
**BRNO UNIVERSITY OF TECHNOLOGY**
**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**

**FACULTY OF INFORMATION TECHNOLOGY**
**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**

**DEPARTMENT OF INTELLIGENT SYSTEMS**
**ÚSTAV INTELIGENTNÍCH SYSTÉMŮ**


# MODEL-BASED REINFORCEMENT LEARNING FOR POMDPS

**ZPĚTNOVAZEBNÉ UČENÍ PRO POMDPS S VYUŽITÍM MODELŮ**

**BACHELOR'S THESIS**
**BAKALÁŘSKÁ PRÁCE**


**AUTHOR** **LUCIE SMÍŠKOVÁ**
**AUTOR PRÁCE**

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



**BRNO 2024**

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

# Bachelor's Thesis Assignment

162598

| | |
|---|---|
| Institut: | Department of Intelligent Systems (DITS) |
| Student: | **Smíšková Lucie** |
| Programme: | Information Technology |
| Title: | **Model-Based Reinforcement Learning for POMDPs** |
| Category: | Artificial Intelligence |
| Academic year: | 2023/24 |

Assignment:

1. Study the state-of-the-art planning and reinforcement learning methods Markov Decision Processes (MDPs) and Partially Observable MDPs (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.
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.


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

| | |
|---|---|
| Supervisor: | **Češka Milan, doc. RNDr., Ph.D.** |
| Head of Department: | Hanáček Petr, doc. Dr. Ing. |
| Beginning of work: | 1.11.2023 |
| Submission deadline: | 31.7.2024 |
| Approval date: | 8.7.2024 |

# Abstract

Partially observable Markov decision processes allow us to model systems containing state uncertainty. They are useful when we have only partial information about the states ( so called observations). The aim of this thesis was to develop a method combining inductive synthesis and reinforcement learning to develop the best possible finite-state controller. This method was then implemented as an extension to the tool PAYNT.



# Abstrakt

Markovské rozhodovací procesy s částečným pozorováním nám umožňují modelovat systémy obsahující stavovou neurčitost. Jsou užitečné, pokud máme pouze částečné informace o stavech (tak zvaná pozorování). Cílem této práce bylo vyvinout metodu kombinující induktivní syntézu a zpětnovazebné učení k vytvoření co nejlepšího konečně stavového kontroléru. Tato metoda poté byla implementována jako rozšíření nástroje PAYNT.



# Keywords

Partially observable Markov decision processes, Finite State Controller, Synthesis, Reinforcement learning, recurrent neural network



# Klíčová slova

Markovské rozhodovací procesy s částečným pozorováním, Konečně stavovový kontroler, syntéza, zpětnovazebné učení, rekurentní neuronové sítě



# Reference

SMÍŠKOVÁ, Lucie. *Model-Based Reinforcement Learning for POMDPs.* Brno, 2024. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor doc. RNDr. Milan Češka, Ph.D.

# Model-Based Reinforcement Learning for POMDPs

## Declaration

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

.....

Lucie Smíšková

July 30, 2024

# Contents


**1 Introduction** **3**

**2 Preliminaries** **5**
2.1 Markov chain . . . . . 5
2.2 Family of Markov chains . . . . . 6
2.3 MDP . . . . . 7
2.4 POMPD . . . . . 9
2.5 Finite State Controller . . . . . 10

**3 Inductive synthesis** **11**
3.1 PAYNT . . . . . 11
3.2 Abstraction refinement . . . . . 12

**4 Reinforcement learning** **13**
4.1 Deep reinforcement learning . . . . . 14
4.2 Recurrent Deep Q–learning . . . . . 17
4.3 Neural network architecture . . . . . 17
4.4 Training . . . . . 18
4.4.1 Environment . . . . . 18
4.4.2 Initialization . . . . . 19
4.4.3 Pre–training phase . . . . . 19
4.4.4 Training cycle . . . . . 19

**5 Combination of reinforcement learning and synthesis** **22**
5.1 Implemented method . . . . . 22

**6 Evaluation** **25**
6.1 Experimental setting . . . . . 25
6.2 Grid . . . . . 26
6.3 Maze . . . . . 28
6.4 Pentagon . . . . . 29
6.5 IFF . . . . . 31

**7 Conclusion** **35**

**Bibliography** **37**

# List of Figures


2.1 Example of Markov chain . . . . . 6
2.2 Example of family of Markov chains . . . . . 7
2.3 Realisation $r_1$ . . . . . 8
2.4 Realisation $r_2$ . . . . . 8
2.5 Example of an MDP . . . . . 9

4.1 Reinforcement learning loop. The agent takes actions resulting in changes in the environment (transitions between states). The agent then receives reward and new observation from the environment. . . . . 14
4.2 General structure of a neural network . . . . . 15
4.3 Recurrent network architecture by Jeffrey Elman [1] . . . . . 16
4.4 LSTM architecture [2] . . . . . 16
4.5 Structure of implemented neural network . . . . . 18

6.1 Grid . . . . . 27
6.2 Maze . . . . . 29
6.3 Graph comparing maximal rewards of each Pentagon experiment constructing 1-FSC, $t_0$ is time needed to find the best found FSC . . . . . 30
6.4 Graph comparing maximal rewards of each Pentagon experiment constructing 2-FSC . . . . . 31
6.5 Graph comparing maximal rewards of each IFF experiment constructing 1-FSC 32
6.6 Graph comparing maximal rewards of IFF experiments constructing incremental FSC . . . . . 34

# Chapter 1

# Introduction

Nowadays, machine learning is very popular for solving various tasks. It is well-suited to a wide range of problems, among the most notable ones are the processing of the natural language and image recognition. Additionally, the field has applications in other areas such as medicine, finance, various analyses. To be able to solve a certain problem, we need to define it at first. This can be done using a variety of models. In certain instances, we want robots to make decisions, so we do not have to control them. For this type of problems, we can use Markov Decision Processes (MDPs). You can effectively build policies for MDPs using probabilistic model checkers such as Storm [20] or Prism [23].

In the real world, we are confronted with uncertainties and randomness. MDPs can only deal with uncertainties to a limited extent. For this reason, it may be necessary to use their extension instead, namely Partially Observable Markov Decision Processes (POMDPs). These are particularly useful in cases we have to deal with state uncertainty. POMDPs have the same properties as MDPs with the difference, that they also contain observations. For example, we do not have the information of which state we are currently in, but we have some incomplete information (observation) about this state (such as where we can move or whether or not we are in a goal state). The potential applications of POMDP vary widely. From basic navigation problems, such as navigating a maze, to the development of self-driving cars. But we can also find applications outside the technical fields, for example in medicine. Further applications can be found in a Survey of POMDP Applications by Anthony R. Cassandra [12].

The goal in POMDP solving is to create a scheduler. The scheduler should be able to find the best possible action in each state for given the observation. Scheduler can be represented either by belief-states or Finite State Controllers (FSCs). In the first case, we select an action to be performed based on a belief [26]. Belief represents a probability distribution over states of POMDP. On the other hand, finite state controller maps observations directly to actions. These controllers can be either deterministic or nondeterministic. In this thesis, I will only deal with deterministic ones. FSCs map directly observations to actions. There are several ways to create the FSC. We can use formal methods such as synthesis. Several synthesizing methods are implemented in the Python tool PAYNT [2]. Synthesis allows us to find optimal controller by exploring the design space. Synthesis is highly effective in term of finding scheduler for smaller models, but the computation time is too high for large models (scales exponentially) and in some cases the problem can be undecidable. For this reason, it is almost impossible to solve really large models using these methods in an acceptable time. Using reinforcement learning methods should help to solve this problem. Machine learning methods are not able to guarantee correctness and efficiency of the found

solution, which is a significant disadvantage. Reinforcement learning methods are capable of finding FSCs even for larger models. Therefore, I will combine both mentioned approaches in this thesis. I will combine reinforcement learning technique which allows us to solve large POMDPs with synthesis method which helps us to achieve the optimality and verify the controller. There is a number of solutions to MDP using reinforcement learning techniques [27], but in the case of POMDP it is more complex and neural networks must be used.

## Thesis structure

In the following Chapter 2, I define several terms that are necessary for the understanding of the following text. This is going to be followed by Chapter 3 where I introduce synthesis with emphasis on the tool PAYNT. After that In Chapter 4 I will move onto reinforcement learning topic and describe Recurrent Deep Q–learning algorithm, I have implemented. In Chapter 4.2 I will introduce combination of synthesis and reinforcement learning, which is the aim of this thesis. the experimental results will then be evaluated In Chapter 6.

# Chapter 2

# Preliminaries

At first, we need to declare definitions of some important terms that will be used further In this thesis. From Section 2.1 to Section 2.4. I will first introduce several stochastic models from the most basic (Markov chains) to the more complicated (Partially Observable Markov Decision Processes). Then I need to define the Finite State Controllers which is a way to represent schedulers. The definitions are provided with illustrative examples for better understanding.

Definitions in this chapter are taken over and adapted from several papers. Specifically, Markov chain definition (Definition 1) is taken from Inductive Synthesis for Probabilistic Programs Reaches New Horizons [3]. The definition of Family of Markov Chains (Definition 2) and definition of Realisation (Definition 3) are taken from Counterexample-Driven Synthesis for Probabilistic Program Sketches [14]. Definitions of Markov Decision Process (Definition 4) and Partially Observable Markov Decision Process (5) are both taken from Inductive Synthesis of Finite-State Controllers for POMDPs [4]. Definitions of an Observation based strategy and Finite State Controller are both taken from Permissive Finite-State Controllers of POMDPs using Parameter Synthesis [22].

## 2.1 Markov chain

Markov Chains (MC) are the most basic mathematical models with Markov property. The other Markov processes, which I will mention later in this chapter, are based on the Markov chain. A stochastic process has the `Markov property` if the future states depend solely on the present state, without regard to the previous states. The probabilities of moving from one state to another can be represented by a transition matrix.

**Definition 1.** A Markov chain is a tuple $D = (S, s_0, \mathbf{P})$, where

- $S$ is a finite set of states,
- $s_0 \in S$ is an initial state,
- $\mathbf{P} : S \to Distr(S)$ is a transition probability function.

We write $P(s,t)$ to denote $P(s)(t)$. The state $s$ is absorbing if $P(s,s) = 1$.

If we get into absorbing state, we are never going to be able to get to any other state (it has only one transition with probability 1 that goes back to this state).

Another property of the state is reachability. Reachability refers to the ability to transition from one state to another state. It means we can get from state $s_0$ to state $s_n$ in one or more steps. The state is not reachable, if we are not able to get into this state after infinite number of transitions from the initial state. By other words, all probabilities of reaching in this state from all reachable states are zero. Otherwise, we can count probability of reaching the state and expected number of steps needed to reach the state.

