SAPIENZA
UNIVERSITÀ DI ROMA

Sapienza University of Rome

Department of Computer, Control and Management Engineering
PhD in Engineering in Computer Science

Thesis For The Degree Of Doctor Of Philosophy

Discovering Logical Knowledge in Non-Symbolic Domains

Thesis Advisor
Prof. Giuseppe De Giacomo

Co-Advisor
Prof. Roberto Capobianco

Candidate
Elena Umili
1491499

Academic Year 2021/2022 (XXXV cycle)

To my dad

Abstract

Deep learning and symbolic artificial intelligence remain the two main paradigms in Artificial Intelligence (AI), each presenting their own strengths and weaknesses. Artificial agents should integrate both of these aspects of AI in order to show general intelligence and solve complex problems in real-world scenarios; similarly to how humans use both the analytical left side and the intuitive right side of their brain in their lives. However, one of the main obstacles hindering this integration is the Symbol Grounding Problem [146], which is the capacity to map physical world observations to a set of symbols.

In this thesis, we combine symbolic reasoning and deep learning in order to better represent and reason with abstract knowledge. In particular, we focus on solving non-symbolic-state Reinforcement Learning environments using a symbolic logical domain. We consider different configurations: (i) unknown knowledge of both the symbol grounding function and the symbolic logical domain, (ii) unknown knowledge of the symbol grounding function and prior knowledge of the domain, (iii) imperfect knowledge of the symbols grounding function and unknown knowledge of the domain. We develop algorithms and neural network architectures that are general enough to be applied to different kinds of environments, which we test on both continuous-state control problems and image-based environments. Specifically, we develop two kinds of architectures: one for Markovian RL tasks and one for non-Markovian RL domains. The first is based on model-based RL and representation learning, and is inspired by the substantial prior work in state abstraction for RL [116]. The second is mainly based on recurrent neural networks and continuous relaxations of temporal logic domains. In particular, the first approach extracts a symbolic STRIPS-like abstraction for control problems. For the second approach, we explore connections between recurrent neural networks and finite state machines, and we define *Visual Reward Machines*, an extension to non-symbolic domains of Reward Machines [27], which are a popular approach to non-Markovian RL tasks.

Keywords: Neurosymbolic AI, Deep Learning, Symbolic AI, Reinforcement Learning, Deep Reinforcement Learning.

Contents

List of Figures vi

List of Tables x

1 Introduction 1
- 1.1 Contributions . . . 3
- 1.2 Thesis Organization . . . 5
- 1.3 Publications . . . 6

I Preliminaries 7

2 Background 8
- 2.1 Symbolic AI . . . 8
  - 2.1.1 Propositional logic . . . 8
  - 2.1.2 First Order Logic . . . 8
  - 2.1.3 How to define logic temporal properties . . . 9
- 2.2 Deep Reinforcement Learning . . . 12
  - 2.2.1 Markov Decision Processes and Reinforcement Learning . . . 12
  - 2.2.2 Abstractions for MDP . . . 13
  - 2.2.3 Neural networks . . . 17
- 2.3 Neurosymbolic integration . . . 18
  - 2.3.1 Deep Learning stepping in the white side: discrete neural networks . . . 19
  - 2.3.2 Symbolic Logic stepping in the dark side: continuous logic . . . 20

3 State of the art 22
- 3.1 Discovering an abstract model for markovian tasks . . . 22
  - 3.1.1 Finite partitioning . . . 23
  - 3.1.2 Continuous encoding . . . 24
  - 3.1.3 Symbol grounding in Reinforcement learning . . . 27
  - 3.1.4 Symbol grounding in NeSy AI . . . 29
- 3.2 Discovering temporal logic knowledge from non-markovian tasks . . . 32
  - 3.2.1 LTL and non-markovian RL . . . 32

II Discovering logical knowledge in markovian non-symbolic domains 35

4 STRIPS-Like Symbolic Abstractions for Control RL Problems 36
- 4.1 Problem Setting . . . 37
- 4.2 The Interaction with The Environment . . . 39
- 4.3 Learning the Models . . . 41
  - 4.3.1 Symbol Grounder . . . 41

4.3.2 Symbolic Transition Model . . . 41
4.3.3 Symbolic Value Model . . . 42
4.3.4 Training Neural Networks with Symbolic or Discrete Layers . . . 42
4.3.5 Value Model Training . . . 43
4.3.6 Transition Model Training . . . 44
4.4 Action Selection . . . 44
4.5 Transfer Learning . . . 45
4.6 Experiments . . . 46
4.6.1 Environments Description . . . 46
4.6.2 Setup . . . 48
4.6.3 Symbol-Set-Size Tuning . . . 48
4.6.4 Training Performance . . . 49
4.6.5 Planning Performance . . . 49
4.6.6 Transfer the Symbolic Domain to New Tasks . . . 50
4.6.7 Representation Insights . . . 51
4.7 Related Work . . . 55
4.8 Discussion . . . 57

III Discovering logical knowledge in non markovian non-symbolic domains 58

5 Symbol Grounding Exploiting LTLf Knowledge 60
5.1 Problem Formulation . . . 61
5.2 Framework . . . 62
5.3 Definition and Examples of Groundability through a Temporal Property . . . 65
5.4 Experiments . . . 67
5.4.1 Dataset . . . 67
5.4.2 Results on MNIST Dataset . . . 68
5.4.3 Discussion on 'Groundability' . . . 69
5.5 Related works . . . 69
5.6 Discussion . . . 76

6 DFA Induction with Neural Networks 78
6.1 Method . . . 80
6.1.1 DeepDFA Definition . . . 80
6.1.2 Temperature Annealing . . . 81
6.1.3 Model Minimization . . . 81
6.2 Experimental Evaluation . . . 81
6.2.1 Target DFAs . . . 81
6.2.2 Dataset . . . 82
6.2.3 Baselines . . . 82
6.2.4 Training Details . . . 83
6.2.5 Results . . . 84
6.2.6 Ablation Study: the Effect of Changing the State Space Size . . . 87
6.3 Related Work . . . 87
6.4 Discussion . . . 88

7 Visual Reward Machines 89
7.1 Non Markovian RL Tasks . . . 90
7.1.1 Minecraft Environment Example . . . 90
7.2 Framework Specifics . . . 91

7.3 VRM as an Extension of DeepDFA . . . 91
7.3.1 From DeepDFA to DeepMooreMachine . . . 91
7.3.2 Embedding Uncertainty over Symbols . . . 92
7.4 Visual Reward Machine Definition . . . 93
7.5 Visual Reward Machine Implementation with NN . . . 93
7.5.1 Reasoning and Learning with VRM . . . 94
7.6 Experiments . . . 94
7.6.1 Offline symbol grounding . . . 95
7.6.2 Reinforcement Learning and online grounding . . . 96
7.6.3 Comparisons with chapter 5 . . . 97
7.6.4 Learning the machine from imperfectly grounded symbols . . . 98
7.6.5 Learning All End-to-End . . . 98
7.7 Discussion . . . 99

IV Conclusions **100**

8 Conclusions **101**
8.1 Summary of Contributions . . . 101
8.2 Future Research . . . 102

Bibliography **104**

A Appendix **119**

List of Figures

2.1 Different formalisms for specifying a temporal behaviour: a) an example of LTLf formula with the corresponding equivalent Deterministic Finite Automaton, b) matrix representation of a Probabilistic Finite Automaton 10
2.2 example of the three categories: left) finite partitioning, center) continuous encoding, right) symbol grounding 14
2.3 example of abstraction: ϕ is the abstraction and f_ϕ is the function induced by the abstraction. Their composition $f_\phi(\phi(\cdot))$ should be a similar as possible to the target function $f(\cdot)$ to have a good representation performance. 15
2.4 a) Architecture of a multilayer neural network. b) Multilayer NN prediction scheme. c) Recurrent prediction scheme. d) A recurrent neural network unfolded. 18
2.5 Some techniques to discretize the output of a neural network in a safe way a) Straight Through estimator c) A Gumbel-Softmax-activated variational autoencoder 19
2.6 An example of Logic Tensor Network, taken from [16]. In this example two variables (x and y), one function (f) and one predicate (p) are represented. Variables x and y are grounded to the available data (the values v_1, v_2 and v_3 for variable x and the values w_1 and w_2 for variable y). The rightmost tensor in the image, $\mathcal{G}(p(x, f(x, y)))$, is the grounding for the binary predicate p when interpreted over x and $f(x, y)$. This is grounded by applying the continuous module p to the grounding of x and the grounding of $f(x, y)$; which is in turn grounded applying the f module to the groundings of x and y. 20

3.1 Discovery of logical knowledge from raw domains 23
3.2 An example of state aggregation satisfying the bisimulation preperty. This 4-states deterministic ground MDP can be partitioned in 2 abstract states without loosing the capability to predict the next reward and abstract state. . 24
3.3 Aligned versus misaligned abstractions: 25
3.4 Training scheme of the discrete world model used in [88] 26
3.5 LatPlan framework from [7] 27
3.6 Some examples of graph representing the state space structure for different domains. They represent the input of algorithm described in [22] 28
3.7 An example of grounding for symbols Z, Z_1, in a robot navigation problem, from [102] 29
3.8 Framework from [39] for grounding digits from high level labels and prior symbolic knowledge through abduction 30
3.9 Categorization of NeSy approaches 31
3.10 A schematic view of Restraining Bolts [74] 32
3.11 Categorization of works on non-markovian RL 33

4.1 A global view on the framework 39
4.2 Continuous codes of a random batch of states with increasing margin. On the y axis: x_1 values between 0 and 1; on the x axis: x_0 values between 0 and 1 . 42

4.3 a) the symbolic quality model (b) computation of the transition model loss . . 43
4.4 Performance with different sizes of the symbol set on the Acrobot environment. 47
4.5 Training performances on Cartpole (a), Acrobot (b), Lunar Lnader (c) and NAO Soccer Player (d). On the y: axis mean training reward; on the x axis: episodes. Solid lines represent mean values, shaded areas represent standard deviations. 48
4.6 Planner performances: Figures show rewards obtained with monitor replanning using the planner acquired through experience with different planning horizons respectively without and with uncertainty estimation, for the Acrobot environment, (a) and (b), for the Cartpole enviornment (c) and (d), and for the NAO robot (e). 50
4.7 Transfer learning to a new task in the Lunar Lander domain. 50
4.8 a) Acrobot environment b)Distances from the final state during an episode of Acrobot: distances are computed respectively in the original MDP state-space (top) and in the abstract representation space (bottom). On the y axis: L1 distance from goal; on the x axis: timesteps. c) Soccer player NAO robot. d) ball-distance from the NAO position computed in the orginal state-space (left) and in the abstract-space (right), smaller values shown in brighter red and bigger values shown in darker red. 51
4.9 Comparison of symbol grounding in the Cartpole environment: (left) symbol grounding obtained with our algorithm, (right) symbol grounding obtained if we train only the symbolic encoder and the Q function, and we interact with the environment without planning, namely choosing in each state the action maximizing the Q function 53
4.10 **Cartpole symbolic representation:** a) distances in the symbolic space from the symbolic code associated with (0,0,0,0); b) different codes, any different code is shown in a random shade of red; c) action to take according to the optimal policy; d) truth value of each symbol in the state space. 53
4.11 **Acrobot symbolic representation:** a) distances in the symbolic space from the symbolic code associated with (0,0,0,0,0,0); b) different codes, any different code is shown in a random shade of red; c) action to take according to the optimal policy; d) Heatmap of the weights assigned to each symbol by the policy network, each column is a symbol, green represent a weight near to 0, yellow an average weight and red an high weight 54
4.12 **NAO symbolic representation:** a) distances in the symbolic space from the symbolic code associated with (0,0); b) different codes, any different code is shown in a random shade of red; c) action to take according to the optimal policy; d) truth value of each symbol in the state space. 54

5.1 When the LTLf symbols are not grounded in observation data the knowledge of the formula is useless to the task 61
5.2 a) An example of LTLf formula with the corresponding equivalent automaton, b) our framework 62
5.3 Pyramid example. (a) Drawing of the pyramid. (b) DFA corresponding to the instructions to build the pyramid. Bricks are not all groundable through the pyramid instructions, in fact brick 1 and 2 can be confused each other. Expected symbol grounding accuracy: 100% or 33% 65
5.4 Gate example. (a) Drawing of the gate. (b) DFA corresponding to the instructions to build the gate. Bricks are all not groundable through the gate instructions, in fact brick 0 and 1 can be confused each other if bricks 2 and 3 are confused each other. Expected symbol grounding accuracy: 100% or 0% . 66

5.5 Experiments over 20 Declare formulas. In the first row: **sequence classification** accuracy, in the second row: **image classification** accuracy. They are obtained by training on three different datasets: (first columns) **complete dataset**, (second column) **restricted dataset**, (third column) complete dataset with **non-mutually exclusive symbols**. Solid lines represent mean values, shaded areas represent standard deviations. 68
5.6 Experiments over 20 Declare constraints training on the **full dataset** in **mutually exclusive symbols**. On the y axis: **sequence classification** accuracy ; on the x axis: epochs of training. Solid lines represent mean values, shaded areas represent standard deviations. 70
5.7 Experiments over 20 Declare constraints training on a **restricted dataset** in **mutually exclusive symbols** setting. On the y axis: **sequence classification** accuracy ; on the x axis: epochs of training. Solid lines represent mean values, shaded areas represent standard deviations. 71
5.8 Experiments over 20 Declare constraints training on the **full dataset** in **non mutually exclusive symbols** setting. On the y axis: **image classification** accuracy ; on the x axis: epochs of training. Solid lines represent mean values, shaded areas represent standard deviations. 72
5.9 Experiments over 20 Declare constraints training on a **restricted dataset** in **mutually exclusive symbols** setting. On the y axis: **image classification** accuracy ; on the x axis: epochs of training. Solid lines represent mean values, shaded areas represent standard deviations. 73
5.10 Experiments over 20 Declare constraints training on the **full dataset** in **non mutually exclusive symbols** setting. On the y axis: **sequence classification** accuracy ; on the x axis: epochs of training. Solid lines represent mean values, shaded areas represent standard deviations. 74
5.11 Experiments over 20 Declare constraints training on the **full dataset** in **non mutually exclusive symbols** setting. On the y axis: **image classification** accuracy ; on the x axis: epochs of training. Solid lines represent mean values, shaded areas represent standard deviations. 75

6.1 a) An example of PFA with three states and two symbols: graph describing the PFA, equivalent representation in matrix form, and produced states and acceptance probabilities while processing the string "ab". b) An exmple of DFA: graph describing the PFA, equivalent representation in matrix form, and produced states and acceptance probabilities while processing the string "ab". In particular, the DFA in (b) is obtained by the PFA in (a) approximing the matrix representation to the closest one-hot vectors. 79
6.2 Results on Tomita 5 with different error rates in the training dataset. a) Number of states of predicted DFAs b) Test accuracy 83
6.3 Results obtained varying the hidden state size hyperparameter with Tomita5 (a-b) and with with a random DFA of size 20 and alphabet size 3 (c-d). For each hidden state size we do 10 experiments. 86

7.1 a) Visual Minecraft environment. b) task specification as Reward Machine . . 90
7.2 Implementation of a visual reward machine for the Visual Minecraft environment 94
7.3 Results obtained by exploiting the reward machine structure to learn the symbol grounding function offline for the Visual Minecraft environment. a) Visual Reward Machine accuracy over sequences. b) Symbol grounding accuracy over single images 95

7.4 Results obtained by exploiting the reward machine structure to learn the symbol grounding function for the Visual Minecraft environment. a) Training rewards Right) Symbol grounding accuracy over single images 97
7.5 Learning DFA from traces composed of imperfectly grounded symbols: results on the 7 Tomita languages 98

A.1 List of Declare formulas as in [49]. We tested on all except last(a). Meaning of modal operators symbols:◯=X, ◊=F, □=G 120

List of Tables

4.1 Percentage of different symbolic states found in the analyzed environments . . 55

6.1 Comparison between DeepDFA, L* extraction and DFA-inductor on the Tomita Languages. We report test accuracy, mean number of states $|\hat{Q}|$ and the number of parameters used $\#W$ 83
6.2 Comparison between DeepDFA and a DFA-inductor when the training set contains 1% of errors. Results on Tomita Languages. 84
6.3 Comparison between DeepDFA and a DFA-inductor when the training set contains 1% of errors. Results on random DFAs. 84
6.4 Comparison between DeepDFA and L* extraction on randomly generated DFAs. The train set does not contain errors. 85
6.5 Comparison between DeepDFA and a SAT-based DFAinductor. The train set does not contain errors. For each DFA we keep the experiment achieving best dev accuracy. 85

Chapter 1

Introduction

Deep learning [113] and symbolic AI [137] correspond to the two main paradigms in current artificial intelligence (AI). Deep learning is based on the use of artificial neural networks (NN), which are modeled and inspired by the human brain. It is generally employed to recognize patterns in large datasets and to classify, predict, and process data. On the other hand, Symbolic AI is based on the idea that knowledge can be represented as symbols, which can then be manipulated and reasoned with to solve problems. It is used for tasks that require logical reasoning and problem-solving, such as planning and discrete decision-making.

Despite the enormous progress made in these two areas, however, machines are still a long way from general intelligence [33], which seems to be a heritage of only our species. The human being, however, has not always been the same, and its intelligence has evolved over millions of years. In particular, the modern human, Homo Sapiens, appeared 'only' around 50'000 years ago during the Upper Paleolithic, which is considered the big bang of human culture, exhibiting more innovation in thousands of years than in the previous six million years of human evolution. Researchers in many different areas, such as psychology, anthropology, archeology, and cognitive science, have so far investigated what led to the "explosion" of our intelligence. Many agree that this time coincides with the birth of complex symbolic language and reflects the onset of the capacity to internally represent complex, abstract, internally coherent systems of meaning using symbols. Other theories suggest that homo sapiens was the first capable of *cognitive fluidity*, namely the ability to shrink or expand the field of attention to favor effortful logical thinking or fast and intuitive thinking depending on the situation [65].

Regarding artificial intelligence, Deep Learning, particularly Deep Reinforcement Learning (DRL) [151][124], deals with the first part of evolution. It allows artificial agents to develop basic cognitive skills, perceive the environment, interact with it, and eventually, other agents. But it is not concerned with constructing an abstract, composite and reusable representation of the environment like the symbolic ones used for general planning. Since most of the representations used for deep learning are vectorized, sparse, distributed, and don't transfer to different tasks or data. Classical symbolic AI, on the other hand, completely avoids the problem of abstraction, and deals only with how to combine logical symbols respecting rules in a handcrafted prior model, but not with how this model can be constructed or adapted to the experienced world. As a result, DRL can solve mainly strategically simple problems

on raw high-dimensional observations and low-level actions. On the other hand, symbolic planning can solve strategically complex problems only once they have been deprived of all their perceptual complexity. Neither can solve strategically complex tasks in a raw and realistic environment.

Neurosymbolic artificial intelligence (in short NeSy AI) [20][43] is the area that mixes machine learning, particularly deep learning, with classical symbolic reasoning. NeSy AI is an attempt to reach that kind of fluid intelligence characterizing the human mind, where induction, deduction, perception, and reasoning are all perfectly integrated into a single agent brain. The most crucial problem for integrating symbolic reasoning and deep perception is the so-called symbol grounding problem.

Symbol grounding is the process of connecting symbolic representations with the physical world [146][145]. Ideally, an intelligent agent should be able to imagine new facts on a mental plane (like reasoning on a knowledge base), grounding concepts and mental abstractions in the perception of real objects in the physical plane, and use both these two levels of knowledge to decide how to act in the physical world. However, this smooth interface between the physical and the logical plane is very hard to implement in artificial agents. Although symbol grounding has been of interest to AI researchers for many decades, it continues to be one of the most important challenges in AI and Cognitive Science today.

In this thesis, we will focus on solving *non-symbolic problems* by extracting *a symbolic logical domain* from it. In other words, we force the agent to create a symbolic abstraction of its environment, and we test its capability to move and act in the physical environment by 'thinking' only in this symbolic model. For this reason, we will concentrate mainly on systems based on this conceptual division in: (i) symbolic world, (ii) subsymbolic world. Although the agent performs the task in the continuous subsymbolic world, we assume there is a latent symbolic representation that can describe the task in a significantly more compact and functional way. Therefore we will review existing methods and describe new models and algorithms, all based on this two-layered view.

This view of the world is more common in NeSy AI, which generally addresses logic domains presenting non-symbolic observations, such as math operations on images [120] and visual logic games such as sudoku [167][14]. However, we will apply this conceptual division within Reinforcement Learning tasks, ranging from control problems to visual tasks in both markovian and non-markovian settings. Although this is pretty unusual, we took inspiration from the huge prior work on state abstraction and representation learning in the scope of RL tasks [116]. The concept of abstraction is really close to that of symbol grounding. RL algorithms have been shown to benefit highly from a wise and abstract representation of the environment learned end-to-end [128] [2]. For this reason, we deem that (continuous or symbolic) abstractions should be able to *emerge* from experience [140].

Based on this division, the task in the real world can be viewed as the combination of two processes: a symbolic logical process that alters the symbolic configuration of the world, and a rendering or emission function that produces the observation that we see in the world [58]. The emission function is the inverse of the symbol grounding function. To uncover the latent logical process in the task, we must therefore identify two functions: (i) a symbol grounding function that maps world observations to the truth values of a finite set of propositional

symbols, (ii) a logical domain that regulates the activity of the symbols.

Uncovering both of these two functions from end-to-end data is extremely challenging, and very few works exist on the subject [12][41]. Most prior works learn only one function by leveraging prior knowledge of the other. For example, some works learn the logical model by already knowing the grounding function [9][153]. In particular, this can be accomplished by directly leveraging labels on symbolic abstraction (which is the simplest case) or by using unsupervised learning techniques, such as clustering, autoencoding, and factorization, which may be adapted to extract discrete features [97]. Another approach is learning the grounding function by leveraging the knowledge of the symbolic logical model. This is mainly used in Nesy AI tasks [42] [54] [174] [16] [120] [176] [171] [39] [95] [154] and is very rarely applied to RL tasks [104].

