is regulated by multiplying it with the output of the sigmoid function (the output is in the range $(0, 1)$) and by combining it with short-term memory combined with the input at time t transformed by tanh (direction of change, range $(-1, 1)$) and sigmoid (size of change)[6] [30, 59].

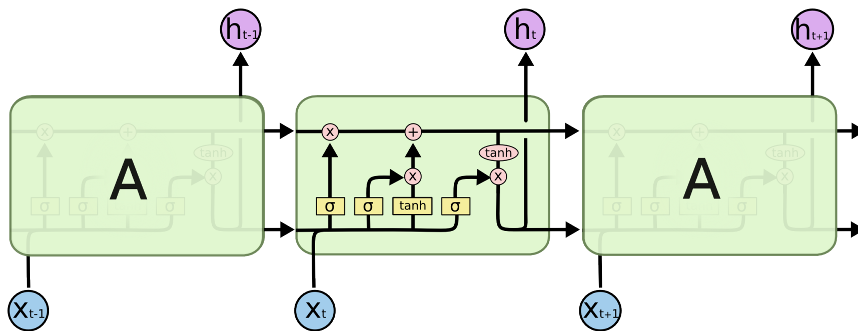


Figure 2.6: Example of LSTM adopted from https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[6]Excellent explanation of LSTM networks created by Josh Starmer: https://statquest.org/long-short-term-memory-lstm-clearly-explained/.

2.4.4 Activation Functions

Aside to architectures of each layer of neural networks, there are types of activation function, which affect the overall behavior of the network and allow us to learn any function f given the universal approximation theorem [30]. Neural networks can utilize a variety of activation functions, which are usually selected depending on the task or layer of the network and are always differentiable [15, 30]. The most common activation functions are:

- **ReLU**: ReLU is often used in image processing (e.g. CNNs) and has a simple derivative that is less prone to the vanishing gradient problem. Its formula is $ReLU(x) = max\{0, x\}$, and its modified version, the leaky ReLU, has a non-zero output for negative values.

- **Sigmoid** and **tanh**: Sigmoid (or logistic function, σ) and tanh (hyperbolic tangens) are used in LSTM or GRU recurrent neural networks, or in binary classification networks. These functions are usually connected to the problem of vanishing gradients. Their formulas are $sigmoid(x) = \frac{1}{1+e^{-x}}$ and $tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$, and their outputs are in the intervals $(0, 1)$ and $(-1, 1)$, respectively, causing a general decreasing gradient through layers.

- **SoftMax**: SoftMax is the activation function that is usually used in the last layer of neural networks for the classification of k classes, where it determines the probability of each class. Its formula is $SoftMax(\mathbf{x})_i = \frac{e^{x_i}}{\sum_j^k e^{x_j}}$.

- **Linear**: Simple function that can adjust our result in some layers. However, if we only use linear functions in neural networks, we do not meet the requirements of the universal approximation theorem, meaning our network cannot approximate general nonlinear functions. The formula for a linear function is $linear(x) = x$.

2.4.5 Backpropagation Algorithm

Numerous learning algorithms exist for training purposes, including evolutionary algorithms, yet the backpropagation algorithm is the favored choice utilized because it is simple and we can adjust it to obtain convergence, and it is used by most neural network frameworks, including TensorFlow Agents [67]. The main idea behind the backpropagation algorithm is to use the chain rule for derivation and use smart bracketing to compute the derivation for complex functions. This algorithm calculates the gradient for the neural network which is subsequently employed by a learning algorithm such as stochastic gradient descent. It operates under the assumption that the network is entirely differentiable, allowing us to compute the gradient for each weight. The process of gradient descent typically involves a loop, where we initially calculate the update (gradient) using the backpropagation algorithm. The size of this update is typically modified by a learning rate parameter or a more complex optimizer like Adam, after which the update is subtracted from the current weights [11, 30].

The standard algorithm is used for both feedforward and convolutional neural networks, which operate with a fixed input size. In contrast, when calculating the gradient in recurrent neural networks with variable input length, we utilize an extension of the original algorithm known as backpropagation through time (BPTT). This method performs the unroll process for the recurrent neural network before executing the algorithm, and then computes the gradient in the standard way [43, 30].

2.5 Reinforcement Learning Algorithms

In this section, we outline the taxonomy and main approaches for reinforcement learning algorithms, first based on the estimation of the value function and second on the policy gradient methods. Then we describe an important concept of actor-critic algorithms, which combines both approaches. Then we make the step towards sequential decision making for POMDPs with reinforcement learning algorithms and, finally, introduce the state-of-the-art algorithm for reinforcement learning, which are used in this thesis.

2.5.1 Taxonomy of Reinforcement Learning Algorithms

Algorithms are initially distinguished based on whether they calculate the policy from a value function, typically $Q(s, a)$, or directly as $\pi(s)$. Or in the case of POMDPs, we note the functions as $Q(o, a, h)$ and $\pi(o, h)$, where o represents observation and h (recurrent) feedback. The first principle mentioned, value-based or critic algorithms, generally have a slower convergence rate and are more efficient in smaller environments where they tend to have better sampling efficiency. They are primarily employed for discrete action spaces, as they generate the policy by enumerating the value function for all actions and choosing the most optimal one. In contrast, policy-based algorithms usually converge more quickly, as they do not need to estimate the value function for the entire environment and simply learn policies from successful trajectories [66, 43]. The final option is the actor-critic principle, a combination of both methods, which typically takes the benefits of both. However, the implementation of these algorithms can vary, with some focusing more on policy (gradient) optimization, such as PPO [61] (proximity policy optimization), and others primarily on Q-value estimation, such as SAC [20] (soft actor-critic). We provide a brief summary in Table 2.1.

Type	Policy-based	Value-based	Actor-critic
Algorithm Goal	$\pi(s)$	$Q(s, a)$	π and Q
Implementations	REINFORCE TRPO, DDPG	Q-learning Sarsa, DQN	SAC, TD3 PPO, A2C
Stochastic	Yes	No	Optional
Convergence	Fast	Slow	Fast
Primary Action Space	Continuous	Discrete	Both

Table 2.1: Policy-based vs value-based algorithms. We consider environments represented by POMDP, thus we also consider history h. Based on various sources, primary [43, 66].

The second division we mention here are the so-called **on-policy** and **off-policy** algorithms. The difference here is the usage of behavioral (collect) policy, where we may use single policy for exploration and for evaluation (on-policy), or implement two different policies (off-policy). While on-policy is more stable, converges faster, is simpler in terms of storing only single policy and it does not require lot data, it also sometimes suffers from getting stuck in local optima and explores less the overall environment. On the other hand, off-policy learning is usually less prone to get stuck in local optima, but it usually leads to less stable learning. In Table 2.2 we again present a brief summary of both approaches.

The final distinction we make is between model-based and model-free approaches. The key distinction lies in the fact that, in model-free algorithms, we formulate a policy or value estimate and make immediate decisions, whereas in model-based algorithms, we build

Type	On-policy	Off-policy
Stability	Stable	Unstable
Past Data	Unavailable	Available
Stuck in local optima	Often	Rare
Behavioral Policy	Target	Different
Implementations	PPO, Sarsa, TRPO	TD3, Q-learning, DQN, SAC

Table 2.2: Comparison of on-policy and off-policy learning. Based on various sources, primary [20, 61, 43, 66, 26].

our own representation (model) of the environment and then make choices using a search algorithm such as Monte Carlo tree search. When the first approach makes decisions fast and is more dependent on the first learning part, the second approach postpones decision making until evaluation. Model-free solutions are usable for a larger variety of tasks, while model-based algorithms are limited by the complexity of the models. However, model-based approaches usually have more stable and safe policies, but it is dependant on the quality of trained model. In this thesis, we do not use any model-based algorithms, because, as we describe later in Chapter 6, the approaches to training policies over POMDPs have many issues and adding one layer, the model-based approach, to them would lead to more problems and instability.

2.5.2 Value Based Learning

The first approach for value-based learning is dynamic programming, which is in general a framework for solving hard problems by splitting solved tasks into smaller sub-problems, where we take advantage of the use of memory. It is used in many domains to optimize existing algorithms, where we usually memorize some results so we do not compute them multiple times. When using dynamic programming to solve reinforcement learning tasks, we estimate the V or Q functions, which are improved with each iteration of the algorithm. This approach is also associated with the Bellman equation:

$$V'(s) = \max_a \left(\sum_{s',r} P(s', r \mid s, a) \left(r + \gamma V'(s') \right) \right) \tag{2.2}$$

where V' is an estimation of the value function, r is the reward for transition (s, a, s'), s' is a reachable state from state s, and P is the transition probability from state s with action a to state s' with reward r. Similarly, we can also consider the Bellman equation for Q-values:

$$Q'(s, a) = \sum_{s',r} P(s', r \mid s, a) \left(r + \gamma \max_{a'} Q'(s', a') \right)$$

Our goal is then make the most optimal estimation V^* or Q^* of value or Q-value, respectively [66]. To achieve it, we may use various multiple algorithms, such as policy or value iteration, where we usually initialize the original arbitrary estimation of the value $V'(s)$ and $\pi(s)$ for all states $s \in S$ and then loop through evaluating the value estimation in the model given some policy, until convergence.

One of the benefits of this approach is that we have an estimation of the value function throughout the entire process, and we can decide when to stop the algorithm. The downside

is that we need to know the entire environment, which can be intractable for large MDPs. The dynamic policy then defines an optimal policy after some learning iterations as the selection of the greatest enhancement of a value function in a given state s.

Dynamic programming is utilized for the precise computation of the value function when the model is fully known. However, in situations where the computation is not feasible due to the large scale of the model or when the entire model is not known, we resort to the Monte Carlo algorithm, which instead of computing estimation for whole model each step uses sampling. These algorithms are based on sampling whole episodes from the environment and then using them to estimate the values or Q-values. The main advantage is that we do not have to compute the estimations for the whole model and with some weighted sampling or similar approaches, we can usually find a relatively fast functional policy. The main drawback is that the convergence is not guaranteed and that if we find a cycle, we may loop in a single episode for the infinite time, as the algorithm works with full episodes for value estimation. In general, these algorithms suffer from high variance of results as the optimization methods suffers from the quality of sampled data.

When both previous approaches have significant drawbacks, we usually use some combination, where we sample a few steps, like in Monte Carlo algorithms, and then use the value estimation for the last state, like in dynamic programming. In general, we consider these approaches under the general name of temporal difference learning [66] (TD learning). There exist multiple approaches [43], such as Q-learning, where the update function is computed as:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a))$$

Sarsa[7]:

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

or eligibility traces:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\delta N(s,a)$$

where α is learning rate, δ is Sarsa temporal difference update $r + \gamma Q(s',a') - Q(s,a)$, and $N(s,a)$ is count of visits of tuple (s,a) decayed with $N(s,a) \leftarrow \gamma\lambda N(s,a)$. All three approaches are based on the use of sampled data to update value functions and are usually improved with the use of an algorithm called *n-bootstrapping*, where we replace the reward r with a sequence of rewards given some actions and $Q(s',a')$ changes in the estimation of values at the end of the trace with length n. Overall, the temporal difference learning algorithms benefit from both approaches, as we do not have to compute data for the whole model and we also do not have to finish whole episodes. These algorithms gave birth to some more recent algorithms such as DQN [52] or TD3 [26].

2.5.3 Policy-Based Algorithms

Quite different approach for obtaining policies given reinforcement learning framework, are policy-based (or policy gradient) methods, which tries to estimate optimal policy π directly instead of computing the value estimation. We usually use the notion with $\boldsymbol{\theta}$ representing the parameters of the policy, when the action selection is described as $\pi(a \mid s, \boldsymbol{\theta})$. The update of the policy parameters is then obtained as $\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \alpha\widehat{\nabla J(\boldsymbol{\theta}_t)}$, where J represents the loss function, and α represents the learning rate.

[7] The name is derived from input tuple (s, a, r, s', a').

One of the simple approaches for policy gradient methods is the REINFORCE [66] algorithm, which is based on Monte Carlo sampling trajectories from the environment and computing using parameter update noted as:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \sum_a \widehat{q}(S_t, a, \boldsymbol{w}) \nabla \pi(a \mid S_t, \boldsymbol{\theta}_t)$$

where $\widehat{q}$ represents some value estimation given current policy π and S_t represents states in trajectories. These algorithms are usually more suitable for tasks with continuous action space and can provide effective learning for complex policies. However, these algorithms suffer from stability problems, are sensitive to hyperparameter selection, and are usually sensitive to the low amount and variance of the data [43, 66, 61].

2.5.4 Actor, Critic and Actor-Critic Agents

We usually use neural networks, more discussed in Section 2.4, in the form of an actor, or critic, where the actor corresponds to policy-based methods (which perform $\pi(s_t, h_t)$) and the critic corresponds to value-based methods (computes $Q(s_t, a_t, h_t)$). The actor is usually implemented for a continuous action space, as we can implement a neural network that performs a regression from input to some actions such as speed or braking in the case of cars [20, 76]. In contrast, the critic is primarily used for discrete action space, as we can simply select the best action by enumerating all possible actions. For example, we can use critic to select the most optimal gear level. However, this differentiation is not fatal, and we can, with some engineering, use both approaches for both action spaces [43].

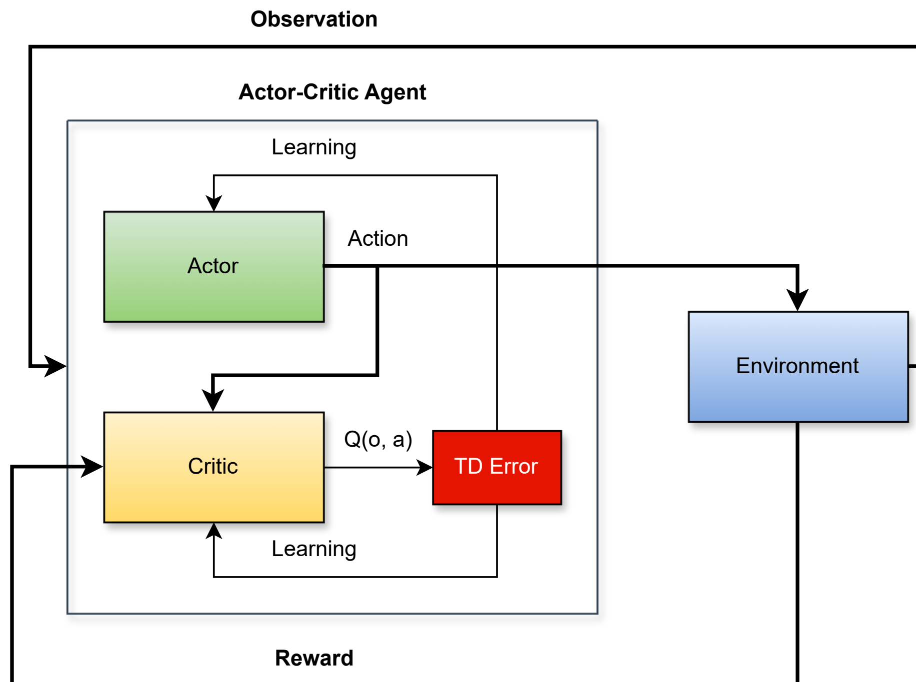


Figure 2.7: Visualization of the actor-critic agent communicating with the environment.

In actor-critic algorithms, we typically employ a combination of both methods, utilizing the actor for action generation and the critic for learning the model and actor. In general,

in this strategy, the reward from the environment is distributed only to the critic component. The visualization of the actor-critic agent can be seen in Figure 2.7. This strategy solves numerous problems of both actor and critic approaches and is typically more stable, accelerates learning, enhances exploration, and overall provides numerous implementation alternatives, such as SAC [20], PPO [61], or TD3 [26].

2.5.5 Reinforcement Learning for POMDPs

One of the emerging areas in the implementation of POMDP policies involves the application of deep reinforcement learning techniques using algorithms such as DQN [32], PPO [61], SAC [20], among others. Although many of these algorithms have been well-crafted and validated for MDPs, their standard versions often fail to address the state estimation challenges posed by POMDPs. Consequently, although these algorithms find utility in various domains like Atari Games, a common benchmark for reinforcement learning algorithms, they are not suitable for environments that require consideration of past observations, such as previous frames. Therefore, subsequent approaches introduce enhancements that facilitate the calculation of policy $\pi(o_t)$ or values $Q(o_t, a_t)$ with respect to a given history h_t. In a formal sense, this is expressed as $\pi(o_t, h_t)$ or $Q(o_t, a_t, h_t)$ [43]. Our goal is to estimate the state, as we can see in Figure 2.8.

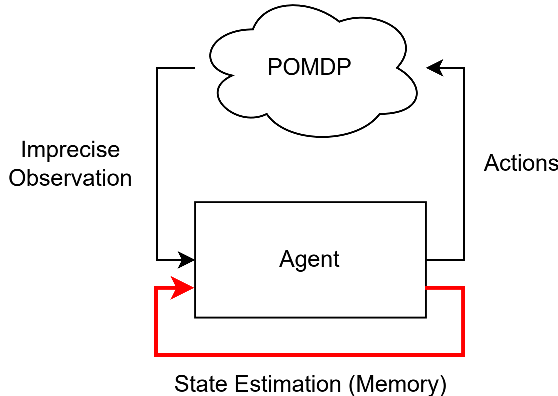


Figure 2.8: General scheme of reinforcement learning for POMDPs. Red lines highlights, what is extra to the MDP-based reinforcement learning.

In deep reinforcement learning, we can consider multiple different approaches to achieve computing policies according to some history. The naive approach may be creating a larger input layer, where we can place current and previous observations to queue. However, this approach suffers from fixed memory size and is not suitable as a general solution for the POMDP framework.

Today, we can find two orthogonal approaches: in the first one, we adjust the architecture of the network to use some layers, which supports consideration of previous observations (computing belief). The most common solutions are based on recurrent neural networks such as LSTM or GRU [33, 5, 17], or solutions based on attention [23] mechanism as transformers. These approaches are based on previously mentioned algorithms, where each actor (or critic; the neural network) uses both feedforward or convolutional layers and as well some recurrent or transformer layers. Current reinforcement learning frameworks such as TensorFlow Agents [67] then support the ability of each algorithm implemented

to transfer the current policy state of each iteration, which usually represents feedback in recurrent neural networks[8].

The second approach for the reinforcement learning algorithm is based on a hybrid solution, where we keep our original reinforcement learning algorithm and combine it with some external tool, which computes beliefs outside of the algorithm. For example, the approach [17] lets Storm [36, 56] to compute belief support[9], which is sufficient statistic for POMDP reinforcement learning algorithms [43, 5]. We may also use belief MDPs from Subsection 2.2.1 or simply use another algorithm outside of the neural network to compute beliefs [10].

In contrast to recurrent neural networks (or transformers), which are usually not stable and require a lot of data to function properly, the belief approach usually brings more stability and faster convergence to an optimal solution [5]. However, the main issue of the latter approach is the need to usually know the whole original model, which is more computationally demanding and may be intractable for common models. We can see a simplified illustration of both approaches in Figure 2.9 In this thesis, we focus on the first approach, as it is more suitable for our interpretation purposes.

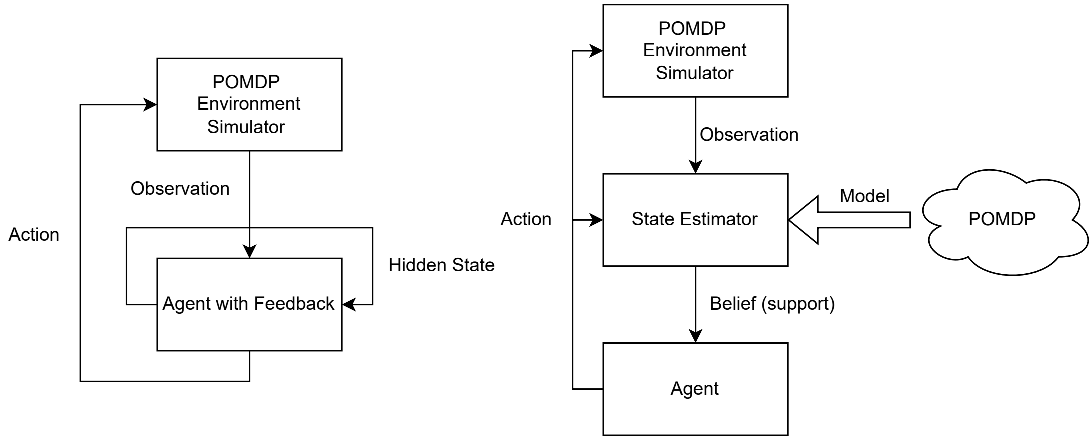


Figure 2.9: Comparison of approach based on agents with feedback (e.g. RNNs; on the left) and belief based approach (right).