**Example 1.** *Basic example is in Figure 2.1. This example consists of three states ($S = s_0, s_1, s_2$). Transition probabilities are given by following matrix:*

$$P = \begin{pmatrix} 0.4 & 0.1 & 0.5 \\ 0.7 & 0.2 & 0.1 \\ 0 & 0.7 & 0.3 \end{pmatrix}$$

*As you can see, the sum of each row equals to one. Probabilities on the diagonal corresponds to staying in the same state after the next transition. Each row represents probabilities of reaching next state given by a column after being in a current state given by a row. In our example the probability of not moving from state $s_0$ is 0.4, the probability of transition from $s_0$ to $s_1$ is 0.1 and probability of transition from $s_0$ to $s_2$ is 0.5.*

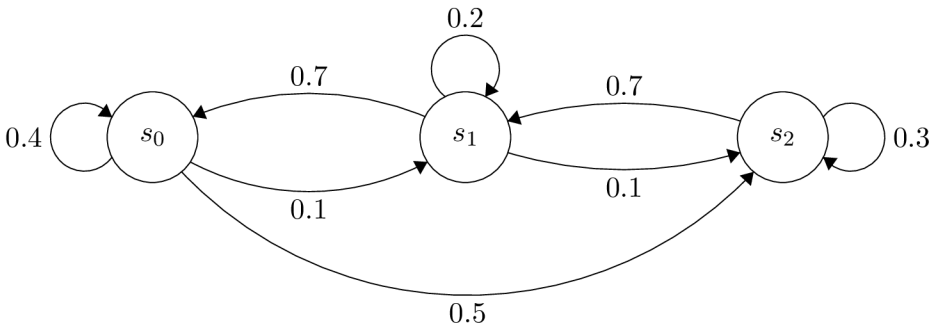



Figure 2.1: Example of Markov chain

## 2.2 Family of Markov chains

We can also describe more than one MC at a time using families of Markov Chains. If we have one set of states with different transition probability, we are able to describe them as a family of Markov chains.

**Definition 2.** A family of Markov chains is a tuple $\mathfrak{D} = (S, s_0, K, \mathfrak{P})$, where

- $S$ and $s_0$ as in definition 1,
- $K$ is a finite set of parameters where domain for each parameter $k \in K$ is $T_k \subseteq S$
- $\mathfrak{P}$ is transition probability function $\mathfrak{P} : S \to Distr(K)$

If we want to get Markov chain from its family, we can do so by instantiating each parameter with value from its domain. Created Markov chain is called a realisation.

**Definition 3.** A realisation of a family $\mathfrak{D} = (S, s_0, K, \mathfrak{P})$ is a function $r : K \to S$ where $\forall k \in K : r(k) \in T_k$. A realisation $r$ yields an $MCD_r := (S, s_0, \mathfrak{P}(r))$, where $\mathfrak{P}(r)$ is the transition probability matrix in which each $k \in K$ in $\mathfrak{P}$ is replaced by $r(k)$. Let $\mathcal{R}^{\mathfrak{D}}$ denote the set of all realisations for $\mathfrak{D}$.

**Example 2.** *As an example, I will use an abstract family of Markov chains consisting of three states $S = \{s_0, s_1, s_2\}$, where $s_0$ is an initial state, and three parameters $K = \{a, b, c\}$. The number of states and parameters in this example is the same, but it is not necessary. We can also have a family of Markov chains with, for example, two states and five parameters or, on the contrary, five states and two parameters.*

*Domains of the parameters are $T_a = \{s_0, s_1\}, T_b = \{s_0, s_1, s_2\}, T_c = \{s_1, s_2\}$. The transition probability matrix is:*

$$2x - 5y = 8$$
$$3x^2 + 9y = 3a + c$$

*You can see this family of Markov chains in picture 2.2.*

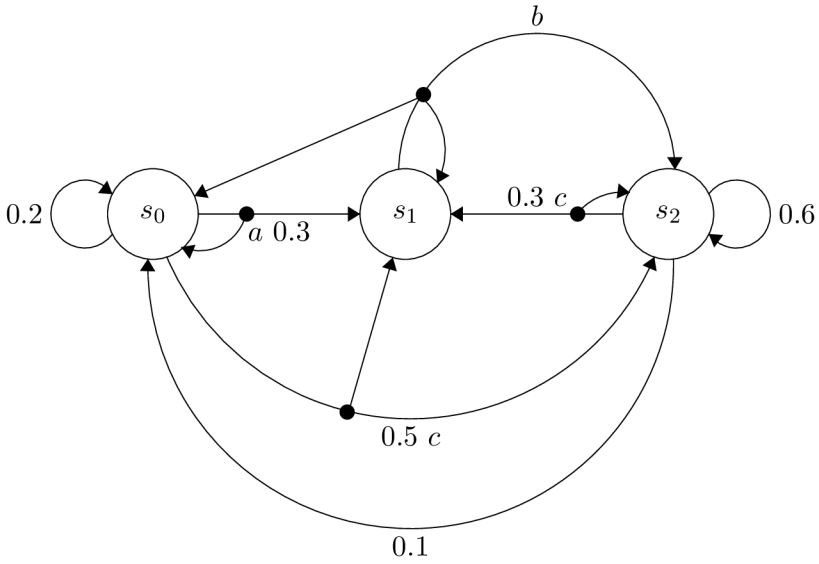



Figure 2.2: Example of family of Markov chains

*There are $2 \times 3 \times 2 = 12$ possible realisations. It is counted by multiplying of set of domains of each parameter. I am going to show you only two realisations as an example.*

*The first realisation is defined as $r_1 : r_1(a) = s_0, r_1(b) = s_0, r_1(c) = s_2$. As you can see in picture 2.3, there is no possible way how to get from initial state $s_0$ to state $s_1$, therefore the state $s_1$ is unreachable. There is no absorbing state.*

*Another realisation can be, for example, $r_2$ defined as $r_2 : r_2(a) = s_1, r_2(b) = s_2, r_2(c) = s_2$. You can see this realisation in picture 2.4. None of the states are absorbing or unreachable.*

## 2.3 MDP

Markov decision processes (MDPs) are models similar to the Markov chains. They both model stochastic processes and have Markov property. The main difference is that MDP

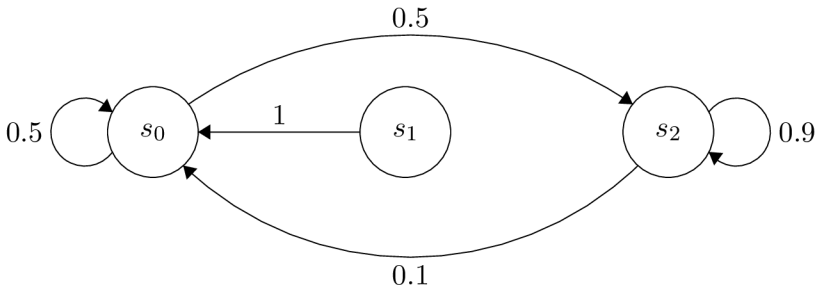



Figure 2.3: Realisation $r_1$

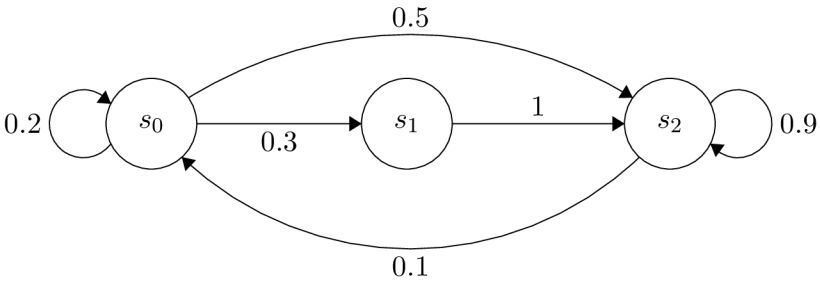



Figure 2.4: Realisation $r_2$

can be controlled by actions. So we have more control over the transition probabilities and we can, for example, maximize the probability of reaching a certain state.

To choose which action should be taken, we use schedulers. Briefly, scheduler is a mapping function that determines what action should be selected in each state (this is not very important for this thesis, so I will not define it formally, but you can take a look at the definition 6 of Observation based scheduler in Section 2.4).

**Definition 4.** MDP is a tuple $M = (S, s_0, Act, P)$ where

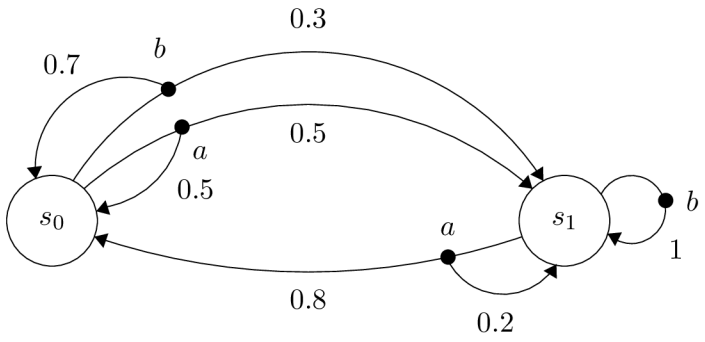
- $S$ is finite set of states
- $s_0 \in S$ is an initial state
- $Act$ is a finite set of actions
- $P(s'|s, a)$ is a transition probability function that gives probability of evolving to $s'$ after taking action $a$ in state $s$

**Example 3.** *As an example, I will use simple MDP consisting of two states and two possible actions from each state. You can see the diagram and transition probability matrix in Figure 2.5.*

*By applying memory-less scheduler on MDP, we get induced Markov chain. For example, if we want to maximize the probability of being in state $s_0$, the scheduler should be: $\sigma = \{s_0 \to b, s_1 \to a\}$. Using this scheduler on given MDP, we get MC with two states and following transition probability matrix:*

$$P = \begin{pmatrix} 0.7 & 0.3 \\ 0.8 & 0.2 \end{pmatrix}$$

*In that case, the number of the states did not change, but in some cases, it is possible to get MC with less states than was in the original MDP. For example, if we have MDP*




(a) MDP diagram

$$P = \begin{matrix} (s_0,a) \\ (s_0,b) \\ (s_1,a) \\ (s_1,b) \end{matrix} \begin{bmatrix} 0.5 & 0.5 \\ 0.7 & 0.3 \\ 0.8 & 0.2 \\ 0 & 1 \end{bmatrix}$$

(b) MDP transition probability matrix

Figure 2.5: Example of an MDP

*consisting of three states, but one of the states is unreachable in the MDP, the MC will consist of only two states (the unreachable one can be excluded).*

Another model important for one of the synthesizing methods, the Abstraction refinement method, is all-in-one MDP. It is derived from Family of Markov Chains and represents all possible realisations in a compact way. The abstraction of all-in-one MDP (called quotient) can be evaluated by model checker (such as Storm).

## 2.4 POMPD

Generalization of MDP is Partially Observable Markov Decision Process (POMDP). They are both used to model decision-making problems. In many real world situations, we do not have complete information. To model these situations, we need something that allows us to deal with uncertainty. The agent has only incomplete information about the state it is currently in. The piece of information is called observation. The agent, unlike the MDP, does not make decisions based on the state, but works with observations instead.

**Definition 5.** POMDP is a tuple $M = (S, s_0, Act, P, Z, O)$ where

- $S$ is finite set of states
- $s_0 \in S$ is an initial state
- $Act$ is a finite set of actions
- $P(s'|s,a)$ is a transition probability function that gives probability of evolving to $s'$ after taking action $a$ in state $s$
- $Z$ is a finite set of observations
- $O$ is an observation function that returns for every state $s$ an observation.

You can imagine POMDP as an MDP with added observation into each state. There are usually multiple states with the same observation. An example of an POMDP is in Section 6.2. It is an example of simple $4 \times 4$ grid which is figured in Figure 6.1.

Similarly to MDP, we are trying to create a scheduler that will tell us which action to perform. The difference is that here we will not handle states, but observations, because

the agent is not always able to clearly determine what state it is in. The Observation-based scheduler therefore maps observations to actions.

Before the actual definition, we have to introduce a few symbols. Set of finite paths $\pi = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \dots s_n \in Paths_f^M in$. By lifting the observation function $O$ to paths we get $\pi = O(s_0) \xrightarrow{a_0} O(s_1) \xrightarrow{a_1} \dots O(s_n)$.

**Definition 6.** An observation-based strategy $\sigma$ for a POMDP $M$ is a strategy for the underlying MDP $M$ such that $\sigma(\pi) = \sigma(\pi')$ for all $\pi, \pi' \in Paths_{fin}^M$ with $O(\pi) = O(\pi')$. $\sum^M$ is the set of observation-based strategies for M.