In this thesis, we present several works on this line. In particular, the thesis first addresses Markovian RL problems and then moves on to non-Markovian problems. The markovian property applies to all those environments where the next state environment outcome depends on the current state and action only, and does not depend on past states and actions. Although this formulation is general enough to model most decision-making problems, it has been observed that many natural tasks are non-Markovian [117]. Solving non-markovian tasks is much harder, since the agent has to consider the entire history of states and actions to make decisions in the environment correctly.

In particular, research on non-Markovian Reinforcement Learning (RL) tends to be more closely connected to symbolic logic. Since defining reward functions for non-Markovian tasks is not straightforward, these tasks are generally defined using formalisms based on automata or Linear Temporal Logic (LTL) [45][27]. Although this solves the problem for finite and discrete domains, using this type of formal symbolic language in non-symbolic tasks requires grounding symbolic abstractions in environment observations. The vast majority of works assume prior knowledge of the grounding function, also known as the labeling function [45][27][67][175][135]. We will explore how this assumption can be removed through the use of neurosymbolic frameworks.

1.1 Contributions

The main contributions presented in the thesis are summarized here.

The first contribution is defining a novel neural network framework to extract a symbolic planning model from a continuous state-space Markov Decision Process. The approach we propose naturally combines interaction, symbolic representation learning, and symbolic online planning. Our system leverages experience-data gained from the environment to autonomously learn a symbol grounding function and a symbolic planning model composed of: (1) a symbolic transition model; (2) a quality function for symbolic states. This model is used at training time to lead the interaction with the world. At each interaction step, we perform fast symbolic online planning over a finite horizon to choose the action to execute in the environment. Let us notice the success of this strategy in the environment implicitly validates our automatically extracted symbolic model, since the system is able to effectively solve the original continuous-state MDP by reasoning only in the finite and symbolic domain.

We evaluate on several OpenAI gym environments and one robotic scenario, successfully addressing different types of control problems. Interestingly, the generated symbol grounding function is very task-related and gives some precious insights. Furthermore, we use the symbol grounding function to shape the reward so as to handle different tasks in the environment.

While this contribution handles the discovery of logical knowledge for markovian tasks, the others focus o non-markovian tasks. In particular, we present three on this topic.

First, we concentrate on grounding the symbols of LTL specifications in image data by exploiting prior knowledge of the formula and a set of image sequences labeled as accepted or not accepted by the formula. The proposed approach consists in translating the formula in an equivalent Finite Deterministic Automaton (DFA) and then interpreting the latter in fuzzy logic, basically transforming it in a recurrent Logic Tensor Network (LTN) [16]. We use this fuzzy model to train the weights of a Convolutional Neural Network (CNN) to maximize the formula's satisfaction when evaluated on the available data. Experiments show that using the discovered symbol grounding function and the given LTLf formula to evaluate the sequences at test time outperforms deep learning models trained only with the data. This demonstrates that formula knowledge can speed up the classification of image sequences, even if we do not know how to recognize the formula symbols in the images. Furthermore, we obtained a very high image classification accuracy without exploiting any image label. This happens for the majority of the formulas except for a few cases. The latter's formulas are not specific enough to impose the correct grounding. We formally define the concept of 'groundability,' specifying in which conditions supervision on the formula output is not supposed to bring the symbol grounder to learn the correct grounding. The second contribution related to temporally extended (or non-markovian) tasks concentrates instead on the induction of the symbolic model from sequences of already grounded symbols. We define a novel recurrent neural network architecture, *DeepDFA*, that we use to learn a DFA from a set of labeled traces. Unlike usual Recurrent Neural Networks (RNN), this model is completely explicable after training. We obtain this feature by defining the RNN as a Probabilistic Finite Automaton where we can control how much the model is stochastic through a temperature parameter. During training, we smoothly decrease the temperature to drive the model to become closer and closer to a DFA. When the temperature is low enough, we extract from the neural model activations a crispy DFA model and use it to classify test sequences. We compared this model with a state-of-the-art algorithm to extract a DFA from a pre-trained RNN [169] and a state-of-the-art logic induction method based on SAT solvers [177]. Results show that DeepDFA is more accurate and faster than both, especially on big-size target DFAs. Furthermore, it is robust to erroneous labels, maintaining top accuracy with up to 15% of errors in the training dataset. At the same time, the SAT-based method completely fails to identify the correct DFA even with 1% of errors.

Finally, the last contribution of the thesis is the definition of *Visual Reward Machines*, a NeSy framework to embed both symbol grounding and DFA induction in a single system. The framework is based on an extension of DeepDFA to embed probabilistic beliefs on symbol truth values. Thanks to this extension, we integrate the previous model with a neural symbol grounder in an end-to-end fully neural system composed of two trainable models: a CNN performing symbols grounding and a specialized RNN performing DFA induction. We can

initialize the system to represent a specific machine and/or grounding function if we know this information. In this case, it will behave exactly as a deterministic machine. Although, in case of missing information in the specifics, we can learn the unknown models from data through back-propagation. Let us notice the system could, in principle, also learn both the symbol grounding *and* the automaton simultaneously from only data. However, experiments on this learning configuration are not promising. In most cases, the system overfits training data, since it can oversimplify the grounding so to oversimplify the machine structure. However, this is reasonable given the weak supervision given to the system. However, we successfully tested other learning configurations. In particular, the system can ground quite a big set of symbols when trained on long sequences exploiting the knowledge of the DFA. Furthermore, it outperforms logic DFA induction methods when trained on symbols grounded by an *imperfect* symbol grounder.

1.2 Thesis Organization

The remainder of the thesis is organized as follows.

- Part I: Preliminaries
  - in Chapter 2, we provide the background knowledge necessary to understand the rest of the thesis. In particular, we give some preliminary concepts related to Symbolic AI, Deep Learning and Deep Reinforcement Learning, and Neurosymbolic AI.
  - In Chapter 3, we review the existing literature on the discovery of logical domains from nonsymbolic environments; specifically, we will explore works related to symbol grounding in Reinforcement Learning abstraction, NeSy AI, and non-markovian RL.
- Part II: Discovering logical knowledge in markovian nonsymbolic domains
  - In Chapter 4, we describe in detail the first contribution of this thesis, which regards the learning of a symbolic planning domain from control RL environments.
- Part III: Discovering logical knowledge in non-markovian nonsymbolic domains
  - In Chapter 5, we discuss how we defined a learning framework to exploit LTLf specifications in a visual sequential classification task, representing our second contribution.
  - in Chapter 6, we explain in detail The definition and results of DeepDFA, our novel recurrent neural model for DFA induction.
  - In Chapter 7, we define Visual Reward Machines, and we apply it to a non-markovian RL task specification with missing information.
- Part IV: Conclusions
  - In Chapter 8, we conclude the thesis with final discussions and directions for future research

1.3 Publications

The contributions reported in Chapters 4, 5 and 7 of the thesis produced the following publications respectively

- Elena Umili, Emanuele Antonioni, Francesco Riccio, Roberto Capobianco, Daniele Nardi, Giuseppe De Giacomo. *Learning a Symbolic Planning Domain through the Interaction with Continuous Environments.* Workshop on Bridging the Gap Between AI Planning and Reinforcement Learning (PRL). 2021.

- Elena Umili, Roberto Capobianco, Giuseppe De Giacomo. *Grounding LTLf Specifications in Images.* In Proceedings of the 16th International Workshop on Neural-Symbolic Learning and Reasoning as part of the 2nd International Joint Conference on Learning & Reasoning (IJCLR). 2022.

- Elena Umili, Francesco Argenziano, Aymeric Barbin, Roberto Capobianco. *Visual Reward Machines.* In Proceedings of the 17th International Workshop on Neural-Symbolic Learning and Reasoning. 2023.

Some results of Chapter 5 and results of Chapter 6 of the thesis are in papers currently under review

- Elena Umili, Roberto Capobianco, Giuseppe De Giacomo. *Grounding LTLf Specifications in Image sequences.* (Submitted to international conference). 2023.

- Elena Umili, Roberto Capobianco. *DeepDFA: a transparent neural network design for DFA induction.* (Submitted to international conference). 2023.

Other publications

- Elena Umili, Marco Tognon, Dario Sanalitro, Giuseppe Oriolo, Antonio Franchi. *Communication-based and Communication-less approaches for Robust Cooperative Planning in Construction with a Team of UAVs.* In Proceedings of the 2020 International Conference on Unmanned Aircraft Systems (ICUAS). 2020.

Part I

Preliminaries

Chapter 2

Background

In this chapter, I introduce the background knowledge necessary to understand the rest of the thesis. In particular, I first present some elements of symbolic AI, with a particular focus on Linear Temporal Logic and its interpretation over finite traces and links with Deterministic Finite Automaton (DFA). Then, I review the key elements and concepts of Reinforcement Learning (RL), abstraction in reinforcement learning, and Deep Learning (DL). Finally, I conclude by giving an overview of neurosymbolic (NeSy) AI, focusing primarily on the symbol grounding problem and describing in detail the NeSy frameworks most closely related to the work described in this thesis.

2.1 Symbolic AI

2.1.1 Propositional logic

Propositional logic (PL) is the simplest form of logic where all the statements are made by propositions. A proposition is a declarative statement which is either true or false. In propositional logic, we use symbolic variables to represent the logic, and we can use any symbol for representing a proposition. The syntax of propositional logic defines the allowable sentences for the knowledge representation. There are two types of Propositions: (i) atomic propositions, (ii) compound propositions. The former consist of a single proposition symbol. The latter are constructed by combining simpler or atomic propositions, using parenthesis and logical connectives. There are five connectives ($\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$), that are called respectively: negation, conjunction, disjunction, implication and biconditional.

2.1.2 First Order Logic

In propositional logic, we can only represent the facts, which are either true or false, therefore PL is not sufficient to represent complex sentences or natural language statements. For this motivation First Order Logic (FOL) was born. FOL is an extention of PL, that develops information about the objects in a more easy way and can also express the relationship between those objects. First Order Logic assumes the following things in the world

- **Objects**: they denote things we want to speak about, such as people, numbers, colors, squares, pits, locations, etc.

- **Relations** on objects: these can be either unary relations that specify a property of an object, such as is_red, is_round, is_on_table, or n-any relations specifying a property that correlates n objects, such as sister_of, father_of, has_color, etc; relations can be either true or false, as example father_of(Jhon, Mary) = true describes the fact that "Jhon is the father of Mary".
- **Functions**: a function takes as input a certain number n of objects as arguments and returns one object as output, like for example father_of(Mary) = Jhon describes the fact that "Jhon is the father of Mary", but in a procedural way.

The basic syntactic elements of FOL are the following types of symbols: constants (to refer to constant objects), variables, predicates (to denote relations between objects), functions, connectives ($\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$), equality ($==$) and quantifiers ($\forall, \exists$).

2.1.3 How to define logic temporal properties

Linear Temporal Logic

Linear Temporal Logic (LTL) [133] is a language which extends traditional propositional logic with modal operators. With the latter we can specify rules that must hold *through time*. Given a set P of propositions, the syntax for constructing an LTL formula ϕ is given by

$$\phi ::= \top \mid \bot \mid p \mid \phi \mid \phi_1 \wedge \phi_2 \mid X\phi \mid \phi_1 U \phi_2 \tag{2.1}$$

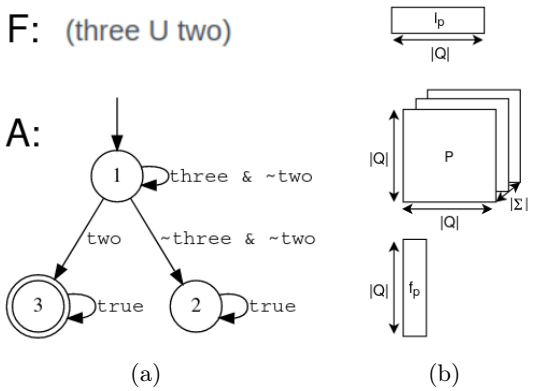
where $p \in P$. We use $\top$ and $\bot$ to denote true and false respectively. X (Next) and U (Until) are temporal operators. Other temporal operators are: N (Weak Next) and R (Release) respectively, defined as $N\phi \equiv \neg X \neg \phi$ and $\phi_1 R \phi_2 \equiv \neg(\neg\phi_1 U \neg\phi_2)$; G (globally) $G\phi \equiv \bot R \phi$ and F (eventually) $F\phi \equiv \top U \phi$. A trace $\rho = \rho[0], \rho[1], ..$ is a sequence of propositional assignments, where $\rho[x] \in 2^P$ $(x \geq 0)$ is the x-th point of ρ. Intuitively, $\rho[x]$ is the set of propositions that are true at instant x. Additionally, $|\rho|$ represents the length of ρ. In this thesis, we will focus mainly on LTL interpreted over finite traces (LTLf) [46]. Such interpretation allows the executions of arbitrarily long traces, but not infinite, and is adequate for finite-horizon planning problems.

Given a finite trace ρ, $|\rho| < \infty$, and an LTLf formula ϕ, we inductively define when ϕ is true for ρ at point x $(0 \leq x < |\rho|)$, written $\rho, x \vDash \phi$, as follows [133]:

$$\begin{aligned}
&\rho, x \vDash \top \text{ and } \rho, x \nvDash \bot;\\
&\rho, x \vDash p \text{ iff } p \in \rho[x];\\
&\rho, x \vDash \neg\phi \text{ iff } \rho, x \nvDash \phi;\\
&\rho, x \vDash \phi_1 \wedge \phi_2, \text{ iff } \rho, x \vDash \phi_1 \text{ and } \rho, x \vDash \phi_2;\\
&\rho, x \vDash X\phi, \text{ iff } x + 1 < |\rho| \text{ and } \rho, x + 1 \vDash \phi;\\
&\rho, x \vDash \phi 1 U \phi 2, \text{ iff there exists y such that } x \leq y < |\rho| \text{ and } \rho, y \vDash \phi_2, \text{ and for all } z, x \leq z < y,\\
&\quad \text{we have } \rho, z \vDash \phi_1.
\end{aligned} \tag{2.2}$$

An LTLf formula ϕ is true in ρ, denoted by $\rho \vDash \phi$, when $\rho, 0 \vDash \phi$.

Any LTLf formula ϕ can be translated in an equivalent Deterministic Finite Automaton



(a) (b)

Figure 2.1: Different formalisms for specifying a temporal behaviour: a) an example of LTLf formula with the corresponding equivalent Deterministic Finite Automaton, b) matrix representation of a Probabilistic Finite Automaton

(DFA) $A_\phi = (2^P, Q, q_0, \delta_t, F)$, where 2^P is the automaton alphabet, Q is the set of states, $q_0 \in Q$ is the initial state, $\delta_t : Q \times 2^P \to Q$ is the transition function and $F \subseteq Q$ is the set of final states. Let be $L(A_\phi)$ the language composed by all the strings accepted by the A_ϕ we have

$$\rho \vDash \phi \text{ iff } \rho \in L(A_\phi) \tag{2.3}$$

Despite the size of A_ϕ is double-exponential in ϕ in the worst-case [46], A_ϕ is often quite small in practice, and scalable techniques are available for computing it from ϕ [180] [18] [73].

LTLf formulas are widely used in Business Process Management (BPM). In particular, the BPM community has selected 21 types of formulas that are particularly significant for describing complex processes declaratively [131]. The latter are at the base of the system Declare [130] and they generate DFAs that are polynomial in the original formula [170]. In the rest of this section I will formally define deterministic and probabilistic finite automata, because of their correlations with LTLf formulas.

Deterministic Finite Automata

A Deterministic Finite Automaton (DFA) A is a tuple (P, Q, q_0, δ_t, F), where P is the alphabet, Q is the set of states, $q_0 \in Q$ is the initial state, $\delta_t : Q \times P \to Q$ is the transition function, and $F \subseteq Q$ is the set of final states. Let be P^* the set of all finite strings over P, and ϵ the empty string. The transition functions over strings $\delta_t^* : Q \times P^* \to Q$ is

$$\begin{aligned} &\delta_t^*(q, \epsilon) = q \\ &\delta_t^*(q, ax) = \delta_t^*(\delta_t(q, a), x) \end{aligned} \tag{2.4}$$

Where $a \in P$ is a symbol and $x \in P^*$ is a string, and ax is the concatenation of a and x. A accepts the string x, and we say that x is in the language of A, $L(A)$, if and only if $\delta_t^*(q_0, x) \in F$. Let be $x = x[0]x[1]...x[l-1]$ the input string, where $x[i]$ is the ith character in the string, we denote as $q = q[0]q[1]...q[l]$ the sequence of states visited by the automaton while processing the string, namely $q[0] = q_0$ and $q[i] = \delta(q[i-1], x[i-1])$ for all $i > 0$.

Moore Machines

A Moore Machine is an extension of DFA that models a finite *output* function. More formally, a Moore Machine is a tuple $(P, Q, O, q_0, \delta_t, \delta_o)$, where P is the input alphabet, Q is the set of states, O is the output alphabet $q_0 \in Q$ is the initial state, $\delta_t : P \to Q$ is the transition function, and $\delta_o : Q \to O$ is the output function. Moore machines belong to the family of *transducers*. A transducer T receives in input a string $x = x[0]x[1]...x[l-1]$ composed with symbols in the input alphabet P and returns an output string $y = y[1]...y[l]$ of symbols in the output alphabet O. In particular, in Moore Machines, the output at time t, $y[t]$ only depends on the state at time t, $q[t]$. More formally

$$y[t] = \delta_o(q[t]) \tag{2.5}$$

We can consider DFAs as Moore Machines with a binary output alphabet $O = \{Acc, Rej\}$.

Probabilistic Finite Automata

A Probabilistic Finite Automaton (PFA) A_p is a tuple $(P, Q, i_p, \delta_{tp}, f_p)$, where P is the alphabet, Q is the finite set of states, $i_p : Q \to [0, 1]$ is the probability for a state to be an initial state, $\delta_{tp} : Q \times P \times Q \to [0, 1]$ is the transition probability function, and $f_p : Q \to [0, 1]$ is the probability of a state to be final. We have therefore $\sum_{q' \in Q} \delta_{tp}(q, p, q') = 1$, and $\sum_{q \in Q} i(q) = 1$ $\forall q \in Q$, $\forall a \in P$.

We can also represent the PFA in matrix form as a *transition matrix* M_t, an *input vector* v_i and an *output vector* v_o. Matrix $M_t \in \mathbb{R}^{|P| \times |Q| \times |Q|}$ contains at index (p, q, q') the value of $\delta_{tp}(q, p, q')$. We denote as $M_t[p] \in \mathbb{R}^{|Q| \times |Q|}$ the 2D transition matrix for symbol p.

The input vector $v_i \in \mathbb{R}^{1 \times |Q|}$ contains at index k the probability of state q_k to be an initial state, while the output vector $v_o \in \mathbb{R}^{|Q| \times 1}$ has in position k the probability of state q_k to be accepting. This matrix representation is shown in Figure 2.1(b).

Given a string $x = x[0]x[1]...x[l-1]$, we denote as $q_{p,0}, q_{p,1}...q_{p,l}$ the sequence of probabilities to visit a certain state, where $q_{p,t} \in \mathbb{R}^{1 \times |Q|}$ is a row vector containing at position k the probability to stay in state k at time t.

$$\begin{aligned} q_{p,0} &= v_i \\ q_{p,t} &= q_{p,t-1} \times M_t[x[t]] \qquad \forall t > 0 \end{aligned} \tag{2.6}$$

The probability of being in a final state at time t is the inner product $q_{p,t} \times v_o$.

Therefore the probability of a string to be accepted is the probability to be in a final state in the last computed state $q_{p,l}$, and it is calculated as follows

$$v_i \times M_t[x[0]] \times M_t[x[1]] \times ... \times M_t[x[l-1]] \times v_o \tag{2.7}$$

2.2 Deep Reinforcement Learning

2.2.1 Markov Decision Processes and Reinforcement Learning

Reinforcement learning (RL) [151] is the area of machine learning related to learning from experience. An agent is supposed to learn how to optimally act in an environment to tackle a task by iterating a process of trial end error. The agent and environment interaction is formally modeled as a Markov Decision Process (MDP). An MDP is a tuple (S, A, t, r, γ). S is the set of environment *states*; A is the set of agent's *actions*; t is the *transition function*, it specifies how the agent actions affect the environment; $r : S \times A \to \mathbb{R}$ is the *reward function*, it specifies an utility value for performing a specific action a in a given state s; and $\gamma \in [0, 1]$ is the *discount factor*, it expresses a preference for immediate over future reward. The state and action spaces can be either finite or infinite, discrete or continuous. The transition function can be either deterministic ($t : S \times A \to S$) or stochastic ($t = P(s'|s, a) : S \times A \times S \to [0, 1]$).

In this setting, transitions and rewards are assumed to be Markovian – i.e., they are functions of the current state only. The agent decisions in the environment are represented through a policy $\pi : S \to A$, that defines the behavior of an agent by mapping states to actions. By executing a policy, an agent interacts with its environment in discrete timesteps and defines a sequence, or a trajectory, of state-action pairs (s_t, a_t), with $t = 0, ..., T$, and an associated cumulative discounted reward $R = \sum_{t=0}^{T} \gamma^t r(s_t, a_t)$. The goal of RL algorithms is to find the optimal agent policy π^*, that is the policy maximizing the expected cumulative reward. To this end, we define the state-value function of a policy π

$$V^\pi(s) = r(s, \pi(s)) + \gamma \sum_{s'} t(s, \pi(s), s')V^\pi(s') \tag{2.8}$$

an the action-value function of a policy π

$$Q^\pi(s, a) = r(s, a) + \gamma \sum_{s'} t(s, a, s')V^\pi(s') \tag{2.9}$$

Intuitively, the state-value function corresponds to the value of the expected return when starting in state s and following policy π from there, while the action-value function represents the value of the expected return when taking action a in state s and then following policy π. Denoting as V^* and Q^* respectively the state and action value function of the optimal policy π^*, we have the optimal policy can be greedily determined with one look-ahead in the following way

$$\pi^*(s) = \operatorname*{argmax}_a (r(s, a) + \gamma \sum_{s'} t(s, a, s')V^*(s')) \tag{2.10}$$

$$\pi^*(s) = \operatorname*{argmax}_a (Q^*(s, a)) \tag{2.11}$$

The goal of RL algorithms becomes therefore to correctly estimate the optimal Q or V function by interacting with the environment [151].