2.5.6 DQN and DDQN

One of the pioneering deep reinforcement learning algorithms used for various tasks is Deep Q-Networks [52] (DQN) and its enhancement, Double Deep Q-Networks [32] (DDQN). Both algorithms are model-free and are based on existing Q-learning methods that incorporate deep neural networks and innovative replay buffers. Training of neural networks in these algorithms is guided by a specific loss function defined as:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\ U(D)} \left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i) \right)^2 \right]$$

Here, θ denotes the parameters of the deep neural network, s and s' represent the current and subsequent states, r is the reward, and Q is the Q-function. The term $U(D)$ refers to

[8]In the case of usage of layers without feedback, the handed policy state is empty.
[9]Vector of states, in which the agent may be present at the time t.

the dataset used for sampling, which is derived from a replay buffer containing experiences in the form of (s_t, a_t, r_t, s_{t+1}). It is useful as it stores not only the most recent experiences, but with some limitations stores even the old ones, thus when the algorithm learns to explore new states, it still has the ability to remember the old trajectories and compute the optimal strategy Q_*.

The double DQN [32] (DDQN) algorithm is very similar to DQN, but adds one more neural network[10]. This algorithm focuses on the fact that the original DQN algorithm tends to overestimate action values given some condition, and is based on previously existing approach for Double Q-Learning, which is a similar method, but only for tabular settings. The deep version consists of two neural networks with the same architecture but used for different purposes, one for action selection and one for action evaluation. This leads to an improved formula:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(D)}\left[\left(r + \gamma Q(s', \arg\max_{a'} Q(s', a'; \theta_i); \theta_i^-) - Q(s, a; \theta_i)\right)^2\right]$$

where the symbols have the same meaning as in the case of DQN and θ_i and θ_i^- are parameters of two different networks.

DRQN – Solution for POMDPs

Shortly after the introduction of the initial DQN algorithm, a revised iteration was created to address memory-related issues in partially observable Markov decision processes (POMDP). This improved algorithm, called Deep Recurrent Q Networks [33] (DRQN), bears a strong resemblance to the original DQN, but integrates recurrent units (LSTM) to aid in memory sequence learning, as shown in Figure 2.10. The formulation of the loss function remains unchanged.

2.5.7 Proximal Policy Optimization

One of the most important algorithms for modern reinforcement learning in many state-of-the-art tasks is Proximal Policy Optimization [61] (PPO) developed by OpenAI. This algorithm is also model-free, but opposite to DQN, is on-policy. This means that we do not use two different policies for exploitation and exploration, but we use only a single behavioral policy.

Proximity policy optimization relies on policy gradient methods, which determine how the network parameters should be adjusted by computing the gradient of the loss function formula [75, 61]:

$$L(\theta) = \hat{\mathbb{E}}_t[\log \pi_\theta(a_t s_t)\hat{A}_t]$$

where t represents time t, π_θ denotes the policy derived from parameters θ, a stands for action, s refers to state (observation), and $\hat{A}_t$ is the estimated advantage function at timestep t. The advantage function is typically computed as $A(s,a) = Q(s,a) - V(s,a)$ and is sometimes viewed as an alternative version of the Q value with reduced variance [75]. However, this function can vary depending on the specific tasks, and for instance, the process of learning for recurrent neural networks involves an enhanced version of this calculation [61]. The authors of [61] criticize this gradient optimization approach as being simplistic and

[10] It may look like creating an actor-critic algorithm, but that is not the case, because both networks are critics!

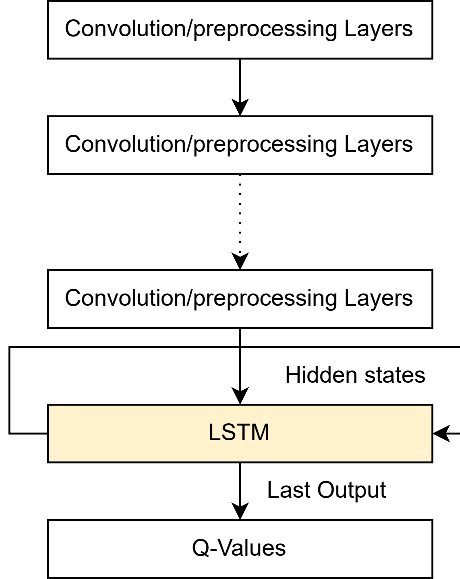


Figure 2.10: General structure of Deep Recurrent Q-Networks. The yellow part indicates the new layer over the original DQN solution.

prone to generating overly large updates that can have a destructive effect on the existing policy.

The authors of PPO developed various versions of the surrogate objectives by adopting a clipped objective inspired by the trusted region policy optimization [60] (TRPO) L_t^{CLIP} or the adaptive Kullback-Leibler penalty coefficient L_t^{KLPEN}. They then selected one of these objective functions and combined it with squared error loss L_t^{SQ}, entropy bonus $S[\pi_\theta](s_t)$ to create a new loss function:

$$L_t(\theta) = \hat{\mathbb{E}}[L^{CLIP \text{ or } KLPEN} - c_1 L_t^{SQ} + c_2 S[\pi_\theta](s_t)]$$

where c_1 and c_2 represent coefficients. This leads to the creation of a family of more stable learning algorithms, which are used in many modern solutions based on reinforcement learning.

Chapter 3

Challenges

In a previous chapter, we presented the fundamental concepts for addressing sequential decision making under uncertainty within the POMDP framework. Although these concepts primarily represent theoretical foundations, the approaches discussed generally exhibit numerous limitations and unresolved challenges. This section will focus on detailing these challenges, examining their impact on model-based and reinforcement learning algorithms, and exploring how various methods deal with these problems.

3.1 Explainability

One of the main goals of this thesis is to create communication between PAYNT and reinforcement learning represented by some agents consisting of deep neural networks. For this purpose, we have to introduce the topic of explainable AI [22, 40], which is a long and complex process for the implementation and deployment process of using machine learning approaches in real-world tasks. For example, in Figure 3.1 we can see some visualization of the deployment process with respect to explainability.

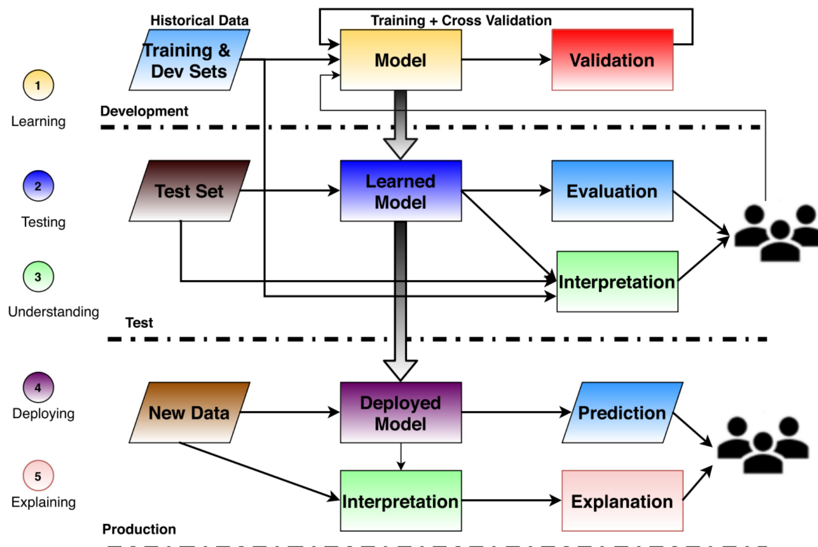


Figure 3.1: Process of implementation and deployment of explainable machine learning solution for real-world tasks. Taken from [22].

In this section, we introduce some underlying concepts of this area, describe some current approaches on how to reach its goals. In addition, we describe our motivation for the explainability of reinforcement learning. The last subsection outlines a possible approach, which is not discussed in this thesis but could provide a deeper understanding of trained agents, and thus provide the best explainability of trained policies.

3.1.1 Goal

In discussions about interpretation, explainable AI [22] or formal verification [46], we generally refer to a learned solution that is based on a machine learning framework, often neural networks, with the aim of gaining a deeper understanding of our creation. There could be numerous inquiries, but the common theme is the transition from a black-box solution to a white-box solution, as illustrated in Figure 3.2. For instance, we might wish to formally verify the learned method according to a certain specification, or we might seek a solution that requires less performance, or even more, and we might aim to develop an interpretation that humans can explain and understand. White-box solutions include decision trees, linear models, symbolic neural networks, etc. In this thesis, we focus on extracting hints from the trained agent to use them for better PAYNT learning.

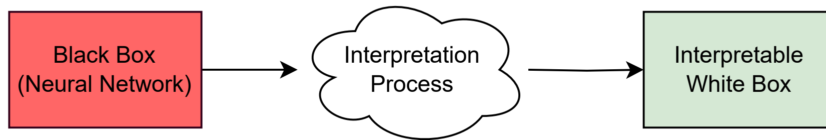


Figure 3.2: Simplified Interpretation Process

3.1.2 Current Approaches

Given a framework of reinforcement learning, there exist multiple approaches which make efforts to derive interpretable policies from trained RL policy. For example, we can consider the programmatic approach [54, 70, 62], or prototype wrapping [40].

The first mentioned, the programmatic approach, depends on the creation of programs which represent policies. It means that instead of creating a large neural network black box, we use some classical programming constructions like *if*, *while*, *for* etc. For example, the study [62] mentions that output policy of programmatic reinforcement learning is a sequence of loops:

```
Do:
    P
Until(e)
```

which means that we run local program `P` until reaching edge `e`. The local program is then represented as a set of instructions in the form of:

```
From s ->
    Target s1, Preference: v1
    Else Target s2, Preference : v2
    ...
    Else Target sk, Preference : vk
```

where `si` represents segment `i` and `vk` represent vertices.

The construction of described programs is based on the recursive traversing tree of the so-called winning regions, which represents the division of the state space to regions with the same preferred actions and which guarantees the achievement of the goal state [62]. However, there exist multiple approaches and definitions of programs for programmatic learning, for example [70] describes an approach with different program structure based on interpreting the behavior of neural network policy.

An alternative method outlined in [18] employs a cyclical process to train neural agents and derive FSCs[1], which are then validated and utilized to improve neural network training or to verify the safety properties of neural networks by verifying FSC. The extraction of policy from neural network is based on sequential increase in the size of the FSC combined with measurement of the entropy. If adding memory nodes to FSC does not lead to a significant improvement (change in entropy), the process ends, and the training process continues. The extraction itself is based on clustering the continuous hidden states of recurrent neural networks to the N memory nodes of FSC.

The final method discussed [40] is prototype-based deep reinforcement learning. Here, the authors introduce prototype wrapping networks with a structure akin to neural networks, where each "neuron" is represented as a human-readable prototype. The authors incorporate transformation layers to generate input for these prototypes. This method yields robust and scalable policies that enable comprehension of the fundamental components and the use of verification techniques to ensure the robustness of these policies.

As the articles [40, 22] mention, there exist two different insights into interpretation and explainability, post hoc and pre hoc. The first one is based on the fact that we obtained some unexplored solution, and now we want to analyze it and obtain some surrogate solution or at least more information about its behavior. The second is based on the assumption that we can interfere during learning and create a surrogate solution parallel to the original one. That is a similar idea which we implement in this thesis.

3.1.3 Difference in Belief Implementation of RL and FSC

Under optimal conditions, we would employ PAYNT to ask straightforward queries to trained agents, such as "*which action should be chosen now?*", but both approaches utilize distinct implementations for handling state uncertainty. This disparity is highlighted in Figure 3.3, where we can see that while the inference process shares similarities, the implementation of the algorithm feedback differs. When dealing with a neural network, we utilize feedback in the form of previous hidden states from recurrent neural networks (LSTM), typically represented by a vector of arbitrary floating numbers. These numbers are the result of the gradient descent training of the employed network and generally appear random. In contrast, for FSC, the policy feedback is represented by an update to the memory node, which is essentially a single integer that points to the subsequent state that will determine the next action to be selected.

Given the distinct nature of the two feedback implementations and the unique ways in which they are worked with, there are currently no known methods for a viable conversion between the two. However, an advantage of PAYNT is its ability to interpret hints in various forms. One such form is a hint regarding relevant actions in response to a certain observation. In Figure 3.3, we could ask which actions an RL agent can choose in response

[1] This method resembles ours but relies on a distinct formal foundation and directly derives FSC policies from neural networks rather than integrating them with PAYNT.

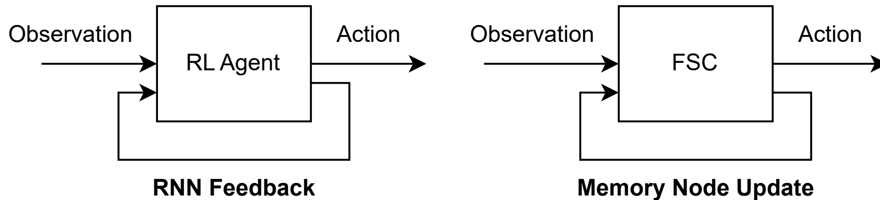


Figure 3.3: Comparison of belief implementation in reinforcement learning based on recurrent neural networks and in FSC.

to a particular observation and any feedback. If this hint is acquired, PAYNT does not need to explore FSCs that involve actions not included in the set derived from RL[2] in response to a certain observation during the synthesis of FSCs.

The second behavior that both approaches include is the variability of actions given some observation. Given various feedbacks, we can obtain different outputs, but both approaches reason for a given observation only about some subset of playable actions. Thus, the information we would like to extract and propagate is information about variability of actions given some observation, because PAYNT synthesize policies given some memory for each observation. However, an important note is that raising the memory size for an observation is usually a very expensive operation, as we have to explore during synthesis exponentially larger design space.

For the extraction of information that we have mentioned, there is no universally available solution that we could use. Thus, in Section 5.4, we describe the interpretation issues and present a novel approach in which we have obtained the requested hints from RNN.

3.1.4 Explaining LSTMs

An alternate method discussed in [6] could be the application of an extension to the algorithm known as Layer-Wise Relevance Propagation (LRP), which aims to explain the decision-making process of LSTM. This approach is based on the modification of LSTMs and the enforcing of uniform credit propagation, which is a feature of LSTMs, when the network propagates long-term information without scaling it. The output of this algorithm is a set of scores for each input variable when each score corresponds to the relevance of the given input to the outcome decision. In the context of our thesis, it would mean detection of some previous observation which led to decision that our network made. This could be used for deeper hints for PAYNT, because it could solve the issue with conversion between FSC and LSTM feedback mentioned in Subsection 3.1.3.

3.2 Stability and Parameter Selection

One of the main issues of the reinforcement learning algorithm is stability. Although model-based approaches such as FSC can just simply use elitism to recall the best solution yet, we observed many problems with stability during our experiments, more described in Chapter 6. For example, we ran three different algorithms on the task *rocks* with parameters $N = 16$ and our results were very unstable, as can be seen in Figure 3.4. We can see that between a few hundred iterations, we obtained a significantly different agent, which

[2] Or other oracle.

performed the task perfectly (achieved the goal in 100% episodes) or did not achieve the goal at all. It is caused by many factors, for example, by the nature of neural networks, which can be sensitive to high gradient values [61], the tendency to overestimate Q values by DQN [52, 33], by exploring the environment and observing new states, or by the nature of RNNs (LSTMs), which are generally not stable and need a lot of careful tuning [30].

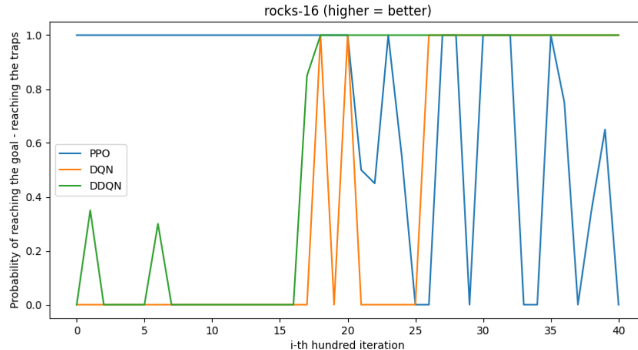


Figure 3.4: Example of learning on task rocks with parameter $N = 16$

The issue of stability is addressed by many approaches. One of the usual approaches is designing a whole new algorithm, which introduces new techniques to address instability, for example, the algorithm PPO [61] reduces instability by cutting gradients with some threshold, the algorithm SAC [20] introduces entropy, and TD3 [26] uses two different networks to stabilize the algorithm. Some approaches are more stable by their nature, as, for example, MuZero [4], which performs tree search in the model and can select actions with the most stable result. However, it also introduces the trade-off between scalability and robustness. Another approach can be the usage of on-policy or off-policy algorithms, where on-policy algorithms are usually more stable. An alternative approach may be the usage of shielding [17], which limits the set of playable actions in each observation only to the safe one. Or we can also use some kind of imitation learning, where we try to learn a strategy similar to another safe one and then improve the agent. However, many approaches lead to lower scalability or a higher tendency to get stuck in local optima, as we explore less aggressive.

Another important joint part of the issue with stability is the selection of parameters. In our case, it includes selection of learning rate, neural network hyper-parameters, reward model, regularization, clipping, size of replay buffer, trajectory length, maximum episode length, activation functions, loss functions, and many other parameters. Unlike other machine learning, evolutionary algorithms [13], or formal model-based algorithms [2, 12], reinforcement learning usually has a significantly higher number of parameters, where there is usually no general agreement on the recommended selection of values.

One of the parameters in which the literature and current approaches differ is in the selection of discount parameters. In practice, we use discount to enable learning for models with infinite horizon, where it allows us to compare two different solutions, as we cannot reach the reward with value ∞. In addition, we use a discount factor to enforce the location of the rewards. For example, as we can see in a trajectory in Figure 3.5, we can have a trajectory with thousands of states obtained by performing 999 actions and obtaining 999 rewards. If we do not use the discount (or set it at 1), every reward obtained for each pair (s_t, a_t) will have the same impact on the value estimation in the state s_1. If we use a

discount factor with a lower value, for example, 0.5, our value estimation will focus more on closer states, as the reward from the latter actions has a lesser and less impact.

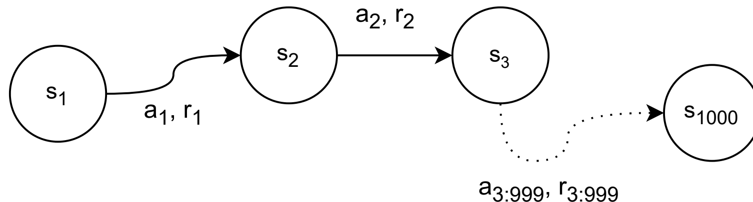


Figure 3.5: Example of trajectory.

As we can find in the literature [66, 43, 42], the values of the recommended discount factor vary for each task, but we usually find values between 0.2 and 0.8. However, in practice, current approaches [17, 5] or even the default implementation of agents in TensorFlow Agents [67] use values close to 1 (usually 0.99). In our experiments, we found that discount factor selection can significantly change the learning outcome in both ways; sometimes higher values can help propagate goal values faster to the trained agent. Sometimes, lower values help to recognize better safe and dangerous states, as the states (observations) closer to traps are impacted more by the negative reward, and further states do not receive that high negative response. In this thesis, we usually prefer lower values, as our experiments have shown a more positive impact of lower values than higher values.

3.3 Scalability

Although reinforcement learning faces stability issues, model-based methods, such as belief MDPs or FSCs, struggle with scalability in real-world applications. Information from their benchmarks indicates that they typically handle grid tasks up to a size of 20×20, where a noticeable drop in performance often occurs [1, 17, 12]. Additionally, the introduction of a basic reward for treasure collection on the map causes the state space of (PO)MDPs to double, necessitating models for states both with and without the reward. In contrast, reinforcement learning methods are generally unaffected by changes in state space, as they operate with a predefined neural network size that is adaptable to various complex tasks. If the network is too small, it can be scaled up by adding layers or enlarging existing ones, enhancing computational power, and retraining. For complete formal model-based strategies, calculating the optimal policy becomes exponentially more challenging, and significant scaling is often unachievable even with increased parallelism or more efficient computation. Thus, a robust heuristic is required. In this context, our aim is to employ a reinforcement learning-based heuristic to explore the design space of PAYNT.

3.4 Robustness

Numerous strategies are available to achieve robustness with solutions that utilize neural networks. For example, one strategy involves extracting FSCs from recurrent neural networks that represent policy and building FSCs directly [18] from them. Another strategy relies on the formal verification of neural networks [46, 35, 78, 25], enabling the verification of neural networks against specific criteria such as local robustness or sensitivity to the bias

field. Or, there exists an approach, which computes the so-called shields [17], which limits the policy to take only safe actions during learning (and even after).

However, the greater the robustness of a method, the lower its scalability. In our experiments, we show that, for example, shielding is not suitable for large tasks at all and is less scalable than the formal-based methods represented by PAYNT [2]. In this thesis, we would like to represent a robust and verified FSC-represented controller created by PAYNT based on hints from trained agents from the reinforcement learning approach.