## 2.5 Finite State Controller

Finite state controllers (FSCs) represent observation-based schedulers. FSCs have finite memory given by number of memory nodes which represents inner states of Mealy Machine.

**Definition 7.** A finite state controller for POMDP $M$ is tuple $F = (N, n_0, \gamma, \delta$ where

- $N$ is a finite set of nodes
- $n_0 \in N$ is an initial node
- $\gamma$ is the action mapping function $\gamma : N \times Z \to Distr(Act)$
- $\delta$ is the memory update function $\delta : N \times Z \times Act \to Distr(N)$

For $|N| = k$, we call an FSC a k-FSC.

Less formally we can say that $\gamma(n, z)$ determines the action when agent is in node $n$ and observes $z$ and $\delta$ updates the memory node, when being in node $n$ and observing $z$ after taking action $Act$.

Finite state controllers can be either deterministic or stochastic. Stochastic FSCs define next node as probability distribution, but we will use deterministic ones in this thesis.

The workflow of controlling the agent in the environment modeled as POMDP using FSC is following. Agent is in node and gets an observation. This observation determines next action. Agent takes the action. This causes POMDP to move to the next state and FCS to the next node. There agent gets new observation and the entire process repeats.

# Chapter 3

# Inductive synthesis

In this chapter, I will firstly describe what how does the synthesis work, then I will talk about the PAYNT tool and then I will focus on one of the inductive synthesis methods implemented PAYNT, which is the abstraction refinement method.

Synthesis is a way to build an FSC for a given POMDP to satisfy the specification. Usually we want to find optimal $k$-FSC which maximizes cumulative reward. This problem is undecidable, because the $k$ can be indefinitely large. The synthesis has two stages. We create family of FSC which is called the design space. In the second stage we try to select the best FSC from the design space. Usually we firstly synthesize 1-FSC, than create new design space using 2-FSC and so on. There are multiple methods for identifying the most optimal FSC within the family of FSCs. Some of these methods are implemented into the PAYNT tool, which will be further described in the following section.

## 3.1 PAYNT

Probabilistic progrAm sYNThesizer, shortly PAYNT, is a tool implemented in Python. PAYNT was, as the name suggests, developed to be able to synthesize probabilistic programs. The goal in this case is to fill holes (undefined parameters) to meet a specification (typically maUsually we create 1-FSC first, then 2-FSC and so on. ximizing reward or reachability). Another task that PAYNT can solve is the synthesis of FSCs for POMDPs and that is what I will focus on.

To be able to accomplish these tasks, it contains an implementation of several synthesis methods. Simply, this tool works by setting up a design space containing a family of FSCs and trying to select the FSC representing the policy that the desired specification the best. The most basic method is one–by–one. This method goes through all the possible FSCs from the family and selects the one that best suits its given specification. Next method is Counter Example Guided Inductive Synthesis (CEGIS). This method is more complicated, as it first selects one realisation from the family and decides whether or not it meets the specification. If the specification is not met, a critical set of states is sampled. Then all realisations that do contain this subset of the states are removed from the family, because it is assumed that they do not meet the required specification either. PAYNT also contains the AR method, which will be discussed more in Section 3.2. By combining the two previously mentioned methods (CEGIS and AR) a hybrid method is developed, which switches between these two methods.

PAYNT needs two inputs to be able to perform synthesis. It needs to know the model (in our case POMDP) and the specification. PAYNT has two input options. The model can be defined as a sketch in PRISM language format [23]. The specification of the property is then in the form of an expression in probabilistic temporal logic, which can for example be PCTL (Probabilistic Computation Tree Logic). Another way of input to PAYNT is the POMDP file format by Cassandra et. al [11], which was created just for defining POMDPs. This format has an enormous advantage, as the parsing of this format is much simpler, because it consists only of preamble, obligatory initial state, and several matrices. Because of its simplicity it is suitable for creating an environment for training neural networks and for this reason it is more useful for this work, and I will discuss it a little further.

In this work I will use Cassandra file format POMDP as an input. The POMDP input file consists of a preamble and a body. The preamble contains a discount, a value type (in our case reward), a number or list of states, actions and observations. The body is composed of several matrices determining transitions between states, observations and rewards. The values in these matrices take the form of probabilities that a given transition/observation/reward will be executed under given conditions (initial state, action, target state ...).

## 3.2 Abstraction refinement

Abstraction refinement (AR) is one of the synthesis methods implemented in PAYNT, as mentioned earlier in this text. It works in quite opposite way than CEGIS method. AR takes family of MCs and transforms it into all-in-one MDP which is then abstracted into quotient. We can obtain an expected minimum and maximum values (probability or reward) using Storm model checker from the quotient and based on the values, we decide if the family satisfies given specification or not. Quotient is a model created from all-in-one MDP by forgetting in which realisation it operates. This process is called an abstraction.

Second part of this algorithm is a refinement loop. It varies according to the type of problem we're solving. The problem can be either maximization or we have given threshold. In case we get threshold, it is a bit easier. We have three values which can be set in three ways (the minimum is always smaller than the maximum and therefore, only the position of the threshold differs). In case thrashold is smaller than minimum, none of the ralisations in this family satisfy the specification. On the contrary, if the threshold is greater than the maximum, all realisations satisfy the specification. If threshold is in the between the minimum and the maximum, we split the set of realisations into subfamilies and consider them separately. The splitting is done by restricting actions of the MDP to the particular subfamily of MCs. This avoids rebuilding MDP after each iteration.

The second one is maximizing synthesis, where we do not have given threshold and are supposed to find the maximum. At the beginning, we set maximum (let me sign it as max*) as a small value. Then, if we get maximum which is bigger than max*, there are another two options, if the maximal scheduler is consistent, we set the max* to value of maximum, otherwise we split and if minimum is bigger than max*, the max* is set to the value of minimum. If the maximum is lower than max*, this realisation is discharged. As you can see, the reinforcement loop is similar to the one with threshold with the difference that instead of threshold we use a value that changes in time according to the maximum and minimum values.

In the case of maximum synthesis, consistency is the crucial factor for us (as it is also used to choose, if maximum should be overwritten in maximizing synthesis).

# Chapter 4

# Reinforcement learning

Reinforcement learning (RL) is one of the fields of Machine learning. Reinforcement learning reflects more the actual way people learn than other methods, such as supervised learning, semi–supervised learning and unsupervised learning do. Said methods differ in the kind of data we have available. In supervised learning, we need to have labeled data. It is really powerful approach, however data-set creation is time and human resources consuming. The best use of this approach is in speak recognition or image classification. In semi-supervised learning, we have part of the data labeled, but not all of them. These techniques are used in case we need a huge amount of data, but do not have resources to label all of them. Semi-supervised learning can be used in image classification if we do not have enough labeled data. The learning itself can be a bit less effective in comparison to the supervised learning, but this is compensated by the difficulty of creating the dataset. Unsupervised learning, as the name suggests, works with unlabeled data. It can be used for various pattern matching problems, where it is difficult to obtain labeled data.

In reinforcement learning, agent learns from experience, same as people do. At first, agent performs random actions and gets new observations with information on how good of a choice he made. We can compare it to children exploring the world by moving around and getting feedback from their surroundings on what to do or not to do. The feedback, the agent receives, has a form of rewards. A reward is a numerical value, and it is typically said that the higher the better. This value can be both positive and negative. A positive value typically signifies the agent's progress towards the goal state, while negative can be moving to an unsatisfactory state. The aim is almost always to learn how to get the biggest cumulative reward possible. We do not need to maximize the reward obtained after each step, but the cumulative reward we get after a sequence of steps is more important for us.

The learning is based on evaluating long term rewards that the agent gets after taking steps in the environment. The environment is a model and, in our case, is represented by POMDP (definition 5). The learning process is a loop. The agent performs an action in the environment. The action determines what is to happen in the environment. This usually determines which transition between states is to be performed (the agent can also stay in the same state). These transitions are given by a transition probability matrix defined in the POMDP model. The agent receives a reward and a new observation from the environment as a response. This observation is corresponding to a different observation that was assigned to the state we get in after performing an action and not the state before the action was performed. This process is shown in Figure 4.1.

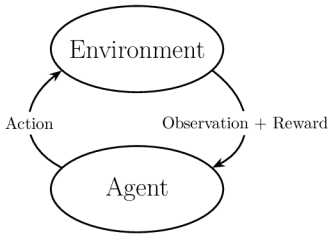



Figure 4.1: Reinforcement learning loop. The agent takes actions resulting in changes in the environment (transitions between states). The agent then receives reward and new observation from the environment.

## 4.1 Deep reinforcement learning

Deep reinforcement learning is a subfield of reinforcement learning. Deep learning methods (in general, also in the case of supervised learning ...) contain neural networks (usually one, but can be multiple). Neural networks (NN) are particularly useful in the case of training huge models. There is a large variety of reinforcement learning method. You can see the taxonomy of reinforcement learning algorithms at OpenAi Spinning Up page [1]. An overview of reinforcement learning algorithms and several examples of their use is presented in the paper Recent Advances in Deep Reinforcement Learning Applications for Solving Partially Observable Markov Decision Processes (POMDP) Problems: Part 1—Fundamentals and Applications in Games, Robotics and Natural Language Processing [30].

In order for a large model to be trained, it is necessary to have a large amount of data available for training the model on. This problem does not need to be solved that much in my case, because the model is trained by simulation in a defined environment that can be reset and reused. Another problem I have to deal with is overtraining. Overtraining of a model occurs when the model is already trained, but we continue training, or we try to train an already trained model. I will discuss this problem and its solution in more details later in Section 5.

Neural networks are made up of layers. These layers are made up of neurons. The neurons usually have some biases and weights, in some cases they may also contain some additional variables. In general, a neural network looks like the one in Figure 4.2. It contains one input layer, several hidden layers and one output layer. Input data is fed into the input layer. These are then processed through the hidden layers and finally we get the values at the output layer. There are several types of layers. Neural networks are often named after the layers they contain (for example, a recurrent neural network contains one or more recurrent layers, but usually some other layers in addition).

Let me describe in more details types of NN layers which I will deal with further in this theses.

### Linear layer

Linear layer can be also called Fully connected layer and that describes exactly how it works. Regardless of which layer follows, all neurons of this layer are connected to all neurons of the following layer (e.g. in Figure 4.2 all layers are linear). It is the most frequently used type of layer and can be found in almost every neural network. For example, in convolutional neural networks it is used to connect individual convolutional layers.

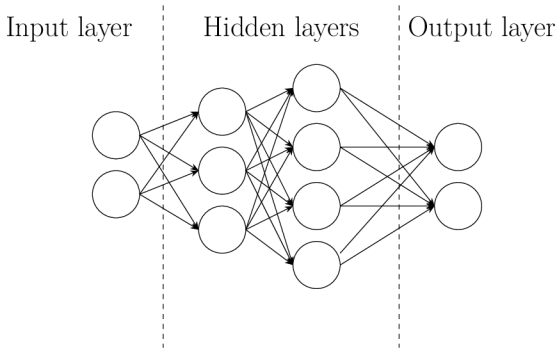



Figure 4.2: General structure of a neural network

### Softmax

Softmax is not so much a layer as a function, but I decided to include it here. Softmax is usually found at the end of the neural network and is usually preceded by a linear layer. Softmax is used to normalize the output so that the values are in the interval $\langle 0; 1 \rangle$ and all of the output values sum to 1.