Non-Markovian decision process

Markovian Decision Processes assume the markovian property over all the functions, namely, the transition and the reward function. The next state and the next reward can therefore be evaluated from the current state and the chosen action only. Although this formulation is general enough to model most decision problems, It has been observed that many natural tasks are non-Markovian [117]. A decision process can be non-markovian because markovianity does not hold on the reward function $r : (S \times A)^* \to \mathbb{R}$, or the transition function $t : (S \times A)^* \to S$, or both. Most of the work, however, focuses on non-markovian reward problems [15]. Learning an optimal policy in such settings is hard, since the current environment outcome depends on the entire history of state-actions pairs the agent has explored from the beginning of the episode; therefore, regular RL algorithms are not applicable. Rather than developing new algorithms to tackle non-markovian Decision Processes (NMDP), research has focused mainly on discovering how to augment the state to make it markovian, in order to apply known RL algorithms to the augmented-state MDP.

Reward Machines Reward machines (RM) are an automata-based representation of non-Markovian reward functions [27]. RMs provide a normal-form representation for reward specification in a diversity of formal languages. Given a finite set of propositions P representing abstract properties or events observable in the environment, RMs specify temporally extended rewards over these propositions while exposing the compositional reward structure to the learning agent. Formally, a simple Reward Machine [27] is a tuple $RM = (2^P, Q, \delta_t, q_0, \delta_r)$, where 2^P is the automaton alphabet, Q is the set of states, $\delta_t : Q \times 2^P \to Q$ is the transition function and $\delta_r : Q \times 2^P \to \mathbb{R}$ is the reward function. Let us notice the RM alphabet is 2^P, namely it is composed of each possible configuration of the symbols in P, contrary to DFAs that considers at each time step only one symbol in P is True and others are False.

2.2.2 Abstractions for MDP

We adopt the definition of abstractions given in [77], for which abstraction consists of mapping from one problem representation to another while preserving some properties. Regarding MDPs, we call ground MDP the original problem model and abstract MDP the one obtained through abstraction. The abstraction mechanism allows us to focus only on relevant pieces of information, discarding the rest. Finding adequate problem abstraction can enormously favor the decision-making process. The problem of creating the right abstractions becomes crucial, especially for complex problems [144], and it should be considered an essential part of solving a task [101]. It is possible to abstract MDPs in different ways [138], which are here summarized.

- **State abstractions**. Abstracting, or *aggregating*, the state consists in finding a new representation for the states that is more convenient for decision making. The ground state space often contains much information that is not useful to the agent to solve the task and only distracts it from solving it [179]. For this reason, state abstraction is usually a compression mechanism.

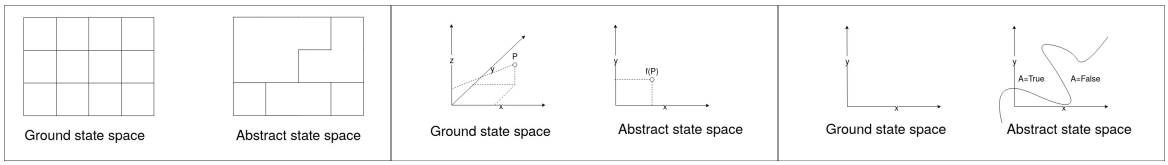


Figure 2.2: example of the three categories: left) finite partitioning, center) continuous encoding, right) symbol grounding

- **Action abstractions**. Abstracting the actions means abstracting in the temporal dimension, by generalizing one-step primitive actions to temporally extended actions called macro-actions or options [150] [148], decomposing in this way the original complex task in a hierarchy of sub-tasks [53]. Action abstractions are very convenient for long-horizon tasks that are hard to solve by considering only primitive actions.

- **State-action abstractions**. Some approaches propose to abstract in both the space and action dimensions, by changing representation for both the state and the action space [44] [163][102]. Usually, one kind of abstraction drives the discovery of the other one.

. In this paper, we focus mainly on state abstractions. We will see that for some kinds of tasks, such as control problems, state abstractions are not very useful if not accompanied by action abstractions, and for this reason, we also review some state-action abstraction methods.

Nature of the ground-abstract state space combination

The ground state space S can be a finite set of states $S = \{s_1, s_2, .., s_N\}$, where $N = |S|$, or a collection of continuous variables $S = \mathbb{R}^N$. The abstract space $\bar{S}$ must be as compact and small as possible while preserving the essential information we want to abstract. In the case of RL agents, this information should allow at least the agent to make reactive decisions in the environment. The abstract state space can be finite or infinite as well. We have different settings depending on whether the two state spaces are infinite and continuous or finite and discrete.

In our formulation, we assume the abstraction is a function $\phi : S \rightarrow \bar{S}$, however, it is possible to express it also as a conditional probability distribution of the abstract state given the ground state, i.e., $\Pr(\bar{s}|\ s)$, but for the sake of simplicity we don't cover this case in this chapter.

- **Finite partitioning:** when both the ground and the abstract spaces are discrete and finite, and we want to find an abstraction function ϕ that associates each different state in $S = \{s_1, s_2, ..., s_n\}$ to one abstract state in $\bar{S} = \{\bar{s_1}, \bar{s_2}, ..., \bar{s_k}\}$, with $k < n$, in this way partitioning S in k chunks or blocks of states.

- **Continuous encoding:** when the mapping is from a continuous feature space $S = \mathbb{R}^n$ to a smaller continuous space $\bar{S} = \mathbb{R}^d$ with $d < n$, usually called *latent space*. This practice is also known as representation learning and it is most common way of abstracting the states.

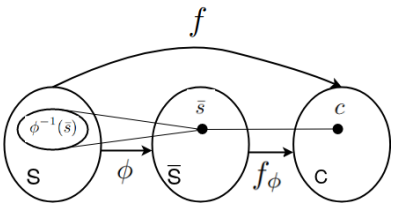


Figure 2.3: example of abstraction: ϕ is the abstraction and f_ϕ is the function induced by the abstraction. Their composition $f_\phi(\phi(\cdot))$ should be a similar as possible to the target function $f(\cdot)$ to have a good representation performance.

- **Symbol grounding:** when the ground state space S is continuous and the abstract space $\bar{S}$ is finite and discrete. This consists in classifying the states in S in a finite number k of classes or *symbols* in $\bar{S} = \{\bar{s_1}, \bar{s_2}, ..., \bar{s_k}\}$. Despite there are fewer works in this category, the research community has started increasing interest in this kind of approaches more related to neurosymbolic AI (NeSy).

Type of environment

RL owes its fame to its enormous versatility and applicability. Countless environments can be modeled as MDPs, and the purpose of this section is not to classify them all, but rather to point out that some are more abstract than others and need different techniques. We divide environments into two categories: **logic** problems and **control** problems. The first category refers to strategically complex problems that are simpler from the point of view of perception and actuation. On the contrary, the second refers to strategically simpler problems made complex by the rawness of observations and actions.

Finite partitioning methods

This section reviews methods for abstracting the state space when finite and discrete. The works focusing on finite abstraction are primarily theoretical and examined in the Lihong et al. survey [116]. Here we briefly resume some definitions and theorems from [116] because they serve as a basis for introducing other concepts later.

Abstraction metrics

When we consider an abstraction function $\phi : S \to \bar{S}$ we can evaluate it respecting coarseness and representation performance.

Coarseness answers the question "how much *wide* or abstract is the abstraction?". The formal definition of coarseness is given in [116] for finite ground and abstract space.

Coarseness definition: given two abstraction functions ϕ_1 and ϕ_2. We say ϕ_1 is *finer* than ϕ_2 (or ϕ_2 is coarser than ϕ_1), denoted $\phi_1 \succeq \phi_2$ ($\phi_2 \preceq \phi_1$), iff for any states $s_1, s_2 \in S$, $\phi_1(s_1) = \phi_1(s_2)$ implies $\phi_2(s_1) = \phi_2(s_2)$. If, in addition, $\phi_1 \neq \phi_2$, then ϕ_1 is *strictly finer* than ϕ_2 (ϕ_2 is strictly coarser than ϕ_1), denoted $\phi_1 \succ \phi_2$ ($\phi_2 \prec \phi_1$).

Representation performance responds to the question, "how much does the abstraction preserve a certain information of interest?". To answer this question, we usually measure the *representation error* for a certain target function of the state $f : S \to C$ we want to preserve.

$$e(s) = |f(s) - f_\phi(\bar{s})| \tag{2.12}$$

where $\bar{s} = \phi(s)$, and $f_\phi : \bar{S} \to C$ is the function induced by the abstraction, see figure 2.3.

The composition of ϕ and f_ϕ must be as similar as possible to the target function f. In case $f_\phi(\phi(\cdot)))$ and $f(\cdot)$ coincide, the performance error is 0. The error generally depends on how we choose the functions ϕ and f_ϕ. In the next section, we formally define ϕ and f_ϕ to have 0 performance error in case we know the target function f.

The combination of performance and coarseness makes a good abstraction. Measuring only one of these is not effective, since the coarsest abstraction is the one mapping all the ground states to the same abstract state $\phi(s) = \bar{s}$, and the most performative abstraction is the one not abstracting at all, namely the one mapping each ground state in a different abstract state. We are interested, instead, in the coarsest abstraction having bounded abstraction error over the whole state space.

$$\forall s \in S \; |f(s) - f_\phi(\phi(s))| < \epsilon \tag{2.13}$$

Exact and approximate abstractions

An abstraction is said **exact** if it has zero representation error. For finite partitioning, we can achieve zero representation error if we choose ϕ so to aggregate each state having the same f-value [116][50], therefore we have.

$$\forall s \in S \; \phi(s_1) = \phi(s_2) \leftrightarrow f(s_1) = f(s_2) \tag{2.14}$$

$$\forall s \in \phi^{-1}(\bar{s}) \; f_\phi(\bar{s}) = f(s) \tag{2.15}$$

In RL, we want to represent in the abstract state space the functions estimated by RL algorithms to make decisions: optimal value V, Q function, optimal policy function π, transition function t, and reward function r. Depending on the algorithm and the application, we could be more interested in one function or another.

[116] defines five exact state abstraction based on the functions of interest of RL:

- model-irrelevance abstraction ϕ_{model} preserves the next step reward and abstract state
- Q^π-irrelevance abstraction ϕ_{Q^π} preserves the Q-value of any arbitrary policy π
- Q^*-irrelevance abstraction ϕ_{Q^*} preserves the Q-value of the optimal policy
- a^*-irrelevance abstraction ϕ_{a^*} preserves the optimal action and its value
- π^*-irrelevance abstraction ϕ_{π^*} preserves the optimal action

[116] also proves these exact abstractions form a chain under coarseness.

Coarseness chain theorem: for any MDP $\phi_{model} \succeq \phi_{Q^\pi} \succeq \phi_{Q^*} \succeq \phi_{a^*} \succeq \phi_{\pi^*}$

As a consequence of this theorem, any one of the five abstractions is an instance of the coarser abstractions. For example, ϕ_{Q^*}, that preserves the value of $Q*$, also preserves a^* and π^*, but it does not necessarily preserve the Q-value of an arbitrary policy Q^π or the one-step model.

2.2.3 Neural networks

Traditional RL algorithms cannot solve problems with continuous or extensive state space, this is also known as the "curse of dimensionality". Deep RL (DRL) [124] [123] tackle the problem by using function approximator such as neural networks (NN) in combinations with dynamic programming. In DRL, we train NNs to approximate any function of interest, such as the optimal policy, the optimal value, etc. Here we give some preliminary concepts on neural networks, specificly on Feed-Forward Neural Networks and Recurrent Neural Networks. The former is most used for Markovian tasks, while the latter is most commonly used for non-Markovian problems.

Multilayer Neural Networks

A multilayer neural network is a parametrized function with a *layered* structure, like the one in Figure 2.4. Each layer at depth i of a neural networks takes as input the output from the layer at depth $i-1$, process it, and return to the layer $i+1$. Each layer is composed of a set of neurons. Each neuron is a matematical procedures extending the older Perceptron model [136]. We denote as $f(x;\theta)$ the neural network f taking as input the tensor x and having trainable parameters θ. The network weights are updated so to minimize a *loss function* through a gradient descent procedure. In supervised learning we can exploit a dataset made of *ground truth* input-output couples (x, y), acquired from the target function $\hat{f}$ we want to approximate. In this setting the loss function $L(x, y, \theta)$ is the distance between the network prediction and the desired output y

$$L(x, y, \theta) = |f(x;\theta) - y| \tag{2.16}$$

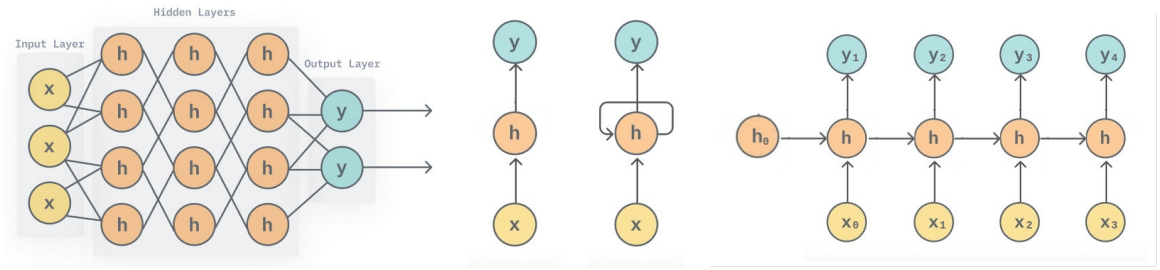
and the trainable parameters θ are updated at each iteration of training in the following way

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta L \tag{2.17}$$

where we denote as θ_t the parameter values at iteration t and α, $0 \leq \alpha \leq 1$, is the learning rate.

Recurrent Neural Networks

A Recurrent Neural Network (RNN) is a parameterized function $h_t(h_{t-1}, x_t; \theta_h)$, having trainable parameters θ_h, that takes as input a state-vector at time $t-1$, $h_{t-1} \in \mathbb{R}^{d_h}$, and the input vector at time t, $x_t \in \mathbb{R}^{d_i}$, and returns the current state-vector at time t,$h_{t+1} \in \mathbb{R}^{ds}$. An RNN can be applied to a sequence $x[0], ..., x[n]$ by recursive application of the function h to



(a) (b) (c) (d)

Figure 2.4: a) Architecture of a multilayer neural network. b) Multilayer NN prediction scheme. c) Recurrent prediction scheme. d) A recurrent neural network unfolded.

the vectors $x[i]$. An example of RNNs are the Elman and Jordan networks [59], also known as Simple Recurrent Networks.

$$\begin{aligned} h_t &= \sigma_h(W_h x_t + U_h h_{t-1} + b_h) \\ y_t &= \sigma_y(W_y h_t + b_y) \end{aligned} \tag{2.18}$$

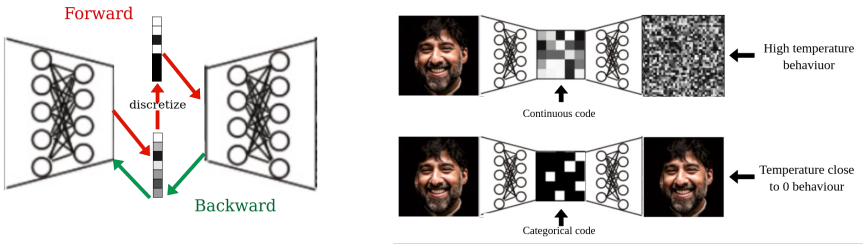
Where x is the network input, h is the hidden state, y is the network output, W_h, U_h, W_y and b_h are the network parameters, and σ_h and σ_y are activation functions.

Although the input must be continuous, we can use a set of discrete symbols as an input alphabet by mapping each symbol to an input vector using either a one-hot encoding or an embedding matrix. RNNs are employed in a variety of tasks on sequential data. In particular, they can be used to classify sequences. A binary RNN-acceptor, is pair of functions h, y, where $y : \mathbb{R}^{d_h} \rightarrow \{Acc, Rej\}$ is the classification module that classifies the RNN's state vectors. An RNN-acceptor is conceptually equal to DFA, except that it processes a sequence by applying continuous state-transitions and output evalutions, that are usually hard to inspect and explain.

2.3 Neurosymbolic integration

Neurosymbolic artificial intelligence (in short NeSy AI) [20] is an emerging field of Artificial Intelligence (AI) combining both neural networks and symbolic reasoning methods to facilitate the development of AI systems that can learn from data and generalize to solve complex tasks. NeSy frameworks are mostly based on a two-layer architecture: we have perception deep learning modules that live in the subsymbolic domain, where representations are continuous; and we have a symbolic part where reasoning happens through boolean symbols manipulation. The subsymbolic modules are responsible for processing the raw data from outside and passing it to the symbolic module, so that the latter is grounded in reality and not disconnected from that. The very crucial part of this design is to implement the subsymbolic-symbolic interface.

This can be implemented in many ways. From a representation point of view, what prevents the integration is the very different means deep learning and symbolic AI use to store and manipulate pieces of information. On one side, neural networks are intrinsically continuous, since continuity ensures they can be optimized using gradient-based algorithms like backpropagation. On the other hand, symbolic AI uses boolean and discrete symbols,



(a) (b)

Figure 2.5: Some techniques to discretize the output of a neural network in a safe way a) Straight Through estimator c) A Gumbel-Softmax-activated variational autoencoder

which could seem impossible to learn with a neural system. However, the integration is possible if one of the two (or both) makes a step towards the other side. Here we review the main methods that aim for smooth neuro-symbolic integration.

2.3.1 Deep Learning stepping in the white side: discrete neural networks

In this section, I briefly review the main techniques used in the literature to predict discrete-valued variables using neural networks. This is a very convenient feature for neurosymbolic integration, since a symbolic reasoning system can use the discrete output of a neural network directly as propositional symbols without needing further operations on the latent space, such as discretization, clustering, or approximation to a finite set of values.

Straight-through estimator

We can obtain discrete values from each neural network layer by simply thresholding the tensor values. However, since the derivative of threshold functions is zero, this poses a problem during back-propagation. The straight-through estimator[19][36] estimates the gradients of a function ignoring the derivative of the threshold function and passing on the incoming gradient as if the threshold function was an identity function. In this way, we use the discrete tensor in the forward step and the continuous tensor in the backward step during back-propagation, as Figure 2.6 (a) shows.

Increasing the activation steepness

Another simple technique used in literature [97] [164] [107] is to increase the steepness of activation functions during training smoothly. We can obtain this effect by dividing the activation argument for a temperature value τ, with $0 < \tau < 1$, starting with a warm temperature and decreasing it smoothly. As the temperature approaches 0, the activation values become closer and closer to discrete values. Except for ReLu-like functions, this technique can be used over all the most popular activation functions, such as Sigmoid, Softmax, Tanh, etc.

Gumbel Softmax activation

Jang et al. propose a stochastic activation function to predict categorical latent variables [97]. The latter are rarely used due to the inability to backpropagate through samples drawn from

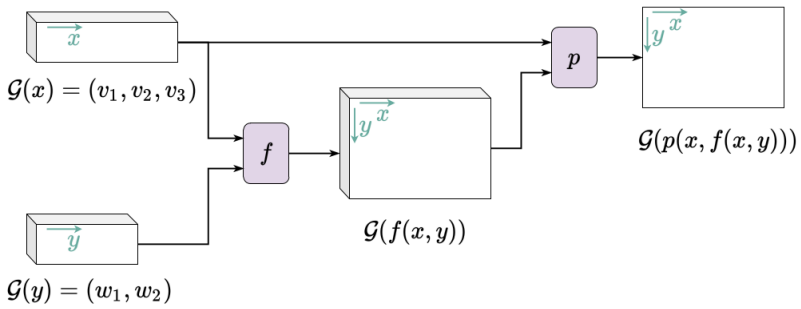


Figure 2.6: An example of Logic Tensor Network, taken from [16]. In this example two variables (x and y), one function (f) and one predicate (p) are represented. Variables x and y are grounded to the available data (the values v_1, v_2 and v_3 for variable x and the values w_1 and w_2 for variable y). The rightmost tensor in the image, $\mathcal{G}(p(x, f(x, y)))$, is the grounding for the binary predicate p when interpreted over x and $f(x, y)$. This is grounded by applying the continuous module p to the grounding of x and the grounding of $f(x, y)$; which is in turn grounded applying the f module to the groundings of x and y.

the categorical distribution. The Gumbel-Softmax activation is a continuous and differentiable version of the Gumbel-Max trick [83] [119] to draw samples from a categorical distribution. The activation function generates a k-dimensional continuous sample vector y. The magnitude of approximation is controlled by a temperature parameter τ, which is decreased by a schedule during training. As the temperature drops, samples from the Gumbel-Softmax distribution become one-hot, and the Gumbel-Softmax distribution becomes identical to the categorical distribution. Figure 2.6 (b) shows an example of a Variational Autoencoder (VAE) using this activation function. The code extracted by the VAE is a $N \times k$ matrix, where N is the number of categorical variables and k is the number of possible categories for each variable. Each row of the matrix is a one-hot encoded vector; therefore, the number of possible codes is K^N.