3.5 Sparse Reward

Another challenge we have faced within this thesis is the so-called sparse reward models, where we do not often obtain positive rewards and we usually get only some small penalty for making steps when moving in the environment. For example, in the case of task *refuel* (more described in Chapter 6), we have to visit three filling stations, in each select action to refuel fuel, and without observing exact position, traverse the whole grid to the goal state. If we consume all the fuel, we lose. In case of reinforcement learning, we have to randomly sample trajectory, which satisfy all these sub-tasks and reach the goal, without obtaining reward in the middle of the route. If we use a naive approach, we change the reward function and add some small reward for visiting stations and refueling fuel. However, this approach, usually called reward shaping [57], can also lead to getting stuck in local optima, where we cycle to refuel and do not leave the station at all.

Reward shaping is generally a challenging field, crucial for developing intricate reward functions that avoid entrapment in local optima and do not restrict the development of superior policies compared to existing ones. In the process of reward shaping, the introduction of certain solution details might hinder the discovery of the true optimal solution, thereby diminishing the primary benefit of reinforcement learning over supervised learning. However, various approaches are available to develop reward shaping. For example, the method described in [7] suggests a potential-based reward shaping, which involves calculating the potential differences between states, analogous to electrical voltage. Alternatively, the strategy in [50] proposes self-supervised online reward shaping, designed to automate the process without compromising the discovery of the optimal solution based on the input model.

During our experiments, we found that the similar issue with sparse reward models was with reinforcement learning and also with model-based approaches, but the model-based approaches usually find at least partial solution, where they can finish the whole task at least with low probability, which can be improved by reinforcement learning. This fact led us to use PAYNT FSCs to improve the exploration strategy of our RL approach.

Chapter 4

Proposed Approach

In this chapter, we will outline the design of our toolkit, including the reinforcement learning part, the interpretation part, and the PAYNT interpretation usage part. In the following sections, we start with a description of the overall schema and then describe individual components of the solution. The technical details of the proposed approach are described in Chapter 5.

4.1 Overall Schema

This diploma thesis examines the combination of PAYNT and reinforcement learning algorithms to create precise controllers for Partially Observable Markov Decision Processes. PAYNT is capable of autonomously generating a robust controller based on FSC (Finite State Controller), but can also request guidance from other approaches like Storm [56], or in our case from a reinforcement learning algorithm acting as an oracle. This information can be used by PAYNT to reduce the exponentially large design space of irrelevant options and synthesize optimal FSCs faster.

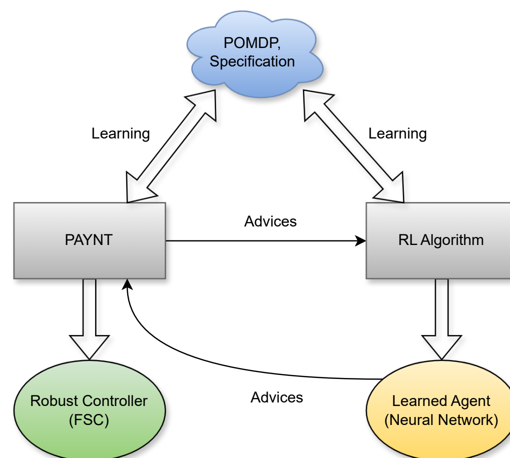


Figure 4.1: Overview schema of this thesis approach. It outlines the communication between PAYNT and RL agents.

Despite the ability of reinforcement learning algorithms to operate in a variety of large-scale environments, their functionality is limited to optimizing a loss function and their performance can only be assessed through environmental simulation. This makes it difficult to determine the point at which a robust agent has been effectively trained. In contrast, PAYNT has the ability to generate the policies represented by FSCs, which can be formally verified based on a given specification, and thus can enhance the RL approach with the desired safety assurances. Moreover, it can provide the RL approach with additional information about the model, aiding in the identification of goal states and the avoidance of dangerous ones. However, it is not as scalable as the RL approach; therefore, it can use the help of a trained agent in the form of advice.

The simplified general schema, as illustrated in Figure 4.1, shows that our toolkit is based on the synthesis of two independent policies: FSC (safe) and neural network (scalable). Our aim is to fuse these to develop more scalable FSCs and possibly safer RL agents (neural networks).

4.2 Reinforcement Learning Design

One of the main parts of this thesis was the development of a functional reinforcement learning pipeline for training agents. We can see the general schema in Figure 4.2, where we can see how we worked with input in the form of the POMDP, how we processed observation, rewards and actions in the overall training process, and the simplified outline of the communication between PAYNT and our reinforcement learning approach. Moreover, we can see four policies that may be created during our training process:

- Target policy – main policy trained by learning algorithm, usually selects actions greedily, tries to maximize cumulative reward.
- Collect policy – the policy for exploration in the environment. Usually, a stochastic version of the agent policy. In case of off-policy (DQN, DRQN) different from target policy, in case of on-policy learning the same but with some exploration element like stochastic selection of action instead of greedy.
- FSC collector policy – policy based on suboptimal FSC synthesized by PAYNT and wrapped by the TensorFlow Agents [67] interface. Used to collect trajectories from the environment similarly to collect policy.
- FSC policy advice – policy based on suboptimal FSC synthesized by PAYNT (similar to collector policy) used for increasing probability of selecting particular actions within stochastic PPO agent.

In the following subsections, we describe some technical details of the left part of the schema in Figure 4.2.

4.2.1 Environment Wrapper and Agents

One of the important parts of this thesis was the integration of multiple tools to one compact toolkit, where we designed and implemented some parts from scratch: the environment wrapper, which is described in more detail in Section 5.1 and implements the interface between the simulator (model), which is represented by Storm *SparsePOMDP* and *Simulator* objects. Furthermore, we implemented the masking process from scratch, which allows us to

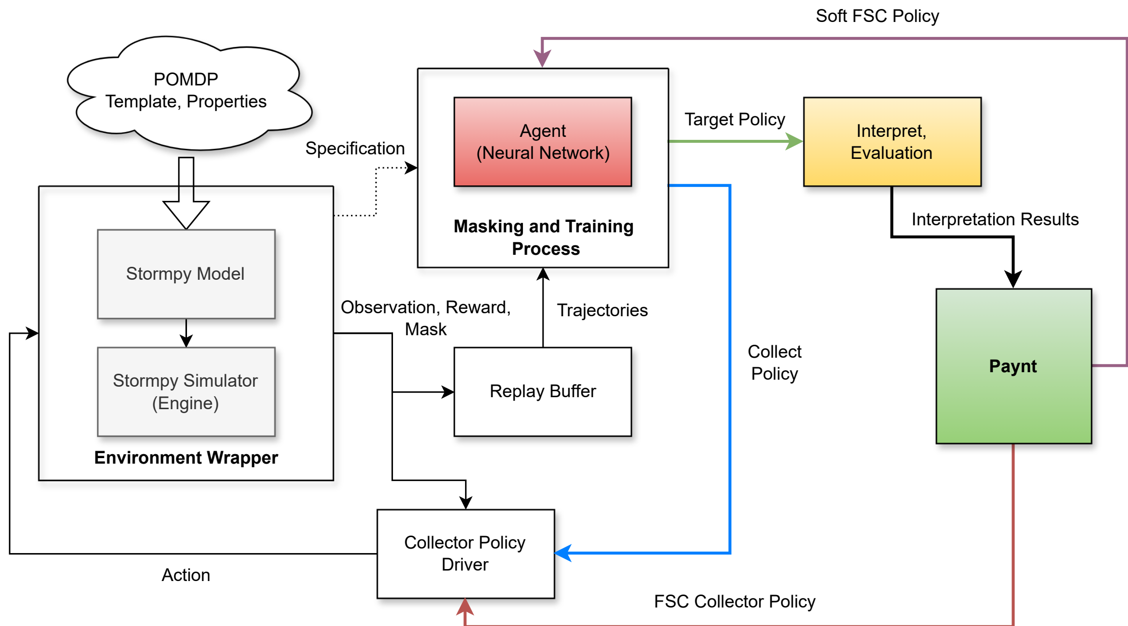


Figure 4.2: Schema of the designed reinforcement learning approach, which consists of environment wrapper that runs the Storm simulator, replay buffer with driver for collecting trajectories (episodes), the agent itself and part for communication with PAYNT.

use algorithms implemented from TensorFlow Agents [67] with dynamic action space (more described in Section 5.3. This framework implements multiple reinforcement learning algorithms such as DQN, DDQN, PPO, TD3, SAC, and more. We take library implementation of the learning algorithms and use our custom neural networks as function approximators.

Furthermore, we explore various agent specifications based on different observation encodings, which are overlooked parts of reinforcement learning for the specific task we are working with. We explore possible encoding options for Storm implementation in Section 5.2.

4.2.2 Replay Buffer and Collector Drivers

When working with the reinforcement learning algorithm, we have to somehow create communication between the environment (simulator) and the learning algorithm (agent). For example, a naive approach may be based on direct communication between the agent and the environment, as shown in Figure 4.3. However, this approach has some significant drawbacks and is usually used only during evaluation, not during training. The main one is that the agent is strictly limited to the knowledge it has during the training, and if it learns some suboptimal policy, it may forget even previously found better traces leading to its desired goal. This approach is usually recommended when we work with some adjusted on-policy actor-critic algorithm, which includes a critic for model learning and an actor for action selection.

Another option is that we use some collect (behavioral) policy, let it take actions in the environment, collect all its experiences, and then give it to some learning algorithm which tries to find optimal estimation of rewards in the environment. The advantage of this approach is that we do not forget any previously obtained experiences and are not limited

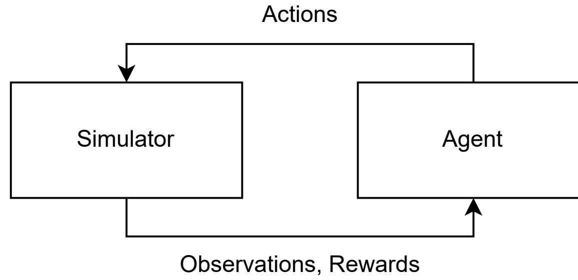


Figure 4.3: Naive approach of communication between agent and the environment.

by the policy we are currently learning. However, this approach lacks communication between the environment and the agent and is also limited to the quality of the exploration strategy.

The main approach with which we have worked within this thesis is an approach that uses replay buffers [74], which store previous experiences obtained during learning in limited buffers. This buffer is then used to randomly generate a dataset consisting of reconstructed sampled trajectories[1], which are taken as input to learning algorithms. In this thesis, we used the implementation of TensorFlow Agents [67], where the buffer is represented by the object of class *TFUniformReplayBuffer*.

Then, when we have the replay buffer, we usually also need a device that fills it with trajectories. For this reason, we use policy drivers that are also implemented in TensorFlow Agents [67]. Its purpose and functionality are clear: take a policy, run it in an environment, and add observed data to replay buffers. We can see an example of the pseudocode of the simplified driver algorithm in Algorithm 1. However, real implementation has some quality-of-life improvements, like starting from the last state of the environment simulator, dealing with obtaining last state during environment exploration, etc.

Algorithm 1: Simple Policy Driver

Input: **Simulator** *sim*, **Policy** π, **Int** *steps*, **ReplayBufferObserver** *observer*,
state ← *sim.reset()*
policy_state ← π*.initial_state*
for $i \leftarrow 1$ **to** *steps* **do**
  action, *policy_state* ← π(*state*, *policy_state*)
  next_state, *reward* ← *sim.step*(*action*)
  observer.add_trajectory(*state*, *action*, *reward*, *next_state*)
  state ← *next_state*
end

As we can see in Figure 4.2, we can use various input policies for our Collector Policy Driver. We can use our trained collector policy, which is the intended purpose of the drivers, but we can also use some external policies, such as the PAYNT FSC policy, or we may also input random policies, which may be useful during the first iterations of our learning algorithm. In our implementation, we use the driver implementation as an object of class *DynamicStepDriver* from TensorFlow Agents [67].

[1] The reconstructed trajectories are represented by batches of multiple following observation, action and reward tuples $(o_1, a_1, r_1, o_2, a_2, r_2 \ldots o_n, a_n, r_n)$, as we want to learn policies using RNNs.

Advanced Approaches for Replay Buffers

In this thesis, we use the uniform replay buffer implemented in TensorFlow Agents. However, there are some approaches that are based on advanced work with replay buffers. One of the considerable approaches is prioritized replay buffers, which focus on prioritized sampling from replay buffers [58]. This is based on increasing the probabilities of sampling trajectories that contain more promising information based on temporal difference (TD error about the environment and that can lead to more efficient agent learning. It also solves the issue that some experiences in the replay buffer may not be sampled at all before leaving the buffer.

Another approach based on advanced work with replay buffers is the so-called Search on the Replay Buffer [24] (SoRB), which uses trained agent for value estimation combined with initialized replay buffer to induce robust policies. It is based on the principle of localizing waypoints in the replay buffer and subsequent running of *ShortestPath* algorithm for finding shortest path between these waypoints to achieve the goal.

4.2.3 Weight Restarting

One of the major issues with learning algorithms is the problem of starting. We cannot see it in Figure 4.2, but deep learning-based reinforcement learning starts with randomly initialized neural agents when we use their initially random policies to sample trajectories from the environment. For some models, we found that sometimes we initialize the agents in the way that its initial policy can find a goal and learn the required policy really fast. In contrast, sometimes the initial policy is stuck in some neighborhood of the initial state and does not explore much.

Because of this, we added the option to initialize the policy multiple times and select the best policy in terms of achieving the objectives. This simple approach has shown that for some models it can significantly boost the start of the learning algorithm and improve the overall learned policy. However, for some environments such as *refuel-20*, it does not help at all, as the policy to reach the goal state is practically impossible to achieve with random policy.

4.2.4 Algorithms Optimization

A critical aspect of employing reinforcement learning algorithms is their demand for high performance, particularly in terms of rapid update computation and fast environment exploration. An identified problem when using the method described in [17] is the absence of compiled TensorFlow graph functions in their implementation, instead of relying on a standard Python function. It was observed that employing compiled versions of the functions *train* or *policy.action*, along with the TensorFlow Agents' compilation features (denoted by *tf.function* and *common.function*), could enhance the speed of our algorithms by more than tenfold. We also use fast CPU multiprocessing for training, inference, and sampling from replay buffers. However, the main bottleneck of this approach is the Storm simulator, as it does not, to the best of our knowledge, include parallel implementation. One of the possible solutions may be to run multiple parallel environments, but this implementation is not included in our approach.

4.3 Communication Between PAYNT and RL toolkit

In Figure 4.1 we show that we mainly try to learn two policies: formal and robust FSC, and scaling and approximating neural network. If we want to use them both to create more advanced policies, we first have to solve the communication between them. In this section, we describe some obstacles that we have not mentioned in the section focused on the whole interpretation problem in Section 5.4. Then we describe how we can distribute information between both of them, and in the next sections we describe how to use the sent information.

4.3.1 Conversion of the Action Mapping

One of the issues of the stochastic nature of the Storm Toolkit, which is the engine for both our approaches, is the issue that if we run both approaches twice on the same model, we obtain a different numbering for each action. For example, consider that we have a model with four fixed actions: *North*, *South*, *West* and *East*. The Storm does not provide this set of actions directly, but, for each observation, provides a range of numbers from 0 to n, where n is the number of legal actions in the current observation. This range is supplemented with labeling for each of these numbers, and from this labeling, PAYNT and approach we have originally been based on [17] construct set of all possible actions.

Unfortunately, both approaches use the Storm construction of the Python implementation of class *Set*, which is based on hashing, and thus after each run of the program, we obtain a differently ordered set of possible actions. Because we wanted to train our agents multiple times after running the code again[2], we solved this problem using Python lists instead of sets. However, we still have to convert the number of actions produced by the neural network to dynamic action space, which uses Storm, and, moreover, add functionality of the action mapping conversion to PAYNT. We can see the algorithm for the conversion of actions in Algorithm 2. We emphasize that this algorithm was used many

Algorithm 2: Action Conversion Algorithm

Input: **Int** *action*, **Dict** *network_labeling*, **Dict** *storm_labeling*
Output: **Int** *storm_action*
action_label ← *find_value_key(action)*
storm_action ← *storm_labeling*[*action_label*]
return *storm_action*

times in our approach, as we had to communicate with Storm simulator during learning, adopt advice from FSCs when combining RL approach with advice from PAYNT, giving advice from RL to PAYNT, etc.

A similar issue comes with the observations, where each observation mapping changes with different properties specification or size of the model. However, this problem has been overcome by always using the same model with the same specification and properties when running PAYNT, training RL agents, and interpreting.

[2]RL agents use as output layer fixed number of output neurons, which corresponds to each action from the set of all actions. Changing their order would mean that each neuron would have different semantics after running the code again.

4.3.2 Format of Communication

For improving learning, we need to distribute two separate information to PAYNT: information on action pruning and information on memory usage for each observation. In general, we send four separate objects:

- Observation to action list dictionary. Used for pruning actions for each included observation. It does not have to include all the possible observations.
- Initial memory guess. Dictionary containing pairs of observations with memory size. It has to include the whole set of possible observations.
- Labeling of network action keywords. Used as input for the action conversion Algorithm 2.
- Ordered list of observation numbers by variance of actions for each observation. Used for memory updates in PAYNT, it does not have to include the whole set of observations.

If we want to communicate in the opposite direction, we would like to send information about the construction of an FSC for the RL approach. For this purpose, we use an implemented class in PAYNT called *FSC*, which includes all important information about the action function, the memory update function, the labeling of action and observation, etc. This class can be used as is, or we can export it to *JSON* and construct an FSC within the RL approach.

4.3.3 File Communication

In this approach, PAYNT does not communicate directly with the RL algorithm. Instead, we train the agent using our selected algorithm, network, and learning arguments until we are satisfied with the results of the algorithm. We may run it with a different model, change the learning rate during the learning iterations, etc. Then we export the results of our interpretation to Python Pickle format, which we may load with PAYNT and use for synthesis.

We also applied the same method in reverse, enabling us to export the resulting FSC to JSON and subsequently utilize it within the RL algorithm. This strategy proves beneficial when our intention is to independently experiment with both methods. The main drawback is that we have to do all the processes by hand or by using an external Bash script.

4.3.4 Integration of RL to PAYNT

The second option to handle is the integration of RL with PAYNT, where PAYNT calls all procedures automatically and does not rely on the export of Pickle or JSON files. The main drawback is that the reinforcement learning algorithms are usually not very stable and that we may obtain significantly different results for each run. However, this approach is the primary option when we perform experiments with the loop of neural network and FSC.

4.4 Usage of RL Oracle in PAYNT

In the application of reinforcement learning as a predictive model for the generation of FSC in PAYNT, our objective is to use two different pieces of information. We need to determine

which actions should be taken primarily into account for each observation and the amount of memory that should be assigned to each observation. These strategies are based on PAYNT's method of policy learning, which operates within the exponentially expanding design space of all k-FSCs, where it endeavors to discard irrelevant solutions and choose the most efficient policy.

4.4.1 Reduction of the Design Space

One of the most important parts of the synthesis of FSC is the reduction of the design space, since the original design space is exponentially large. PAYNT implements various approaches on how to deal with the reduction of the design space for a more directed policy search, including the Symbiotic [1] approach based on cooperation of the belief-based Storm and the policy-search-based PAYNT. In this thesis, we experiment with a similar approach, where we reduce the design space by the advice of the RL oracle.

When considering the design space, PAYNT considers three different terms: family, main family, and subfamilies. The family includes all possible k-FSCs given a different memory for each observation. This means that for each observation, we have to consider all playable actions. For example, consider three different observations with dynamic action space and assigned memory:

$$(o_1, A_1, m_1) = (1, \{1, 2, 3, 4\}, 2) \tag{4.1}$$

$$(o_2, A_2, m_2) = (2, \{1, 2, 3, 4, 5\}, 3) \tag{4.2}$$

$$(o_3, A_3, m_3) = (3, \{1, 2, 3\}, 1) \tag{4.3}$$

if we had for each observation assigned only a single memory cell, the total number of possible FSCs would be $4 \cdot 5 \cdot 3 = 60$. If we also consider the memory assigned, the number of possible FSCs would be $4^2 \cdot 5^3 \cdot 3^1 = 6000$. As we can see, the size of the family grows exponentially.

The second term, main family, takes into account only a trimmed-down version of the design space, where we reduce the number of actions that can be considered. For example, imagine that the oracle informed us that the only logical action for observation observation 3 is the action *west*, which is symbolized by the number 2. This would imply that our main family's size was cut by a factor of three: to 20 when memory is not taken into account and to 2000 when memory is considered. The last term, subfamilies, is the opposite of the main family. This design space considers only the FSCs which have been thrown away in the main family by considered restrictions. In this case, it would be FSCs, where we consider actions 1 and 3 for observation 3.

While operating within the given constraints, we have two potential strategies for policy search: we can first prioritize k-FSCs from the main family and then, if required, explore the subfamilies. This does not mean that we have shrunk the design space; we simply altered the exploration schedule within it. The alternative strategy is to ignore subfamilies and focus solely on the main family. Each strategy has its own set of benefits and limitations. The first strategy allows for a more efficient traversal of the design space but restricts the possibility of exploring the entire space. However, the latter strategy allows for a potential exploration of the entire design space (given the significantly smaller size of the main family), but it risks discarding viable solutions that the original oracle may have missed. In this thesis, we experiment with both approaches.