### Recurrent layer

Recurrent neural networks are very extensive in the creation of large language models. Other types of neural layers have no memory and this limits them because they can only work with the last information. But for example, if we want to translate a sentence, we need to know the context in the form of the previous words in the sentence and this is what the recurrent layer solves. There are several types of recurrent neural layers. One of the first recurrent neural networks was introduced by John Hopfield [21]. I will not describe it in detail, because I did not use this NN in this work. The one I use is in `Pytorch module: torch.nn.RNN` which was introduced in 1990 by Jeffrey Elman [18]. You can see the architecture of recurrent layer in Figure 4.3. Each neuron in this layer has a cell. The cell is used to store information from previous passes. The output of this layer will be added to the next input with some weight, which is usually lower than the weight of the input. In this way we achieve that with the passage of time the first value will have less and less influence than the following values.

### Long short–term memory

Long short–term memory (LSTM) is another type of recurrent layer. The recurrent layer as described previously has one disadvantage. In some cases, we do not have, for example, words that are related directly next to each other, but the context is stretched out in a larger section where unnecessary words are found. For this it is useful to have a tool that allows us to remember some stuff and forget others and this is what the LSTM layer solves. LSTM has a hidden state In addition to the cell state. The cell state serves to store information in long-term and the hidden state represents short-term memory (hidden state represents current output). LSTM contains three gates that determine what data should be stored in the long-term memory (cell state). These gates control the flow of information which is needed to predict the output in the network. All three gates are Sigmoid function

[1]The picture was taken over from https://pabloinsente.github.io/the-recurrent-net [8]

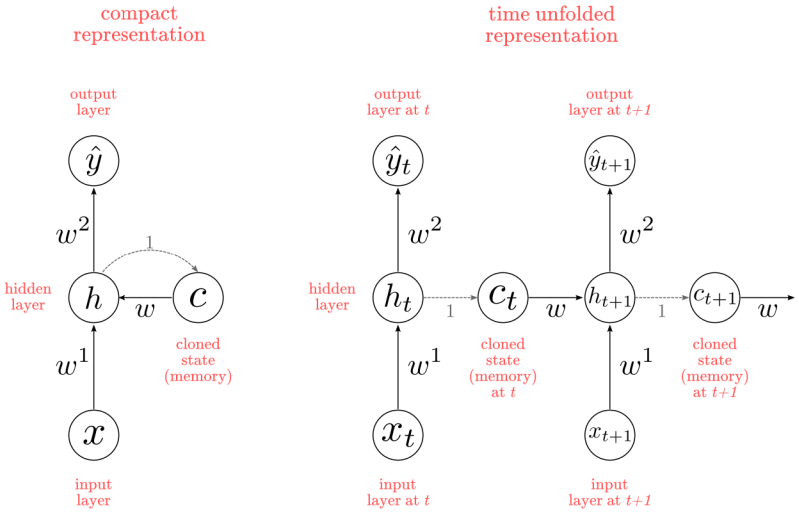



Figure 4.3: Recurrent network architecture by Jeffrey Elman [1]

and their results are constant on the interval $\langle 0; 1 \rangle$. The first gate is an input gate. It determines how important the new data is to remember. Next, there is a forget gate. This gate determines how important the data stored in cell state are (if we can forget them or keep them). Lastly, there is output gate. This gate decides if the data stored in cell state are relevant for current output. The workflow is as follows. At first, we multiply cell state with result of forget gate, then we add input with input gate and multiply this result with cell state (which is already affected by the previous calculation). Then we generate output applying the activation function to the output of input multiplied by output gate. The LSTM architecture is described in Long Short–Term Memory based recurrent neural network architectures for large vocabulary speech recognition by Hasim Sak et. al. [29].

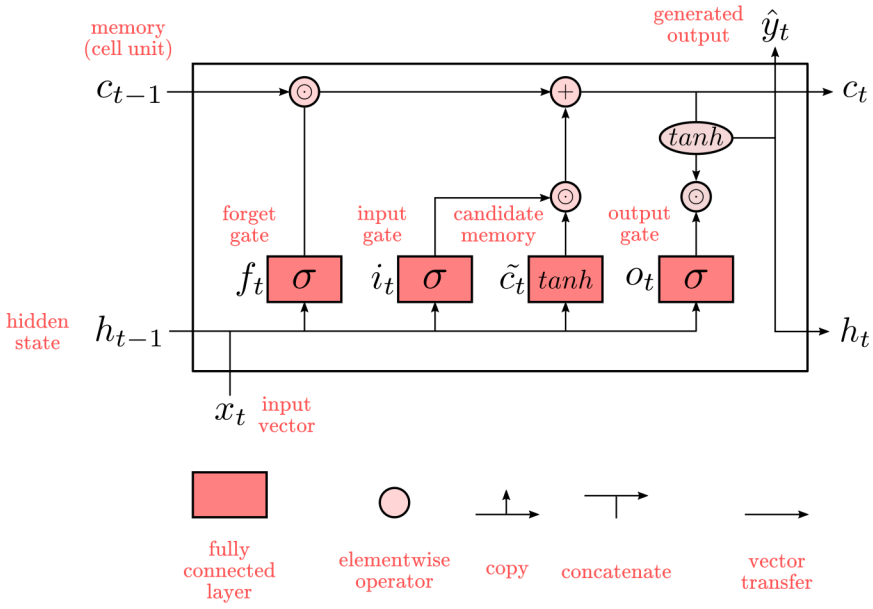



Figure 4.4: LSTM architecture [2]

## 4.2 Recurrent Deep Q–learning

As a base of reinforcement learning method I have implemented, I have chosen to use Deep Q–Learning algorithm (DQL) described in Human-level control through deep reinforcement learning by Mnih et. al. [25]. The aim of its task is to find optimal action–value function (Q–function) using Deep Q–Network (DQN) agent. Authors of referenced paper have evaluated this algorithm on several Atari 2600 games. A fundamental insight into Deep Q–learning is also provided by Maxim Egrov in Deep Reinforcement Learning with POMDPs [17].

Neural networks are a powerful tool, but they are not suitable for dealing with long term dependencies between data. However, recurrent neural networks allow us to work with memory. Due to that fact, I have decided to extend this method on recurrent neural network. One of the recurrent neural network architectures is described by Clare Chen in Deep Q–Learning with Recurrent Neural Networks [16]. Deep Recurrent Q–Learning in the context of POMDPs is in Deep Recurrent Q-Learning for Partially Observable MDPs by Matthew Hausknecht and Peter Stone [19]. In the lastly mentioned paper, authors used LSTM as a recurrent layer in combination with three convolutional layers.

In the following sections of the text, I describe firstly the architecture of the neural network that was used, then the used machine learning algorithm and finally I describe how this algorithm was implemented and connected with the synthesis in the framework of PAYNT.

## 4.3 Neural network architecture

The neural network was in Python using PyTorch library [5]. I have assessed two networks, which are the same, besides the fact that one of them contains Elman Recurrent layer, and the second one has Long-Short-Term Memory layer instead. Both of these layers gave me similar outputs which will be discussed In more depth in the Section 6. The neural network architecture is illustrated in Figure 4.5. All of the layer's functions were mentioned earlier in this text. I have chosen to use two recurrent layers, where the input size of the first one is 1, which is because we can have only one observation at time. As a hidden size (that is also number of output size of the recurrent neural network and input size of the linear layer) I have chosen random number, which seems to work fine. The output size of the linear layer equals number of actions of the POMDP model as this output gives us distribution over actions. To be able to have the probability distribution we have to normalize the output using Softmax function.

Both modules `torch.nn.RNN` and `torch.nn.LSTM` use the same activation function as default. The activation function is `Tanh` function. For RNN it is also possible to use `ReLU` as an activation function. For RNN, the function that computes hidden state is following:

$$h_t = tanh(x_t W_{ih}^T + b_{ih} + h_{t-1} W_{hh}^T + b_{hh})$$

where,

- $x$ is an input
- $t$ is and index in sequence
- $h$ is hidden state

[2]The picture was taken over from https://pabloinsente.github.io/the-recurrent-net [8]

- $W_{ih}$ is input–hidden weight
- $W_{hh}$ is hidden–hidden weight
- $b_{ih}$ is input–hidden bias
- $b_{hh}$ is hidden–hidden bias

The previous equation is adapted from PyTorch documentation [28].

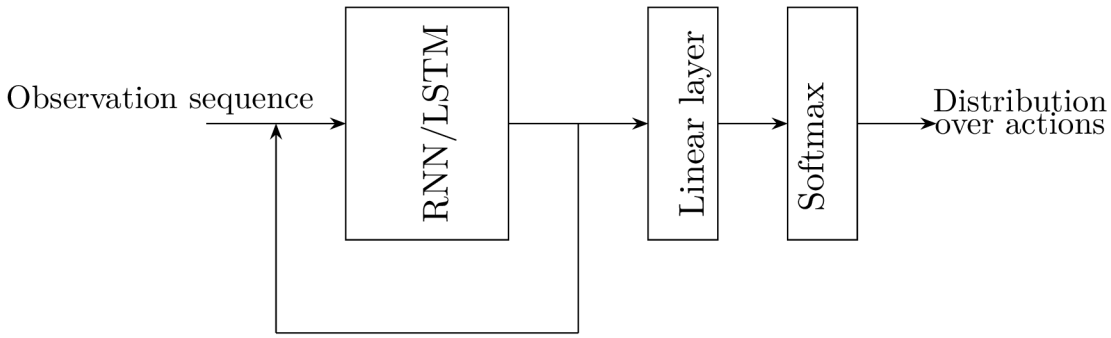



Figure 4.5: Structure of implemented neural network

## 4.4 Training

Training has two phases. Firstly, we need to initialize Q–function, target Q–function and replay buffer. Then, there is an inner cycle. All the initialization is done while the learning agent is initialized, because we call the training function repeatedly until the neural network is trained well enough. In the beginning of the cycle, we choose an action based on $\varepsilon$–greedy policy. We use this action to perform a step. As a result of performing a step, we get a reward and new observation. We append this transaction to replay buffer and update reward of the episode. After that we sample batch of transitions from replay buffer. Then we count target value. The calculation of target value varies on whether we are or are not in the goal state after last step. At the end of the loop, we perform gradient descent step. On every C of steps, we need to update parameters of target Q–function from online Q–function before the loop ends. Now, I will describe each of the steps in more details. Hyperparameters used for training and their values are described in Table 4.1

### 4.4.1 Environment

To be able to train the neural network, we need to define the environment. We use the environment to simulate performing steps (taking actions which leads to transitioning between states). We get all the necessary information, specifically observations and rewards, from the environment. The environment is generated from POMDP file we get as an input to the synthesis. For parsing the input model in POMDP file format and generating environment, I have used Python tool called `Gym-pyro` developed by Andrea Baisero [6]. The environment is created using the `Open AI Gym` library [7].

The environment consists of three spaces: `state_space, action_space, observation_space`, which are initialized from the input POMDP model in my case. Two most important methods, used to move in the environment, are `env.step(action)` and `env.reset()`. The reset

function resets the environment into its initial state. The step function performs steps in the environment based on the action, environments current state and the model. After the step is done, we get observation, action, reward and info. The first three variables are stored in the replay buffer. The info consists of transition distribution, observation distribution and reward based on state (before step is performed) and action.

### 4.4.2 Initialization

At first, we need to initialize Neural network and replay buffer. I have skipped initialization of replay buffer, and instead, I have only defined it as a list. The values to the buffer will be added later during pre–training cycles, which is why I do not see any reason to initialize it now with random values. There are two neural networks that need to be initialized. Weights of both of them are initialized to random uniform values. I also initialize optimizer. As an optimizer I will use Adam.