2.3.2 Symbolic Logic stepping in the dark side: continuous logic

Some works take a diametrically opposite direction, and base the integration on the use of real-valued logic such as fuzzy logic [89] or probabilistic logic. In particular, fuzzy Logic is a multi-valued generalization of classical logic, where truth values are reals in the range $[0, 1]$, and the logic operators are replaced by continuous functions having both inputs and outputs in the continuous range $[0, 1]$. Fuzzy logic has shown to be suitable in several real-world applications where a statement can be only partially true or exceptions can be present. Notably, fuzzy interpretations are based on continuous and differentiable functions, so neural networks can co-exist with the logical knowledge and actually implement some elements of the logic knowledge base. The use of continuous logic has seen many successes in neurosymboilc AI [161], and many framework are based on it, such as Logic Tensor Networks (LTN) [16] and Lyrics [121].

Logic Tensor Networks

Logic Tensor Networks (LTN) [16] are a neurosymbolic framework that can reason and learn by exploiting both structured symbolic knowledge and raw data. It implements a logic called Real Logic, which contains constants, function and predicate symbols, as First Order Logic (FOL).

LTN also implements connectives ($\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$) and quantifiers (universal, existential, diagonal universal, and guarded universal and existential). Any logic formula in Real Logic is interpreted using fuzzy logic semantics, namely, it is assigned with a continuous truth-value between 0 and 1. Every element of Real Logic is grounded in real tensor, so that it can be an assignment to available data, the output of a neural network, or a satisfaction level of a logic formula between 0 and 1.

LTN can be used for querying, reasoning and learning: here we focus on learning. LTN can learn from both data and symbolic knowledge by imposing the knowledge available, and searching for the groundings that maximize the satisfiability of that knowledge. This is done by defining a loss objective that is inverse to the given formula's satisfaction level and optimizing the system's trainable weights by back-propagation. This is also known as *learning by best satisfiability*, and it is implemented by constructing the neural computational graph with the logical operators present in our logic knowledge base.

Chapter 3

State of the art

In this chapter, I review existing methods in the literature to discover logical knowledge from a non-symbolic domain. Recall that this problem can be schematized as shown in Figure 3.1 as the composition of two subproblems

1. the discovery of an abstract representation, possibly symbolic, as compact as possible, that can be used in place of raw observation data
2. the discovery of a model, as structured and simple as possible, capable of describing the functioning of the observed system by exploiting the abstractions found in the previous point

In particular, I divide this chapter into two main sections, following the division line of the entire thesis. In the first part, I focus on the so-called Markovian problems, in which each observation can be considered separate from the previous ones. In the second part, I review learning approaches for non-markovian problems, i.e., domains whose observations come in temporal sequences that must be considered in their entirety since they cannot be split thanks to Markovianity.

3.1 Discovering an abstract model for markovian tasks

In this section, I will focus on finding an abstract model for *Markovian* tasks. I start by reviewing the literature in the area of Reinforcement learning abstraction, which was introduced in the background chapter. Abstraction is very related to symbol grounding; in this thesis, I consider symbol grounding as a particular case of state abstraction. In particular, I will follow the categorization of state abstractions given in Section 2.2.2, and I will review in the order: finite partitioning, continuous encoding, and symbol grounding methods for abstracting an MDP. Then I will review how the NeSy AI literature tackles the symbol grounding problem. In the latter, the tasks considered are not necessarily modeled as MDPs, and generally do not include the concept of actions. However, they are of interest to the thesis as they foresee a part of perception and a part of logic. Even for these works, I will focus in this section on those focusing on intrinsically "Markovian" problems, borrowing the definition from the RL literature.

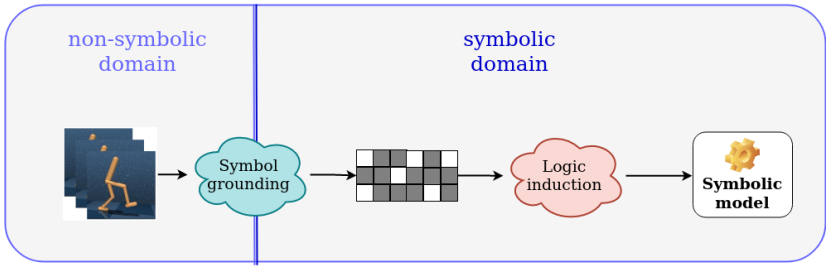


Figure 3.1: Discovery of logical knowledge from raw domains

3.1.1 Finite partitioning

In this section I review *finite* state-abstraction of *finite*-state MDPs. In this setting abstraction consists in *partioning* the ground state space in a certain number of partitions and in considering the partition index as the new state representation in the abstract MDP. Random partitions are not supposed to bring any benefit to the RL algorithm, on the contrary, they can destroy any ability of the agent to make reasonable decisions in the abstract environment. The goal of this kind of abstractions is therefore to partitionate the ground state-space in a number of sets as small as possible, while preserving the capability to estimate one or more of the RL functions of interest, such as the optimal policy, π^*, the optimal Q function Q^* and so on. Based on which function we want to preserve we obtain an abstraction of one of the five classes defined by [116]. Of particular interest are the abstractions belonging to the class ϕ_{model}, since this is the finest class of abstractions. An example of ϕ_{model} is the *stochastic bisimulation* [50] [78], which aggregates states that are bisimilar, defining as bisimilar two states if they have the same expected reward and equivalent distributions over the next bisimilar states. [50] proposes an iterative algorithm for aggregating bisimilar states in polynomial time given a finite state-space MDP.

Some approaches relax the requirement of having 0 representation error by defining **approximate** abstractions. In this case, two ground states are aggregated if they have a representation error smaller than a positive constant ϵ.

[1] proposes approximate abstractions based on the model, the optimal q-value Q^*, and multinomial and Boltzmann distributions over Q^*. It proves the optimal policy found in the abstract MDP is *near optimal* in the ground space. Namely, the representation error of the optimal value is bounded, and the bound depends on ϵ.

The works mentioned so far aim to define abstractions for theoretical purposes, and not to make reinforcement learning algorithms better, e.g., more efficient or robust. Both exact and approximate state aggregations can be defined only once we *know* the function of interest. However, knowing this function is not an assumption of RL; quite the opposite. To say it more clearly, with these methods, we infer s_1 and s_2 must be associated to similar (the same) abstract states because *we know* they have a similar (the same) f-value. In learning setting, where we are trying to *estimate* function f, we would like to *infer* that $f(s_1)$ and $f(s_2)$ are similar (or the same), speeding up the learning of f, because we know that s_1 and s_2 are associated to similar (the same) abstract state. So looking at the theory mentioned in this section, from the RL perspective, it could seem that RL agents will not gain anything from abstracting the state and will only lose in a bounded way. It is true for problems with little

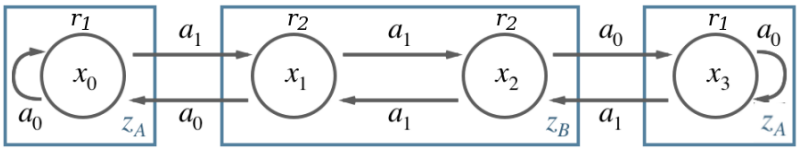


Figure 3.2: An example of state aggregation satisfying the bisimulation preperty. This 4-states deterministic ground MDP can be partitioned in 2 abstract states without loosing the capability to predict the next reward and abstract state.

state-spaces, where states are in practice *already abstract* [101], and performances pay a cost for cutting off some states. However, it is not true in realistic problems with huge state spaces, where very different observations have the same meaning and must be mapped in the same abstract state. In the next section, we review methods for abstracting the state in such environments where neural networks are employed to manage the high dimensionality of the state space.

3.1.2 Continuous encoding

This section considers abstractions that are continuous functions taking continuous inputs. Furthermore, in this section and the subsequent (symbol grounding), we focus on learning abstractions instead of their definition and properties. However, the theory introduced in the previous section guides us in categorizing the works.

Traditional tabular RL algorithms cannot solve problems with continuous or extensive state space, this is also known as the "curse of dimensionality". Deep RL (DRL) [124] tackle the problem by using function approximator such as neural networks (NN) in combinations with dynamic programming. In DRL, we can train NNs to approximate any function of interest, such as the optimal policy, the optimal value, etc. In particular, it has shown that encoding the high-dimensional observations in compressed *latent representation* can be beneficial for DRL algorithms [128] [2]. Considering this latent space as a continuous abstract state space $\bar{S}$, we call ϕ the parametric function that maps the environment observation to the abstract state; and f_ϕ the parametric function, that maps the abstract state to the value of interest. In this way, we divide the network into two parts: one learning the abstraction and the other learning the function of interest in the abstract space. We obtain different abstractions depending on how we train these two parts and which learning objective we use for training. Some of them are more aligned with the abstraction theory than others.

Misaligned deep abstractions

One way to train ϕ and f_ϕ is through *staged training*. It consists of two explicit training stages with two different learning objectives: (1) first, we train the abstraction ϕ minimizing an auxiliary loss; (2) then, keeping the abstraction function fixed, we train the outer function f_ϕ. The most classical auxiliary loss used is the reconstruction error [110] [111] [85] $e_{rec} = |s - d(\phi(s))|$ where $d : \bar{S} \to S$ is a *decoder*, mapping the abstract state back to the ground state. A more modern approach is to use a contrastive loss as the auxiliary objective to train the encoder [31] [160] [112]. Although staged training is stable and easier to implement, it

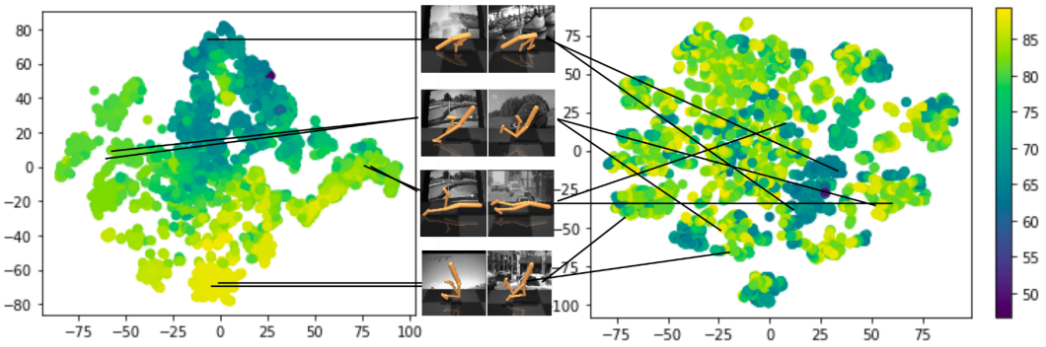


Figure 3.3: Aligned versus misaligned abstractions:

has some throwbacks. (1) Since we do not know the value or the Q function during the first training, the only data we can use for training the encoder is the one coming from *random exploration* in the environment. This data should be sufficiently similar to those we will observe under improved policies to have a good encoder, and this is not a suitable assumption for many environments. 2) Training with an auxiliary loss that does not depend on the task makes the encoder capture all the predictable features in the observation. This differs from what we want in terms of abstraction, so these representations are misaligned with the abstraction theory.

Aligned deep abstractions

A different way to train ϕ and f_ϕ is through *end-to-end training*. It consists in training the composition of the two functions directly in one stage of training by minimizing the representation error $|f(s) - f_\phi(\phi(s))|$. This training modality is aligned with the abstraction theory since it automatically reduces the representation error. Furthermore, the continuity of the abstract space and the use of function approximators ensure we have an abstraction even for unseen ground states, speeding up the abstraction learning. Depending on which function the network approximates, we obtain abstractions of the five kinds defined by [116] and described before. For example, Deep Q Network (DQN) [124] can solve tasks in rich observation spaces, like video games, through a convolutional encoder followed by a multi-layer perceptron trained end-to-end. Since the Q network aims to estimate the optimal Q-function, the convolutional encoder can be considered a *Q^*-irrelevance abstraction*. Model-irrelevance abstraction can be obtained through model-based reinforcement learning on a latent space [58]. It consists in learning an abstraction function that maps observations to a compressed latent space, $\phi : S \to \bar{S}$, and the next state and reward functions as functions of the abstract space $r_\phi : \bar{S} \to R$, $\delta_\phi : \bar{S} \to \bar{S}$. The reward function in the latent space can be trained with supervised learning by minimizing the representation error

$$L_r = |r(s,a) - r_\phi(\phi(s),a)| \tag{3.1}$$

By contrast, using the representation error to train transitions in the latent space can be unpractical.

$$L_t = |\phi(t(s,a)) - t_\phi(\phi(s),a)| \tag{3.2}$$

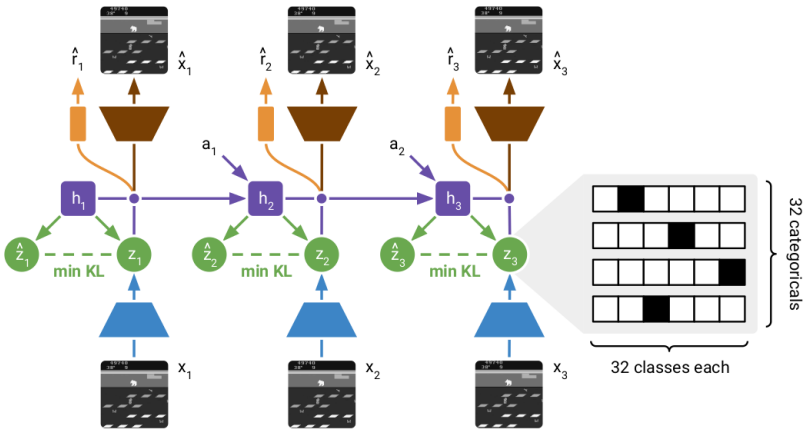


Figure 3.4: Training scheme of the discrete world model used in [88]

This loss is also called *model coherence*. Let us notice that the transition derived by the abstraction also has *the output* in the abstract space; therefore, a neural architecture directly minimizing this error can lead to *representation collapsing*, since the model prediction and target both depend on ϕ. This issue can be mitigated by 'bootstrapping the encoder' [81] [128], by adding an explicit regularizing loss [64] or adding other learning objectives that influence the abstraction, like for example learning the inverse dynamics model [129] [2]. The latter takes the feature encoding $\phi(s_t)$, $\phi(s_{t+1})$ of two consequent states as input and predicts the action a_t taken by the agent to move from state s_t to s_{t+1}. Unlike reconstruction error, adding this objective does not misalign the encoder from the abstraction purpose, because it does not encode any further information in the latent space except the capability to not collapse. Model-irrelevant abstraction has proven to be beneficial for transfer policies between environments with different observations that only differ regarding task-irrelevant information [179]. Furthermore, learning the model in a compressed latent representation that does not contain any useless information for the task allows planning in the latent space, speeding up the agent's imagination components and improving sample efficiency [64] [86]. Last but not least, model-irrelevant abstractions are also used to assess the novelty of observations and implement curiosity-driven exploration strategies using the model prediction error as curiosity signal [129] [128]. In this section, for the sake of simplicity, we call model-irrelevant abstractions the feature spaces designed to encode information about the next reward and the next abstract (or latent) state, discarding all the other information we consider nonrelevant for the task. They are therefore aligned to the theory conceptually, even if they do not formally provide any guarantees to find an exact model-irrelevant abstraction. Some works are more attuned to the formal theory of MDP abstraction, and they are based on the definition of a bisimulation metric [60] [61] [29]. The latter softens the concept of state partitions introduced in [50] for finite state MDPs. They instead define a pseudometric space (S, d), where a distance function $d : SxS \rightarrow R_{\geq 0}$ measures the "behavioral similarity" between two states. This relaxation allows applying general concepts from the finite abstraction theory to continuous state MDPs. In particular, [179] trains an encoder using bisimulation metric as target distance in the latent space. The paper demonstrates that the policy on ϵ-discretized representation is near optimal. In other words, the optimal value representation error $|V*(s) - V*(\phi(s))|$

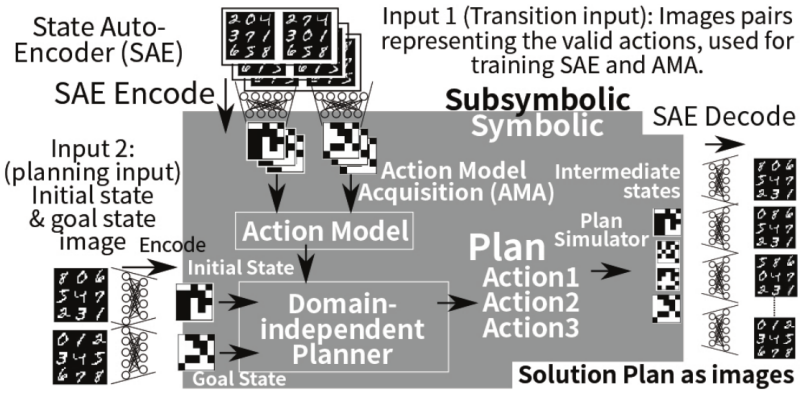


Figure 3.5: LatPlan framework from [7]

is bounded, and the bound depends on epsilon. This means that states can be clustered according to the bisimulation metrics without losing too much performance.

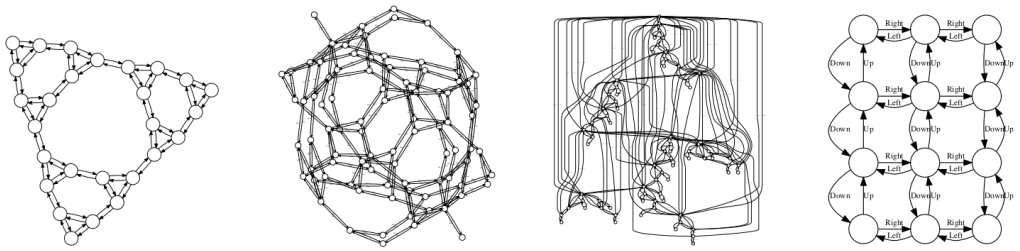
3.1.3 Symbol grounding in Reinforcement learning

In this last section on abstraction for reinforcement learning, we focus on *finite and discrete* abstractions of a *continuous* ground-state MDP. All the training objectives and neural network architectures introduced in the previous section for continuous encoding can be combined with discrete outputs neural networks, e.g., the ones described in section 2.3.1, so to obtain a *discrete*, and therefore symbolic [70], RL model. In this sense, the encoder with discrete activation is used as an *automatic symbol grounder*, and the symbols are nothing else than boolean latent variables. Depending on the loss objective used for training, we can obtain symbolic interpretations more or less aligned with the abstraction purpose. Many works [105][88] show that discrete world models can accurately predict the agent action outcome in a non-symbolic environment. These models can be used to plan towards a specific goal [105] or to simulate the agent-env interactions and learn a policy entirely from simulations [88].

This kind of work is the most similar to classical DRL. Some others aim to build a tighter connection with classical symbolic planning domains [7][10][5][12][55]. From a certain point of view, latent symbols extracted by a neural network have the same strengths and weaknesses as continuous latent representations: they can be easily learned without supervision, but they are usually hard to inspect and connect with a meaningful set of concepts from outside.

However, a significant difference that distinguishes them from continuous representations is that they are *compatible* with the learning of *discrete logical domains*, for which several off-the-shelf solvers are applicable.

Therefore, once we have learned the symbol grounding method and the domain over these symbols, we can apply the usual tools designed to solve that kind of domain. Indeed, semantics only matter to humans. Tools efficiently manipulate symbols regardless of their meaning or how they are grounded, and this is one of the main assumptions of classical symbolic AI. An example of this is LatPlan [7][10][5][12], which is a framework able to learn a symbolic STRIPS [62] first-order logic domain in the latent space. The framework is trained with a set of transitions in the form $(image_t, image_{t+1})$, *without* the information about the action performed to pass from $image_i$ to $image_{t+1}$, and its objective is to learn a symbolic represen-



(a) Towers of Hanoi (1 label) (b) Gripper (3 labels) (c) Blocksworld (3 labels) (d) 4×3 Grid (4 labels)

Figure 3.6: Some examples of graph representing the state space structure for different domains. They represent the input of algorithm described in [22]

tation for images and a set of action-schemas describing the actions preconditions and effects. They are both grounded by a Gumbel-softmax function. Once the training is complete, it uses the learned strips domain for planning. The framework obtains encouraging results in visual, logical tasks like solving puzzles and constructing Hanoi towers. The first LatPlan version [7] trains the models in a staged fashion. First, the images are grounded into symbols with a Gumbel variational autoencoder. Then the system learns the action schema by using another autoencoder-like neural architecture that takes as input no longer the pixels of images but only their associated symbolic codes. In its last version [12] LatPlan changes the network architecture and the mode of training: symbols for the images and actions over these symbols are grounded *at the same time* with end-to-end training. Similarly, [55] trains a discrete VAE to learn relevant features in Atari games given images as training data. Then it uses the discrete features for planning with RolloutIW [17], achieving impressive results. This prior work proves that discrete representations can be effectively used not only for RL but also for general planning.

Differently from the previous works, in [22] [134], authors address the problem of determining the first-order representation from the state-space structure, rather than observations. They learn the planning model without using deep learning. They extract it from a labeled directed graph modeling transitions, by encoding the search problem as a satisfiability problem and solving it with a SAT solver.