4.4.2 Selection of the Initial Memory

Another option to improve the search for policies within PAYNT is "jumping" to achieve an optimal observation memory distribution. This means that we tell PAYNT how many memory nodes it should consider for each observation. If we consider our example from the previous subsection of observation, action, and memory tuples, it means that we know the exact memory values from the start.

In practice, PAYNT starts with a memory set to 1 for each observation, and thus first explores the design space with memoryless policies. Experiments show that this configuration is actually relatively optimal, as exploration of this set of policies is not overwhelming. However, for larger models, this solution may lead to not exploring the design space with higher memory assignments, and there is a space for improving models with information from the RL agent. In this thesis, we simply use the memory approximation we describe in Section 5.4.3.

4.4.3 Memory Update Prioritizing

Another option, how can the reinforcement learning approach help PAYNT learning policies, is the use of information on memory updates for each observation. If we raise the memory too much, we significantly affect the performance of exploration of design space. If we raise the memory too slowly, we will explore a design space very similar to the original one, and thus we will be obligated to increase the memory more times. Existing approaches work on the principle that when an oracle (for example, Storm [56]) recommends multiple actions for some observation, it increases memory for exactly these observations. We observed that the number of observations with recommended actions is usually $\frac{1}{10}$ of all observations. For this reason, within our approach, we sort our hints by the variance of recommended actions for each observation and increase the memory for the first $\frac{1}{10}$ observations with the highest variance. However, this approach was selected only on the basis of a small empirical experience, and further research may be beneficial.

4.5 Usage of FSCs in RL Approach

In this thesis, we investigate two strategies to improve the learning processes of RL agents using PAYNT. The initial strategy, depicted in Figure 4.2, involves the use of an FSC as a collector policy. The second strategy uses an FSC policy to modify the action probabilities of the PPO policy[3], with the aim of subtly favoring actions that achieve the objective. This method is detailed in Algorithm 3, which illustrates the initial setup of the loop with initial agent training and the first pruning of the design space, followed by a loop with synthesized FSCs and insights from trained agents. Throughout the training phase, we accumulate experience in our replay buffer and maintain its contents between training sessions.

Before we describe both approaches, we should note that the FSC and TF Agent algorithms usually work with different forms of feedback (more details are described in Section 3.1) and we have to address this issue when running these different policies.

[3]This applies solely to the PPO policy, as it is the only stochastic policy.

Algorithm 3: Reinforcement Learning with FSC

Input: **Int** pre_iter, **Int** fsc_time, **Int** fsc_iter, **Int** $train_iter$
$hints \leftarrow RL_training(pre_iter)$;
$prune_design_space(hints)$;
$assignment \leftarrow synthesis_phase(fsc_time)$;
$FSC \leftarrow construct_FSC(assignment)$;
while *True* **do**
  $RL_training(fsc_iter, FSC)$;
  $hints \leftarrow RL_training(train_iter)$;
  $enhance_memory(hints)$;
  $prune_design_space(hints)$;
  $assignment \leftarrow synthesis_phase(fsc_time)$;
  $FSC \leftarrow construct_FSC(assignment)$;
end

4.5.1 FSC as Hard Collector Policy

The mentioned approach has some significant drawbacks. The main one is that, as we play with some deterministic policy with a sharp "probabilities" set to 1 for the selected action and 0. In case of the DQN and DDQN, it does not imply problems, as these policies are based on greedy selection of action with the highest estimated return. However, in case of stochastic PPO, it causes an explosion of the loss function, as it also needs probabilities (logits) of categorical distribution, from which a selected action was sampled. For correction of this issue, we have to imitate the distribution from PPO and combine it with the sampling actions from the FSC. This approach shows that after couple of learning iterations, PPO can learn the policy represented by PPO.

The main motivation for this approach is the issue of locating goal states in models with sparse reward. For example, in the case of *refuel-20*, to achieve the goal, we have to randomly select at least 43 consequent actions[4], which, at least according to our experiments, is not realistic at all. The main goal of the usage of FSC is selection of initial collect (behavioral) policy to find these goals, filling the replay buffer with these trajectories, and then training it with, at least, some suboptimal possible solution.

As noted in the opening of this section, it is important to recognize that RL and FSCs possess distinct hidden states (memory nodes versus recurrent feedback). Consequently, to replicate the PPO policy while implementing the FSC policy, we must incorporate hidden memory into the policy that encapsulates FSC[5]. This is achieved by introducing a hidden attribute that stores the prior hidden state in the class designated for the FSC policy. Although not the perfect approach, it remains the most feasible solution we identified that avoids intricate engineering. Moreover, in the case of FSCs, we use an episodic driver instead of a step driver. It means that we run full episodes instead of couple of steps in environment, as we do not want to mix hidden states of drivers of neural and FSC policy.

[4]Moving north, east and refueling fuel. Number 43 is the lowest possible number if we expect the best result for each move

[5]Implemented by a class derived from *TFPolicy* in TF Agents [67].

4.5.2 Soft FSC with PPO Policy

The second approach that we designed and implemented is the integration of PPO with the FSC policy. For our approach, we use our policy wrapper described in Subsection 5.3.3 where we extract logits generated by PPO to construct a categorical distribution. To integrate our FSC hints to PPO, we slightly modify the constructed distribution, where we raise logits with the preferred action by a small constant. Formally, we simply change logits as:

$$logit'_i = \begin{cases} logit_i + \gamma boost & \text{if } p == i \\ logit_i & \text{otherwise} \end{cases}$$

where p is the index of the preferred action, *boost* is some constant, and γ is its multiplier, which changes over time. In this case, *boost* is assigned a value of 1, while γ starts at 2.0 and is halved when PAYNT failed to improve FSC, as we do not want to propagate the same FSC to our solution multiple times. This forces PPO collector policy to play primarily the actions from FSC within the first iterations of RL with PAYNT loop, and then continue with its own action selection. However, the approach for changed logit computation should be investigated more, and some improvements may be beneficial.

Similarly to the previous approach, we have to remember the previous hidden state of FSC aside from the hidden state of the recurrent neural network. The solution is the same, as we just add an object attribute to remember FSC state.

Chapter 5

Technical Details

In this chapter, we explore the primary theoretical and practical ideas encountered during the research for this thesis. We go into the fundamental principles underlying the functionality of the algorithms employed and the way in which we depict the tasks under consideration. In addition, we outline the key obstacles encountered, such as dealing with a dynamic action space and representations of observations.

5.1 Environment

In one of the previous sections, we explored some significant reinforcement learning algorithms that would be of little value without a suitable environment to train our agents on [43, 66]. As this thesis focuses on the integration of PAYNT [2, 3, 48] with reinforcement learning, it is essential to select environments that are compatible with both systems. Given that the current implementation of PAYNT is designed for a formal environment with a fully known model, adjustments must be made to the reinforcement learning simulator to align with the environments supported by PAYNT.

5.1.1 Template and Properties

Before we discuss other POMDP topics such as representation, environments, simulators, wrappers, or agent interpretation, we should describe the first building block of our tasks. These are PRISM [44] templates and PCTL properties, which we use as a shared specification for both reinforcement learning and PAYNT.

PRISM is a framework designed to create templates that include descriptions of states, transition probabilities, reward mechanisms, labels, observables, observation probabilities, and equations that explain the current or previous states of (PO)MDPs. These characteristics are captured by specific models that depict aspects of the behavior of POMDP. It also features a section on holes that represent areas of incomplete information within a system. In reinforcement learning, a crucial element is the description of environmental rewards, which might be represented as follows:

```
rewards "penalty"
    [up]    true : penalty;
    [down]  true : penalty;
    [left]  true : penalty;
    [right] true : penalty;
```

```
    [wait]  true : penalty;
endrewards
```

where we can see part of PRISM template, which describes reward function for each action (eg. `up` or `wait`). This means that our agent obtains some reward with the reward `penalty` for taking any action. In our approach, we usually suppose that the model contains only a single reward model and we give our agent the negative values. If there are more reward models, we will use only the last one.

The second mentioned part, the PCTL specification, describes our goal for learning policies in the environment. In this thesis, we usually work with one of the two types of PCTL properties:

- `Popt = [cond]` – there we want to optimize probability of fulfilling some condition, where `opt` is `min` or `max` and `cond` is some LTL formula. For example, property for some grid model `Pmax = ["notbad" U "goal"]` describes maximizing probability of being in `notbad` states until reaching `goal`.

- `R{var}opt = [cond]` – reward criterion, where we want to optimize `opt` (`min` or `max`) some reward for a given variable `var` given some condition `cond`. For example, property `R{"steps"}min=? [F "goal"]` corresponds to minimizing the steps before reaching the goal. This specification was usually more complex to fulfill, as we have to find a solution that optimizes the previous condition with probability 1.0.

Sometimes, we find models described not only in PRISM, but also in Jani, Cassandra, or DRN, which are compatible with Stormpy [36] and PAYNT, which have different structures, but also describe POMDPs.

5.1.2 Storm

In this thesis, we need to use a toolkit for parsing the mentioned templates and specifications and for representation of the POMDP simulator. In our case, this is performed by the Storm toolkit.

Storm is a modular verification toolkit for probabilistic model checking based for various Markov models (MDP, POMDP, DTMC, CTMC etc.), Petri nets, or Markov automatas. It supports various verification specifications, including those for multiobjective verification, conditional probabilities and rewards, or for long-term rewards [37]. One of the important features of this toolkit is the presence of the Python API, which the authors call Stormpy [56].

Moreover, the Storm toolkit may be used for many other functions. For example, the authors of [17] used Storm for the calculation of belief support to provide its toolkit capability to use it for state estimation. Other approaches, such as, for example, [5] are using belief MDPs for learning policies, and we may obtain them from Storm. However, in this thesis, we primarily use Storm for its ability to process POMDPs and the creation of simulators over them. In the following subsections, we describe how we work with models and how we use POMDP simulators.

5.1.3 Stormpy Model and Simulator

When dealing with POMDPs using formal methods, we usually rely on a model that includes states, observations, and transition probabilities. However, in the context of reinforcement

learning, the focus shifts towards executing our model. Fortunately, Stormpy provides functionalities for constructing a simulator based on our sparse POMDP model, which is created by Stormpy from the PRISM template and the PCTL property. This simulator enables us to reset the environment and then navigate through it by selecting actions using the `step()` method. In addition, it allows us to access current observations, provide labels for these observations, and receive rewards. In essence, Stormpy's simulator encompasses all the necessary components for us to develop a simulator for our reinforcement learning agent.

5.1.4 TensorFlow Wrapper

In previous section, we described the overall framework for framework simulation. However, for compatibility with agents implemented from the TF Agents [67] library, we have to use some wrapper. In our case, the authors of TF Agents developed the wrapper class `TFPyEnvironment`, which takes as a parameter during the initialization object of class `PyEnvironment` with overwritten methods to perform the step in the environment (`step`), environment restart, and all important TF Agents specifications such as observation, action, or reward specification. Thus, for the creation of a proper environment, we can use our Stormpy simulator from the previous subsection and wrap it with the mentioned objects. However, there are some issues that we have to solve first.

The primary concern revolves around the reward structure. Although the PRISM [44] framework incorporates rewards (or penalties) for actions within specific contexts, the main focus is guided by the properties chosen. These properties typically articulate objectives such as maximizing the likelihood of reaching a goal while avoiding obstacles or minimizing the number of steps taken in the environment. Although such properties are valuable for formal approaches like PAYNT, they do not meet the requirements of reinforcement learning, which requires a precise reward system to motivate the agent to progress toward achieving the goal [43, 66]. In our approach, we use **virtual goal values** to motivate the agent to finish the task and anti-goal values for other sink states like traps.

Other problems we have to solve when wrapping simulator by our TensorFlow wrapper are technical details when to restart the environment, because while formal methods uses some threshold for relevancy of following steps, we have to stop and restart the environment sometimes. This is, in our case, solved by restarting the environment from the inside after reaching some constant number of steps, or it can be solved by detecting loops.

An additional important aspect to consider is ensuring full compatibility with the Stormpy environment in terms of maintaining consistent observation and action definitions. Although the observation remains unchanged after initialization using the same template and properties files, the configuration of the action space differs significantly from standard reinforcement learning algorithms. In the Stormpy simulator, each observation presents a distinct set of feasible actions, represented by a sequence of numbers ranging from 0 to n, where n corresponds to the available actions in that specific observation. Resolving this disparity requires addressing the mapping between our agent's action space and that of Stormpy, as well as implementing measures to prevent the agent from selecting invalid actions, a topic that is further elaborated in Section 5.3 and in Subsection 4.3.1.

5.1.5 Reward Shaping

As we described in Section 3.5, many of the tasks presented have only a sparse reward. It means that we obtain reward only after finishing some long period of steps, which leads

to some goal or checkpoint. This causes issues for problematic start of learning and issues with stability, as reaching the goal with random strategy is usually difficult. In this thesis, we try to develop some simple variations of reward shaping, which we based on the different forms. The first is to add a small reward for reaching the checkpoint in the form of a fuel station to *refuel* tasks for the first time. The second form was calculated by computing the length between the desired goal and the last step of the episode[1]. The distance from the goal state is computed in grid tasks as:

$$distance(x_i, y_i) = \sqrt{(x_i - x_f)^2 + (y_i - y_f)^2}$$

where (x_i, y_i) is the last position of agent and (x_f, y_f) is the position of the goal state. The reward if we did not achieve the goal or trap state is then computed as:

$$reward(distance(pos), goal_value) = \frac{goal_value}{distance(pos)}$$

where $goal_value$ is some constant value for achieving goal state.

We conducted a couple of experiments with this implementation for tasks *evade* and *refuel-20*, but the results were not satisfactory enough and usually only led to getting stuck in some local optima. We suppose that this approach may lead to a significant improvement of trained policies, but it would need some difficult engineering and multiple experiments, which is not the main point of this thesis.

5.2 Observation Representation

One challenge with existing approaches to the tasks addressed in this thesis is the way observations are processed for the RL algorithm. Although many current approaches [61, 32, 55, 20] deal with observations (or states) in high-dimensional spaces, these tasks typically involve simple environmental information such as X and Y coordinates, fuel levels, or trap locations. The issue arises when one attempts to apply deep learning algorithms with feature extraction to learn policies in environments with only a few features per task. Furthermore, established solutions such as [17, 18] often simplify the problem by obtaining not just a single observation from the environment for their POMDP solution, but also a belief distribution over possible states at each time step, which is considered a sufficient statistic [43], instead of incorporating memory or recurrence in LSTM.

Consequently, these solutions typically do not use recurrent neural networks and are typically trained in simpler tasks. During the course of this research, three distinct forms of observations from the Stormpy environment were identified. The first approach, which was proposed as an alternative to the belief distribution in [17], involves the use of a single numeric observation from the environment simulator. However, this method poses challenges, since the number of potential observations is often lower than the number of weights in the neural networks employed by the agents. Furthermore, these observations lack consistent semantic relationships when they are close in proximity. For instance, given observations 2 and 3, and 2 and 5000, it is difficult to determine which pair is more similar or dissimilar semantically due to Stormpy's unique numbering of observations.

The second approach, which represents an enhancement of the initial numeric observation method, is one-hot encoding [11, 30], where a single observation is transformed into

[1]We set the limit of steps in episode to some constant for each task, because some trajectories may lead to infinite cycle.

a vector. This vector comprises $n - 1$ zeros, where n represents the number of possible observation options, with a single one indicating the position of the observed number. For example, if there are 5 possible observations and observation number 2 is observed, the resulting vector would be $(0, 0, 1, 0, 0)$. This approach is useful because it erases the problems with similarity and dissimilarity of observations, which have a close number of observations.

5.2.1 Stormpy Valuations

However, Stormpy usually provides for each environment model a few observable variables, which, to our knowledge, have not yet been used by any implemented solution. For example, if we work with the evade model described in PRISM [44], we can find in the template file lines:

```
observables
start, dx, dy, turn
endobservables
```

which tell us, that we can observe some variables *start*, *dx*, *dy* and *turn*. We found that we can extract valuations of these observations from our Stormpy model simulator and use it for training our RL agents, which provides more stable learning. Moreover, it enables us to learn the actual semantics of each observation, which may also lead to better generalization for unseen observations.

5.3 Dynamic Action Space

One of the main obstacles facing reinforcement learning in the tasks examined in this thesis was the dynamic nature of the action space $A(s)$. The challenge arises from the way Stormpy [56, 37] handles the action space of POMDPs compared to the conventional approach of reinforcement learning algorithms. For example, in simpler critic-based reinforcement learning algorithms such as DQN, our model estimates the Q values, denoted by a function $Q_\pi(o, a)$ with parameters o representing current observation[2], and a representing an action such as moving *north* in grid-based tasks. This function calculates the total reward expected when following a specific policy π. Typically, this function is implemented using a neural network with a fixed number of neurons, resulting in a fixed number of outputs, each corresponding to an action in a finite discrete space.

However, this approach differs from the approach employed by Stormpy. Stormpy typically involves a predefined set of action labels such as *north*, *south*, *placement*, *refuel*, etc. However, in each state (observation), only a limited set of permissible (playable) actions are available. For example, in Figure 5.1, there is an agent that aims to reach the green state. When using DQN or another RL algorithm, the agent initially considers all feasible actions across the entire grid (move *up*, *down*, *left*, *right*, *stand up*), and during the learning process it can choose any of these actions. However, as depicted in the left subplot, the only viable actions are *go right* or *go down* (2 out of 5), while in the right subplot, the agent can only *stand up* – this defines how Stormpy defines the action space.

We explored two possible solutions to solve the dynamic action space problem – from the side of the environment (wrapper) and from the side of the agent, as we can see in Figure 5.2. However, the ideal path does not exist because it may lead to lower sampling efficiency or to problematic compatibility of the RL algorithm and the environment representation.

[2] And memory, in the case of RNNs such as LSTM.

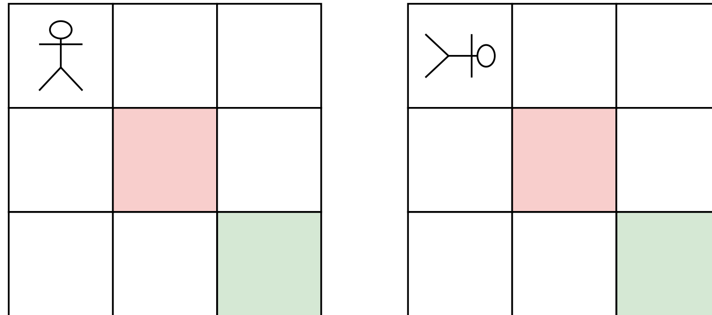


Figure 5.1: Agent in some grid model task. In the right picture agent fell down and have to stand up.

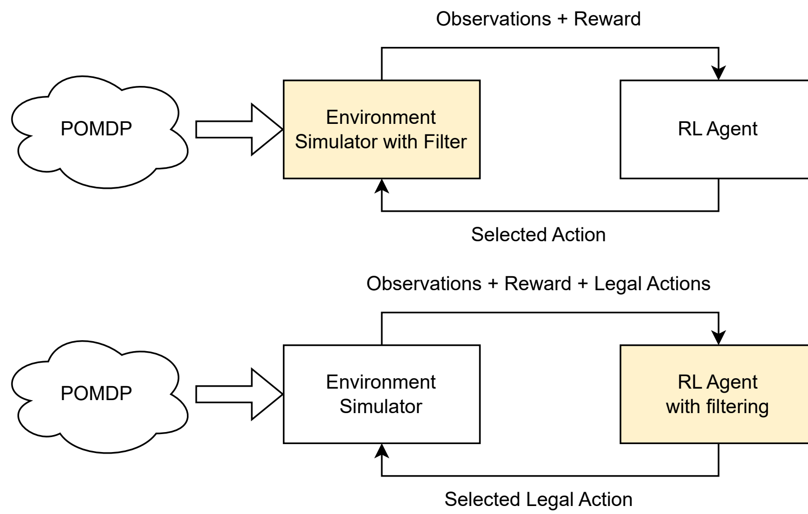


Figure 5.2: Two possible paths to solve problem with dynamic action space. Yellow color tells, who takes main responsibility.

5.3.1 Masking

One strategy to restrict the action space involves employing a technique known as masking, or in the context of TensorFlow agents [67], utilizing observation and action constraint splitters. Referring to Figure 5.2 in the preceding subsection, this corresponds to the bottom option. This method involves artificially adjusting the outputs of the implemented policies by setting the outputs (logits) of prohibited actions to a value close to negative infinity. Consequently, this ensures that the agents select actions (via a deterministic or stochastic method) with zero probability of selecting an illegal action.

This approach involves dividing the observation structure into two distinct parts, the observation section and the mask section. As illustrated in Figure 5.3, the modified time steps containing an extended observation with both observation and mask components are sent to the agents. Adapting this method to a custom environment requires minor adjustments in computing the mask, updating the observation specifications, and returning the modified time steps. This strategy significantly improves sampling efficiency by ensuring that no samples contain invalid actions. Furthermore, our algorithm eliminates the need to learn a substantial portion of the model specification, since action filtering is handled separately.