### 4.4.3 Pre–training phase

Before we start training, we need to run several cycles. Transitions from this cycle are used as initial values of replay buffer. That is why the number of steps in this cycle corresponds to the size of the replay memory. Agent chooses all of the actions within this cycle randomly. The random action is chosen using the command `self.env.action\_space.sample()`.

After the cycle, few more variables have to be defined. One of them is counter, which is used to count steps. We need this information to be able to identify the moment, of when is best to update the target network. We create variable `epsilon` which is used to store probability of selecting the action randomly. There are also several variables which will be used later to store a batch of transitions. They are initialized as zero, but later I found out that it may not be necessary.

### 4.4.4 Training cycle

There are two cycles; the inner cycle and the outer cycle. Each iteration of outer cycle represents one episode. Number of episodes is a multiple of a constant. That is, because the training function is called repeatedly, and we do not know how many times (it varies task from task (and can differ also why running same program repeatedly with same parameters due to the randomness of selecting actions). The inner cycle does not have number of iterations, but it is limited. the maximum number of iterations is derived from number of states of POMDP (is double the number of states). In our case, we do not have defined terminal state in our model, so the condition on ending the inner cycle when the goal state is finished is ignored and we use maximum number of iterations to break the inner cycle. Iterations of the inner cycle represent individual transitions.

In the outer cycle, there is not so much going on. It is a wrapper around the inner cycle. Before getting into the inner cycle, some values (specifically reward and counter of steps of the current episode) are being set to zero.

We spend most of our training time performing iterations in the inner cycle. At first, we increment counters and decrement $\varepsilon$. At first, we need to generate observation–action sequences, which are then used as an input for training neural network. That is done by selecting actions using $\varepsilon$–greedy policy. After that, we perform a step using selected action. We store the transition, which we obtained as a reaction to the step taken. The transition is stored into replay buffer. Each transition consists of previous observation, action that

was taken, reward, flag done (that indicates if we are in the final state; in our case always set to False) and new observation we obtained after performing the step. The cumulative reward of the current episode increases by the value of the obtained reward. If the inner cycle has finished, cumulative reward and number of steps are added to lists.

After this, we generate a batch of random transactions from the replay buffer. This set of transactions is used to perform a gradient descent step. In the original article, this part was done once every several iterations, but I do it after each one. We need to count target values. This is done by equation:

$$y_j = \begin{cases} reward_j & \text{if episode terminates at j+1} \\ reward_j + discount * y_{max} & \text{otherwise} \end{cases}$$

where $y_j$ is target and $y_{max}$ is maximum value we got as a result from target neural network from previous transactions in this batch. Than we need to get value $q_j$. This value is a result from online neural network. We use these two values to count error with the use of loss function.

After we count value of loss function, we have to remove gradients from the last iterations. We do so, by setting them to zero. Now, we can compute gradients and perform gradient descent step, using an optimizer. As the optimizer I use Adam.

After every $n$ iterations, where $n$ represents the update frequency, we have to update the target network with values from online network.

After each training loop (50 episodes) is finished, we save the model in file `model.pt`..

### $\varepsilon$–greedy policy

While deciding which action to use, there are three options. We can select an action, based on a synthesized assignment or a random action or select the best option in the Neural Network available for last observation. Which of these two options to choose, is decided by probability. The probability of which action we choose is set by $\varepsilon$(at first, we select random actions more often and later, we rely more on trained neural network). If the random generated number in range $(0; 1)$ is smaller than $\varepsilon$, we have 50% chance agent performs random action from the action space of the environment and 50% chance agent selects action based on the result of synthesis. Otherwise, the action that seems the best to the agent is taken. The best action from an agent's point of view is selected based on the output from the neural network, while using the last observation as an input.

### Loss function

As a loss function, I used Mean squared error loss function, as well as the authors of the paper that I used as a base of my work. The error is calculated by equation:

$$loss = (y_j - Q(p))^2,$$

where $y_j$ is calculated from the reward, the maximal Q–value for the selected action and $Q(p)$ is an actual output from the neural network, $p$ stands for parameters, which were the same as the ones that were used to calculate the $y_j$.

Hyperparameters I have used for training are described in Table 4.1 with their assigned value.

| | | |
|---|---|---|
| batch size | 32 | Number of transitions to be chosen to compute gradient descent update |
| buffer size | 10000 | Number of pre–training cycles |
| update frequency | 10 | the frequency in which target network is updated |
| epsilon start | 1 | Initial value of $\varepsilon$ |
| epsilon end | 0.1 | Final value of $\varepsilon$ |
| epsilon decrease | 0.000007 | the number subtracted from $\varepsilon$at each step |
| number of episodes | $50 \times k$ | Number of training episodes (excluding pretraining cycles) |
| discount | 0.99 | Used in Q–learning update |

Table 4.1: Hyperparameters used for training

# Chapter 5

# Combination of reinforcement learning and synthesis

The possibilities of using synthesis in combination with machine learning are a matter of current research. One approach is presented by Steven Carr et. al. In Task-Aware Verifiable RNN-Based Policies for Partially Observable Markov Decision Processes [9]. In contrast to the above approach, I have chosen not to deal with memory nodes at all during reinforcement learning, but to focus only on assigning actions to observations. Therefor my approach is significantly different.

## 5.1 Implemented method

I have combined two previously mentioned algorithms, Abstraction refinement and Recurrent Deep Q–learning. The entire process has three phases, which will be described in this chapter. Both algorithms will run in separate processes, for this reason we need to initialize these processes at first. Then they run independently until one of them is finished and after this, the results have to be processed. You can see simplified workflow in following pseudocode.

```
run parallel:
 AR synthesis
 training NN

if synthesis finished:
 terminate
else:
 restrict design space using NN output
 run AR using limited design space
```

First I run the synthesis and training of the neural network in parallel. What happens next depends on which of these two processes ends first. If the synthesis is finished (this happens mainly with smaller models), all possible FSCs in the family are processed and the program can be terminated (or continue with 2-FSCs for incremental synthesis ...). If we manage to train the neural network first (this happens with larger models), the following process is slightly more complex. We use the output of the neural network to restrict the family of FSCs. We do this by removing the possibilities that are less likely to be good and

leaving only those that have a better chance of containing a better contorter. We will then synthesize over the modified FSC family again using the AR method. The different parts of this process are described in more detail in the following sections.

## Preparation

As the method is implemented inside the original method `synthesize_one` originally used for Abstraction refinement method, I had to add initialization of the agent used for training the neural network.

To be able to run both algorithms as two separate processes, I have used Python library `multiprocessing`. As getting variable `satisfying_assigment` from process after termination is a bit complicated, I created only one new process for training, and abstraction refinement synthesizer remained almost the same. I have created a shared list where the output data from the neural network will be stored at the end of the learning. This data will then be used to create a new family for synthesis.

## Synthesis

Synthesis proceeds exactly as stated in Section 3.2, except that it can be aborted early if the learning ends before synthesis. In case, there are no more families to consider before the training of the neural network has finished, the learning process is aborted and this method finishes as if there had never been any training. This happens especially for small models, for which is the use of a neural network too robust. Since the learning algorithm runs in a separate thread, this method is almost comparably as fast as the original AR method (it is only slightly slower).

## Neural network training

The learning loop runs in cycles of fifty episodes. After each cycle, it is evaluated whether or not there has been an improvement compared to the previous fifty episodes. If there has been an improvement, the scheduler is updated according to the current state of the neural network. If there is no improvement, it means that there has been a probable overtraining and further training is pointless. After this occurs, the previous scheduler is returned and the learning process ends.

After each cycle (50 episodes), learning agent updates the assignment according to the current synthesis progress.

## Post–training

There are two possibilities that can occur, either the synthesis finishes as the first one and thus this method is terminated as already said, or the learning of the neural network finishes before the synthesis. We will discuss the second scenario in more detail.

We get Observation based scheduler from neural network. Since the output of the neural network is a probability distribution over actions corresponding to a specific observation that is on the input to this neural network, we select the action that has the highest probability. In some cases, it may happen that this probability is not very high, and it is possible that another action may be more suitable. For example, if the probability of one

action is 49 % and the probability of the other is 51 %, it is reasonable to consider both possibilities. This was a pretty extreme case, usually the difference is a bit wider. In a downturn, when there is a large number of actions, we can consider even more. The key to selecting the number of actions is that the sum of their probabilities exceeds the threshold I set as 0.6. The family is then restricted using obtained scheduler and synthesis is run again, but now with the restricted family.

**Example 4.** *For example, we have two Observations and three actions for each observation. During the training, the rewards stopped increasing and started decreasing, we caught the overtraining. We have a scheduler that we created in the last cycle (at the time of peak rewards before we have caught the drop). At the time of creating the scheduler, we got the following probability distribution over actions as the output of the neural network:*

$$O_0 : P(a_0) = 0.1; P(a_1) = 0.2; P(a_2) = 0.7$$
$$O_1 : P(a_0) = 0.4; P(a_1) = 0.35; P(a_2) = 0.25$$

*As you can see, the best action for the first observation is clearly $a_2$, its probability is bigger than 0.6 which means we can use it. The best action for the second observation is $a_0$, but its probability, according to the neural network, is only 0.4. It is smaller than threshold which is 0.6. This means that we must also take into account the second–best action which is $a_1$. We create family containing one action for the first observation and two possible actions for the second observation. This is a very simple example, which will never occur, because it would be solved very quickly using synthesis. In more complex cases, it is possible to have more than one possible action for one observation, there is no upper bound, except for the sum of probabilities.*

The method of creating list of actions for given observation is following:

```
def best_actions(self, obs):
    obs_tensor = torch.as_tensor(obs, dtype=torch.float32)
    q, _ = self(obs_tensor.unsqueeze(0), torch.zeros(2 ,1,10))
    i = 0.0
    j = 0
    while i < 0.7:
        j += 1
        options = torch.topk(q, j)[1]
        i = torch.topk(q,j)[0].sum()
    return  options.tolist()
```

# Chapter 6

# Evaluation

In this chapter, I will compare the results of a method combining synthesis with reinforcement learning against the results of an abstract refinement method.

This chapter will be divided into several sections. Firstly, the experimental conditions and the models that will be used in the experiments will be described. Following this, the experiments that have been performed with each model will be evaluated.

## 6.1 Experimental setting

All of the experiments were run on Lenovo Thinkpad E490 equipped with Intel®Core$^{TM}$ i7-8565U CPU @ 1.80GHz×8 and 16 GB of RAM. I will compare the results of experiments using a method combining reinforcement learning with the Abstraction Refinement method against the results obtained using only the Abstraction Refinement method, which is implemented in PAYNT.

I will use two different types of model as a benchmark. First, I will use the simple Grid and Maze models. I obtained these models by overwriting the models that were already in PAYNT [2] into Cassandra format. The Maze model was introduced in the paper PAYNT: A Tool for Inductive Synthesis of Probabilistic Programs [2]. Secondly, I will perform experiments using more complicated models Pentagon and IFF obtained from [10].

Since the results using reinforcement learning can differ while using the same input, I run each experiment using reinforcement learning multiple times. I will compare the mean results of the experiments performed using the implemented method with the results of the Abstraction Refinement method and also evaluate how much the results of each experiment using reinforcement learning differ and compare these results with AR method.

By performing experiments I will try to answer the following questions:

- Do we obtain similar results for Maze and Grid models using method combining RL with AR as using the AR method only?
- We will be able to get better results for models Pentagon and IFF using combination of RL with AR than using AR method only?
- Will we be able to get at least as good results for models Pentagon and IFF, but in less time using combination of RL with AR in comparison to AR method only?
- How stable the method using reinforcement learning is?