All the works mentioned so far for symbolic planning address environments with a discrete, even if sometimes complex, setting, e.g., videogames and puzzles, that I classified as *logic problems* in Section 2.2.2. More rare are applications of unsupervised symbol grounding and planning domain discovery to fully continuous and dynamic problems with multiple continual variables, i.e., control problems [102][143][142]. In the logic problems considered, the environment produces high-dimensional rich states in the form of images. However, transitions are almost already *abstract*, in the sense that one elementary action in the environment produces a *substantially different* new state that we can associate easily with another symbolic representation. In a robot-control scenario, instead, both states and actions are very low level. Therefore, we cannot expect a *meaningful* change in the state space by executing only one elementary action, and macro actions extended in time are needed. One way to represent macro actions is through *options* [150]. An option is a tuple $< \pi_o, I_o, \beta_o >$, where $\pi_o : S \rightarrow A$ is the *option policy*, specifying which low-level action must be taken in the low-level states when option o is activated; $I_o \subseteq S$ is the *initiation set*, namely the set of low-level states

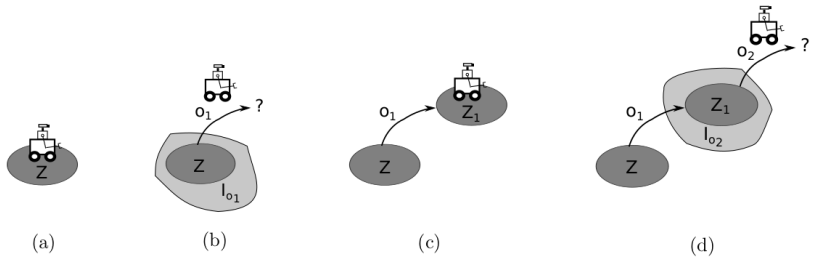


Figure 3.7: An example of grounding for symbols Z, Z_1, in a robot navigation problem, from [102]

from which we can execute the option; $\beta_o : S \to [0, 1]$ is the *termination condition*, specifying the probability that an option terminates in a specific low-level state. In particular, in [102], authors assume to know robot controllers as options, transforming them into FOL action schemas. This work particularly focuses on *subgoal* options, that terminate in a compact set of states and do not depend on where the option started. This kind of option is particularly of interest in robot controlling since usually motor controllers of a robot are designed to reach a specific, generally small, set of goal states. This definition of subgoal skills is also extended to options that match a particular set of conditions in a subset of the state variables, letting the other unchanged. In this prior work, Konidaris demonstrates these skills lead to an abstract high-level symbolic representation of the problem, and off-the-shell probabilistic planners can use that to solve different robotic tasks in that domain. The symbolic representation is obtained by naming the different sets with symbolic names and eventually performing feature selection without using machine learning. Finally, in [108] the authors tackles continuous planning problems by the use of an *initial draft* of an abstract PDDL domain. The purpose of this work is to learn how to map continuous observations into the abstract states and how to *adjust* or *complete* the prior PDDL domain by interacting with the environment in a plan-act-learn interleaved schedule.

3.1.4 Symbol grounding in NeSy AI

In this section, I review the topic of symbol grounding from the point of view closer to the NeSy literature. NeSy AI generally addresses logic domains made more challenging by using non-symbolic observations. The latter requires the introduction of a perception system which is generally implemented with neural networks. Examples of these tasks are math operations on images [120] and visual logic games such as sudoku [167][14].

In this setting, symbol grounding implements the *perception* function, while logic induction with [139] or without [38] neural networks discover the logic model from symbolic data.

Symbol grounding through self-supervision or supervised learning

As for the abstraction function in Deep RL, the symbol grounding function can be learned with objectives unrelated to the logic task in a separate learning stage. Many works use reconstruction error, which can be used as the unique objective [9], or in combination with a *logic* loss [153]. Many other works exploit labels directly on observation data and implement symbol grounding with a neural classifier trained with supervised learning. Although this is not always applicable since direct labels are not always available, supervised classification

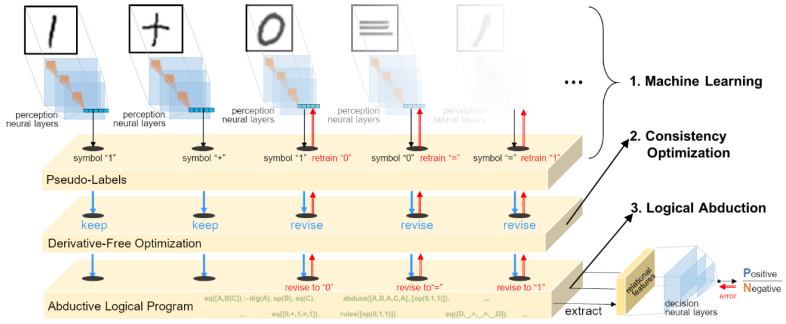


Figure 3.8: Framework from [39] for grounding digits from high level labels and prior symbolic knowledge through abduction

is the easiest and more stable way to implement symbol grounding. Connecting symbols predicted by a neural network with a logic induction algorithm is not always straightforward [38], since some of them are not resilient to errors in the training data or cannot exploit probabilistic beliefs on symbols.

Example: in a domain with two symbols, A and B, which are predicted by a softmax-activated network, a (boolean) logic induction system might be completely confused by a symbol prediction like [0.49, 0.51]. The latter is treated precisely as a configuration [0, 1] (A =false, B =true) when it is closer to the prediction [0.5,0.5], semantically corresponding to (A='do not know', B='do not know').

For this reason, the research on NeSy problems has moved mostly on probabilistic [23] [120] [171], or fuzzy [16] [54] [121] reasoning systems.

Symbol grounding by exploiting prior logical knowledge

The majority of works approach the symbol grounding problem by assuming perfect knowledge of the symbolic model [42] [54] [174] [16] [120] [176] [171] [39] [95] [154]. These works infer the most probable grounding of a certain set of symbols P given prior logical knowledge expressed on the alphabet P and high-level labels on the whole NeSy process. This practice is known as semi-supervised symbol-grounding.

Example: A benchmark for semisupervised symbol grounding is the *digit addition* problem. The system takes as input two images of handwritten digits between 0 and 9, and it has to predict their sum. The system can be trained to recognize the symbols 0, 1, ..., 9 in the single images without single-image-labels, by exploiting rules on how addition works (0+0=0, 0+3=3, etc.) and a dataset supervising *the addition process*, made of associations (couple of images, sum of the two digits represented in the images).

Symbol grounding exploiting the logical knowledge is tackled mainly with two families of methods.

The first approach [42] [54] [174] [16] [120] [176] [171] consists in encoding the prior logical knowledge as a differentiable module $lm()$ taking the output of the classifier $sg()$ as input. The two functions are combined as two layers of a NN, and their composition $lm(sg(\cdot))$ is trained to maximize the given symbolic knowledge on the high-level label available.

The second approach [39] [95] [154] instead maintains a crisp boolean representation of the logical knowledge and uses a process of logic *abduction*. Even in this approach, a neural

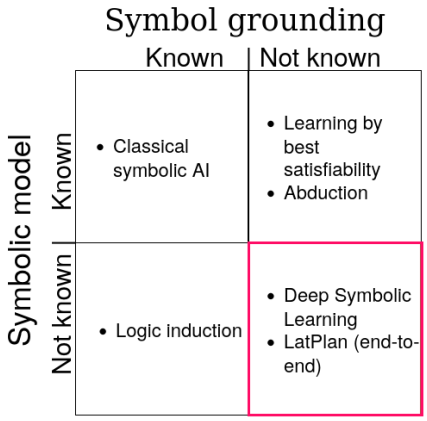


Figure 3.9: Categorization of NeSy approaches

classifier implements the symbol grounding function. The classifier outputs its current belief on the symbol's truth value, and an abduction module corrects its predictions to make them more consistent with the prior symbolic knowledge. After that, the classifier is *retrained* with the corrected symbols in a supervised fashion. This two-step training process has been shown to make the classifier converge to the target symbol grounding.

Learning symbol grounding and logical knowledge at the same time

Let me notice that in the previous section 3.1.3, I considered all works aiming at discovering *both* the symbol grounding and the symbolic model *at the same time* (with some exceptions where the grounding is solely determined by reconstruction). In contrast, I mostly review works that address one problem at a time in this section. Indeed, Nesy's approaches to tackling both of these problems simultaneously are actually few [12][41]. An approach designed initially for this is SATNet [167]. It consists in encoding MAXSAT in a semi-definite programming based continuous relaxation, and integrating it into a larger deep learning system. However, it has been shown that it can only learn the symbolic model when supervision on symbol grounding is given [30]. A follow-up work [153] extends SATNet to learn also the symbol grounding. However, this extension relies on a pre-processing pipeline that uses InfoGAN [32] based latent space clustering, which does not rely on the symbolic model. This makes this approach fall into the class of methods that use an auxiliary objective to determine symbol grounding. In [41], the authors propose Deep Symbolic Learning (DSL), a method for making discrete symbolic choices within an end-to-end differentiable architecture. It uses a policy function that, given confidence values on an arbitrarily large set of symbols, can discretely choose one of them. DSL achieves competitive results with NeSy frameworks that assume prior knowledge of the symbolic model on visual, mathematical operations on MNIST digits [120]. However, it suffers scalability problems when the number of symbols is big, which prevents its application to large domains.

The literature shows that research in the discovery of logical knowledge from non-symbolic

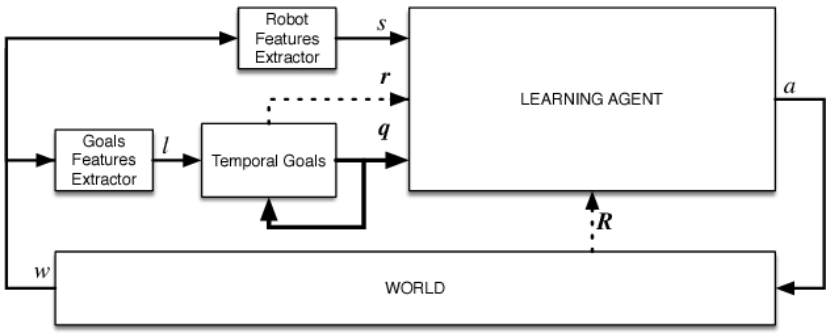


Figure 3.10: A schematic view of Restraining Bolts [74]

data is at the very beginning. In particular, there is currently no *formal criterion* (such as the bisimilarity for abstraction in RL) to define *how* the perception task and the logical induction task should influence each other, and *whether* they should influence each other or not. Moreover, the discovery of logical knowledge from non-symbolic domains is only one of the goals of NeSy AI, which most commonly aims to improve the efficiency, explainability, or reliability of frameworks based on neural networks [21] [68] [76].

3.2 Discovering temporal logic knowledge from non-markovian tasks

This section will focus on discovering logical symbolic knowledge describing a non-symbolic non-markovian domain. I consider, therefore, tasks that are *extended in time* that a formula can capture in Linear Temporal Logic (LTL). In particular, I will focus on LTL over finite traces and DFAs, since we can assume without loss of generality that the task has a finite duration in time. The main interest is still to investigate the integration between symbolic data, such as those processed by the LTLf formula, and non-symbolic data, such as those coming from directly observing the domain.

3.2.1 LTL and non-markovian RL

Non-markovian RL tasks are extremely hard to solve because intelligent agents must consider the entire history of state-action pairs to act rationally in the environment. However, the idea behind the current line of research in this area consists in bypassing the non-markovianity augmenting the state space with a set of *features* that encode the environment history and solving the augmented-state MDP with known RL algorithms.

The main point remains, therefore, how to construct these features. In the case of non-symbolic-state-MDPs, the most popular approach is to combine RL algorithms with the use of Recurrent Neural Networks [91][84][100] that automatically extract features from data sequences. In case of problems with a discrete and finite state space, most works use LTL or LTLf to specify the temporally-extended task. This specification is then compiled into an automaton, and the automaton state is combined with the environment state to make the decision process markovian. Examples of this approach are the so-called Reward Machines [27] and the Restraining Bolts [45]. Both simplify and automate the creation of reward functions

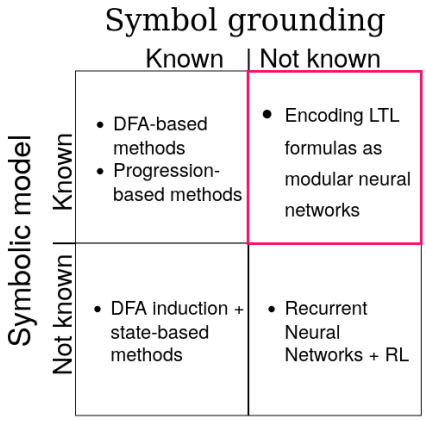


Figure 3.11: Categorization of works on non-markovian RL

for non-Markovian decision processes exposing the structure of the reward function to the agent.

Some other works focus on *learning* the temporal specification of the task from observation and rewards received by the environment by using known methods for DFA induction [67][175][135]. Another work tackles "advice-guided" non-markovian problems, where the reward machine is *partially* known in the form of a set of *advice*, and the agent refines it using rewards from the environment [126].

All these works use symbolic data and do not consider the problem of discovering latent symbols in the data. For this reason, they are applicable only in discrete-state environments or continuous problems for which a mapping between the continuous state and a symbolic interpretation is known, also known as labeled MDP [166]. This mapping is also called *labeling function*.

Many works do a preliminary step towards integration with non-symbolic domains by considering *imperfect* labeling functions [25][162][115]. Namely, functions that sometimes make mistakes in predicting symbols from states, or that predict a set of *beliefs* over the symbol set instead of just one perfect value. These functions closely resemble the output of deep symbol grounder, so considering them is important for integration with deep learning.

A notable work [104] uses LTLf specification of the task without assuming any knowledge of the grounding of symbols of the formula into the environment states. This work employs a neural network architecture that is *shaped as the LTL formula*. The formula is considered a tree of operators and propositional symbols. Each operator or propositional symbol is implemented as a neural network module, and the modules are connected as in the formula tree, and reused among different formulas. The output of the tree root is used as state representation by an Actor-Critic (A2C) agent [123]. The whole system is trained end-to-end with agent actions and outcomes from the environment, as in the A2C algorithm. The authors exploit the framework compositionality by training on a sizable amount of different formulas (corresponding to various tasks in the same domain). Doing so, the neural modules

are supposed to *learn through the interaction with environment* the grounding for symbols and operators of the formula. Therefore they can be reused in other formulas without retraining on that particular task on new data, with zero-shot transfer.

Part II

Discovering logical knowledge in markovian non-symbolic domains

Chapter 4

STRIPS-Like Symbolic Abstractions for Control RL Problems

As we have seen in the previous chapter, the ability to make decisions autonomously is a key feature of artificial intelligence agents. When the agent has to perform a task in a dynamic and complex environment, its decision-making capabilities are strictly influenced by what the agent knows about its environment and how well it can predict its evolution. Automated planning is centered on finding a correct plan to solve a specific task, reasoning on the knowledge of the scenario, often expressed in a symbolic form. Just as humans manipulate ideas in their heads in order to plan their future actions, so do planners with *propositional symbols*. In this way, they can easily perform long-term projections in the future and plan actions accordingly. For example a human, or a symbolic planner, can plan the actions to turn on a car, knowing that: ″when I *turn the key* my car is *turned on*" - where 'turn the key' is an action modifying the truth value of the symbol 'turned on'- avoiding modeling irrelevant details such as the car color, the weather, and so on. Although this approach can be very performant on symbolic domains [72], formalizing continuous and dynamic environments in such symbolic form is extremely hard and it is usually manually carried out by a human expert. On the other hand, there has recently been an increasing interest in Deep Reinforcement Learning (DRL) [4]. DRL techniques automatically extract from data an effective policy to achieve the task in the environment, interacting with it through a trial-and-error process, and they do not require any models of the environment's states and transitions. However, by using DRL, the policy and the learned model are a black box, given the lack of explicability of the final result at the end of the training.

In this chapter I propose an algorithm inspired by both symbolic planning and Reinforcement Learning to *automatically* extract a symbolic planning domain for complex dynamic environments, this allows to exploit the advantages of planning techniques in this particular domains without the need of expert modeling. We develop an interactive learning algorithm that maps the environment state to an arbitrary size finite set of propositional symbols, tackling the symbol grounding problem [145], and jointly learns the environment dynamics over this symbolic space. The system interacts with the environment formalized as a Markov Decision Process (MDP) over continuous variables. The experience data gained from the interaction are exploited to learn the following function approximators: a function for grounding

the propositional symbols from the continuous observations; a value function approximator for evaluating symbolic states; a transition function approximator over the symbolic state space to predict the next symbolic states from the previous one and the corresponding action. This automatically learned planning domain is used to lead the interaction with the environment. At each interaction step, we perform online planning [72] over a finite horizon, and we select the first action of the computed plan as the action to take in the environment.

We evaluate the approach on several OpenAI gym environments with continuous state, including control problems and games, and in a RoboCup scenario with a real robot. We also perform experiments setting different sizes for the finite symbols-set modelling the environment. Experiments show that, with a sufficiently large symbols-set, it is actually possible to plan the continuous environment's actions, reasoning only in the symbolic domain extracted from data. Furthermore, we show that we can use the learned planning domain to achieve different tasks in the same environment, through a reward shaping based on the learned symbolic representation. Experiments confirm that the symbolic representation-based reward and the transition model efficiently guides the agent towards different goals in the environment. The contents of this chapter have been published in [156]. I organize this chapter as follows: first, I define our problem setting and the goal we want to achieve in section 4.1; then, I illustrate the schedule used by the agent to interact with the environment in section 4.2; in section 4.3, I describe the models composing our framework and how they are learned from interaction data; in section 4.4 I describe the planning module, in section 4.5 I present how the framework models can be transferred to new tasks in the same domain, in section 4.6 I report experiments on different domains; related work is reported in section 4.7, finally, in section 4.8 I report my final considerations on the results and directions for future research in this area.

4.1 Problem Setting

We consider an agent interacting with an unknown environment. The environment is modeled as an infinite horizon Markov Decision Process MDP $= \{S, A, t_g, r, \gamma\}$. Where $S \subseteq \mathbb{R}^n$ is a continuous state space, $A = \{a_0, a_1, .., a_m\}$ is a discrete action space, $t_g : S \times A \to S$ is the environment transition function, $r : S \times A \to \mathbb{R}$ is the reward function and γ is the discount factor. The agent knows the current environment state s_t, it can take an action $a_t \in A$ and observe the action-outcome, namely the new state $s_{t+1} \in S$ and a reward $r_{t+1} \in \mathbb{R}$ received by the environment.

We want to learn a planning-based symbolic representation model of the reference environment interacting with the MDP. The idea of learning the environment model of MDPs, alongside the optimal action policy, is not new in Reinforcement Learning and is known also as model-based Reinforcement Learning (RL) [125]. Model-based RL algorithms usually learn the transition function and the reward function from the experience data gained through the interaction with the environment and use them to increase sample efficiency and temporal efficiency.

Our purpose in this work is not to increase the efficiency of the learning process, but to learn *another*, alternative, intrinsically different, representation of the same world, as similar

as possible in its essence to a symbolic planning domain. Furthermore, we want use this model to effectively select actions to take in the MDP, so to achieve the desired task, as expressed by the MDP reward function. More formally, let's call $a_p(s)$ the action chosen by the planner when the environment is in state s, we want to maximize the expected cumulative discounted reward obtained by choosing actions according to the planner.

$$\sum_{t=1}^{\infty} \Big[\gamma^t r(s_t, a_p(s_t)) \Big]_{s_t = t_g(s_{t-1}, a_p(s_{t-1}))} \tag{4.1}$$

Let us consider a symbolic state-space X composed of d_X propositional symbols.

$$X = \{0, 1\}^{d_X} \tag{4.2}$$

At each time instant t, we can think of the observation s_t as the realization of a certain *situation* x_t identified by a specific truth value of the symbols in X. We define a situation as a boolean d_X-dimensional vector $x \in X$, and we can think of it as the latent symbolic representation of s. Since actions in A modify the observation that arises from the situation, we expect them to modify in some cases the underlying situation as well, and we want to model how this happens. In other words, we want to find a transition model t_s for the situation-space.

$$t_s : X \times A \to X \tag{4.3}$$

This model is conceptually similar to the effects of a PDDL-like [71] actions schema: given an action a and a situation x we want to learn which symbols to delete and add to x in order to have the next situation x'.

Furthermore, we want to know a measure of how much being in a particular situation is decisive for achieving the task in the environment. This is equivalent to know a numeric value function v_s over the symbolic space.

$$v_s : X \to \mathbb{R} \tag{4.4}$$

Last but not least, we want learn a *grounding* method for the symbols, namely a model e able to return the truth values of the symbols in every observed state $s \in S$.

$$e : S \to X \tag{4.5}$$

The symbolic planner learned with the interaction can be seen therefore as a tuple of three functions.

$$P = \{e, t_s, v_s\} \tag{4.6}$$

Once we have learned P from experience we can use it to perform online planning during the agent-environment interaction in the abstract symbolic space. In other words, we may say that the agent is able to *abstractly imagine the future*, as it is shown in Figure 4.1. Abstract imagination is a fundamental part of our human intelligence: as humans we naturally make projections of our future reality by manipulating abstract ideas in our heads and we base our actions in the real world on those projections. In the same way the agent, starting from

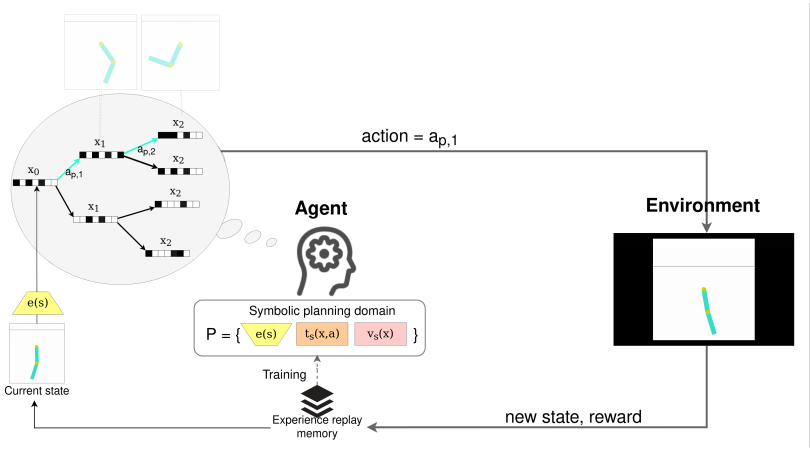


Figure 4.1: A global view on the framework

its current state s, can abstract the situation $x = e(s)$, imagine many possible finite-horizon-evolutions of the episode in its own idea-space, and choose the action leading to the best 'story in its head', which corresponds to the simulated path achieving the best value according to the estimated value v_s (see section 4.4 for further details on the planning process).

4.2 The Interaction with The Environment

Here I describe the interactive learning method from an high level point of view, as it is illustrated in Algorithm 1.