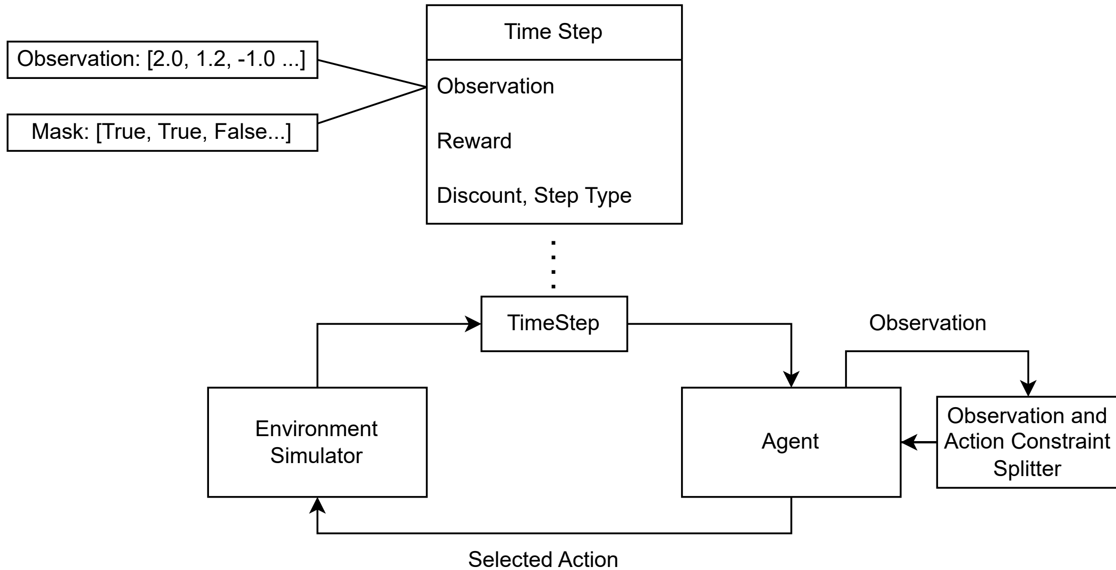


Figure 5.3: Implementation of action masking.

However, this method has several notable disadvantages. The primary issue is that it could hinder the model's ability to generalize since we manually restrict the output of the function approximator. Consequently, the algorithm may not learn, for example, that moving *upward* is not viable in the top section of the grid, information that could be valuable for similar observations. Additionally, a major drawback is the limited compatibility of masking implementation in contemporary RL frameworks. For example, as illustrated in Table 5.1, only the simplest algorithms (critic-based) support this feature. Although the PPO algorithm offers greater stability and other attractive attributes, its lack of masking implementation prevents its direct use with action masking.

Some of the approaches overcome the compatibility issue by reimplementing the algorithms from scratch [17], but with some notable limitations, such as the absence of

Algorithm	Masking	Primary Action Space
DQN	Yes	Discrete
DDQN	Yes	Discrete
PPO	No	Both
SAC	No	Continuous
DDPG	No	Continuous
TD3	No	Continuous

Table 5.1: Compatibility of observation and action constraint splitter for algorithms implemented in TensorFlow Agents [67].

compatibility with RNNs (replaced by the use of belief support instead of direct observations). Moreover, own implementation of learning algorithms can lead to bugs combined with problematic compatibility with complex computation-optimization methods, and we found that their implementation runs below expectations.

5.3.2 Environment Action Filtering

The second approach to addressing illegal actions involves action filtering, which is considered somewhat simplistic. This approach allows the agent to choose any action it desires and then filter out actions within the environment (wrapper). If an illegal action is taken, the agent can either stay in place (resulting in a negative reward) or choose a random, permissible action.

However, in the tasks examined in this thesis, where only a limited set of actions can be taken based on the observations (e.g., 1 out of 7), this significantly reduces the effectiveness of the algorithm in terms of sampling, causing it to become stuck and wait until a different action is selected during training. Opting for a random action in such cases can prevent this deadlock, but the challenge remains that the agent is unaware of whether the action it took was blocked or randomized. This method is advantageous for its compatibility with various discrete algorithms in TF Agents (like PPO), yet no notable enhancements have been identified compared to simpler learning algorithms that incorporate masking.

Prioritised Action Randomizing

Small improvement over basic action filtering may be the use of stochastic algorithms. In this approach, we take an algorithm that may select actions with some probabilities (such as PPO) and, in addition to action, send this distribution (or output logits) to the environment. The environment then uses the action, if it was legal, or selects a random action according to the distribution given by formula:

$$selected_action(action, distribution) = \begin{cases} action & \text{for legal action} \\ distribution.sample() & \text{for illegal action} \end{cases}$$

If the formula samples another illegal action, we may simply repeat it until we get some result, or use greedy option, when we select the most probable legal action.

This approach improves learning in terms of exploring more valuable observations (states) but still does not improve the learning issues with the knowledge of the real action outcome, as there is no feedback information about sampled action.

5.3.3 Policy Wrapping

The novel option that we have designed and implemented is to wrap the policy. This approach consists of wrapping the original policy, which must be stochastic[3], in the policy wrapper, which serves as an interface between the simulator and the original policy. This allows us to use original policy without masking (like PPO) to work with dynamic action space without the drawback of not knowing which actions were really selected (unlike in the previous Subsection 5.3.2). We can see the general idea in Figure 5.4.

We can implement wrapper filtering in multiple ways. For example, we can sample an action multiple times and refuse it until we get a legal action, take the legal action with the highest probability, or create an improved categorical [11] distribution from the output of the original policy. The last approach is similar to the approach used in the implementation of masking in TensorFlow Agents [67].

As we have multiple options, we select the latter two. We select the action with the highest probability when evaluating the policy because the other approaches can draw "dangerous" actions when evaluating. For collector policy, we sample from the improved categorical distribution for the exploration policy, as it explores not only the best current paths.

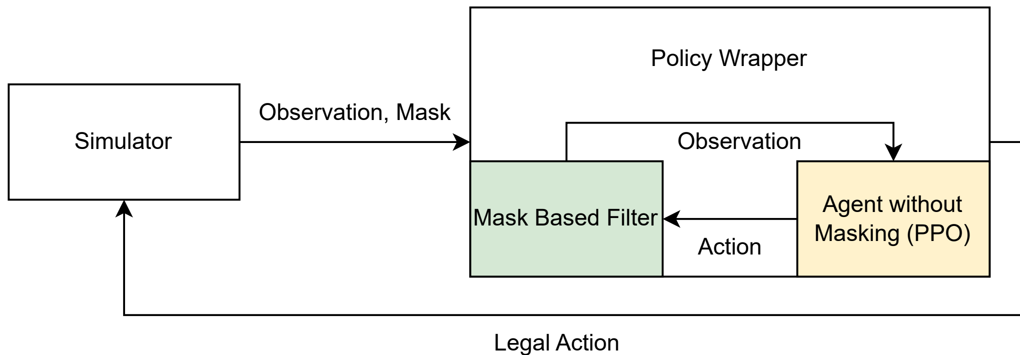


Figure 5.4: General idea of policy wrapping.

5.4 Interpretation

In this section, we describe our approach on how to interpret trained RL agents to information usable by PAYNT. We first start with a general idea of what we want to implement. Then we describe the proposed naive model-free approach, which has some limitations and in the final version is not used, but its idea best describes our goal. In the last subsection, we describe our simple novel approach, which effectively obtains the interpretation result for PAYNT by evaluating trajectories in the environment.

5.4.1 RNN Feedback Approximation

In our task to derive all possible actions from a given current observation and any previous history, our aim is to feed our RL agent with all possible hidden states in combination with the given observation. A naive method that could be employed involves the uniform

[3] It has to pick different actions when called multiple times.

sampling of hidden states $\mathbf{h}$ from the set $\mathbf{H_o} = \{\mathbf{h} \in \mathbb{R}^n \mid l_i < h_i < u_i, \text{ for } i = 1, 2, \ldots, n\}$, where n represents the dimension of a single hidden state, and l_i and u_i denote the lower and upper bounds of the component h_i of the hidden state, respectively. Using this method implies that we are capable of calculating the upper and lower bounds for each component of the hidden state vector[4]. Furthermore, it is essential to ensure that even when the boundaries are accurate, the individual components are independent and can be derived through uniform sampling from the mentioned set.

If we satisfy these conditions, we could compute action given some history h, observation o and neural network f as:

$$action = f(o, h)$$

and if we would like to compute all possible actions for a given observation, we could simply use:

$$actions = \bigcup_{h}^{\mathbf{H_o}} f(o, h)$$

As noted previously, this method raises numerous problems, such as determining the boundaries, the manner in which we should sample to investigate the limitless realm of potential histories, and primarily the question of whether this method aligns with the functionality of RNNs. Moreover, we should count with the possibility that sampling one extreme value can break the entire selection process, and we can possibly simply reconstruct the entire original set of actions A for each observation o, which would not help PAYNT prune the design space.

5.4.2 Model-Free Approximation

The first method that we propose and test is a model-free algorithm that generates history samples from agents by constructing imaginary trajectories. In this approach, we work with the function f representing the neural network (policy π), which has input in the form of a pair o_t, h_{t-1} returns the pair (a_t, h_t), where o_t is observation, a_t is action, and h_t is action in time t. The algorithm is based on two independent steps, where the first one estimates distribution of possible histories for each observation, and the second one then generates samples from distribution and combine it with observations. We outline the distribution estimation step in Algorithm 4, where we can see that we estimate the boundaries for the uniform distribution[5].

Algorithm 4: Distribution Bounds Estimation

Input: Policy π, **Int** *granularity*, **Int** *number_of_observations*
Output: Dict *min_limits*, **Dict** *max_limits*
min_limits, *max_limits* ← *initialize_limits*()
for $i \leftarrow 1$ **to** *number_of_observations* **do**
    mins, maxs ← *emulate_trajectories_to_observation*(π, *i*, *granularity*)
    min_limits[*i*] ← *minimum*(*min_limits*[*i*], *mins*)
    max_limits[*i*] ← *maximum*(*max_limits*[*i*], *maxs*)
Return *min_limits*, *max_limits*

---

[4] In the case where we employ multiple layers of LSTM, our hidden states take the form of a matrix.
[5] Better approach would estimate distribution

The subsequent phase involves traversing the set of potential observations along with sampling from a uniform distribution determined by the boundaries described above. This method is detailed in Algorithm 5, in which we determine the distribution boundaries, followed by sampling a collection of possible histories. Using this history, we obtain the action and update our statistics. These updated statistics are then used to generate hints for PAYNT.

Algorithm 5: Model-Free Interpret

Input: Policy π, **Int** *distribution_granularity*, **Int** *granularity*, **Int** *observations*
Output: ResultStruct *observation_action_stats*
observation_action_stats ← **ResultStruct**()
min_limits, *max_limits* ← *distribution_bounds_estimation*(π, *distribution_granularity*, *observations*)
for *obs* ← 1 **to** *observations* **do**
    for *i* ← 1 **to** *granularity* **do**
        hidden_state ← *sample*(*min_limits*[*obs*], *max_limits*[*obs*])
        action ← π(*obs*, *hidden_state*)
        observation_action_stats.update(*obs*, *action*)
Return *observation_action_stats*

However, this approach is very naive and during our experiments we found that it usually tends to generate actions for observations that are visited very rare or never, it is performance-demanding for higher sampling strength, and it does not provide real distributions as we use imaginary trajectories and with poorly trained agents leads to estimate all possible actions for each observation. Moreover, we faced various implementation issues, as recurrent neural networks usually do not have the same shape of history for different agents. For these and many other reasons, such as the problem of constructing real trajectories by computing reachability[6], we focus on another approach, which in theory uses the same principle but is more efficient and works with a real model.

5.4.3 Tracing Interpret

The main issue with model-free interpretation of RL agents is that we have to construct reasonable histories to obtain an accurate interpretation. We could also use various different approaches, but we were inspired in the article [40], in which they discuss that the usual approaches for explainable AI are based on post hoc analysis, where we take a trained agent and try to interpret it. In contrast, they boast approaches based on pre hoc analysis, where they construct during prototype-based agents during learning of neural network. However, our approach is still primarily post hoc (even if it can be used as pre hoc), but it takes slight inspiration from the mentioned approach.

In this approach, we do not construct imaginary trajectories, but instead run the agent in the environment and observe its behavior. We outline the method in Algorithm 6, where we can see that we work with our policy π in the form of a neural network, the number of evaluation episodes, and the POMDP simulator. The principle is based on standard evaluation of policies, but in this approach, we extend the evaluation with counting

[6]In our experiments, we started with shared observation history for all observations and thus eliminated this issue by other trade-off.

stats during learning to our dictionary of dictionaries, represented by **ResultStruct** in the algorithm. This dictionary contains as keys numbers of observations and each element of dictionary is represented by another dictionary, which contains keys in the form of action numbers and values in the form of the number of occurrence of each action. We outline this structure in Figure 5.5, where the colors represent the observation and the letters represent actions.

Algorithm 6: Tracing Interpret

Input: **Policy** π, **Int** *episodes*, **Simulator** *simulator*, **Bool** *refusing*, **Label** *goal*
Output: **ResultStruct** *observation_action_stats*

```
observation_action_stats ← ResultStruct()
for i ← 1 to episodes do
    hidden_state ← π.initial_state()
    state ← simulator.restart()
    aux_stats ← ResultStruct()
    while not state.is_last() do
        action, hidden_state ← π(state, hidden_state)
        next_state ← simulator.step(action)
        aux_stats.update_stats(state, action)
        state ← next_state
    if not refusing or state.label = goal then
        observation_action_stats.merge_stats(aux_stats)
return observation_action_stats
```

The output of Algorithm 6 then goes through another processing, where we construct objects that we distribute in Subsection 4.3.2. This process includes construction of a dictionary of observations and actions, where we simply take all actions belonging to each observation in the mentioned structure. The second part consists of filtering actions from each observation given a mean and variance of the distribution of actions. The resulting set then corresponds to the following:

$$\mathbf{A}_{o,reduced} = \{a \in \mathbf{A}_{o,original} \mid occurrence(a) \geq mean - variance \cdot multiplier\}$$

where A is set of actions, *occurrence* computes number of usage of action a given observation o and original structure, *mean* and *variance* corresponds to statistics of the action sub-dictionaries and *multiplier* is variable argument for tuning the strength of pruning. From this reduce action, we compute the initial memory size for each memory node in FSC. In addition to this process, we recall variances and compute a prioritizer, which is an ordered set of observations given the variance. The observation with the highest variance should increase the memory first, etc.

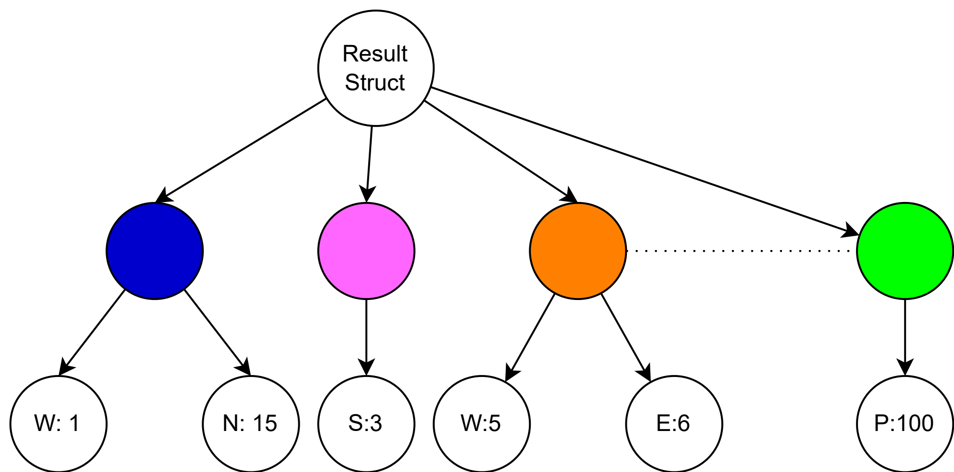


Figure 5.5: Visualisation of example of output structure from Algorithm 6.

Chapter 6

Experiments

In this chapter, we outline multiple experiments that answer various experimental questions. We separately evaluate two main categories: quality of our agents, and performance of combination of PAYNT with hints from reinforcement learning. In Section 6.2, we outline five different research questions related to our proposed approach and evaluate them.

6.1 Experimental Setting and Benchmark Selection

Before we proceed to the experiments themselves, we first describe our experimental setting and briefly introduce our experimental goals and benchmarks that we have been using.

The experiments were carried out locally on a computer with an AMD Ryzen 5 5600 processor, 16 GB of RAM, and a GPU 3070. Within our experiments, we try to explore how our novel encoding method combined with our implemented reinforcement learning approach improves the overall performance in terms of stability and the best attainable performance. We explore how our agents using the DQN, DDQN, and PPO algorithms can solve various benchmarks, and then we explore how these agents can improve the synthesis of PAYNT. First, we explore a combination of PAYNT and a pre-trained agent, and then we perform experiments of loop training of agents and synthesis performed by PAYNT.

Benchmark models were obtained from the repositories of the PAYNT [2] and shielding [17] projects. These models are tailored for tasks such as maze navigation, rock collection on maps, or network packet management. Generally, controllers for these models strive to minimize the rewards collected in the environment[1] or enhance the likelihood of reaching the final state when the achievement of the goal state is not guaranteed. These models were specifically chosen and adjusted to present a challenge to PAYNT in terms of size and complexity, allowing reinforcement learning to exhibit superior scalability compared to PAYNT, when PAYNT cannot explore the design space of k-FSCs with $k > 1$. We present a summary of these models in Table 6.1, with additional details in Appendix A in for models primarily in Table A.6. The most challenging models are distinguished by a significant imbalance between the number of observations and states, particularly in the cases of *refuel* and *grid-large*, where the information about the current state is extremely limited.

Articles from PAYNT [1, 3] also discuss alternative models, typically based on the Cassandra language, which is different from our models' PRISM language. Due to specific implementation aspects, these models are only operable within the PAYNT with RL loop,

[1] This implies reducing the steps required to achieve the objective.

Model Name	Constant Setting	# of Obs	# of Act	# of States	# of Trans
evade	N=7, R=2	4172	7	8108	57570
evade	N=5, R=23	981	7	1961	12905
grid-large	N=30, NDIV=5	37	4	900	7075
intercept	N=7, R=1	2002	6	4705	18049
intercept	N=7, R=2	2598	6	4705	18049
intercept	N=15, R=1	17346	6	100801	399617
mba	*none*	8	5	10	68
mba-small	*none*	8	5	10	68
network	K=20, T=8	2205	5	17253	93128
refuel	N=10	84	6	892	5367
refuel	N=20	174	6	6834	47915
rocks-16	N=16	2761	10	11017	68116

Table 6.1: Summary of used benchmark models. Constant setting refers to situation, when we adjust parameters of used PRISM model to change the size of grid etc. # of Obs refers to number of all possible observations of used POMDP, # of Act to number of maximum possible actions of a given POMDP (models in general work with dynamic action space), # of States describes the number of states of underlying MDP and # of Trans refers to number of transitions of the underlying MDP. The constant N usually refers to size of 2-dimensional grid, R to radius for scanning and in case of network, the K refers to number of time periods and T to number of slots for packets. Additional details are described in Table A.6.

as PAYNT is capable of converting the Cassandra model into the Stormpy model format. Moreover, these models do not contain a straightforward observable variable, and the reward model also differs. Thus, these models are available to experiment with, but because of these limitations, we do not consider them in this chapter.

Remark 6.1.1 (Stochastic PPO Evaluation) *There are two options, how to evaluate PPO agents: greedy and stochastic. We used stochastic evaluation for the PPO algorithm in rocks model. However, within other models, the stochastic evaluation of the PPO algorithm leads to higher penalties and lower probability of reaching the goal state. We explain it with the nature of the task rocks, where we have to collect multiple rocks from the environment and when the greedy evaluation of PPO misses some of them, it cannot return, whereas the other tasks have a significantly different goal condition. Thus, in the following subsections, we work primarily with greedy evaluation of PPO policy. Training with stochastic and greedy evaluation remains the same, as the behavioral policy is always stochastic.*

6.2 Research Questions and Answers

In this section, we try to answer these five different research questions:

- **Q1**: Comparison of Encoding Methods: We proposed a novel encoding technique based on Stormpy valuations, aimed at enabling our reinforcement learning agents to acquire semantic knowledge of the environment. We are interested in determining whether this new method enhances the agents' overall effectiveness compared to traditional techniques that utilize integer observations or one-hot encoding.