Parameters of models used for experiments are in Table 6.1. I will give more details to each model in following sections of this text.

| Model | $\|S\|$ | $\|Z\|$ | $\|A\|$ | MDP $\|S\|$ | MDP $\|A\|$ | $\|Designspace\|$ |
|---|---|---|---|---|---|---|
| Grid $4 \times 4$ | 16 | 9 | 4 | 33 | 126 | $10^6$ |
| Grid $10 \times 10$ | 100 | 9 | 4 | 201 | 798 | $10^6$ |
| Maze | 14 | 7 | 4 | 27 | 102 | $10^5$ |
| Pentagon | 212 | 28 | 4 | 5646 | 22578 | $10^{16}$ |
| IFF | 104 | 22 | 4 | 668 | 2666 | $10^{13}$ |

Table 6.1: Benchmark models. $|S|$ stands for number of states in POMDP, $|Z|$ stands for number of observations in POMDP, $|A|$ stands for number of actions in POMDP , MDP $|S|$ stands for number of states of underlying MDP after determination, $|A|$ stands for number of actions in underlying MDP after determination and $|Designspace|$ is size of the design space. All of the values are for 1-FSCs, in case of 2-FSCs . . . some of the values are bigger.

## 6.2 Grid

The grid is one of the simplest navigation problems modeled as POMDP. If the grids size is $n \times n$, the POMDP has $n^2$ states, always have 4 actions and 9 observations. The actions define the directions the agent can move (north, south, east, west). The probability of successfully moving into the neighboring state equals one, except for impossible directions. If the agent tries to get into the impossible direction (out of the grid), the agent stays in the current state. The initial state is set by uniform distribution over all states, except the goal state. The goal state is the last state (in bottom right corner) and after reaching this state, the next state is not selected by action, but is random (one from all possible states except the goal). You can see an example of $4 \times 4$ grid in Figure 6.1. The reward in each state is -0.1, except for the goal state which has reward one.

As you can see in Table 6.2, the method using reinforcement learning is a bit slower than the abstraction refinement method. This was expected in this case, because this POMDP is easy to solve using only AR in almost no time, but training the neural network takes time even while using small environment. The method including combination of the reinforcement learning and the synthesis in this case did not use any outputs from the reinforcement learning, because the AR method finished a lot quicker, than we were able to obtain any useful data from the learning. For this reason are the results of both methods the same.

| Model | Abstraction refinement | | AR + RL | |
|---|---|---|---|---|
| | Max Reward | Duration [s] | Max Reward | Duration [s] |
| Grid $4 \times 4$ | 2.62 | 0.01 | 2.62 | 0.04 |
| Grid $10 \times 10$ | -1.29 | 0.06 | -1.29 | 0.07 |

Table 6.2: Grid experimental results

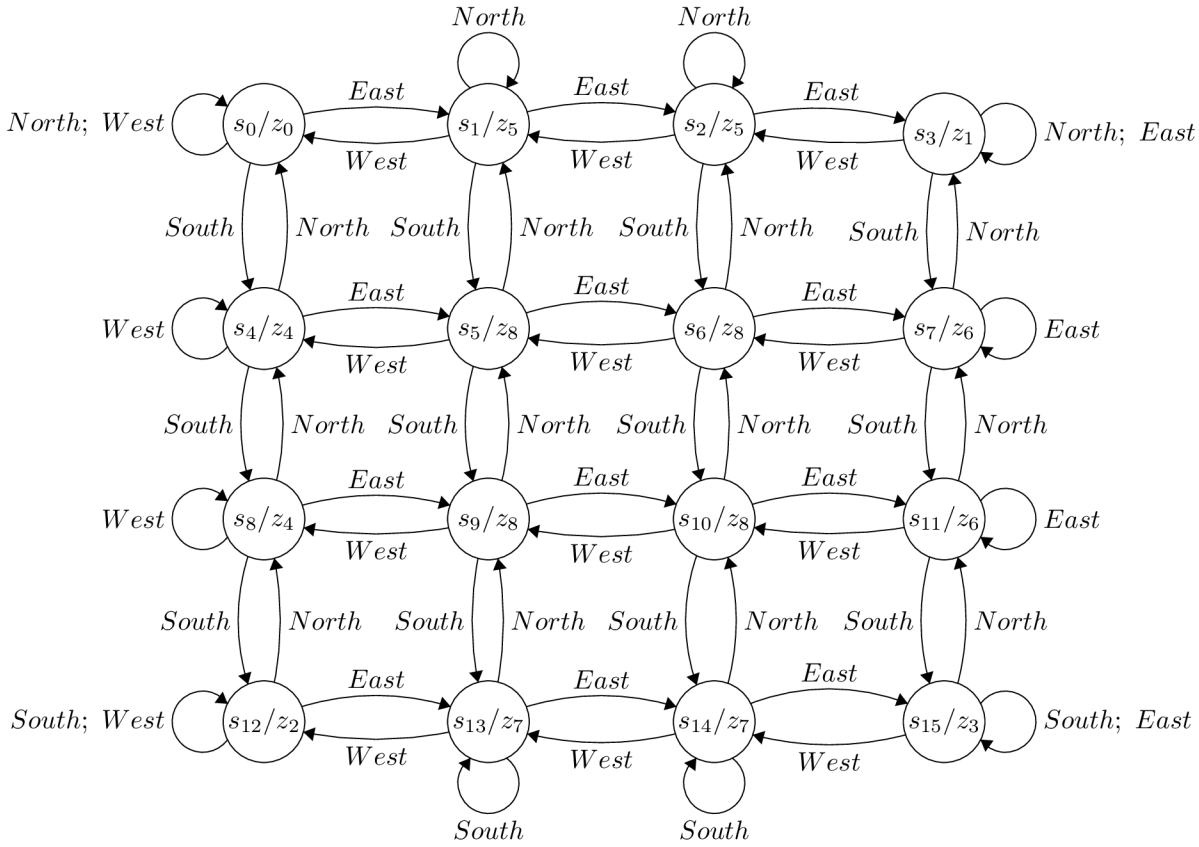



Figure 6.1: Grid

### 3-FSC

I have also performed experiments creating 3-FSC. As you can see the reward is the same using both methods same as in previous case using 1-FSC. That is because the AR method was able to find optimal 3-FSC really fast. The difference is in times needed for the program to terminata. The technique combining Abstraction Refinemenet with Reinforcement Learning sometimes terminates faster. The results are shown in Table 6.3. All experiment using both methods results in finding the same optimal reward (2.62 for grid $4 \times 4$ and -1.29 for grid $10 \times 10$), for this reason the table shows only time needed for the experiment to terminate.

| Model | AR | AR + RL | | | | | AR + RL mean |
|---|---|---|---|---|---|---|---|
| Grid $4 \times 4$ | 14395 | 31.36 | 33.62 | 60.06 | 106.16 | 29.7 | 52.18 |
| Grid $10 \times 10$ | 14395 | 14397 | 14396 | 14396 | 707.41 | 148.13 | 8808.91 |
| Termination | timeout | timeout | timeout | timeout | finished | finished | - |

Table 6.3: Grid 3-FSC experimental results

As you can see in Table 6.3, the Abstraction Refinement method terminated always due to timeout. While using reinforcement learning in addition to Abstraction Refinement, the synthesis in some cases terminates before the set timeout.

All experiment using grid $4 \times 4$ as an input terminated a lot faster using combination of synthesis with learning. Four out of five experiments terminated in about a minute. The experiment that took the most time ended after 106 seconds. The mean time of run using combination of RL an AR methods is 52.18 seconds which is less than a minute, whereas while not using reinforcement learning, the process terminates due to timeout after four hours.

For the bigger grid $10 \times 10$, is the difference between used methods not that obvious. The combination of synthesis and learning terminates in only two out of five before the timeout, but in this two cases, the time difference is significant. The mean time when using the combination of RL and AR in the performed experiments is less than two and a half hours, which is a bit over half of the time than in the case of using only the AR method only, which ends after four hours with a timeout.

As expected this model is too small for reinforcement learning to be useful. The AR method was in all cases able to give us optimal controller really fast. The reason why the 3-FSC results are not better than the 1-FSC results is that the model is so simple that we are able to obtain the optimal 1-FSC. For example if the goal state is $s_0$, we optimally should take action north in all states besides the ones on the top where we need to go west. We do not need any memory to do this so we are able to get optimal 1-FSC.

## 6.3 Maze

The Maze is similar to the Grid mentioned previously. You can see the visualization of the Maze in Figure 6.2. It consists of 13 states, where $s_13$ is goal state and $s_0$ and $s_{10}$ are unwanted states. There are 4 actions, same as in the Grid, and 6 observations.

I have performed a bit different experiments using this model than using grid. I run 1 experiment using AR method and 5 experiments using combination of AR with RL. In these experiments I was doing incremental synthesis which means 1-FSC synthesis is run first, than the experiment continues with 2-FSC .... Unlike the grid here, we are able to

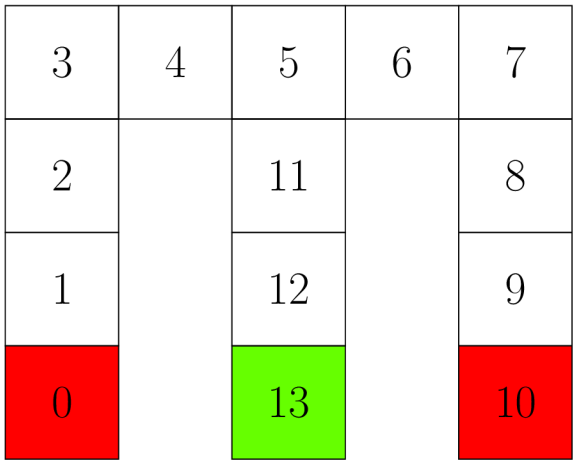



Figure 6.2: Maze

get 2-FSC, which is better than the best possible 1-FSC. In all experiments we were able to find 1-FSC with the best reward -0.4 in about 0.05 seconds. The best found k-FSC was 2-FSC in all of the experiments and that gave us reward 1.231. The difference in the time to find the optimal 2-FSC was 0.39 in the case of the AR method and about 0.48 in the case of the combination of AR and RL. The result is very similar to the grid results. More precise results of individual experiments are in table 6.4. The timoeout was set to 4 hours so I was able to explore k-FSC for $k > 2$, but were not able to find any better controller.

| Maze 2-FSC | AR | AR + RL | | | | | AR + RL mean |
|---|---|---|---|---|---|---|---|
| Duration [s] | 0.39 | 0.45 | 0.56 | 0.43 | 0.44 | 0.52 | 0.48 |

Table 6.4: Maze 2-FSC experimental results

## 6.4 Pentagon

Pentagon is larger navigation problem in compare to Grid. It consists of 212 states, 4 actions and 28 observations. This model is noisy, which means the transitions between states are partially random. The observations are also partially random. Due to these facts this model is a lot more difficult to solve and the induced MDP is large consisting of 5646 states and 22578 actions. The size of the design space is $10^{16}$ for 1-FSC and $10^{49}$ for 2-FSC. I have performed two different kinds of experiment constructing 1-FSC and 2-FSC which will be discussed in following sections.