The agent interacts with the environment by following a classic RL scheme. At each step it chooses an action $a \in A$ and executes it in the environment env with the function $env.execute(a)$. This function returns the next state s', that is resulting by applying a in the current environment state s, and the obtained reward r. The function $env.episode\_terminated()$ return True if the current episode is finished and False otherwise. The tuple (s, a, s', r) is the experience acquired through the step and it is stored in the experience buffer. Then, we train the models of P with a batch of data extracted randomly from the experience buffer. In particular, each model is implemented as a feed-forward neural network and is trained online - namely only with the data obtained through experience - no offline training is required. The action to execute is chosen with a classic epsilon-greedy exploration strategy: the agent performs random action choice with probability ϵ and abstract online planning with probability $1-\epsilon$. ϵ is initialized to a large value and it is decreased over time. In this way, the agent relies more on the planner policy when the networks output becomes more consistent and robust. This loop plan-interact-learn is executed until the agent is able to achieve in the environment a satisfying cumulative reward value, (**while** $episode\_reward \leq desired\_episode\_reward$), and $desired\_episode\_reward$ is a desired cumulative reward value defined by the environment. In order to select the action, the planner takes as input the current state s_t and the desired finite planning horizon T. It plans future trajectories in the abstract space starting from the current situation $x_t = e(s_t)$, using the model t_s to expand abstract states for T steps. The symbolic-state trajectories are evaluated using the value function v_s, and finally the first action a^1* of the computed plan $(a^1*, a^2*, ..., a^T*)$ is selected for execution. The planning

Algorithm 1: Interactive learning algorithm

Input: env, desired_episode_reward
Output: P = (e, t_s, v_s)

```
P = initialize_models_randomly();
experience_buffer = empty_buffer();
ϵ = initialize_epsilon();
episode_reward = 0 ;
while episode_reward ≤ desired_episode_reward do
    s = env.initial_state();
    episode_reward = 0 ;
    while not env.episode_terminated() do
        mode = epsilon_greedy(ϵ);
        if mode == explore then
            a = random_action();
        else
            T = set_planning_horizon();
            a = plan_action(P, s, T);
        s', r = env.execute(a);
        experience_buffer.append(s, a, s', r);
        episode_reward += r;
        s = s';
        batch = experience_buffer.sample();
        P = train_models(P, batch);
    ϵ = decrease_epsilon();
```

horizon is also adapted during training, starting from small length horizon when expansion and evaluation of plans are more prone to commit errors, and it is increased as the agent gains experience by the function *set_planning_horizon*().

The interaction stops when the agent achieves the task in the environment by choosing at each step the action returned by the symbolic planner, namely when the sum of the rewards obtained in the last episode reaches a certain desired cumulative reward value that depends on the task.

4.3 Learning the Models

In this section we give details about each function of $P = \{e, t_s, v_s\}$ and how they are learned from data.

4.3.1 Symbol Grounder

The mapping between the environment state-space to the symbolic space is implemented using an encoder neural network $e(\cdot;\theta) : S \to [0,1]^{d_X}$ with a sigmoid activation function on the last layer, with the code length d_X equal to the number of symbols we want to predict. The boolean output is obtained by discretizing the encoder output in 0/1 values by using discretization with the straight trough estimator [19] [36]. Therefore, the encoder output space is composed of only 2^{d_X} possible different codes, and every single code is a possible interpretation over d_X propositional symbols.

$$e(s) = \text{Discretize}_{\{0,1\}}(e(s;\theta)) \tag{4.7}$$

4.3.2 Symbolic Transition Model

For the transition function in the symbolic space we trained a neural network model $t_\Delta(\cdot;\theta) : X \times A \to [-1,1]^{d_X}$, that learns the effects of actions on symbols in a PPDL-like formalism. t_Δ^{net} takes as input the symbolic state x and the action a, and it predicts which symbols the action modifies in the state. In particular, the output is discretized in three possible values: -1 for the symbols to delete from the state, 0 for the symbols that remain unchanged, 1 for the symbols to add to the state. The next symbolic state x' is therefore

$$t_s(x,a) = x + \text{Discretize}_{\{-1,0,1\}} t_\Delta(x,a;\theta) \tag{4.8}$$

Where we use the following activation function in the last layer of the t_Δ neural network.

$$\phi(x) = 1.5\text{tanh}(x) + 0.5\text{tanh}(-3x) \tag{4.9}$$

This activation suggested by [132] assume values between -1 and 1 as the hyperbolic tangent (tanh(x)), but, differently from tanh, it has three flat points with values -1, 0 and 1, hence it supports 3-valued discretization.

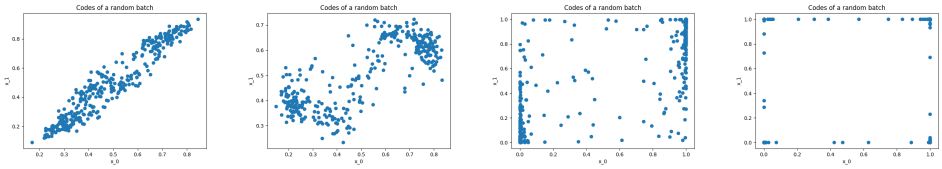


Figure 4.2: Continuous codes of a random batch of states with increasing margin. On the y axis: x_1 values between 0 and 1; on the x axis: x_0 values between 0 and 1

4.3.3 Symbolic Value Model

To express the value of a symbolic state we analyzed two possible strategies:(1) learning the reward function r over the symbolic space, (2) learning the state-action pair quality function Q over the symbolic space.

The function $Q_\pi(s, a)$ is defined as the expected discounted cumulative reward of taking action a in the state s and then following the policy π. It is commonly used in model-free reinforcement learning algorithms like Q-learning [96]. Although learning a Q function is much harder than learning a reward function, Q describes how much current actions influence future rewards, bringing much more information than the simple current reward function. For this reason we implemented the second choice, even if option one may still be a valid choice.

We define a Q-network taking as input the state in its symbolic form and returning an m-dimensional continuous vector $Q(\cdot;\theta) : X \to \mathbb{R}^m$, where m is the number of possible actions, and the i-th component of the output, $Q(x;\theta)_i$, is the expected cumulative discounted reward for taking action $a_i \in A$ from the symbolic state $x \in X$. To eliminate the dependency on a we define the value of a symbolic state as

$$v_s(x) = \max_i Q(x;\theta)_i \tag{4.10}$$

Our system trains therefore three neural network models: $e(s,\theta)$, $t_\Delta(x, a;\theta)$ and $Q(x;\theta)$, that are shown in Figure 4.3.

4.3.4 Training Neural Networks with Symbolic or Discrete Layers

Symbolic representations are compatible with symbolic planners, and they are often more interpretable and more computationally efficient than their continuous analogs. However, their discrete nature makes it difficult to be learned with neural networks.

Since discretization is a non-differentiable operation, in order to use backpropagation we cannot embed discretization as a layer in the network. On the other hand, since the discrete output is used as input by other models, we cannot discretize the layer outside the network. In fact, if we train the other models with *almost discrete* inputs, they will not predict the expected output when they are fed with *perfectly discrete* inputs.

Regarding our approach, we need both discrete and continuous outputs to be predicted by our system. For these reasons, discrete-output layers are trained using a the Straight-Through Estimator [19][36]: both the discretized output tensor t and the continuous one $\widetilde{t}$ are maintained in the computational graph; the former is used during forward propagation, while the latter is used in the backward step of backpropagation. This technique, that was introduced

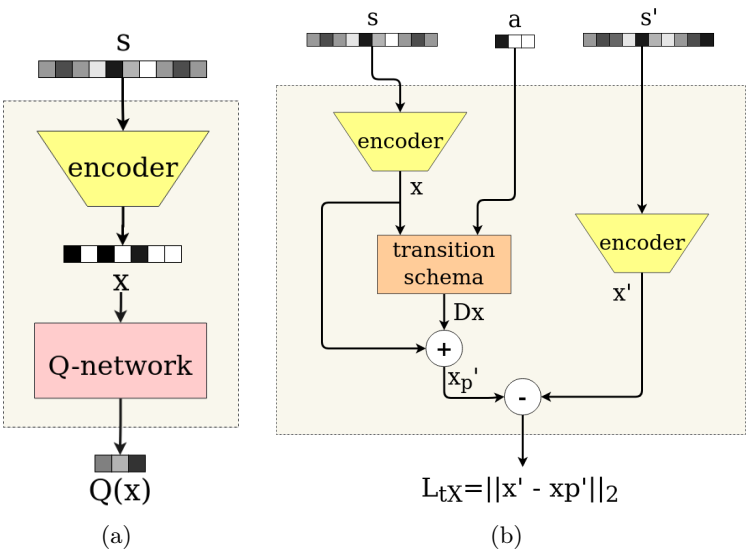


Figure 4.3: a) the symbolic quality model (b) computation of the transition model loss

in section 2.3.1, solves both the issues mentioned above, because we use the continuous tensor for calculating the gradients necessary for updating the weights and we use the discrete tensor to feed the next layer of the model.

Furthermore, in order to decrease as much as possible the discretization error, namely the difference between the continuous tensor $\widetilde{t}$ and its discrete t, we use the following loss function.

$$L_{disc} = 1/m_1 * \text{ReLU}(m_1 - ||\widetilde{t} - Y||_2) \tag{4.11}$$

Where Y is a constant value and m_1 is a margin. The effect of minimizing this loss is to move $\widetilde{t}$ to have a distance from Y of at least equal to the margin m_1. This loss is applied to $\widetilde{t} = e(s;\theta)$ with $Y = 0.5$, to push the codes to stay close to 0/1 values, and to $\widetilde{t} = t_\Delta(x, a;\theta)$ with Y equal to -0.5 and 0.5 so to cluster the output of t_Δ^{net} around the values of -1, 0, and 1. The effect of increasing the margin m_1 on the continuous tensor representing the symbolic state before discretization is shown in Figure 4.2. For visualization purposes the figure shows only the first two tensor components. Notice that setting the margin large enough push the codes really close to the 4 possible configurations of 2 boolean variables: (0,0), (0,1), (1,1) and (1,0).

Another possibility is to use the Gumbel-Softmax [98] as the activation function of the discrete layers. We tried to use it and we observed that the agent is able to increase the cumulative episode reward for a while, but finally fails to reach the desired episode reward value, even with very large size codings.

4.3.5 Value Model Training

Here we describe how we use the tuple (s, a_i, s', r) taken by the experience replay to update the value model, shown in figure 4.1. For sake of clarity, we use the notation $Q(x, a_i;\theta)$ to indicate $Q(x;\theta)_i$.

We use two Q networks as in Double Deep Q-Network (DDQN) [124] to enhance the

Q network stability. We call the two models prediction and target network and we denote respectively with $Q(x;\theta)$ and $\bar{Q}(x;\theta)$. We train only the prediction network, the target network is a copy of the prediction network, which is updated every t_{synch} steps. We jointly train the encoder $e(s;\theta)$ and the symbolic q-network model $Q(x;\theta)$ by minimizing the following loss function.

$$L_q(s, a_i, s', r) = ||Q(e(s;\theta), a_i;\theta) - V(s', r)||_2 \tag{4.12}$$

Where $V(s', r)$ is the expected cumulative reward according to the target Q-network model and it is calculated with the Bellman equation.

$$V(s', r) = r + \gamma \max_{a_i \in A} \bar{Q}(e(s';\theta), a_i;\theta)) \tag{4.13}$$

4.3.6 Transition Model Training

Everything in the tuple except the reward is used to update the transition model. Given (s, a, s'), we jointly train $e(s;\theta)$ and $t_\Delta(x, a;\theta)$ minimizing the transition prediction squared error *in the symbolic space* X, L_{tX}. Figure 4.3 shows how this error is calculated. Let us denote as x'_p the next symbolic state, as predicted by the transition model.

$$x'_p(s, a) = e(s;\theta) + t_\Delta(e(s;\theta), a;\theta) \tag{4.14}$$

We want to minimize the distance in the abstract space X between x'_p and the symbolic interpretation of s': $x' = e(s';\theta)$.

$$L_{tX}(s, a, s') = ||x'_p(s, a) - e(s';\theta)||_2 \tag{4.15}$$

Let us notice that according to loss L_{tX} only, the optimal e and t_Δ are the constant functions $e(s;\theta) = \bar{x}$ and $t_\Delta(x, a;\theta) = 0$, since this configuration of the models always leads to 0 error in predicting the next symbolic state. We prevent this degenerate solution by using the following regularization loss.

$$L_{dist}(s, s') = 1/m_2 * \text{ReLU}(m_2 - ||e(s;\theta) - e(s';\theta)||_2) \tag{4.16}$$

Akin Equation 4.11, L_{dist} is used to forces the distance between the encoding of s and s' to be at least equal to the margin m_2.

4.4 Action Selection

Here I describe how the models are used for online planning. We assume to know the planner functions e, t_s and v_s, since they are approximated by the neural networks trained with samples of the experience replay. The planner takes the current continuous state s_t, and a desired planning horizon T and it selects the action plan showing the best expected total value over the next T future steps.

First the current continuous state s_t is encoded in its symbolic version x^0, then x^0 is expanded using the transition schema t_s to simulate every possible action $a \in A$ for T con-

secutive steps. In this way a full-breadth limited-depth planning tree is built.

$$x^0 = e(s_t), \quad x^i = t_s(x^{i-1}, a^i) \tag{4.17}$$

We denote with a^i the action simulated at step i, with $p^i = (a^1, a^2, .., a^i)$ the sequence of action simulated in the first i steps, and with x^i the symbolic state at the end of simulation p^i. Hence a complete simulation p^T corresponds to a path from the root x^0 to one leaf of the tree. We evaluate each complete simulation using the symbolic value model v_s and an approximation of the transition epistemic uncertainty. In particular the total value V of a simulation is

$$V(p^T) = \sum_{i=1}^{i=T} v_s(x_i)c(p^i) \tag{4.18}$$

Where $c(p^i)$ is the confidence we have in the transitions simulated up to step i. Planning over a learned model shows different risks in terms of stability. But, estimating and propagating the model uncertainty allows to robustly plan over long horizons [51][66][34]. In our case we found that approximating the epistemic uncertainty of a transition with the discretization error is quite effective to find better plans. In fact, at each simulation step, we first calculate a continuous next state $\widetilde{x'} = x + t_\Delta(x, a; \theta)$. Then, we discretize $\widetilde{x'}$ to its closest boolean vector $x' = \text{Discretize}_{\{0,1\}}(\widetilde{x'})$, and we consequently compute the discretization error as $e_d(x, a) = |\widetilde{x'} - x'|$. which is considered as the uncertainty of the transition (x, a). The confidence we have in the final state of a simulation long i transition steps, denoted as $c(p^i)$, is therefore recursively calculated as

$$c(p^i) = c(p^{i-1})(1 - e_d(x^{i-1}, a^i)), \quad \text{with } c(p^0) = 1 \tag{4.19}$$

Where $(1 - e_d(x_{i-1}, a_i))$ is our confidence in the last simulation step (based on disrcetization error), and $c(p^{i-1})$ is the confidence we had in the simulation before performing the last step.

Finally the best plan $p*$ is calculated as the complete simulation maximising the total value V.

$$p* = \underset{p^T=(a^1,a^2,...,a^T) \text{ with } a^i \in A}{\arg\max} V(p^T) \tag{4.20}$$

and the first action of $p*$ is selected as the optimal action to take in the environment and it is executed by the agent.

4.5 Transfer Learning

A benefit of using planning systems is that they can be re-used infinite times, in order to calculate a plan for every possible couple (initial state, final state) in the domain. In RL instead, since the goal is expressed as a reward function, changing the goal is not so straightforward, and it consists of solving another MDP with a different reward function. Some techniques exist in transfer learning for RL [181], such as reward-shaping [127], that allow re-using part of the knowledge acquired solving an $MDP^1 = (S, A, t, r_{G_1}, \gamma)$ to solve an $MDP^2 = (S, A, t, r_{G_2}, \gamma)$, where r_{G_1} is the reward shaped to guide the agent to the goal G_1, and r_{G_2} is another reward

function defined to lead the agent to the goal G_2. Regarding our approach, we would like to investigate if, once we have learned our symbolic planner $P^1 = (e^1, t_s^1, v_s^1)$ solving MDP^1, we can use it to guide the agent's actions towards a different goal G_2; or, in other words, if we can use P^1 to solve MDP^2.

In principle, we should be able to re-use the grounding method, e^1, and the transition function t_s^1, but we cannot re-use for sure the quality function v_s^1, since it depends on r_{G_1}.

Since the symbolic representation tends to map closer states that were subsequent during execution, we examined the use of Jaccard distance between symbolic states, according the symbolic representation learned for goal G_1, to guide the agent toward goal G_2. We call this distance function Jd_{e^1,G_2}.

$$Jd_{e^1,G_2}(s) = 1/d_X \sum_{i=1}^{i=d_X} |e_i^1(s) - e_i^1(G_2)| \tag{4.21}$$

It is the distance between a generic state s, in its symbolic form according to e^1, and the goal $G2$, also converted in the same representation through e^1.

We tested the use of $1 - Jd_{e^1,G_2}$ as reward function to lead the agent toward G_2, with a generic model-free RL algorithm and with our learning algorithm.

In the second case, this is equivalent to define a new planning domain $P^2 = \{e^2, t_s^2, v_s^2\}$, initialize the models of P^2 as

$$e^2 = e^1, \quad t_s^2 = t_s^1, \quad v_s^2 = \text{RandomInitialization}() \tag{4.22}$$

and re-train the models running Algorithm 1 (by relying on everything as before, but the value function) in the $MDP^2 = (S, A, t, f_{e^1,G_2}, \gamma)$. This second training adjusts the models e^2, d^2 and t_s^2 and learns a new value function v_s^2 from zero, that is specific for the new goal.

Experiments show that model-free RL algorithms are able to converge to the new goal using the reward $1 - Jd_{e^1,G_2}$ that is based on the encoder learned by our algorithm for the old goal G_1. Furthermore, if we use our algorithm instead, the convergence is faster, since the planner can re-use also the transition model learned for the old task.

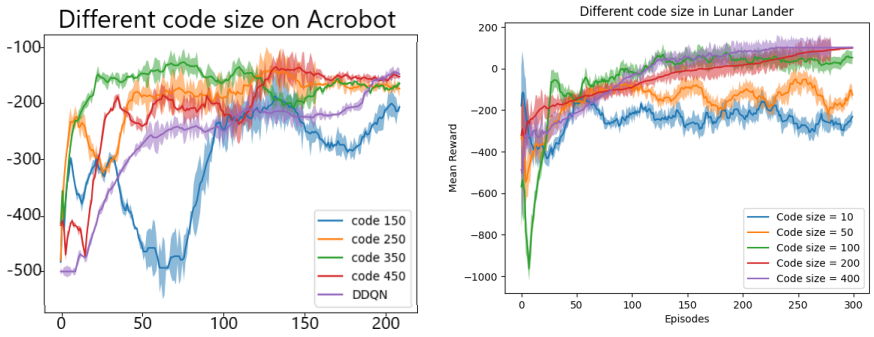
4.6 Experiments

In this section we propose a set of three experimental setup. The problems faced are: (1) under-actuated robot control problems from the OpenAI gym suite,such as CartPole and Acrobot, (2) continuous OpenAI gym problems subject to scattered rewards, such as Lunar Lander, and (3) a decision making task in the contest of robotic soccer.

4.6.1 Environments Description

Cartpole

The CartPole environment requires the agent to balance a pole connected to a motorized car. The goal is to keep the pole at an angle between +15° and -15° using the cart's movement.



(a) (b)

Figure 4.4: Performance with different sizes of the symbol set on the Acrobot environment.

The state definition of the environment is: $[p, \dot{p}, \theta, \dot{\theta}]$, where p is the cart position and θ is the pole angle with respect to the cart.

Acrobot

In the Acrobot environment, the agent can act only on the intermediate joint of a two revolute joints planar arms. The two links are initially hanging downwards, the goal is to reach a certain height with the end of the second link. The state definition of the Acrobot is so composed: $[\cos(\theta_1), \sin(\theta_1), \cos(\theta_2), \sin(\theta_2), \dot{\theta_1}, \dot{\theta_2}]$, where θ_1 is the first joint angle and θ_2 is the second one.

Lunar Lander

The Lunar Lander environment models the classic rocket trajectory optimization problem. The agent maneuvers a spaceship, and the goal is safely land the spacecraft on the ground. To do so, the agent disposes of four actions: do nothing, fire the left orientation engine, fire the main engine, and fire the right orientation engine. The state is an 8-dimensional vector $[x, y, \dot{x}, \dot{y}, \theta, \dot{\theta}, b_1, b_2]$: the coordinates of the lander in x and y, its linear velocities in x and y, its angle, its angular velocity, and two booleans representing whether each leg is in contact with the ground or not. A small negative reward is given to the agent each time it fires an engine. The agent receives positive rewards only when it reaches the ground at the end of the episode. This makes the reward function quite sparse and complicates the credit assignment problem.