- **Q2**: Comparison with Other Implementation: We have created a custom toolkit for reinforcement learning and are interested in evaluating its performance against other existing implementations that target similar models.
- **Q3**: Training of RL on Various Benchmarks: We selected various models and we would like to find whether our approach is successful in training reasonable policies given some number of iterations. There are two important metrics of our training, the stability of learning, and the overall performance of trained agents.
- **Q4**: PAYNT with RL Oracle: We trained multiple agents for given models and we would like to know whether these agents can help with our designed hint system to improve PAYNT performance.
- **Q5**: Closed Loop with RL and PAYNT: Another approach for the distribution of hints is to run RL and PAYNT in closed loop, where both RL and PAYNT give hints to each other as described in Algorithm 3.

These questions cover the main areas which we described in this thesis and we try to answer them properly. However, in addition, we performed some experiments to adjust the learning parameters combined with information from the literature, but these are not included in this thesis.

Q1: Comparison of Encoding Methods

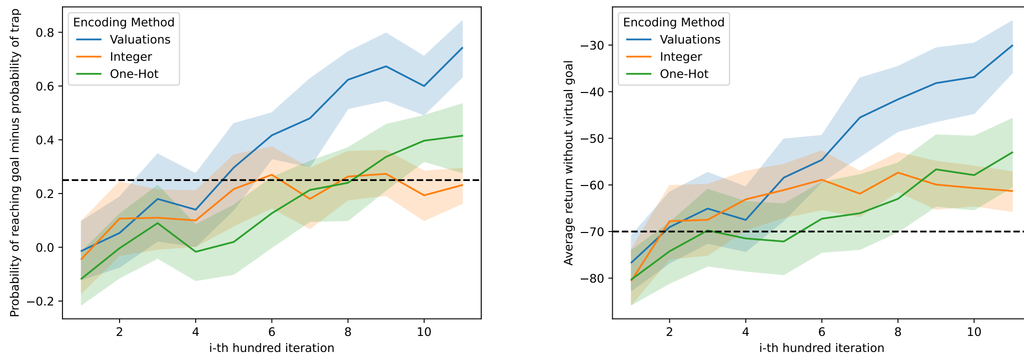
In Section 5.2, we describe three different methods to encode observations. The initial method utilizes a single integer for encoding, while the second employs one-hot encoding of that integer. Additionally, we observed that Storm usually assigns valuations to each observation, which indicate certain observation characteristics. This subsection presents an analysis of the *intercept* task, conducted with parameters $N = 7$ and $RADIUS = 2$ in 15 runs given 1000 iterations of the PPO algorithm, applying these various encoding techniques. We chose this task because we found that we can train satisfying policies with our RL approach and the training is usually more stable compared to other models.

In Figure 6.1 we can see, how often our approaches achieved goal if we subtracted probability of reaching traps[2]. The figure shows that our proposed method based on Stormpy valuation dominates both approaches, as it can learn a reasonable policy in a short period of time, while both other approaches struggle to learn at least a bit reasonable policies, which could overcome balanced dice. For example, the authors of [17] also struggle with integer encoding and usually cannot overcome random policy with their learning algorithms without a time-consuming shielding process.

Figure 6.1 also shows that policies based on one-hot and integer encoding also struggle to minimize the number of steps in the environment (the maximum number of steps is 100) and usually beat the random policy only by a few steps. In contrast, valuation encoding significantly decreases the number of steps in the environment during the entire training period, and if we train policies with this encoding more time, we can improve the policy even more.

Summary: We also compared the encoding methods with other models and the results were usually the same – Stormpy valuations significantly improve the training convergence of the learning algorithms. It leads to faster learning and also usually to more stable learning.

[2] In this case, it is label *exits*.



(a) Probability of reaching final state

(b) Cumulative reward

Figure 6.1: Comparison of multiple training runs with different encoding methods given probability of reaching goal state and cumulative reward given average of 20 episodes each 1000 iterations on *intercept* model. The plots are created from 15 separate training runs with each encoding method. The lines represents medians and shadow areas interquartile ranges. Black line shows approximated performance of random policy, which is usually better than usage of single Integer encoding.

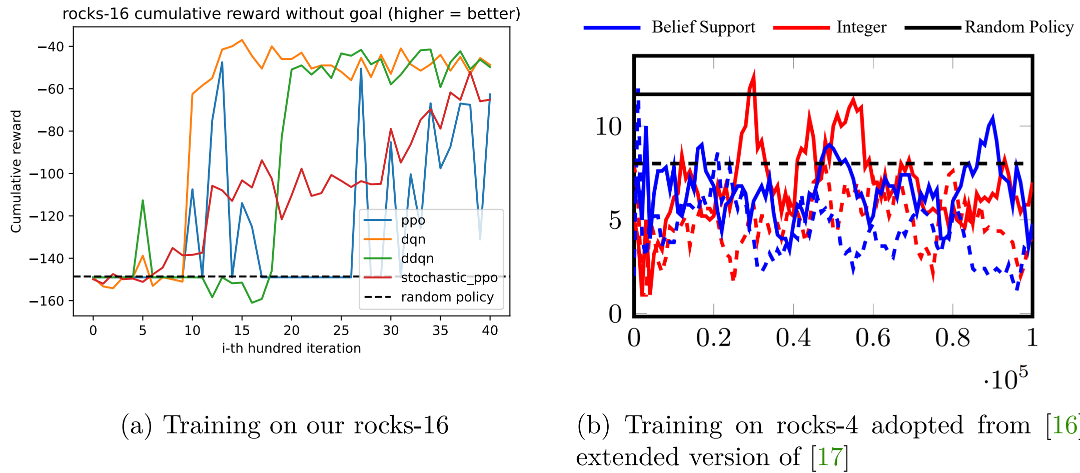
However, some models, mainly in Cassandra, do not include observation valuations, and also some models contain only a few observable valuations (e.g. *obstacle*). Overall, we try to use valuations as much as possible, but some models do not contain them. In these cases, we recommend using one-hot encoding, which usually has slightly more reasonable results than the integer encoding method.

Q2: Our RL Compared to Related RL Approaches

Since we have developed our own implementation of the reinforcement toolkit for selected models represented by the Storm simulator, we should compare it with other implementation. However, the main issue with comparison is that there are not many approaches that solve similar problems. Many existing approaches are tested on benchmarks based on OpenAI Gym, DeepMind puzzles, or some physics simulators[3], which are significantly different from our Stormpy simulator, as it usually depends on frames consisting of various multi-dimensional values, while Stormpy provides only single integers, or valuations, as we mentioned in previous subsection. Given this situation, there exist only few similar approaches to compare with, and one of them is the approach for shielded reinforcement learning [17], where they solve similar tasks with reinforcement learning. This article describes state-of-the-art (2023) implementation for training robust and safe controllers for POMDPs. However, in our context, it is worth mentioning that their approach is primarily focused on their shielding method, and many of their algorithms (PPO, DQN, DDQN, TD3, and REINFORCE) are trained primarily for memory-less policies, as they do not use recurrent layers. Instead, they use for their algorithms as inputs integer observations, or primarily, belief support computed from Stormpy[4].

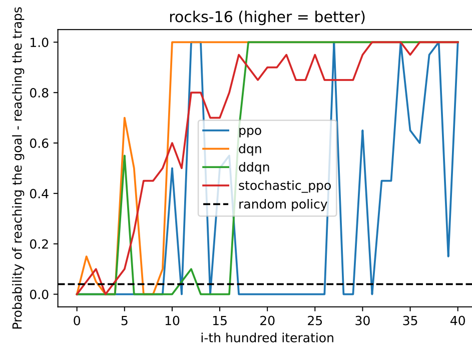
[3]Some of the popular benchmarks are summarized in https://neptune.ai/blog/best-benchmarks-for-reinforcement-learning.

[4]Belief support is vector of states with non-zero probability given current time step.



(a) Training on our rocks-16

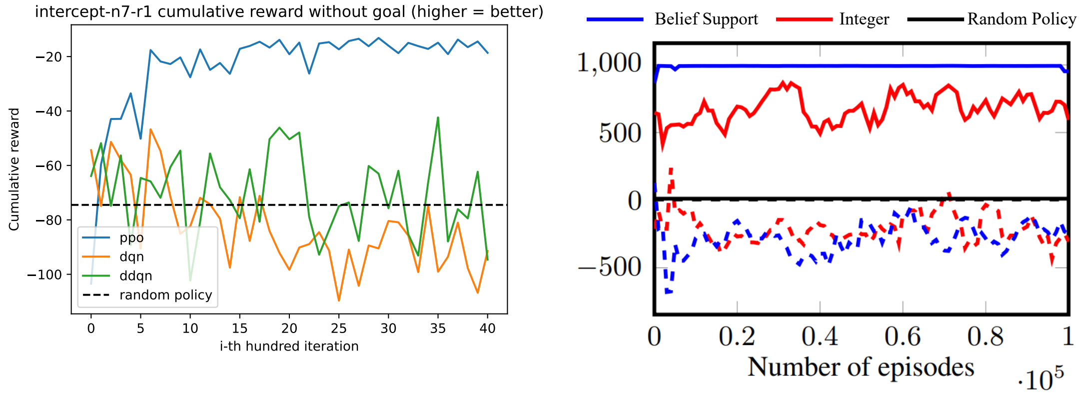
(b) Training on rocks-4 adopted from [16], extended version of [17]



(c) Our rocks-16 with goal reaching probabilities

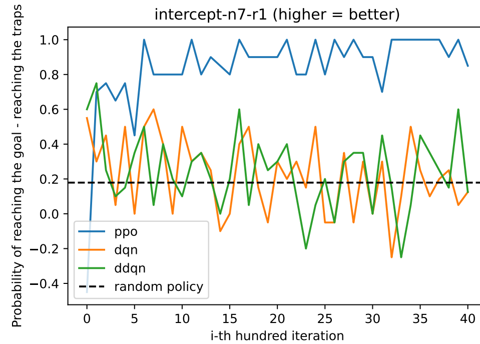
Figure 6.2: Comparison between training on our larger model rocks-16 and adopted experiment on smaller rocks-4 with RL algorithm SAC using LSTMs. The solid and dashed lines in (b) represents option with and without shielding. Subfigures (a) and (b) used cumulative reward evaluations given the used reward, while the Subfigure (c) used evaluation based on the probabilities of reaching the goal. Our implementation in all cases dominates the random policy, while the implementation of [17] is usually worse than the random policy in a much smaller model.

In our case, we try to compare our implementation with their implementation of soft actor-critic (SAC), as it is the only algorithm where they used LSTM layers for memory estimation, and thus it is only agent that can estimate belief purely by RL. However, direct comparison is not possible, as they usually use slightly different models with their own dense reward models developed to improve learning stabilization. That significantly changes the evaluation output[5], and it also rules out the possibility to use their models with our training and our models in their training, because they use a dedicated reward model in their code, while we use the reward models from the provided model specification.



(a) Our training intercept n=7 and r=1

(b) Their training on intercept n=7 and r=1 adopted from [16], extended version of [17]



(c) Our training intercept n=7 and r=1 with objective of reaching goal probability

Figure 6.3: Comparison between our training and training with RL algorithm SAC using LSTMs from [17]. The solid and dashed lines in (b) represents option with and without shielding. Subfigures (a) and (b) used cumulative reward evaluations given the used reward, while the Subfigure (c) used evaluation based on the probabilities of reaching the goal. Our DQN and DDQN provides more unstable results compared to the adopted figure, while PPO dominates all other results except for shielded training with belief support, which has comparable performance.

In their evaluations, they usually compare their agents with random policies, which select the actions in each observation using balanced dice. **This allows us to relatively**

[5] For clarity, within evaluations, we compute average episode reward from reward model formally described in each model without any additional rewards for agents motivation. We also performed a few experiments with reward shaping to create a dense reward, but our experiments led to getting stuck in local optima.

compare the learning curves of our algorithms and theirs without knowing the exact reward values. For this reason, we added a random policy evaluation in the previous question in Figure 6.1 to show how we perform in the case of using valuations instead of integer encoding. In Figure 6.2 we show how our RL algorithms trained on *rocks-16* compared to their training with the smaller model *rocks-4*. The comparison shows that the quality of our learning is significantly better as our learning produces a relatively stable learning curve that improves over time, while the approach [17] does not learn at all, even with shielding usage.

We also found that we can adjust our model intercept to obtain a similar model of *intercept* as they used in their experiments, only with different reward model. We show a comparison of our training with their training in Figure 6.3, where we can see that our implementation of DQN and DDQN agents struggles with this task and is generally very unstable. However, our PPO agent outperforms all other possible agents except for their implementation of shielded policy with state estimation in form of Stormpy belief support, which produces similar results[6].

A major limitation of the shielding method [17] is its time-intensive shield computation process. Before developing our own implementation, we conducted several experiments using both PAYNT and the shielding method with their models. Not only did PAYNT achieve superior results for certain models, but it also managed to develop a viable policy for the scenario of *evade* with the grid size $N = 7$ and radius $R = 2$ faster than the shielding method could generate shields and initiate training. Although an exact time complexity was not determined, the shield calculation for environments characterized by a grid size of $N \times N$ where $N > 10$ proved infeasible.

Summary: The answer to our question about the performance of our RL agents compared to other existing RL approaches is that we created a better implementation in terms of stability, time complexity, and overall performance. However, there are still limitations, and the comparison is limited for technical reasons.

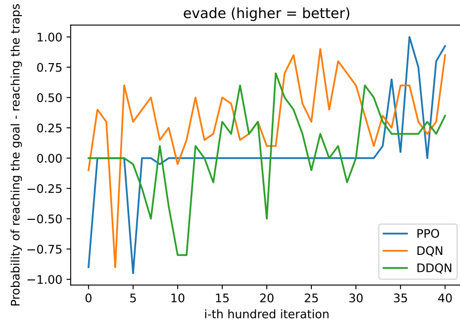
Q3: Training of RL Agents on Selected Benchmarks

In this subsection, we try to answer the question of whether we can train reasonable policies for various models. We would like to use these policies to generate hints for PAYNT, so we expect that the learning produces reasonable policies that can solve tasks given metrics of maximizing reward and maximizing probability of reaching the goal. In addition, we would like to obtain stable learning curves that improve over time.

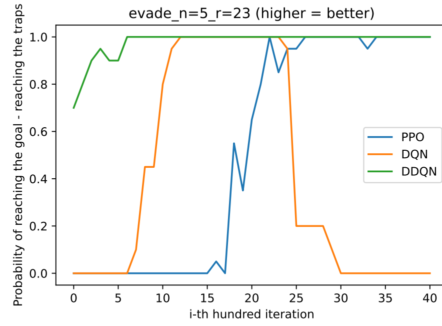
We work with **two different evaluations of policies**. One is based on an evaluation of probabilities for reaching a goal, while the other is based on an evaluation of average cumulative reward. This corresponds, with some differences, to two main specifications, which we use for PAYNT[7]. This separation of two different evaluations more describes the overall behavior of our agents, as in some cases we reach a goal with probability 1.0 and we want to optimize the number of steps to reach. In some other cases, we can optimize the number of steps well for some episodes and at the same time reach the goal only with a low probability.

[6] If we reach goal with probability 1.0, we can add to our average cumulative reward value 1000, because the reward for reaching goal is in their case just 1000.

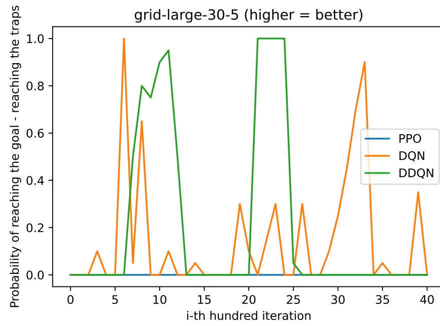
[7] The distinction lies in our probability estimation method, which is $P(goal) - P(trap)$. This approach was selected because the outcome remains unknown if an episode exceeds a certain time limit, and subtracting these probabilities can improve our understanding of the agents' behavior.



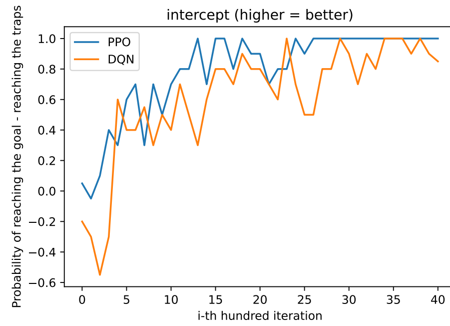
(a) Evade with N=7, R=2



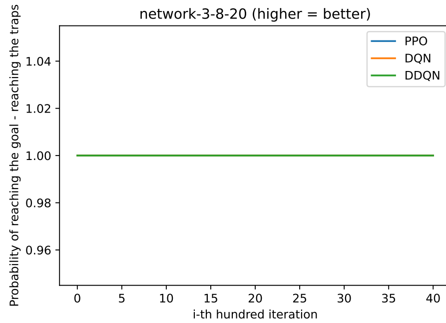
(b) Evade with N=5, R=23



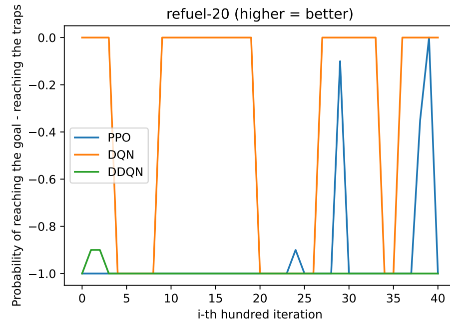
(c) Grid Large



(d) Intercept with N=7, R=2



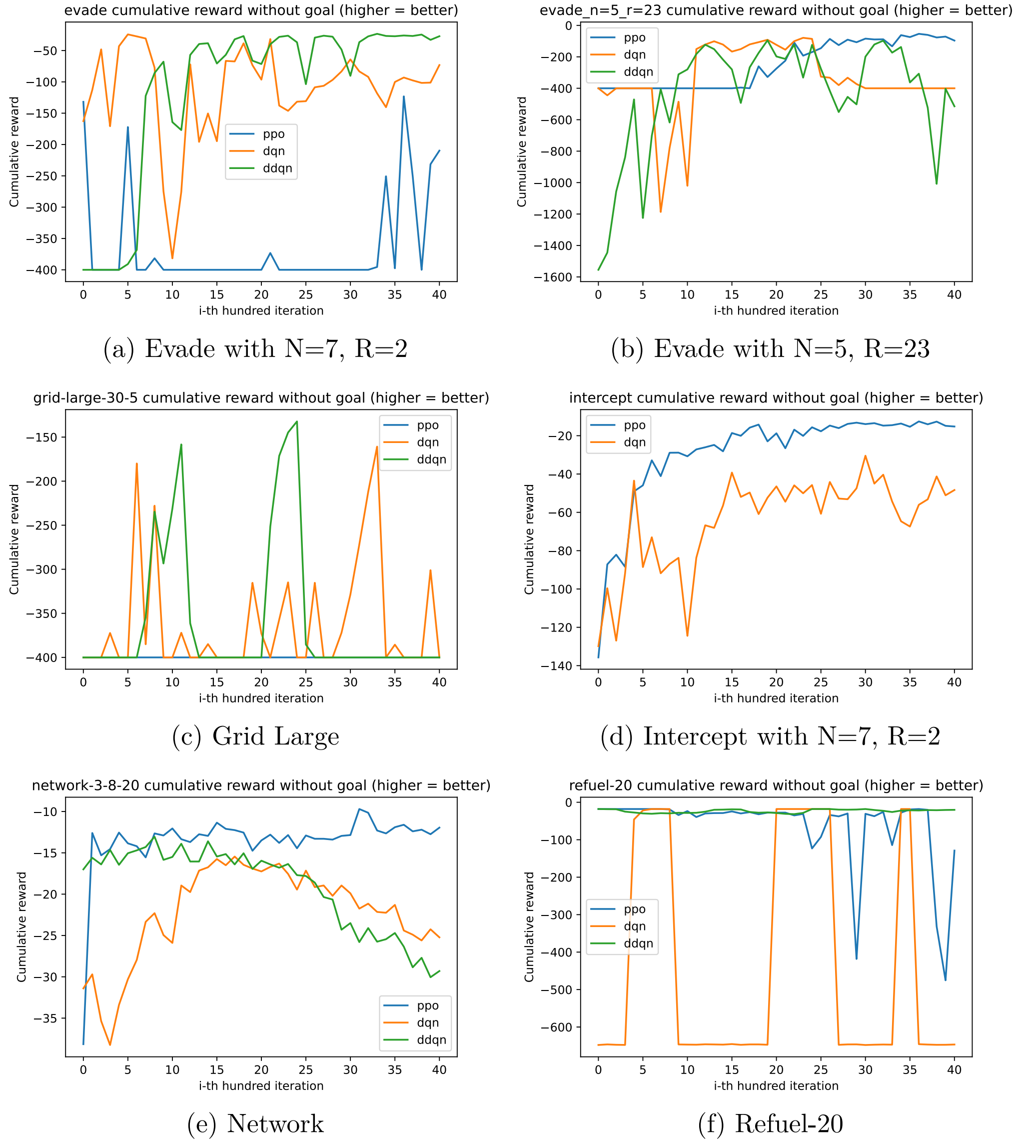
(e) Network



(f) Refuel-20

Figure 6.4: Figures of probability of reaching final state for various models. In case of *network*, the agents achieve the goal with probability 1.0 all the time, while in the case of *refuel-20*, they cannot achieve the goal at all and only loop in the environment or go to the trap.