### 1-FSC

Results of performed experiments constructing 1-FSC are consistent. The results are in Table 6.5. Max rewards are either -1.585 (same as using AR method only) or 0.0 (which is optimal reward, because rewards can be zero or negative). Three out of five experiments two of them were able to find 1-FSC with reward 0. The mean reward is -0.634 which is by 0.951 better then using AR method only. The mean time is slightly higher in case of using combination of AR and RL method, that is because most of the experiments were killed

by operating system probably because of lack of memory (including experiment using AR method only). You can see the results also in graph 6.3. The result of second experiment using combination of AR + RL methods is not visible in the graph, that is because it terminated too quickly.

| Pentagon 1-FSC | AR | AR + RL | | | | | AR + RL mean |
|---|---|---|---|---|---|---|---|
| Max Reward | -1.585 | 0.0 | 0.0 | -1.585 | 0.0 | -1.585 | -0.634 |
| Duration [s] | 9590 | 14360 | 14373 | 11452 | 452.47 | 11284 | 10384.3 |
| $t_0$ | 6953 | 6849 | 2978 | 8865 | 25 | 8852 | - |
| Termination | memory out | timeout | timeout | memory out | finished | memory out | - |

Table 6.5: Pentagon 1-FSC experimental results

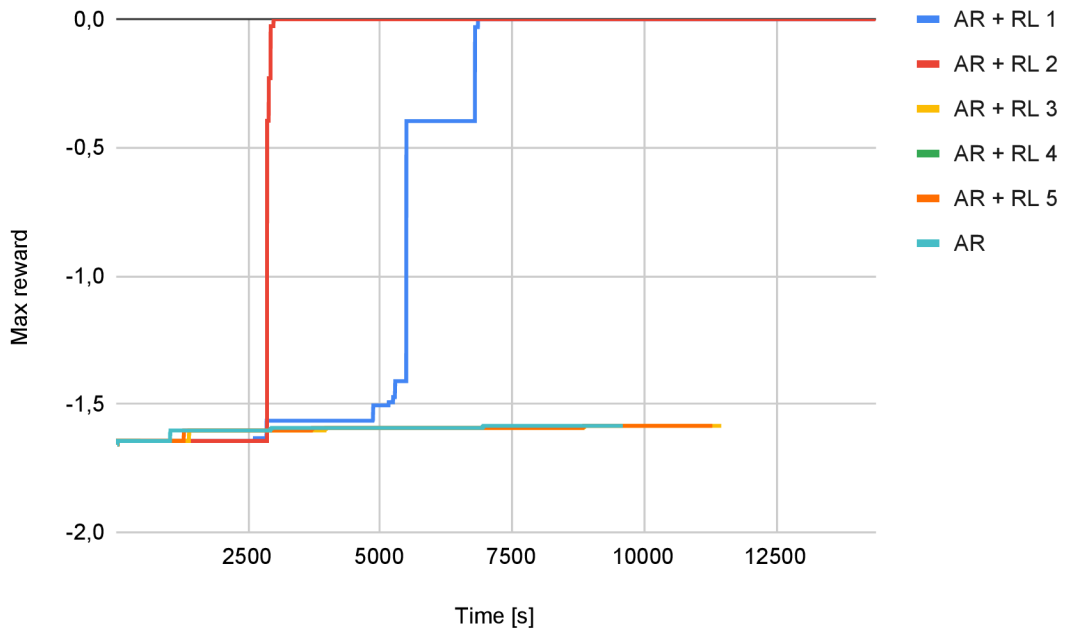



Figure 6.3: Graph comparing maximal rewards of each Pentagon experiment constructing 1-FSC, $t_0$ is time needed to find the best found FSC

The reason why using RL in combination with the AR method gave us better results is that by just using AR we ran out of memory before we were able to explore all possible controllers, in the case of not running out of memory it is possible that we would run out of time before we would be able to go through all possibilities and even with more memory we have no guarantee that we are able to find a better controller in a given time. By using machine learning I am able to reduce the number of controllers by those that are less likely to be good, I can find a more optimal controller faster. So it is possible (and from some experimental results this has been confirmed) that I may not go through all the possibilities, but I am able to find a better controller without running out of memory and in less time.

### 2-FSC

Results of experiments constructing 2-FSC are also pretty consistent. They are either -1.704 of -0.393. In three out of five cases is the best found reward -1.704 which is the same

as using AR method only. In other two cases the result is -0.393 which is better. The mean reward of experiment using combination of AR and RL method is -1.108. The mean time is almost the same as using AR method. This experiments were also terminated due to lack of memory as in the case of the constructing 1-FSC (in this case all of the experiments terminated for this reason).

| Pentagon 2-FSC | AR | AR + RL | | | | | AR + RL mean |
|---|---|---|---|---|---|---|---|
| Max Reward | -1.704 | -0.393 | -1.704 | -1.704 | -0.393 | -1.704 | -1.180 |
| Duration [s] | 7826 | 5799 | 9279 | 10191 | 5143 | 9459 | 7974.2 |
| $t_0$ | 207 | 564 | 153 | 189 | 3302 | 207 | - |

Table 6.6: Pentagon 2-FSC experimental results, $t_0$ is time needed to find best found FSC

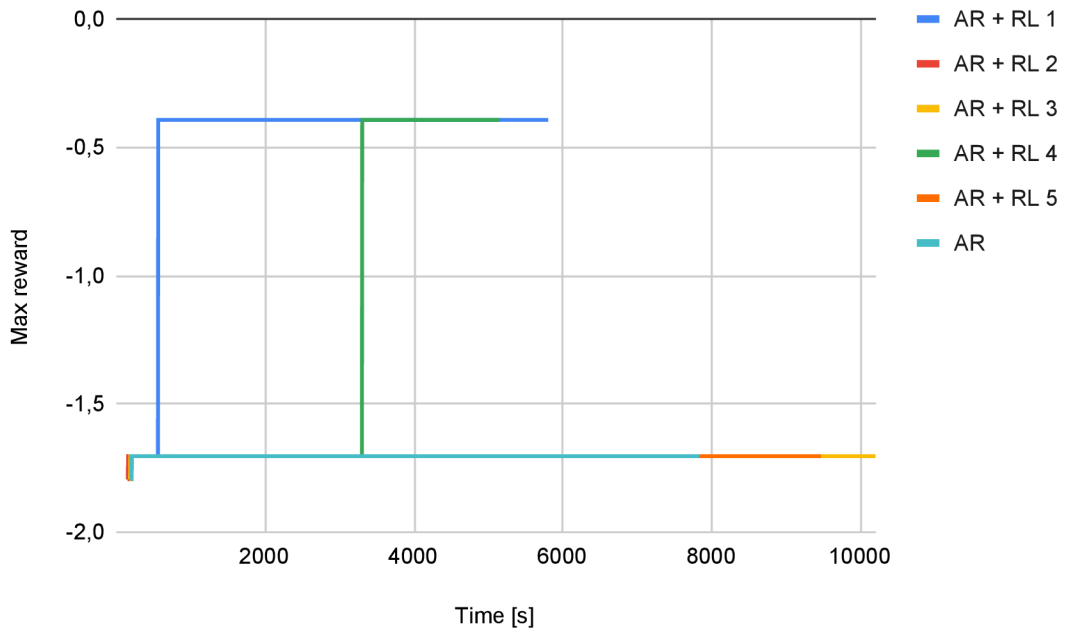



Figure 6.4: Graph comparing maximal rewards of each Pentagon experiment constructing 2-FSC

As you can see from the results, we were not able to find a better 2-FSC than 1-FSC. This is due to the fact that 2-FSC has a significantly larger design space ($10^{49}$ while 1-FSC is $10^{16}$). For this reason, we can go through a significantly smaller part of the possibilities before we run out of memory.

## 6.5 IFF

The IFF (Identifidation Friend or Foe) is an aircraft identification problem. Model I have used consists of 104 states, 4 actions and 22 possible observations. I have split this section into two parts, firstly I constructed only 1-FSC. Secondly, I was construction k-FSC, where $k$ increases from 1 to bigger number. both parts contain a table with the results of exper-

iments, a graph of the process of experiments and the evaluation of experiments for this model.

### 1-FSC

As you can see in 6.7, results of each experiment differ. All of the maximal computed rewards are above two. The mean reward using combination of AR and RL method is 2.5096 which is by 0.2656 more thabn using only AR method. Only one out of five of the experiments using AR + RL method gives worse results, than using only AR method. In one case the result is same and other three cases the results were better. The best result of these experiments was 2.822 and the worst was 2.196.

The durations of the experiments also differ. As is shown in graph 6.5, two experiments terminated a way faster than others and gave us the best results. One experiment terminated in only 231 seconds and that is the one with the worst results (reward 2.196 while the reference reward is 2.244, but in four hours. The second experiment has the same duration as the reference experiment, therefore we do not really see it in the graph. The last experiment is most of the time same as the reference experiment, but its results near the end slightly outperform and the experiment terminates a bit earlier giving us better results.

| IFF 1-FSC | AR | AR + RL | | | | | AR + RL mean |
|---|---|---|---|---|---|---|---|
| Max Reward | 2.244 | 2.976 | 2.244 | 2.822 | 2.196 | 2.311 | 2.5096 |
| Duration [s] | 14397 | 1191.16 | 9353.65 | 1196.84 | 231.19 | 14396 | 5273.77 |
| $t_0$ | 303 | 772 | 348 | 762 | 72 | 12916 | - |

Table 6.7: IFF 1-FSC experimental results, $t_0$ is time needed to find best found FSC

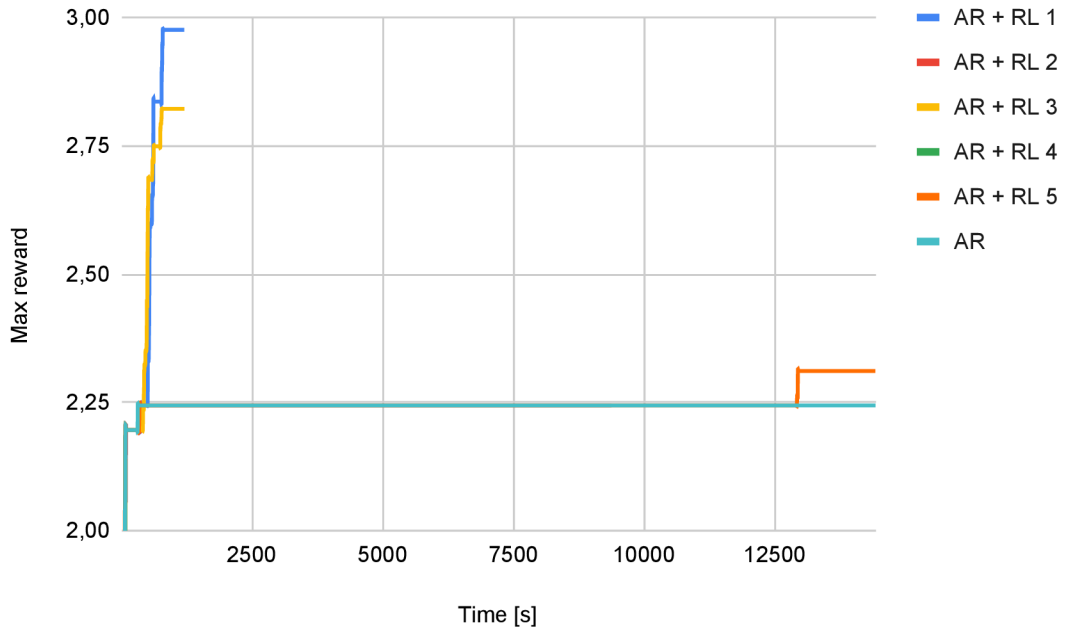



Figure 6.5: Graph comparing maximal rewards of each IFF experiment constructing 1-FSC

### Incremental FSC

The results where the $k$ in $k - FSC$ is increasing are very similar to results of constructing only 1-FSC. In the experiment all of the best computed rewards only 1-FSC. As in the previous experiments, the method combining AR and RL has one worse, one equal and three better results than the AR method. But one of the better ones is only slightly better and the other two are not as good as the best result in the previous experiments. As you can see in Table 6.8, three of the experiments started computing 2-FSC, but in given time (four hours including solving 1-FSC) were not able to find better results, the other two experiments terminated in four hours while solving 1-FSC same as reference experiment using AR method.