NAO Soccer Player

The platform considered is the NAO robot produced by SoftBank robotics. The chosen task concerns the robotic soccer scenario inspired by the RoboCup competition [1], more specifically, the ability to walk with the ball, keeping it in the internal space of the robot's feet. The robot can perform three different actions: (1) Walk forward; (2) Adjust its position by moving to the right; (3) Adjust its position by moving to the left. The state is composed of two continuous

[1] `https://2023.robocup.org/`

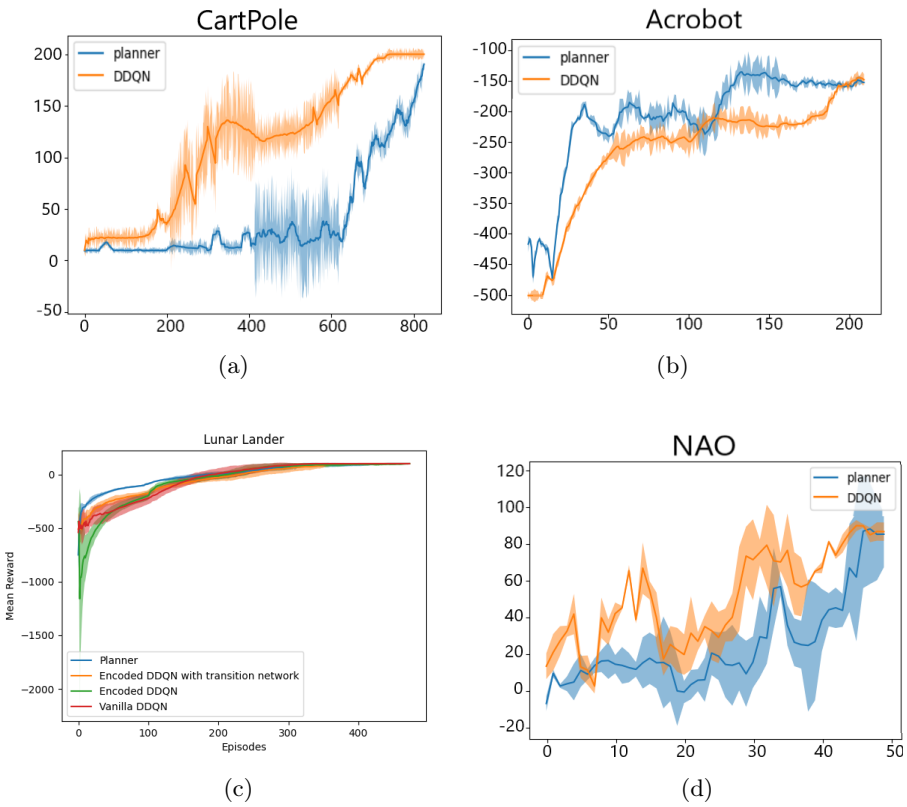


Figure 4.5: Training performances on Cartpole (a), Acrobot (b), Lunar Lnader (c) and NAO Soccer Player (d). On the y: axis mean training reward; on the x axis: episodes. Solid lines represent mean values, shaded areas represent standard deviations.

variables: $[p_x, p_y]$ with p_x and p_y being the relative position of the ball with respect to the robot on the x and y-axis. The robot receives a positive reward proportional to the distance it can move, keeping the ball close to it. The episode terminates when the ball reaches a maximum distance from the robot.

4.6.2 Setup

The chosen setting for the DDQN is $lr = 0.001$, $\gamma = 0.99$, the number of hidden layers is 4, and the number of units for each hidden layer is between 100 and 450 units, depending on the experiment. The experiments have been conducted ten times with ten different seeds for each environment and approach.4.6.7.

4.6.3 Symbol-Set-Size Tuning

To show the relationship between the symbol-set size and the system performance, we evaluate our approach results with different sizes of the symbolic code. Figure 4.4 shows the mean rewards obtained during training in the Acrobot and the Lunar Lander environment using different code sizes for the symbol grounder. The graph clearly shows that properly setting the symbol-set size is critical, and affects the system capability to converge quickly to a solution. In the Acrobot domain, the system almost cannot converge to the desired cumulative reward value by using 150 symbols. Still, it can converge even quicker than DDQN on the continuous

space with a number of symbols greater or equal to 250. In the Lunar Lander, convergence to the desired final reward of 100 is achieved only with code sizes greater or equal to 100. Also experiments in this domain show how the training sample efficiency increases by increasing the number of symbols. This is completely coherent with the abstraction theory. Increasing the number of symbols, we create a finer abstraction that can potentially better represent the continuous domain we are abstracting. Let us recall that the system needs to encode the information about the next symbolic state and the next rewards in the symbolic representation since it is trained end-to-end to minimize the representation error on the next state and Q value. This classifies the system in the model-irrelevant abstractions. In order to balance coarseness and representation performance, we decide to keep the number of symbols as small as possible to achieve convergence in a *reasonable* number of episodes. We consider reasonable a sample efficiency similar to that obtained by DDQN in the same domain without abstraction. For this reason, we set the symbol set size to 200 for all the domains in the experiments that will follow. Keeping the number of symbols quite small also allows us to investigate the representation more easily, as we will see in Section

4.6.4 Training Performance

We analyze the system performance on OpenAI simulated control problems and a robotic task. To demonstrate that we achieve competitive results, in terms of cumulative reward and sample efficiency, whilst exploiting a symbolic and discrete state representation we compare our system against DDQN a state-of-the-art deep RL baseline released within the OpenAI Baselines framework [52]. The results obtained by the system on the test environments demonstrate its ability to solve different types of continuous and dynamic environments as control problems of underactuated systems: Acrobot, and CartPole; systems with sparse rewards as Lunar Lander; and robotic tasks. Figures 4.5 (a)(b) and (c) show the mean rewards achieved by our algorithm during training in gym environments (a-b-c) and in the task with the NAO robot (d). Regarding experiments with the robot, the models were previously trained using a simulator, then brought back to the real robot after a retraining phase. In all the experiments the symbol set size is set to 200. It is important to remark that our model-based approach – in order to successfully extract the planning domain – has to learn in a more constrained state-space, while the model-free DDQN acts in the original continuous state-space. Nevertheless, our solution reaches a competitive average reward in a number of episodes which is smaller o comparable with the baseline.

4.6.5 Planning Performance

Figures from 4.6 show the results obtained in the different environments using the models obtained from training for planning with increasing planning horizons. We used the planner with and without uncertainty estimation and propagation for the gym environments and only with the uncertainty estimation in the NAO Soccer Player environment. Results show how uncertainty estimation positively affects the overall results, achieving better performances in all the environments. In particular, we found that on the CartPole environment performances using uncertainty estimation almost double those obtained without uncertainty. Furthermore,

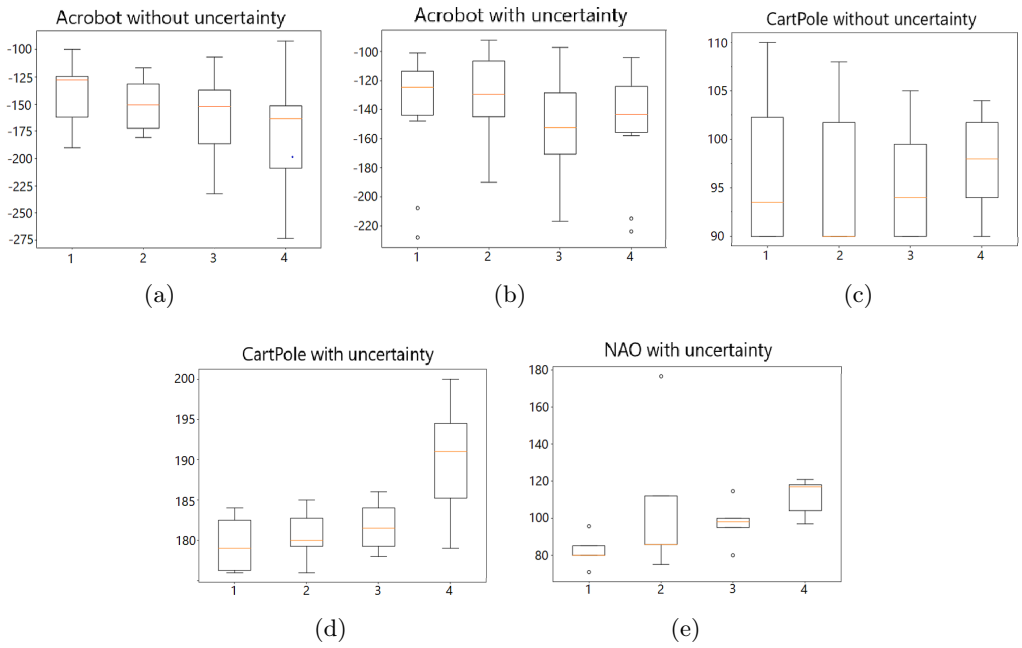
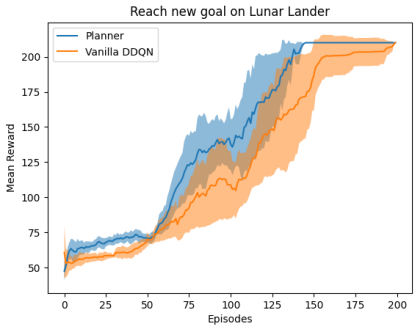


Figure 4.6: Planner performances: Figures show rewards obtained with monitor replanning using the planner acquired through experience with different planning horizons respectively without and with uncertainty estimation, for the Acrobot environment, (a) and (b), for the Cartpole enviornment (c) and (d), and for the NAO robot (e).



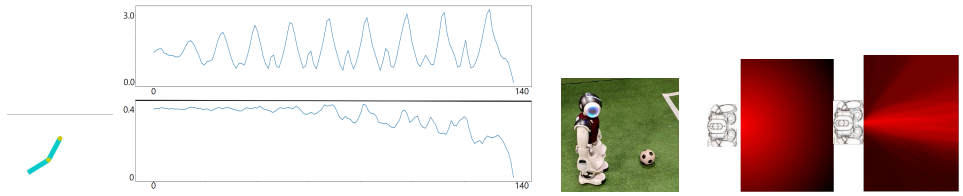
(a)

Figure 4.7: Transfer learning to a new task in the Lunar Lander domain.

in CartPole and NAO environments, we always have a significant improvement in performance increasing the planning horizon. That confirms the importance of reasoning about future actions' outcomes in dynamic scenarios and underlines the robustness of the prediction system proposed in this paper.

4.6.6 Transfer the Symbolic Domain to New Tasks

In order to access the capability of the extracted symbolic domain to represent the Continuous MDP, we also conduct some transfer learning experiments. In particular, we focus on using the planning domain acquired in one MDP to perform a different task in the same environment. The new task is a *reaching* task. Therefore, we choose one continuous state of the MDP as the new goal G_2 and test the agent's capability to exploit the symbolic domain previously



(a) (b) (c) (d)

Figure 4.8: a) Acrobot environment b)Distances from the final state during an episode of Acrobot: distances are computed respectively in the original MDP state-space (top) and in the abstract representation space (bottom). On the y axis: L1 distance from goal; on the x axis: timesteps. c) Soccer player NAO robot. d) ball-distance from the NAO position computed in the orginal state-space (left) and in the abstract-space (right), smaller values shown in brighter red and bigger values shown in darker red.

acquired to reach G_2. To do this, we shape the reward for the new task based on the symbol grounder, as explained in Section 4.7. Here we report one experiment in the Lunar Lander environment, shown in Figure **??**. First, we train a planning domain over 200 symbols solving the original task expressed by the Lunar Lander reward, namely safely landing the spacecraft on the landing pad. Then we re-train the planning models with the reward based on the encoder previously trained, whit the new goal equal to (0,1,0,0,0,0,0,0); hence, we want the spacecraft to stay straight in the center of the screen with zero angular and linear velocity. Figure 4.7 shows the comparative results of the planner and a Vanilla DDQN trained on the new reward. Let us notice that both DDQN and our algorithm have to learn a Q model from scratch, since the value model depends on the old reward function and cannot be reused. The state representation they use differs from the two approaches: DDQN simply uses the ground state, while the Q of our system uses the symbolic state. The experiment shows that the trained encoding speeds up the learning of our system compared to a Vanilla DDQN. This supports the idea that the symbolic representation is sufficiently generic to allow the execution of multiple tasks in the same environment.

4.6.7 Representation Insights

Finally, we investigate the abstract representation learned by our system with training in the different environments. Although the representation is not interpretable from a human point of view, we notice that the abstract space enjoys an interesting property. The transition loss L_{tX} drives the learning to a configuration of the encoder that maps close together the discrete representation of states that are temporally related at training time. Hence, we observed that the distance between two abstract states x and x' is proportional to the *temporal distance* between the associated continuous states s and s', namely the number of steps needed to reach s from s' or viceversa. This property is also at the base of our transfer learning experiment, described in the previous section. Figure 4.8(a) shows the Acrobot in its environment trying to solve a balancing task. Accordingly, Figure 4.8(b) reports the evolution of the distance between the current and the goal state both in the original MPD state-space (top), and the abstract encoded state-space (bottom). The former is computed as $||s_t - s_G||_2$, while the latter is computed as $||e(s_t) - e(s_g)||_2$, where s_t and s_g are respectively the states at time t and at the end of the episode. The two distance profiles show that, as we get closer to the

goal state, in the abstract representation, the distance profile decreases almost monotonically, while in the original state-space, it keeps oscillating until the very end (intuitively, this is due to the inherent nature of a balancing task). Hence, we can easily observe that the distance computed in the abstract state space provides a more robust cost metrics to reach the goal. Similarly, Figure 4.8(c) shows the NAO robot performing a kicking task. The state space used for the task is the ball position in the robot reference frame. Figure 4.8(d) shows for each state point its distance from the state (0,0) (corresponding to the robot position) in the original MPD state-space (left), and the abstract state-space (right). Distances are computed as mentioned before, points closer to the origin are colored in a brighter red. Let us notice the distance metric has consistently changed in the abstract space: the figure shows that points with the same angle with respect to the robot sagittal direction are mapped so as to have the same distance from the robot in the abstract space. It seems the system has learned that the most important state feature to represent, to perform the task, is the ball angle with respect to the robot sagittal direction, and this is perfectly coherent with the task.

Investigating the learned ground-abstract state space mapping, we also individuated another interesting property: the abstraction tends to partition the ground state space more finely around decision thresholds and in a much coarser way away from that. This is shown in Figures 4.10(b), 4.11(b) and 4.12. In figure 4.10 (a), we show distances in the abstract space, as explained before, for the Cartpole environment. In Figure 4.10(b), we show instead how the ground state space is partitioned in the finite abstract state representation, by plotting points in the ground state space in different shades of red when they are mapped to different symbolic states. Let us notice that there are more symbolic configurations than shades of red (255); therefore, colors are reused randomly for different codes. However, it is clear from the figure that the vast majority of symbolic configurations are all concentrated around a particular line. In Figure 4.10 (c), we show the action policy on the ground state space, by encoding in different colors the two different actions of Cartpole. Figure c shows that the line around there is the major concentration of symbolic states, and the action threshold line correspond to each other. In figure 4.10 (d), we show the truth value of each symbol in the symbol set encoded in black (false) and white (true) on the ground state space. In figure 4.9 we compare the symbol grounding for Cartpole when we train and use all the models of our framework (left) and when we train only the encoder and the Q module, and we choose actions without planning as in Q-learning (right). The figure shows clearly that symbols tends to concentrate on the decision threshold because of the training of the transition function. In fact, when the symbol grounding is influenced only by the Q loss, symbols are very more homogeneous in the ground space. Figures 4.11 and 4.12 show the same kind of analysis for AcroBot and NAO Soccer Player, respectively.

In particular, we plot the ground state partition in the different symbolic configurations by sampling ground states on a grid with a given small step and calculating the symbolic code on that ground state. In table 4.1, we report the number of ground states sampled and the number of different symbolic states found processing the ground states with the symbol grounder. We found that even if the possible number of symbolic configurations is 2^{d_X}, where d_X is the symbol set size, the number of symbolic configurations actually used to encode the state space is much smaller. We also found that the three environments analyzed

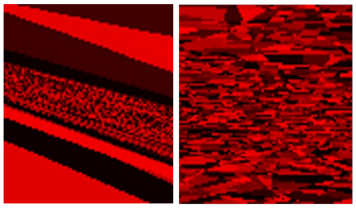


Figure 4.9: Comparison of symbol grounding in the Cartpole environment: (left) symbol grounding obtained with our algorithm, (right) symbol grounding obtained if we train only the symbolic encoder and the Q function, and we interact with the environment without planning, namely choosing in each state the action maximizing the Q function

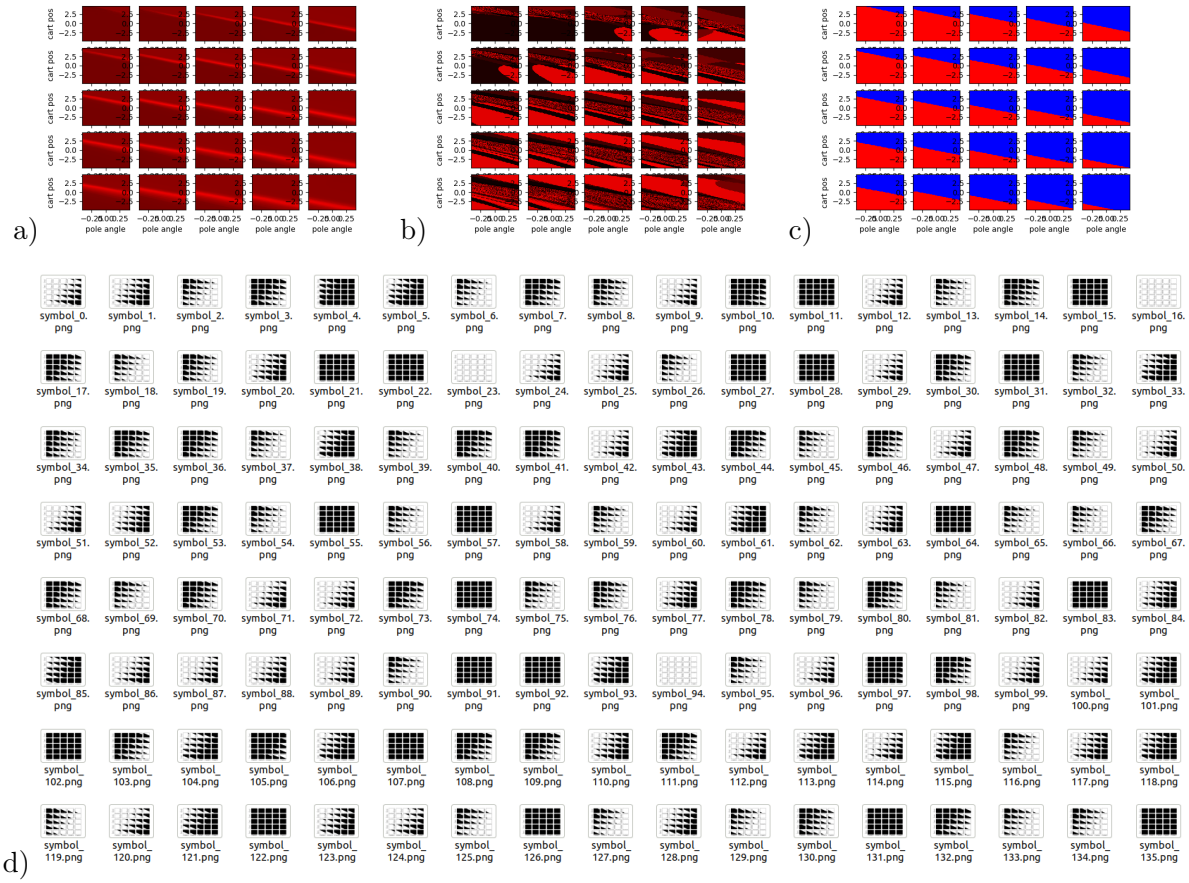


Figure 4.10: Cartpole symbolic representation: a) distances in the symbolic space from the symbolic code associated with (0,0,0,0); b) different codes, any different code is shown in a random shade of red; c) action to take according to the optimal policy; d) truth value of each symbol in the state space.

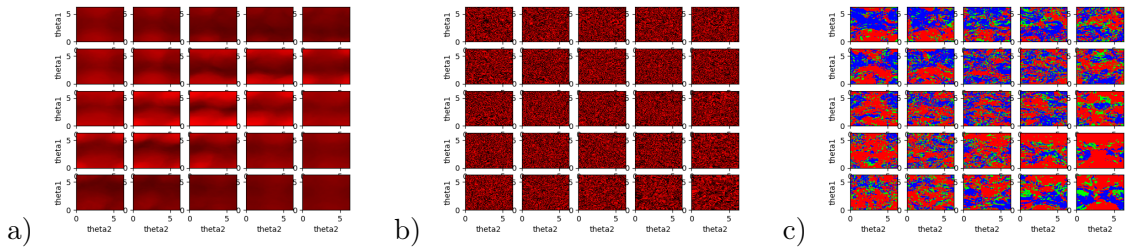


Figure 4.11: Acrobot symbolic representation: a) distances in the symbolic space from the symbolic code associated with (0,0,0,0,0,0); b) different codes, any different code is shown in a random shade of red; c) action to take according to the optimal policy; d) Heatmap of the weights assigned to each symbol by the policy network, each column is a symbol, green represent a weight near to 0, yellow an average weight and red an high weight

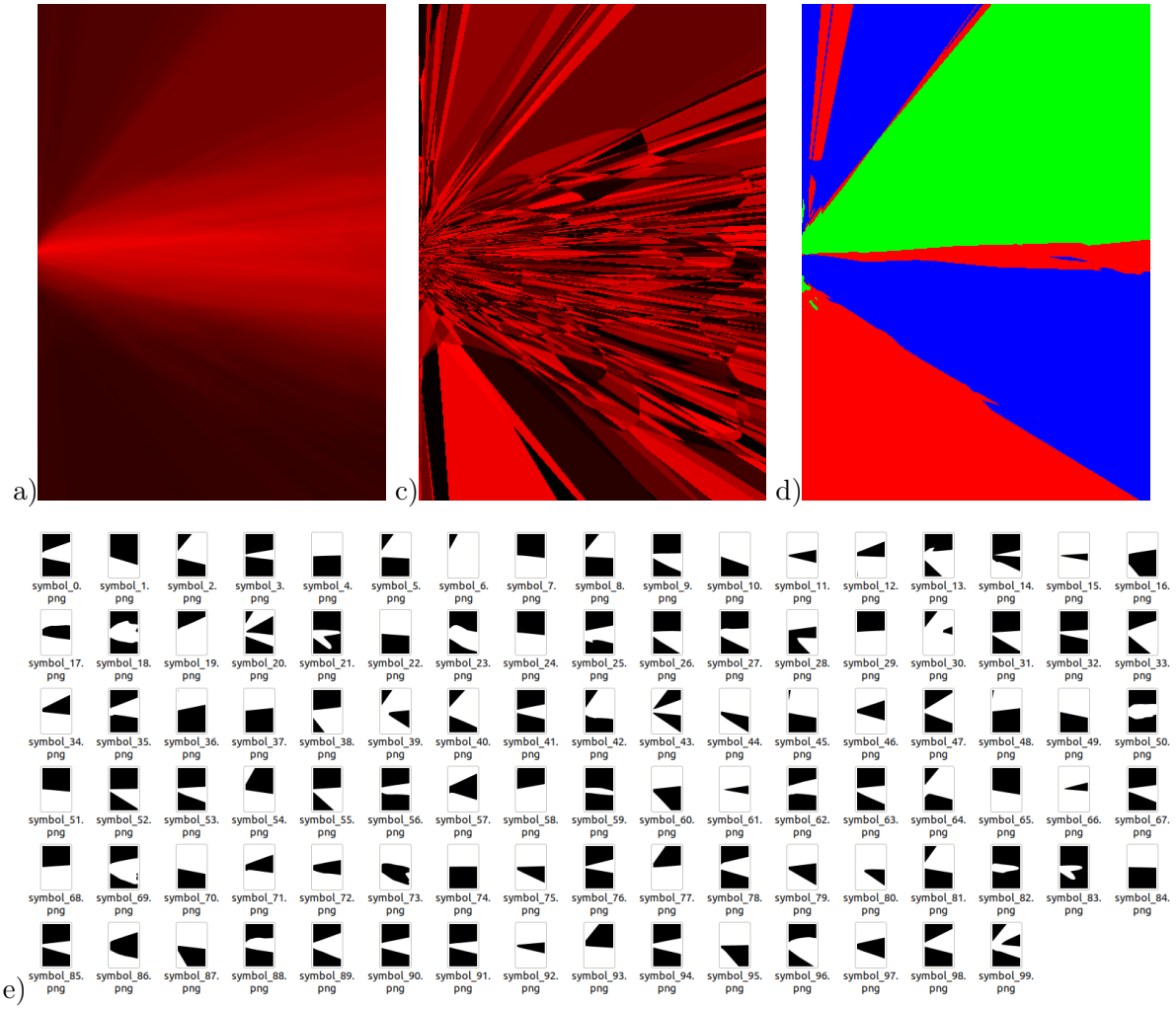


Figure 4.12: NAO symbolic representation: a) distances in the symbolic space from the symbolic code associated with (0,0); b) different codes, any different code is shown in a random shade of red; c) action to take according to the optimal policy; d) truth value of each symbol in the state space.