Figure 6.4, shows the progress of the training with different learning algorithms in terms of probabilities of reaching the goal state. Reasonable convergence curves were achieved for certain algorithms and models, such as PPO and the models *intercept*, *evade* with a large radius, and for every algorithm applied to the model *network*. However, as we can see in the case of *evade* or *grid large*, the stability of learning algorithms varies from really good results to very poor results between couple of iterations. In case of *refuel*, our sole algorithms never reach the goal state, and it is the most challenging environment we have faced.



(a) Evade with N=7, R=2

(b) Evade with N=5, R=23

(c) Grid Large

(d) Intercept with N=7, R=2

(e) Network

(f) Refuel-20

Figure 6.5: Figures of cumulative reward for various models. For some models and algorithms, we can see a stable learning curve, while for the others, the learning is unstable. In case of *refuel-20*, the agent cannot achieve the goal and decides between going through the environment until the max steps, or it runs out of fuel.

Alternatively, our primary objective is to reduce the number of steps or negative rewards received from the environment. As illustrated in Figure 6.5, PPO usually outperforms other

algorithms by learning effective policies. However, being a policy algorithm, it is prone to encountering local optima, as seen with the more challenging scenarios such as *grid large* or the variant of *evade* with a larger environment.

Summary: Broadly speaking, we managed to develop reasonably effective policies for certain environments, though the training process often faces challenges with instability and occasionally fails to produce even moderately suboptimal outcomes for models with sparse rewards.

Q4: PAYNT with RL Oracle

During our reinforcement learning experiments, we obtained a variety of results for each model. We employ various algorithms and interpretation configurations and occasionally execute the learning algorithm multiple times. The objective of this subsection focuses on achieving the maximum enhancement in the optimal value derived from PAYNT, following hints from the pre-trained RL oracle (agent). Table 6.2 compiles all the optimal results of our experiments, conducted with a timeout of 600 seconds, beyond which substantial improvements were generally not noted. In general, we obtained better or similar results with the RL oracle if the agent was well trained, with the exception of the task *rocks*. In this case, we trained a significantly better agent, but the agent did not improve the synthesis at all. We should also note that the models are that large that PAYNT cannot usually explore the whole 1-FSC design space even after a higher time limit.

Model	PAYNT	k-FSC	*Max RL	PAYNT + RL	k-FSC	Goal
evade	0.932	1-FSC	**1.0**	0.964	1-FSC	Pmax
evade-5-23	17	1-FSC	53.6	17	1-FSC	Rmin
grid-large	nan	1-FSC	160.0	**118.734**	3-FSC	Rmin
intercept n=7, r=2	0.999	1-FSC	**1.0**	0.979	1-FSC	Pmax
intercept n=15, r=1	0.0	1-FSC	0.65	**0.836**	2-FSC	Pmax
network-3-8-20	11.066	1-FSC	**9.7**	10.694	4-FSC	Rmin
refuel-20	0.004	1-FSC	0.0	**0.214**	1-FSC	Pmax
mba	6.265	3-FSC	22.975	6.265	3-FSC	Rmin
mba-small	4.598	3-FSC	92.75	4.598	3-FSC	Rmin
rocks-16	46.0	1-FSC	**37.0**	46.0	1-FSC	Rmin

Table 6.2: Summary of experiments with hints from pre-trained oracles obtained by various approaches. The states and actions represent the size of the constructed quotient MDP, *Pmax* indicates probability reaching final state and *Rmin* minimization of cumulative reward. PAYNT was used with option *–fsc-synthesis*, which may be outperformed by other options. In case of *grid-large*, sole PAYNT could not find any controller, that achieves the goal state with probability of 1. *Maximum performance of RL algorithm is obtained from evaluation of 20 episodes each 100 iterations of training.

We can see that in some tasks we obtained some small improvements, for example in case of *evade* and *network-3-8-20*. Within two tasks, *grid-large* and *refuel-20*, we obtained a more significant improvement, but in the first case it was caused primarily by aggressive memory increase and in case of *refuel-20*, the RL agent cannot achieve the goal but gets stuck in local optima of the refuel station. We suppose that PAYNT can use this information to achieve the first gas station and derive the rest of the task. In addition, we conducted experiments on a significantly expanded version of the *intercept* task. Here,

PAYNT failed to identify viable solutions, achieving an optimal probability of 0.0. In this scenario, our reinforcement learning algorithm served as a catalyst, initiating the search for viable solutions and resulting in a significant improvement of the optimal value. This is one of the ideal cases for improving the RL oracle. However, the main issue with larger models is that the PAYNT has a problem with construction of the model if we increase the size significantly and cannot start to use our hints to prune the design space. In contrast, our RL implementation usually has similar performance for various sizes of grid in terms of computation time given with the same type of model.

Ablation Study of RL Oracle

The primary concern of the previous table is the synthesis of optimal results derived from extensive experiments, which utilize diverse configurations for interpretation and RL hints to PAYNT. Table 6.3 illustrates our engagement with several different approaches. We employ three different algorithms for agent training: DQN, DDQN, and PPO. Occasionally, we obtained the best results with the DQN and DDQN algorithms based on critics, capable of developing more aggressive and less stable exploration strategies. Consequently, DQN was identified as the most effective method for the grid-large scenario, as the PPO algorithm failed to achieve the goal state and got stuck in local optima, whereas DQN succeeded in quickly identifying and retaining a viable policy.

Model	Algorithm	Refusing	Initial RL Mem.	Pruning	Agent
evade	DQN	True	False	False	Last
evade-5-23	DQN, DDQN	False	False	False	Last
grid-large	DQN	False	True	Both	Best
intercept n=7, r=2	PPO	False	False	False	Last
intercept n=15, r=1	PPO	Any	True	False	Any
network-3-8-20	PPO	False	True	True	Last
refuel-20	DDQN	False	False	False	Best
mba	PPO	True	False	False	Best
mba-small	PPO	True	False	False	Last
rocks	All	Both	Both	False	Any

Table 6.3: List of used hints for each tasks to obtain the best results. It describes best combination of benchmarked model, training RL algorithm, interpretation option for refusing wrong episodes, setting initial memory defined by RL oracle, PAYNT action pruning/prioritizing and the time of selection of agent – the best trained agent or the agent after whole training period.

The second option, refusing, modifies the interpretation process as described in Algorithm 6, where setting refusing to *True* excludes episodes that do not reach the goal state. Initial memory sets the starting memory for the PAYNT from RL advice, while pruning determines whether to reduce (prune) the design space or prioritize some observation-action pairs. The final parameter, *agent*, specifies whether to use the agent from the last training cycle or the agent that performed the best during the evaluations.

In general, we found that different tasks prefer different configuration. Only parameter that was the most preferable was pruning set to off, as our algorithms usually prune the design space that much, that we disable the ability to learn at least sub-optimal solutions.

Only in the case of *network* model, we found that pruning can achieve slightly better results, as our policy provided multiple actions for different observations.

Question Summary

Summary: We found that we can significantly improve the resulting PAYNT-synthesized policy if we trained a proper agent and PAYNT could start solving the task. However, there are still some limitations with memory estimation, where only in a few cases did memory initialization with hints improve the overall synthesis. In general, our RL approach can improve the synthesis of FSCs but still has some limitations, mostly in the ability to train reasonable policies for complex models, for example, for *refuel*. Moreover, there exist multiple options on how to interpret the agents and provide hints and there is not any dominant one.

Q5: Closed Loop with RL and PAYNT

An alternative method for the distribution of the hints involves integrating the RL oracle into the FSC synthesis process, as shown in Figure 4.1. This section details the outcomes of training the PPO reinforcement learning algorithm in conjunction with PAYNT synthesized FSCs, employing two distinct strategies. For the hard FSC approach, the agent is trained over 100 episodes using only data derived from FSC sampling. In the soft FSC approach, the trajectory sampling from the environment is utilized as outlined in Subsection 4.5.2. The training cycle is described in Algorithm 3.

Within our experiments, summarized in Table 6.4, we found that this symbiosis of both approaches can, with some drawbacks, reduce the limitations of both approaches. For example, in the case of *refuel-20*, it was the first time that we reached the goal state with our RL algorithms. However, training suffers from great instability and sometimes we experience significantly different results. For example, in the case of *refuel-20*, we can obtain much better results with more trained agents and interpretation results. Moreover, we can use only a single agent trained during these experiments, and it usually suffers from the worse synthesized first FSC from the first advice. For this reason, we adjusted the *refuel* to smaller grid, where our learning is more stable. In this task, we found that the agent can overcome PAYNT-synthesized FSCs after 6 or 7 loop cycles and then significantly improve the PAYNT procedure.

Model	Pure PAYNT	Pure RL	Hard FSC	Soft FSC	Goal
refuel-10	0.0774	0.0	0.3512	0.3374	Pmax
refuel-20	0.00083	0.0	0.0101	0.0158	Pmax
network-3-8-20	11.129	9.7	13.2471	14.6835	Rmin
rocks-16	46.0	37.0	47.0	46.0	Rmin
evade	0.932	1.0	0.921	0.932	Pmax
intercept n=15, r=1	0.0	0.65	0.537	0.427	Pmin

Table 6.4: Results of FSC cycling with PPO algorithm, when hard FSC samples data from the environment itself, while soft FSC combines the sampling policy with PPO logits. Original PAYNT results are produced with option *fsc-synthesis*. Different options can have better results. Pure RL results represents best obtained result during training of multiple RL agents for mentioned model outside of the loop from Section 6.2.

One of the largest advantages over the previous mentioned approach without the loop is that we use only a single implementation without refusing episodes, and we use only the last trained agent. Another advantage is that with time limitation of synthesis of FSC, we can continuously increase memory for observations, where it is needed. However, this approach has many drawbacks. The most significant one is the tendency to overfit the RL agent to the policy represented by FSC. We tried to reduce this issue with our soft FSC implementation, but it has slightly similar issues. This led to worse results with the task *network*, where the RL algorithm started to converge to the optimal solution, but the synthesis slowed the training, as the algorithm was fitted to data from an equally good FSC solution. The last but not least issue of this approach is general instability of training algorithms, which leads to significantly different results if the experiments are rerun with the same parameters. However, we found that the training instability is reduced compared to the independent training of RL and PAYNT.

Another difference between the previous approach and the PAYNT-RL loop is that in the first case the RL agent learned to avoid wrong sink states, while in the second approach, PAYNT without adjusted specification does not care about other states than the goal one. That opens a wide area of potential research for adjusting the specifications of both approaches, where we could change the specification of PAYNT to provide more safe policies and then focus on different, the main one.

Summary: In general, this approach shows a lot of potential, but still needs some further research. Both PAYNT and RL can help each other, but in general, the improvements are worse than in the case of RL hints outside the loop since we cannot train in a shorter period of time better RL policies.

6.3 Summary

In this chapter, we discuss various research questions. The first focused on the comparison of three distinct encoding methods for reinforcement learning, where we show that our novel valuation encoding based on data provided by Storm dominates both other standard approaches in terms of stability, performance, and the ability to learn semantics. Moreover, if we compare our results with the baseline RL results obtained in [17] with LSTM networks, we found that we can solve significantly larger tasks without the need for outer help, such as shielding or belief support instead of integer inputs.

We also describe our experiments with training various reinforcement learning algorithms on multiple tasks, followed by experiments with combinations of various interpretation outputs and PAYNT. We found that we can improve finite-state controller synthesis with well-trained agents, but the general issue is the instability of training process and limited abilities of learning quality policies by state-of-the-art approaches. We also found that our interpretation method usually leads to similar results as provided by the agents themselves, if we use it as advice to PAYNT.

The last research question focused on the combination of reinforcement learning and PAYNT in loop. We found that we can improve the learning of our agents in environments with sparse reward, as the FSC usually finds at least some solution. Moreover, the learning procedure is usually much more stable than in the case of independent training of both approaches. However, the main disadvantage of this approach is the usual stabilization of learning of both approaches, as the reinforcement learning usually adapts to the policy provided by the FSC from PAYNT and PAYNT usually adapts to the policy provided by reinforcement learning.

Chapter 7

Conclusion

In this thesis, we develop and implement a novel approach for merging two different solutions for making sequential decisions under uncertainty under the framework of a partially observable Markov decision process (POMDP). The first, a complete model-based system, is represented by PAYNT, while the second, an approximate and model-free system, is represented by the reinforcement learning framework. We outline the primary challenges associated with both approaches, review some methods to face these challenges, and discuss how our strategy addresses them. Furthermore, after identifying substantial shortcomings in the current state-of-the-art methods represented by [17], we created a new reinforcement learning solution from the ground up, utilizing the TensorFlow Agents [67] framework.

In addition, we introduced a novel method for encoding observations using data from the Stormpy [56] simulator, a novel strategy for policy wrapping to manage dynamic action spaces, and a straightforward weight-resetting algorithm to improve the early stages of learning in reinforcement learning algorithms. We also designed a novel method for interpreting policies based on recurrent neural networks by analyzing trajectories within the environment. Furthermore, we introduced advanced hard and soft FSC techniques to improve reinforcement learning with PAYNT hints, aiming for improved outcomes in tasks with sparse reward. The principal contribution of this thesis is a toolkit that enables the training of policies using reinforcement learning algorithms like DQN, DDQN, and PPO, their interpretation to create hints, the provision of hints to PAYNT, the extraction of finite-state controllers (FSCs) and their integration into reinforcement learning. Our toolkit can perform this sequence in a single run or in a complex training loop.

7.1 Future Research

In this thesis, the main bottleneck of our presented approach is the limitation of reinforcement learning in terms of stability, problematic behavior in sparse reward models, difficult parameter configuration and non-existing scalable explainability. Even with our improvements and overcoming current state-of-the-art approaches, we sometimes could not overcome the results provided by the formal-based PAYNT.

One of the important paths that further work should explore is the selection of learning parameters. In this thesis, we selected learning arguments inspired by the existing literature [5, 17, 18, 43] and some parameters were adjusted by multiple experiments. However, the number of arguments of our training algorithm, interpretation, environment, PAYNT-RL loop etc. come up to a higher order of tens. Further research could perform more

experiments manually or develop some automated algorithm for argument selection based on, for example, evolutionary algorithms. This could possibly improve stability or decrease problems with sparse reward settings.

In this thesis, we achieved the main improvement in reinforcement learning with novel observation encoding using Stormpy values. However, some models do not contain many (or any) observable valuations and some models contain only a few valuations that do not tell anything about the environment. Further research could focus on approaches, how to increase the ability of agents to create reasonable encoding for each observation. Moreover, further research could use information about state valuations and use it to add semantic meta-information about the environment, because, for example, in the case of task *refuel-20*, the agent does not have the information that it operates on the grid.

Another possible approach is the adjustment of the selected reinforcement learning algorithms. In this thesis, we worked with implementations of DQN, DDQN and PPO from TensorFlow Agents [67], but in some cases, we use them differently to achieve better results. For example, we found that using replay buffers with PPO is more beneficial than using it in the original sense with on-policy learning based only on current experience. Moreover, there exist many other algorithms, such as TD3, SAC, DDPG etc., which may provide, combined with our novel methods, somewhat better training results. Another option, which we briefly introduced and experimented with, is reward shaping, which is one of the disciplines that leads to creating state-of-the-art solutions based on reinforcement learning.

The last option, which we would like to outline, is the usage of formal methods for explaining recurrent policies. In this thesis, we briefly introduce the existing approach for the explainability of slightly modified LSTM based on Layer-Wise Relevance Propagation [6], which could be used for exact mapping between LSTM-based policy and the synthesis of finite-state controllers performed by PAYNT. Moreover, there exist various approaches for formal verification of neural networks, such as Crown [72, 77, 78], VeriNet [34, 35] or MN-BaB [25], that could support verification of recurrent neural networks in the near future. These approaches could be used, with some reasonable specification, to interpret exactly the neural network-based policies.

Bibliography

[1] Andriushchenko, R., Alexander, B., Češka, M., Junges, S., Katoen, J.-P. et al. Search and Explore: Symbiotic Policy Synthesis in POMDPs. In: *Computer Aided Verification*. Springer Verlag, 2023, vol. 13966, p. 113–135. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). DOI: 10.1007/978-3-031-37709-9_6. ISBN 978-3-031-37708-2.

[2] Andriushchenko, R., Češka, M., Junges, S., Katoen, J.-P. and Stupinský, Š. PAYNT: A Tool for Inductive Synthesis of Probabilistic Programs. In: Silva, A. and Leino, K. R. M., ed. *Computer Aided Verification*. Cham: Springer International Publishing, 2021, p. 856–869. ISBN 978-3-030-81685-8.

[3] Andriushchenko, R., Češka, M., Junges, S. and Katoen, J.-P. Inductive Synthesis of Finite-State Controllers for POMDPs. In: *Conference on Uncertainty in Artificial Intelligence*. Proceedings of Machine Learning Research, 2022, vol. 180, no. 180, p. 85–95. Proceedings of Machine Learning Research. ISSN 2640-3498.

[4] Antonoglou, I., Schrittwieser, J., Ozair, S., Hubert, T. K. and Silver, D. Planning in Stochastic Environments with a Learned Model. In: *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022.

[5] Arcieri, G., Hoelzl, C., Schwery, O., Straub, D., Papakonstantinou, K. G. et al. POMDP inference and robust solution via deep reinforcement learning: An application to railway optimal maintenance. *CoRR*. 2023, abs/2307.08082. DOI: 10.48550/ARXIV.2307.08082.

[6] Arras, L., Arjona-Medina, J. A., Widrich, M., Montavon, G., Gillhofer, M. et al. Explaining and Interpreting LSTMs. In: Samek, W., Montavon, G., Vedaldi, A., Hansen, L. K. and Müller, K., ed. *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning*. Springer, 2019, vol. 11700, p. 211–238. Lecture Notes in Computer Science. DOI: 10.1007/978-3-030-28954-6_11.

[7] Badnava, B., Esmaeili, M., Mozayani, N. and Zarkesh-Ha, P. A new Potential-Based Reward Shaping for Reinforcement Learning Agent. In: *13th IEEE Annual Computing and Communication Workshop and Conference, CCWC 2023, Las Vegas, NV, USA, March 8-11, 2023*. IEEE, 2023, p. 1–6. DOI: 10.1109/CCWC57344.2023.10099211.

[8] Ballestar, M. T., Grau Carles, P. and Sainz, J. Predicting customer quality in e-commerce social networks: a machine learning approach. *Review of Managerial*

Science. June 2019, vol. 13, no. 3, p. 589–603. DOI: 10.1007/s11846-018-0316-x. ISSN 1863-6691.

[9] BAUDER, M., PAULA, D., KUBJATKO, T. and SCHWEIGER, H.-G. Evaluation of the vehicle behaviour when not responding to the take-over request of Tesla Autopilot and Volkswagen Travel Assist. *Transportation Research Procedia*. 2023, vol. 74, p. 450–457. DOI: https://doi.org/10.1016/j.trpro.2023.11.167. ISSN 2352-1465. TRANSCOM 2023: 15th International Scientific Conference on Sustainable, Modern and Safe Transport.

[10] BHATTACHARYA, S., BADYAL, S., WHEELER, T., GIL, S. and BERTSEKAS, D. Reinforcement Learning for POMDP: Partitioned Rollout and Policy Iteration With Application to Autonomous Sequential Repair Problems. *IEEE Robotics and Automation Letters*. march 2020, PP, p. 1–1. DOI: 10.1109/LRA.2020.2978451.

[11] BISHOP, C. M. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN 0387310738.

[12] BORK, A., KATOEN, J.-P. and QUATMANN, T. Under-Approximating Expected Total Rewards in POMDPs. In: FISMAN, D. and ROSU, G., ed. *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2022, p. 22–40. ISBN 978-3-030-99527-0.

[13] BRABAZON, A., O'NEILL, M. and MCGARRAGHY, S. *Natural Computing Algorithms*. 1stth ed. Springer Publishing Company, Incorporated, 2015. ISBN 3662436302.

[14] BRANQUINHO, A. A. B., LOPES, C. R. and BAFFA, A. C. E. Probabilistic Planning for Multiple Stocks of Financial Markets. In: *28th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2016, San Jose, CA, USA, November 6-8, 2016*. IEEE Computer Society, 2016, p. 501–508. DOI: 10.1109/ICTAI.2016.0083.

[15] BURKOV, A. *The Hundred-Page Machine Learning Book*. Andriy Burkov, 2019. ISBN 9781999579517.

[16] CARR, S., JANSEN, N., JUNGES, S. and TOPCU, U. Safe Reinforcement Learning via Shielding under Partial Observability. *CoRR*. arXiv. 2022.

[17] CARR, S., JANSEN, N., JUNGES, S. and TOPCU, U. Safe Reinforcement Learning via Shielding under Partial Observability. In: *Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence and Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence and Thirteenth Symposium on Educational Advances in Artificial Intelligence*. AAAI Press, 2023. AAAI'23/IAAI'23/EAAI'23. DOI: 10.1609/aaai.v37i12.26723. ISBN 978-1-57735-880-0.