Since the synthesis process for 1-FSC takes a long time, we were not able to get any better results for incremental FSC synthesis. While using the AR method only, we were not even able to finish 1-FSC synthesis thus there is no difference in results in comparison to 1-FSC synthesis. The method using RL is faster so we were in three out of five experiments able to begin 2-FSC synthesis, but did not have enough time to get any better results.

In the graph 6.6 you can see plotted duration of the experiments is similar to 1-FSC. The main difference is in this case all of the experiments terminates due to timeout, because when 1-FSC synthesis is completed, 2-FSC synthesis begins.

| IFF incremental FSC | AR | AR + RL | | | | AR + RL mean |
|---|---|---|---|---|---|---|
| Max Reward | 2.244 | 2.244 | 2.714 | 2.714 | 2.291 | 2.196 | 2.4318 |
| $t_0$ | 291 | 330 | 7469 | 9361 | 369 | 3 | - |

Table 6.8: IFF incremental FSC experimental results, $t_0$ is time needed to find best found FSC

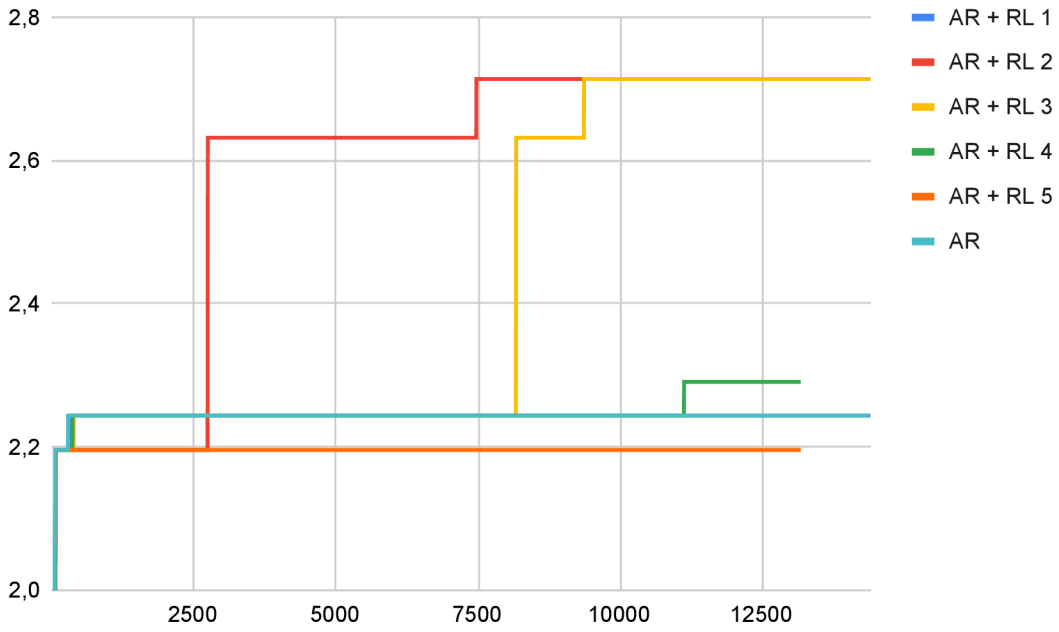



Figure 6.6: Graph comparing maximal rewards of IFF experiments constructing incremental FSC

# Chapter 7

# Conclusion

The aim of this theses was to combine inductive controller synthesis with reinforcement learning method. I have implemented the combination of these two techniques in the Python tool PAYNT.

The implemented method contains two individual processes that run in parallel. One is recurrent deep Q–learning and the other one is the abstraction refinement synthesis method. The preliminary results of the synthesis are used for training the neural network. After the neural network is good enough, the learning is finished. Based on the output of the neural network, an Observation-Based scheduler is created, which is then used to constrain the family of Finite State Controllers. The synthesis is then run again to optimize the Finite State Controller, but now only on the restricted family.

A number of experiments comparing this method with the abstraction refinement method was performed. The experiments were performed on several different models. In the case of using the Grid and Maze models, the method I have implemented had the same results as the AR method, however my method was slightly slower. Although that was to be expected since the model was too small to be able to achieve training the neural network before the syntesis was finished.

I have also performed several experiments using larger models – Pentagon and IFF. These models are noisy and have significantly larger design space. The size of the design space for 1-FSC synthesis of the Pentagon model is $10^{16}$ and for 2-FSC it is $10^{49}$. The synthesis using the AR method was not able to finish due to a lack of memory. I was able to get better 1-FSC using my method in three out of five cases. While constructing 2-FSC for the Pentagon model I was able to get better results in two out of five cases, however I was not able to finish the 2-FSC synthesis, for the lack of memory, with neither one of the methods. The resutls of the IFF model were similar to the results of the Pentagon. The size of the IFF design space is $10^{13}$. Since the IFF model has a slightly smaller size of the design space than the Pentagon model, I did not have problems with memory. The results of my method in comparison to the AR method were once worse, once the same and in three cases better. The mean value of the rewards of the experiments performed using my method were about 2.5, while using the AR method the reward was about 2.244.

From the results of performed experiments, we can see that the combination of the reinforcement learnign method with Abstraction Refinement method does not outperform Abstraction Refinement method for constructing FSCs within smaller models. I was able to get better results for more complicated models in shorter time than the AR method. In the future, it could be interesting to try out different model sizes and find the boundaries

of the usefulness of using reinforcement learning. It would also be useful to improve the stability of reinforcement learning.

# Bibliography


[1] *Part 2: Kinds of RL Algorithms - Spinning Up documentation* [online]. 2018 [cit. 2024-05-03]. Available at: https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html#a-taxonomy-of-rl-algorithms.

[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 for Probabilistic Programs Reaches New Horizons. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer International Publishing, 2021, p. 191–209. Lecture Notes in Computer Science. DOI: 10.1007/978-3-030-72016-2_11. ISBN 978-3-030-72015-5. Available at: https://www.fit.vut.cz/research/publication/12498.

[4] Andriushchenko, R., Češka, M., Junges, S. and Katoen, J.-P. Inductive synthesis of finite-state controllers for POMDPs. In: Cussens, J. and Zhang, K., ed. *Proceedings of the Thirty-Eighth Conference on Uncertainty in Artificial Intelligence*. PMLR, 01–05 Aug 2022, vol. 180, p. 85–95. Proceedings of Machine Learning Research. Available at: https://proceedings.mlr.press/v180/andriushchenko22a.html.

[5] Ansel, J., Yang, E., He, H., Gimelshein, N., Jain, A. et al. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In: *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM, April 2024. DOI: 10.1145/3620665.3640366. Available at: https://pytorch.org/assets/pytorch2-2.pdf.

[6] Baisero, A. *Gym-pyro* [online]. GitHub, 2024 [cit. 2024-05-05]. Available at: https://github.com/abaisero/gym-pyro.

[7] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J. et al. *OpenAI Gym*. 2016.

[8] Caceres, P. *The Recurrent Neural Network - Theory and Implementation of the Elman Network and LSTM* [online]. 2020-04-16 [cit. 2024-05-05]. Available at: https://pabloinsente.github.io/the-recurrent-net.

[9] 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. Available at: https://doi.org/10.1613/jair.1.12963.

[10] Cassandra, A. R. *POMDP Example Domains* [online]. [cit. 2024-06-05]. Available at: https://www.pomdp.org/code/pomdp-file-spec.html.

[11] Cassandra, A. R. *POMDP File Format* [online]. [cit. 2024-05-05]. Available at: https://www.pomdp.org/code/pomdp-file-spec.html.

[12] Cassandra, A. R. A survey of POMDP applications. In: *Working notes of AAAI 1998 fall symposium on planning with partially observable Markov decision processes*. 1998, vol. 1724.

[13] Cassandra, A., Kaelbling, L. and Kurien, J. Acting under uncertainty: discrete Bayesian models for mobile-robot navigation. In: *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems. IROS '96*. 1996, vol. 2, p. 963–972 vol.2. DOI: 10.1109/IROS.1996.571080.

[14] Češka, M., Hensel, C., Junges, S. and Katoen, J.-P. Counterexample-Driven Synthesis for Probabilistic Program Sketches. In: Beek, M. H. ter, McIver, A. and Oliveira, J. N., ed. *Formal Methods – The Next 30 Years*. Cham: Springer International Publishing, 2019, p. 101–120. ISBN 978-3-030-30942-8.

[15] Češka, M., Jansen, N., Junges, S. and Katoen, J.-P. Shepherding Hordes of Markov Chains. In: Vojnar, T. and Zhang, L., ed. *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2019, p. 172–190. ISBN 978-3-030-17465-1.

[16] Chen, C. Deep Q-Learning with Recurrent Neural Networks. In:. 2016. Available at: https://api.semanticscholar.org/CorpusID:29201646.

[17] Egorov, M. Deep Reinforcement Learning with POMDPs. In:. 2015. Available at: https://api.semanticscholar.org/CorpusID:15312806.

[18] Elman, J. L. Finding Structure in Time. *Cognitive Science*. 1990, vol. 14, no. 2, p. 179–211. DOI: https://doi.org/10.1207/s15516709cog1402_1. Available at: https://onlinelibrary.wiley.com/doi/abs/10.1207/s15516709cog1402_1.

[19] Hausknecht, M. and Stone, P. *Deep Recurrent Q-Learning for Partially Observable MDPs*. 2017.

[20] 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*. august 2022, vol. 24, no. 4, p. 589–610.

[21] Hopfield, J. J. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*. 1982, vol. 79, no. 8, p. 2554–2558. DOI: 10.1073/pnas.79.8.2554. Available at: https://www.pnas.org/doi/abs/10.1073/pnas.79.8.2554.

[22] Junges, S., Jansen, N., Wimmer, R., Quatmann, T., Winterer, L. et al. Permissive Finite-State Controllers of POMDPs using Parameter Synthesis. *CoRR*. 2017, abs/1710.10294. Available at: http://arxiv.org/abs/1710.10294.

[23] 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.

[24] Littman, M., Cassandra, A. and Kaelbling, L. Learning policies for partially observable environments: Scaling up. *California*. february 1970. DOI: 10.1016/B978-1-55860-377-6.50052-9.

[25] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J. et al. Human-level control through deep reinforcement learning. *Nature*. february 2015, vol. 518, no. 7540, p. 529–533.

[26] MYKEL J. KOCHENDERFER, K. H. W. Algorithms for Decision Making. In:. Cambridge, Massachusetts, London, England: MIT PRESS, 2022, chap. 19, p. 379–405. ISBN 9780262047012. Available at: https://algorithmsbook.com/files/chapter-19.pdf.

[27] MYKEL J. KOCHENDERFER, K. H. W. Algorithms for Decision Making. In:. Cambridge, Massachusetts, London, England: MIT PRESS, 2022, chap. 15-18, p. 299–375. ISBN 9780262047012. Available at: https://algorithmsbook.com/#.

[28] PyTorch, C. *RNN* [online]. 2023 [cit. 2024-05-05]. Available at: https://pytorch.org/docs/stable/generated/torch.nn.RNN.html#rnn.

[29] Sak, H., Senior, A. W. and Beaufays, F. Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition. *CoRR*. 2014, abs/1402.1128. Available at: http://arxiv.org/abs/1402.1128.

[30] Xiang, X. and Foo, S. Recent Advances in Deep Reinforcement Learning Applications for Solving Partially Observable Markov Decision Processes (POMDP) Problems: Part 1—Fundamentals and Applications in Games, Robotics and Natural Language Processing. *Machine Learning and Knowledge Extraction*. july 2021, vol. 3, p. 554–581. DOI: 10.3390/make3030029.