[18] CARR, S., JANSEN, N. and TOPCU, U. Task-Aware Verifiable RNN-Based Policies for Partially Observable Markov Decision Processes. *J. Artif. Int. Res.* El Segundo, CA, USA: AI Access Foundation. jan 2022, vol. 72, p. 819–847. DOI: 10.1613/jair.1.12963. ISSN 1076-9757.

[19] DAFARRA, S., BERTRAND, S., GRIFFIN, R. J., METTA, G., PUCCI, D. et al. Non-Linear Trajectory Optimization for Large Step-Ups: Application to the Humanoid Robot Atlas. In: *IEEE/RSJ International Conference on Intelligent*

Robots and Systems, IROS 2020, Las Vegas, NV, USA, October 24, 2020 - January 24, 2021. IEEE, 2020, p. 3884–3891. DOI: 10.1109/IROS45743.2020.9341587.

[20] Delalleau, O., Peter, M., Alonso, E. and Logut, A. Discrete and Continuous Action Representation for Practical RL in Video Games. *ArXiv e-prints*. december 2019, p. arXiv:1912.11077. DOI: 10.48550/arXiv.1912.11077.

[21] Deshpande, S. V., Harikrishnan, R., Sampe, J. and Patwa, A. An algorithm to create model file for Partially Observable Markov Decision Process for mobile robot path planning. *MethodsX*. 2024, vol. 12, p. 102552. DOI: https://doi.org/10.1016/j.mex.2024.102552. ISSN 2215-0161.

[22] Dwivedi, R., Dave, D., Naik, H., Singhal, S., Omer, R. et al. Explainable AI (XAI): Core Ideas, Techniques, and Solutions. *ACM Comput. Surv.* New York, NY, USA: Association for Computing Machinery. jan 2023, vol. 55, no. 9. DOI: 10.1145/3561048. ISSN 0360-0300.

[23] Esslinger, K., Platt, R. and Amato, C. Deep Transformer Q-Networks for Partially Observable Reinforcement Learning. *CoRR*. 2022, abs/2206.01078. DOI: 10.48550/ARXIV.2206.01078.

[24] Eysenbach, B., Salakhutdinov, R. R. and Levine, S. Search on the Replay Buffer: Bridging Planning and Reinforcement Learning. In: Wallach, H., Larochelle, H., Beygelzimer, A., Alché Buc, F. d', Fox, E. et al., ed. *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2019, vol. 32.

[25] Ferrari, C., Mueller, M. N., Jovanović, N. and Vechev, M. Complete Verification via Multi-Neuron Relaxation Guided Branch-and-Bound. In: *International Conference on Learning Representations*. 2022.

[26] Fujimoto, S., Hoof, H. van and Meger, D. Addressing Function Approximation Error in Actor-Critic Methods. In: Dy, J. G. and Krause, A., ed. *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*. PMLR, 2018, vol. 80, p. 1582–1591. Proceedings of Machine Learning Research.

[27] García, J. and Fernández, F. A Comprehensive Survey on Safe Reinforcement Learning. *J. Mach. Learn. Res.* JMLR.org. jan 2015, vol. 16, no. 1, p. 1437–1480. ISSN 1532-4435.

[28] Gemini Team, Anil, R., Borgeaud, S., Wu, Y., Alayrac, J.-B. et al. Gemini: A Family of Highly Capable Multimodal Models. *CoRR*. 2023, abs/2312.11805.

[29] GitHub, Inc. *GitHub Copilot*. 2024. Available at: https://github.com/features/copilot.

[30] Goodfellow, I. J., Bengio, Y. and Courville, A. *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016. ISBN 978-0262035613. http://www.deeplearningbook.org.

[31] Google DeepMind. *Gemini*. 2024. Available at: https://deepmind.google/technologies/gemini.

[32] Hasselt, H. v., Guez, A. and Silver, D. Deep reinforcement learning with double Q-Learning. In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. AAAI Press, 2016, p. 2094–2100. AAAI'16.

[33] Hausknecht, M. J. and Stone, P. Deep Recurrent Q-Learning for Partially Observable MDPs. In: *2015 AAAI Fall Symposia, Arlington, Virginia, USA, November 12-14, 2015*. AAAI Press, 2015, p. 29–37.

[34] Henriksen, P. and Lomuscio, A. DEEPSPLIT: An Efficient Splitting Method for Neural Network Verification via Indirect Effect Analysis. In: Zhou, Z.-H., ed. *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*. International Joint Conferences on Artificial Intelligence Organization, August 2021, p. 2549–2555. DOI: 10.24963/ijcai.2021/351. Main Track.

[35] Henriksen, P. and Lomuscio, A. Robust Training of Neural Networks against Bias Field Perturbations. *Proceedings of the AAAI Conference on Artificial Intelligence*. Jun. 2023, vol. 37, no. 12, p. 14865–14873. DOI: 10.1609/aaai.v37i12.26736.

[36] Hensel, C., Junges, S., Katoen, J.-P., Quatmann, T. and Volk, M. The probabilistic model checker Storm. *International Journal on Software Tools for Technology Transfer*. 2022, vol. 24, no. 4, p. 589–610. DOI: 10.1007/s10009-021-00633-z. ISSN 1433-2787.

[37] Hensel, C., Junges, S., Katoen, J.-P., Quatmann, T. and Volk, M. The probabilistic model checker Storm. *International Journal on Software Tools for Technology Transfer*. Aug 2022, vol. 24, no. 4, p. 589–610. DOI: 10.1007/s10009-021-00633-z. ISSN 1433-2787.

[38] Imanian, A., Zhang, S. and Fan, C. Safe and Scalable Collision Avoidance Model for Small Unmanned Aircraft Systems: An Artificial Intelligence Approach. In: *AIAA SCITECH 2024 Forum*. DOI: 10.2514/6.2024-1081.

[39] Kalliamvakou, E. *A developer's second brain: Reducing complexity through partnership with AI*. 2024. Available at: https://github.blog/2024-01-17-a-developers-second-brain-reducing-complexity-through-partnership-with-ai/.

[40] Kenny, E. M., Tucker, M. and Shah, J. Towards Interpretable Deep Reinforcement Learning with Human-Friendly Prototypes. In: *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.

[41] Kesiraju, S., Beneš, K., Tikhonov, M. and Černocký, J. BUT Systems for IWSLT 2023 Marathi - Hindi Low Resource Speech Translation Task. In: *Proceedings of the 20th International Conference on Spoken Language Translation (IWSLT 2023)*. Toronto (in-person and online): Association for Computational Linguistics, 2023, p. 227–234. DOI: 10.18653/v1/2023.iwslt-1.19.

[42] Kochenderfer, M. J. *Decision Making Under Uncertainty: Theory and Application*. The MIT Press, july 2015. ISBN 9780262331708.

[43] Kochenderfer, M. J., Wheeler, T. A. and Wray, K. H. *Algorithms for Decision Making*. MIT Press, 2022. ISBN 978-0262047012.

[44] Kwiatkowska, M., Norman, G. and Parker, D. PRISM 4.0: Verification of Probabilistic Real-time Systems. In: Gopalakrishnan, G. and Qadeer, S., ed. *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*. Springer, 2011, vol. 6806, p. 585–591. LNCS.

[45] Lecun, Y., Bottou, L., Bengio, Y. and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*. 1998, vol. 86, no. 11, p. 2278–2324. DOI: 10.1109/5.726791.

[46] Leucker, M. Formal Verification of Neural Networks? In: Carvalho, G. and Stolz, V., ed. *Formal Methods: Foundations and Applications - 23rd Brazilian Symposium, SBMF 2020, Ouro Preto, Brazil, November 25-27, 2020, Proceedings*. Springer, 2020, vol. 12475, p. 3–7. Lecture Notes in Computer Science. DOI: 10.1007/978-3-030-63882-5_1.

[47] Li, M., Lv, T., Cui, L., Lu, Y., Florêncio, D. A. F. et al. TrOCR: Transformer-based Optical Character Recognition with Pre-trained Models. *CoRR*. 2021, abs/2109.10282.

[48] Macák, F. *Improving Synthesis of Finite State Controllers for POMDPs Using Belief Space Approximation*. Brno, 2023. Master's thesis. Brno University of Technology, Faculty of Information Technology.

[49] Manzanas Lopez, D., Johnson, T. T., Bak, S., Tran, H.-D. and Hobbs, K. L. Evaluation of Neural Network Verification Methods for Air-to-Air Collision Avoidance. *Journal of Air Transportation*. 2023, vol. 31, no. 1, p. 1–17. DOI: 10.2514/1.D0255.

[50] Memarian, F., Goo, W., Lioutikov, R., Topcu, U. and Niekum, S. Self-Supervised Online Reward Shaping in Sparse-Reward Environments. *CoRR*. 2021, abs/2103.04529.

[51] Mitchell, T. *Machine Learning*. McGraw-Hill Education, 1997. McGraw-Hill Series in Computer Science. ISBN 978-0-07-042807-2.

[52] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J. et al. Human-level control through deep reinforcement learning. *Nature*. 2015, vol. 518, no. 7540, p. 529–533. DOI: 10.1038/nature14236. ISSN 1476-4687.

[53] OpenAI, :, Achiam, J., Adler, S., Agarwal, S. et al. *GPT-4 Technical Report*. 2023.

[54] Qiu, W. and Zhu, H. Programmatic Reinforcement Learning without Oracles. In: *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022.

[55] Ray, P. P. ChatGPT: A comprehensive review on background, applications, key challenges, bias, ethics, limitations and future scope. *Internet of Things and Cyber-Physical Systems*. 2023, vol. 3, p. 121–154. DOI: https://doi.org/10.1016/j.iotcps.2023.04.003. ISSN 2667-3452.

[56] RWTH Aachen. *Stormpy Documentation*. 2024. Available at: https://moves-rwth.github.io/stormpy/.

[57] SAMMUT, C. and WEBB, G. I. *Encyclopedia of Machine Learning.* 1stth ed. Springer Publishing Company, Incorporated, 2011. ISBN 0387307680.

[58] SCHAUL, T., QUAN, J., ANTONOGLOU, I. and SILVER, D. Prioritized Experience Replay. In: BENGIO, Y. and LECUN, Y., ed. *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings.* 2016.

[59] SCHMIDT, R. M. *Recurrent Neural Networks (RNNs): A gentle Introduction and Overview.* 2019.

[60] SCHULMAN, J., LEVINE, S., ABBEEL, P., JORDAN, M. and MORITZ, P. Trust Region Policy Optimization. In: BACH, F. and BLEI, D., ed. *Proceedings of the 32nd International Conference on Machine Learning.* Lille, France: PMLR, 07–09 Jul 2015, vol. 37, p. 1889–1897. Proceedings of Machine Learning Research.

[61] SCHULMAN, J., WOLSKI, F., DHARIWAL, P., RADFORD, A. and KLIMOV, O. Proximal Policy Optimization Algorithms. *CoRR.* 2017, abs/1707.06347.

[62] SHABADI, G., FIJALKOW, N. and MATRICON, T. Theoretical foundations for programmatic reinforcement learning. *CoRR.* 2024, abs/2402.11650. DOI: 10.48550/ARXIV.2402.11650.

[63] SHERSTINSKY, A. Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) network. *Physica D: Nonlinear Phenomena.* march 2020, vol. 404, p. 132306. DOI: 10.1016/j.physd.2019.132306.

[64] SIMÃO, T. D., SUILEN, M. and JANSEN, N. Safe Policy Improvement for POMDPs via Finite-State Controllers. In: *Proceedings of the AAAI Conference on Artificial Intelligence.* AAAI Press, 2023, p. 15109–15117.

[65] SKINNER, G. and WALMSLEY, T. Artificial Intelligence and Deep Learning in Video Games A Brief Review. In: *2019 IEEE 4th International Conference on Computer and Communication Systems (ICCCS).* IEEE, 2019, p. 404–408. DOI: 10.1109/CCOMS.2019.8821783.

[66] SUTTON, R. S. and BARTO, A. G. *Reinforcement Learning: An Introduction.* Secondth ed. The MIT Press, 2018. ISBN 978-0262039246.

[67] TENSORFLOW AGENTS CONTRIBUTORS. *TensorFlow Agents: Scalable Reinforcement Learning Library for TensorFlow.* 2024. Available at: https://www.tensorflow.org/agents.

[68] TESLA INC. *Autopilot.* 2024. Available at: https://www.tesla.com/support/autopilot.

[69] TSOUKALAS, A., ALBERTSON, T. and TAGKOPOULOS, I. From data to optimal decision making: a data-driven, probabilistic machine learning approach to decision support for patients with sepsis. *JMIR medical informatics.* JMIR Publications Inc., Toronto, Canada. 2015, vol. 3, no. 1, p. e11. DOI: 10.2196/medinform.3445. ISSN 2291-9694.

[70] Verma, A., Murali, V., Singh, R., Kohli, P. and Chaudhuri, S. Programmatically Interpretable Reinforcement Learning. *CoRR*. 2018, abs/1804.02477.

[71] Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A. et al. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*. November 2019, vol. 575, no. 7782, p. 350–354. DOI: 10.1038/s41586-019-1724-z. ISSN 1476-4687.

[72] Wang, S., Zhang, H., Xu, K., Lin, X., Jana, S. et al. Beta-CROWN: Efficient Bound Propagation with Per-neuron Split Constraints for Complete and Incomplete Neural Network Verification. *CoRR*. 2021, abs/2103.06624.

[73] Wang, Y., Chaudhuri, S. and Kavraki, L. E. Bounded Policy Synthesis for POMDPs with Safe-Reachability Objectives. In: André, E., Koenig, S., Dastani, M. and Sukthankar, G., ed. *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2018, Stockholm, Sweden, July 10-15, 2018*. International Foundation for Autonomous Agents and Multiagent Systems Richland, SC, USA / ACM, 2018, p. 238–246.

[74] Weltevrede, M., Spaan, M. T. J. and Böhmer, W. The Role of Diverse Replay for Generalisation in Reinforcement Learning. *CoRR*. 2023, abs/2306.05727. DOI: 10.48550/ARXIV.2306.05727.

[75] Weng, L. *Policy Gradient Algorithms*. 2018. Available at: https://lilianweng.github.io/posts/2018-04-08-policy-gradient/.

[76] Wurman, P. R., Barrett, S., Kawamoto, K., MacGlashan, J., Subramanian, K. et al. Outracing champion Gran Turismo drivers with deep reinforcement learning. *Nature*. February 2022, vol. 602, no. 7896, p. 223–228. DOI: 10.1038/s41586-021-04357-7. ISSN 1476-4687.

[77] Xu, K., Zhang, H., Wang, S., Wang, Y., Jana, S. et al. Fast and Complete: Enabling Complete Neural Network Verification with Rapid and Massively Parallel Incomplete Verifiers. In: *International Conference on Learning Representations*. 2021.

[78] Zhang, H., Weng, T., Chen, P., Hsieh, C. and Daniel, L. Efficient Neural Network Robustness Certification with General Activation Functions. In: Bengio, S., Wallach, H. M., Larochelle, H., Grauman, K., Cesa-Bianchi, N. et al., ed. *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. 2018, p. 4944–4953.

Appendix A

Experimental Setting

In this appendix, we describe our experimental setting. We describe what we can set for our agents, the environment, the models we used, and some arguments for our interpretation algorithms. We also outline some typical values, which may vary a bit given various tasks. However, some of the implementation related arguments, like the path to saving files or names for models, are not included.

A.1 Agents Arguments

As we mentioned in Chapter 3, one of the most important issues in reinforcement learning algorithms is the number of parameters that are used in training with particular algorithms. In this section, we summarize all parameters that we had to select combined with our commonly used values[1].

One part of agent parameter settings is common for all learning algorithms; we summarize them in Table A.1. The Table A.2, Table A.3 and Table A.4 then describe the selection of arguments for each algorithm used.

Variable	Typical Value	Description
Learning rate	$1.6e-5$	Learning rate of RL algorithms
Batch Size	32	Number of trajectories for a training iteration
Number of steps	25	Length of trajectories for training
Buffer Length	10000	Length of the replay buffer
Number of Runs	5000	Number of iterations of training
Weight Restarts	0-10	Number of restarts of NN (more in 4.2.3)
FSC Collector	*False*	Sets the behavior policy to FSC for a few episodes
Optimizer	Adam	Optimizer of learning algorithms

Table A.1: Common agents arguments.

A.2 Environment Setting

Given each experiment, we can set various variables in the environment. The most significant one is number of steps in the environment and virtual goal (and anti-goal) values, as

[1]May differ in particular experiments).

Variable	Value	Description
ϵ-greedy	0.14	Configuration of ϵ-greedy policy
ffnn layers	$(100, 100)$	Sizes of fully connected layers
lstm layers	$(100,)$	Sizes of LSTM layers
td error	squared loss	Loss function for temporal difference learning

Table A.2: Arguments of DQN Agent

Variable	Value	Description
ϵ-greedy	0.1	Configuration of ϵ-greedy policy
ffnn layers	$(50, 50)$	Sizes of fully connected layers
lstm layers	$(64,)$	Sizes of LSTM layers
td error	squared loss	Loss function for temporal difference learning

Table A.3: Arguments of DDQN Agent

Variable	Value	Description
Num. epochs	25	Number of epochs for each iteration
greedy eval	False	Greedy evaluation of policy
on-policy	False	Turn-off replay buffers
actor layers	$(50, 50)$	Sizes of fully connected layers of actor net
actor lstm	$(50,)$	Sizes of LSTM layers of actor net
critic layers	$(50, 50)$	Sizes of fully connected layers of critic net
critic lstm	$(50,)$	Sizes of LSTM layers of critic net

Table A.4: Agruments of PPO Agent

the models does not include rewards for achieving goal and PAYNT is usually driven by some PCTL specification. We usually select settings that correspond to given set of tasks, but we usually set the goal value equal to the maximum steps of the environment, because we want to force RL agents to prefer reaching goal over taking steps in the environment. We could set larger goal values, but we found that larger values lead to higher gradients (loss functions), which may lead to breaking the policy. However, more research would be beneficial.

Variable	Typical Value	Description
Max Steps	100-400	Max. available steps in the environment
Discount Factor	0.75	Value of discount provided to agents
Encoding Method	*Valuations*	Method for encoding observations
Action Filtering	*False*	Blocking of illegal actions
Evaluation Goal	*Max Steps*	Virtual reward for agent motivation
Evaluation Anti-goal	$-$*Evaluation Goal*	Reward for trap states

Table A.5: Summary of variables in the environment setting.

A.3 PAYNT Setting

The toolkit PAYNT allows multiple different approaches on how to solve given tasks. In this thesis, we usually compare our approaches with the selected argument `fsc-synthesis`, which, given some time[2], constructs some controller (if possible) that solves the given template and specification. However, we should mention that PAYNT is still under development, and contains multiple different and potentially better settings, which can outperform the settings we have been using. For running PAYNT with the RL oracle, we use options `fsc-synthesis` and `storm-pomdp` with modified implementation – some parts of Storm hints are used, and some are replaced and ignored.

A.4 Used Models

In Table A.6, we summarize all the models used in this thesis and information about its main configurable constants, number of observable variables usable for the encoding of the valuation, more described in 5.2, and the type of reward model described in the model. Moreover, we also mention the format of the input file.

A.5 Interpretation Arguments

In Table A.7, we summarize all the arguments used in the implemented interpretation algorithm.

A.6 PAYNT-RL Loop Arguments

In Table A.8, we summarize all the arguments used in the loop approach implemented.

[2] Usually in order of minutes.

Model Name	Settings	# of Observables	Reward Model	Format
evade	N, RADIUS	8	steps	PRISM
intercept	N, RADIUS	8	steps	PRISM
grid-large	*none*	3	steps	PRISM
mba	*none*	6	steps	PRISM
network	K, T, channels	6	dropped packets	PRISM
obstacle	N	3	steps	PRISM
refuel	N, CAPACITY	8	cost	PRISM
rocks	N, rocks (2-3)	8	cost (rocks)	PRISM

Table A.6: Summary of additional details for used models in the experiments. Reward model always describes penalties.

Variable	Typical Value	Description
Granularity	50	# of episodes for interpretation
Refusing	*True*, *False*	Refusing trajectories not leading to the goal
Memory Variance	0.5	Parameter for reduction of memory estimation
Prune Actions	*False*	Prune unusual actions in observations

Table A.7: Summary of arguments used in tracing interpret.

Variable	Typical Value	Description
Pre-loop iterations	500	Iterations of learning algorithm before the loop
Loop iterations	300	Iterations of learning algorithm in the loop
FSC iterations	100	Iterations of learning algorithm with FSC hints
Synthesis time	60	Number of seconds for FSC synthesis by PAYNT
Soft FSCs	False/True	Soft or hard action selection by FSC in RL
Soft multiplier	2/0.5	Update of the strength of soft FSC hints

Table A.8: Summary of arguments used in loop between RL and PAYNT