Env	#ground states	# symbolic states	% symbolic states over ground states
Cartpole	201000	17000	8%
AcroBot	200000	145000	72%
NAO	175000	3000	1.7%

Table 4.1: Percentage of different symbolic states found in the analyzed environments

use very different amounts of symbolic states. For example, in AcroBot, 72% of the ground states sampled are assigned to unique symbolic configurations. To solve this environment, we need the state to be continuous; therefore, the ground state space is discretized very finely everywhere, pushing the symbol grounder capacity to the max. To solve Cartpole, instead, we only need to be very precise around the equilibrium point (corresponding to the action decision threshold); therefore, the grounding is very fine around that and coarse elsewhere. As a result, the symbolic states used are only 8% of the sampled ground states. In the NAO environment, we found the smallest percentage of symbolic states: only the 1.7% of sampled ground states. We hypothesize that it is because the NAO's actions are more high-level than those of the other environments, which encourages a coarser abstraction.

In conclusion, examining the symbolic abstraction, we found that

- distances in the abstract space tend to be proportional to the temporal distance between the associated ground states.
- As expected, the symbolic representation is task-based. In particular, it is not uniform, and we have a very bigger number of codes around the decision threshold. This is very easy to visualize in the Cartpole and Nao environments, where decision thresholds have simple shapes. In Acrobot, instead, changing the state of even just a little changes the optimal action to take; it follows that decision thresholds are entangled, and the number of discrete codes is very bigger.
- The method to obtain the symbolic representation is very elastic, giving no prior to the discretizer, the system is able to reduce the number of states consistently with the complexity of the task (e.g. Acrobot recquires very much more states than Cartpole and Nao).

4.7 Related Work

The use of learning approaches for determining compressed and interpretable representations of the agent environment has been a very active research topic in the last years. Following this idea in [70], authors propose an end to end reinforcement learning architecture based on a neural network back-end and a symbolic front-end for addressing tabular game environments. In this work, a neural network is used to perform a symbolic representation of the agent state-space, then the symbolic state is used to obtain an effective action policy. The idea of determining a symbolic state to learn a propositional planning domain is introduced in [6, 8, 11, 13], the learned planner is then used to solve a puzzle task. The system is based on the use of two different autoencoders: 1) a State Autoencoder for the propositional

representation of the image, 2) an Action Autoencoder for obtaining the action transition definition on the domain. The works by Asai and collegues address the theme of the stability of symbols, that is, the system's ability to maintain the representation unchanged for small perturbations; this goal is obtained by using a State Autoencoder. Authors have extended the planner learning algorithm introducing First-Order State Autoencoder, unsupervised architecture for grounding the first-order logic predicates and facts. In this architecture, each predicate models a relationship between objects by taking the interpretable arguments and returning a propositional value. A similar task has been faced in [149] using an unsupervised learning approach. Differently from the previous works, in [22], authors address the problem of determining the first-order representation from a non-symbolic input. The model is not learned with deep learning approaches, but it is extracted from the state-space structure. The required input is a labeled directed graph. All the works mentioned so far address environments with a discrete, even if sometimes complex, setting. Our system is meant to address also fully continuous and dynamic problems with multiple continual variables; this allows to address more problems closer to the real world environment. Different kinds of scenarios are addressed in [55]. In this case, a variational autoencoder is used to learn relevant features in Atari games given images as training data. The planning is done with RolloutIW using the features learned by the VAE. The algorithm proves that VAEs can learn meaningful representations that can be effectively used for planning with RolloutIW. Similarly, in [35], an external variational autoencoder is used to obtain a propositional representation of the environment's state space. The propositional state is then given as input for a tabular reinforcement learning system and a neural network approach for learning state transitions. The combined architecture is exploited to learn with improved sample efficiency plans for solving the proposed tasks. Both these works involve the use of external VAE's for the states' propositional encoding. In our work, the code is directly learned through the Q-network and transition network, creating a symbolic representation encoded following the task specification. This allows to have state task-based state representation, useful for representing both the environment and the task associated to it. The idea of determining a model for a dynamic environment is addressed in [106]. In this paper, authors present infoGAN, a system based on Generative Adversarial Networks [37] to learn a plannable representation of dynamic problems. This approach identifies intermediate observation points in the agent task execution and the formalization of them in several kinds of symbolic representation. However, this approach does not reason about the action plan, but about intermediate state formalization for trajectory path planning problems. In [87] the authors propose Dreamer, a model based RL algorithm that learns behaviours by using latent imagination. In a follow up work [88], they propose Dreamer v2, which use a categorical representation for the latent space, that appear to be very beneficial when applied to Atari games. Both Dreamer v1 and Dreamer v2 do not learn an abstract representation of the environment, because the representation highly depends on the reconstruction error, while we learn the abstract state-space encoding and transition and values over abstract states *at the same time* in a end-to-end fashion. In particular, Dreamer v2 use discrete representation only in visual games, like Atari games, and does not test it on continuous control problems. Furthermore it use a categorical code of size 32×32, which correspond to a huge discrete state space made of 32^{32} possible different symbolic configura-

tion, very much larger than the one used in our work. Finally, in [63], authors present CRAR (Combined Reinforcement via Abstract Representations), a hybrid architecture composed of a model-based and a model-free component that jointly infer a sufficient abstract representation of the environment. This is achieved by explicitly training both the model-based and the model-free components, including the joint abstract representation. The CRAR agent creates a low-dimensional representation that captures task-specific dynamics, even without any reward. This last work introduces an architecture for obtaining a restricted meaningful abstract representation of the state-space but still relies on a continuous encoded space; Our work instead relies on a fully propositional symbolic representation of the environment.

4.8 Discussion

In conclusion, this chapter proposes an interactive learning algorithm inspired by both planning and reinforcement learning, for automatically learning a symbolic planning domain from a continuous-state MDP. The data gained by experience in the MDP are used to train online three neural network models: a discrete encoder which extract symbols from ground states, a STRIPS-like transition network and a Q-network over the symbolic state. These models allow for fast symbolic online planning over a finite horizon at each interaction step. The planner can reason only with the automatically grounded symbolic representation, without the need for complex trajectory propagation and evaluation in the original continuous state space. We have shown that reasoning in the symbolic space is enough to effectively guide the agent's action in the original continuous environment, and that increasing the planning horizon the agents can improve the performance of the previously trained models.

We started analyzing the use of the learned symbolic domain as a *complete* domain, namely a domain that can be used to plan a sequence of actions for any possible couple initial state-final state in the original state space. We have shown that the planning domain can be *retrained* so to achieve different goals in the same environment through a reward shaping based on the found symbolic representation. As a future direction, we aim at focusing on the domain re-usability trying to speed-up or even eliminate the second training necessary for achieving a new goal.

Furthermore, we observed that incorporating our uncertainty approximation based on the discretization error is high beneficial for planning, especially for larger planning horizons.

We remark that our system autonomously learns how to ground the symbols from the environment observation, even for very complex continuous environments. Therefore, this mapping could not be perfectly understandable by a human. However, this does not affect the validity of the symbolic space. If the training successfully ends, this implicitly guarantees the learned symbolic space is enough variegate and informative to express the environment dynamics. In particular, since we are forcing the system to use a very much smaller representation than the original continuous one, possibly cutting out a lot of information, we found that the system learns to capture only very task-specific characteristics of the state with the symbols, leaving out the rest.

Part III

Discovering logical knowledge in non markovian non-symbolic domains

In this part of the thesis, I focus on discovering logical knowledge in non-symbolic and intrinsically non-Markovian tasks. It means the task generates time sequences that must be analyzed entirely; in other words, we cannot consider each step separate from the others. Many tasks can be modeled as non-Markovian, especially in the field of Natural Language Processing (NLP), Business Process Management (BPM) [75], and robotics [90]. In particular, we focus on non-Markovian tasks that generate *non-symbolic* observations. So, also in this case, we consider the entire model that underlies the task as the composition of two modules: 1) a *perception* module, which deals with extrapolating the truth values of a symbolic representation from the observations, 2) a wholly logical and symbolic module, that describes the task from a formal point of view, without any link to the non-symbolic data that it generates. Different from the approach we followed in the last part, in this part, we focus on one module at a time. In particular, Chapter 5 focuses on exploiting the knowledge of the symbolic model to learn the perceptual model. Chapter 6 focuses on how to learn the symbolic temporal model in the presence of *uncertain* symbolic observations and in a way that can be integrated with a perception model. Finally, chapter 7 explores possible integrations of the two modules previously illustrated in a single neurosymbolic framework.

Chapter 5

Symbol Grounding Exploiting LTLf Knowledge

A crucial problem in neurosymbolic integration is handling the symbol grounding problem without direct supervision. We refer to symbol grounding [146] as the process of mapping raw data into an interpretation over a finite boolean symbolic alphabet, where each symbol expresses a meaningful high-level concept. In particular, we focus on grounding symbols in raw data sequences using some prior symbolic knowledge expressed in Linear Temporal Logic interpreted on finite traces (LTLf) [46]. LTLf is used in a big variety of domains, from robotics [90] to Business Process Management (BPM) [75], for specifying temporal relationships, dynamic constraints and performing automated reasoning. It is unambiguous compared to natural language, yet easy to use and understand. Evaluating if a symbolic sequence is compliant with a given LTLf formula is straightforward. In several real-world applications, however, such sequences are not symbolic but appear 'rendered', or grounded in raw data such as images, videos, words, audio, etc. In some application domains, such for example in BPM [103], we could know a high-level specification of the process expressed in terms of symbols, yet exploiting this knowledge is impossible unless it is grounded in the data. Therefore, symbol grounding represents the first preliminary step to be made to perform any logical reasoning, included evaluation.

Deep neural networks perform extraordinary well in perception tasks on raw data [79]. Supervised classification can be seen as grounding a set of classes in the dataset, by training directly on a set of (data, class) examples. Despite the success of deep learning in this area, the main drawback remains the acquisition of the labeled data necessary for training. State-of-the-art self-supervised approaches have seen enormous progress lately; they can compress high-dimensional data and cluster it in a meaningful way [82] [28] without using any label. In particular, some approaches can also extract a discrete representation of the input data [99], which can be considered an interpretation over a symbolic alphabet [9] [56] [156], as we have seen in the last chapter. However, we do not know the meaning of these automatically-extracted symbols, and inspecting them or connecting them to some human-designed knowledge remains very hard.

In this work, we take a step in the direction of grounding a known meaningful set of symbols in perceptual data, with as little supervision as possible, by exploiting some prior

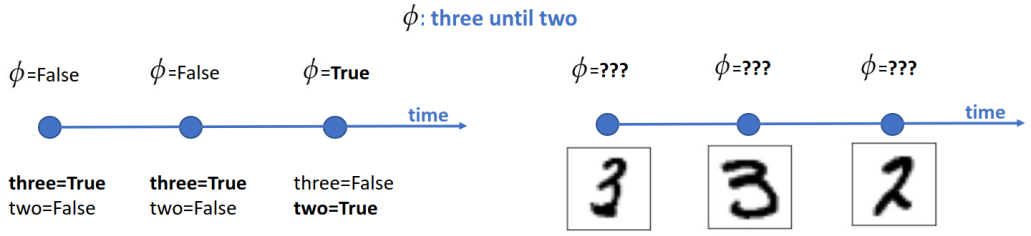


Figure 5.1: When the LTLf symbols are not grounded in observation data the knowledge of the formula is useless to the task

knowledge about expected sequencing expressed as an LTLf fomula in the same symbolic alphabet. Our framework is based on translating the LTLf symbolic knowledge into an equivalent deterministic finite automaton (DFA) and encoding the latter using fuzzy logic. The use of fuzzy logic has seen many successes in neurosymboilc AI [161], and many framework are based on it, such as Logic Tensor Networks (LTN) [16] and Lyrics [121]. Unlike prior work, we focus on grounding knowledge into time-extended data sequences. Our work is similar to LTN, but extends it to temporal logic and time-extended data. LTN extends First Order Logic (FOL) to make it compatible with machine-learning tasks. For example introducing the concept of a *dataset* containing more data samples, and the concept of *feature*. However, the concept of *time* is still missing, in the sense that encoding knowledge on a set of examples (batch dimension), each represented by a sequence of data (time dimension), eventually multidimensional (feature dimension), is not straightforward. In our work, we manage the time dimension by applying recursion over different time steps, in the same way recurrent neural networks do.

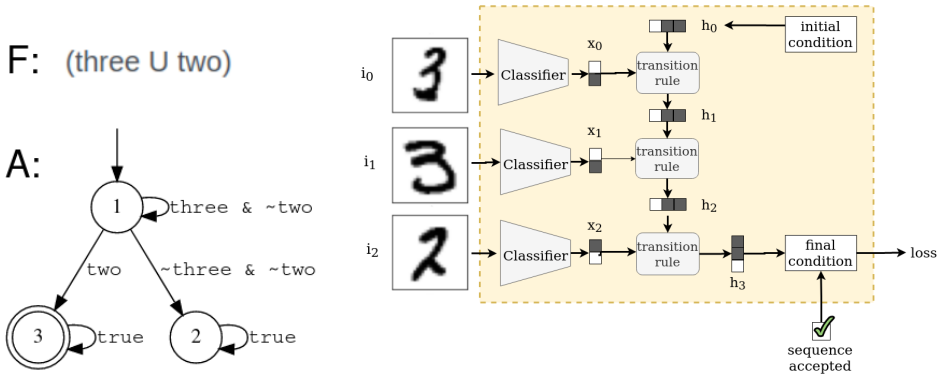
In summary, the main contribution of this chapter is a framework able to encode temporally extended specifications and ground them on sequences of images of any length, through a recursive structure. Experiments show that our method effectively classifies both sequences and single images. In particular, it is faster, requires less data, and is more robust to overfitting than a classical end-to-end classical neural approach that cannot use high-level knowledge.

I organize this chapter as follows: first I formulate the problem we want to solve in Section 5.1; in Section 5.2, I illustrate in detail the developed framework used to solve it; I report the experiments evaluating our approach in section 5.3; in Section 5.4, I discuss the limitations of our method and in particular the problem of 'groundability'; I review related work in section 5.5; and finally I conclude and discuss directions for future work in section 5.6.

The content of this chapter was published in [159].

5.1 Problem Formulation

We consider the problem of classifying a sequence of images $I = i_0, i_1, ..i_{l-1}$ as compliant or not with a certain specification expressed as an LTLf formula ϕ. Each image is the 'rendering' of a symbolic interpretation over the formula alphabet P. This means that there exists a function $c : \mathcal{I} \rightarrow 2^P$, where $\mathcal{I}$ is the space of images, that maps each image into the truth values of symbols in P. If we map each image in the symbolic space with this function we obtain a trace $p = p[1], p[2], ..., p[l-1]$, where l is the sequence length and $p[t] = c(i_t)$ $\forall 0 \leq t \leq l$. We



(a) (b)

Figure 5.2: a) An example of LTLf formula with the corresponding equivalent automaton, b) our framework

denote with $A_\phi = (2^P, Q, q_0, \delta_t, F)$ the DFA corresponding to the formula ϕ, where 2^P is the automaton alphabet, Q is the set of states, q_0 is the initial state, δ_t is the transition function and F is the set of final states. If we run the trace in the DFA we obtain a sequence of $l+1$ automaton states $q = q[0], q[1], ...q[l]$, where $q[0] = q0$ is the initial state and the last state $q[l] \in F$ if the sequence of images is accepted or $s[l] \in (S - F)$ otherwise.

We are interested in the classifying function c. We assume that we can discover it in a weakly supervised way, namely without using any single-image label in the form of associations $(i[t], p[t])$. In particular, we assume to know the following information: (i) the formula ϕ, from which we can build the DFA A_ϕ, (ii) a set of training data $D = \{< I_1, \bar{y_1} >, < I_2, \bar{y_2} >, ..., < I_n, \bar{y_n} >\}$ where I_k is an image sequence $i_0, i_1, ..i_{l-1}$ and $\bar{y_k} \in \{0, 1\}$ is the ground truth label denoting whether the sequence is accepted or not. We will show this information is enough to learn the mapping from images to symbols, without exploiting any single image labels.

5.2 Framework

We consider our framework as a neural network composed of two parts:(i) a *perception part*, represented by a trainable convolutional neural network $CNN(i; \theta)$ that classifies symbols from images, having parameters θ and approximating the function c we want to discover; (ii) *a logic part*, represented by a non-trainable recurrent structure, that is a fuzzy correspondent of the automaton A_ϕ. Figure 5.2 shows an example of the functioning of our framework.

The sequence of images $I = i_0, i_1, ..., i_{l-1}$ is passed one by one to the classifier, producing l continuous vectors of dimension $|P|$ where P is the set of propositions used by the formula.

We define a fuzzy predicate $Class(c_k, t)$ denoting whether the t-th image in the sequence belongs to class k. The classifier implements the grounding of $Class$. In fact the component k of the CNN prediction for the image i_t in the sequence is the truth value of $Class(c_k, t)$.

$$Class(c_k, t) = CNN(i_t; \theta)_k \tag{5.1}$$

Where we denote with $CNN(\cdot; \theta)_k$ component k of the CNN output. We denote as $x_t = [Class(c_0, t), Class(c_1, t), ..Class(c_{|P|}, t)]$ the fuzzy interpretation over propositions in P at

time t, corresponding to the complete output vector of the CNN. This fuzzy interpretation can be used to proceed on the automaton. Let us notice x_t is a fuzzy relaxation of the t-th point in the trace $\rho[t]$, referring to the background section 2.1.3.

In particular, at any time t we are in a state q_k of the automaton, we encode this information with another fuzzy predicate $State$, where $State(q_k, t)$ is true if we are in state q_k at time t. Again, we define as h_t the interpretation at time t over the state symbols $h_t = [State(q_0, t), State(q_1, t), ..., State(q_{|Q|}, t)]$, with $|Q|$ equal to the number of states in the DFA. Let us notice h_t is a fuzzy relaxation of the state at time t denoted for (bolean) DFAs $q[t]$, see section 2.1.3.

We denote the input and the state at time t of the fuzzy DFA x_t and h_t, respectively, to stress the connection with recurrent neural networks.

We start at the initial state q_0 of the automaton, and we have therefore

$$State(q_0, 0) = \top \wedge (State(q_i, 0) = \bot \; \forall 1 \leq i \leq |Q|) \tag{5.2}$$

Then we simulate a run of the automaton using the fuzzy symbolic interpretations x_0, x_1, x_{l-1} the classifier has predicted classifying the images.

We recall that for (boolean) DFAs, if at time t we are in a state q_i and we receive a certain interpretation $\rho[t]$ over the set of symbols, at time $t+1$ we transit to the state q_j linked to q_i by the edge $e_{i,j}$ that is made true by the interpretation of symbols in $\rho[t]$. For example, if we are in state 1 of the DFA in Figure 5.2(a),and we receive the interpretation $['three' = False, 'two' = False]$ we move to state 2 because the interpretation satisfies the formula $\neg three \wedge \neg two$ on the arc $e_{1,2}$. More formally $State(s_j, t+1) = (State(s_i, t) \wedge e_{i,j}(\rho[t]))$, where we denote as $e_{i,j}(\rho[t])$ the truth value of the formula on arc $e_{i,j}$ when evaluated on the interpretation $\rho[t]$.

We apply the same transition rule in fuzzy logic. In particular:

$$State(q_j, t+1) = \bigcup_{i:(i,j) \text{ is an edge of } A_\phi} State(q_i, t) \wedge e_{i,j}(x_t) \tag{5.3}$$

Finally we evaluate the state in the last step h_l and we impose this must be either: one state in F if the sequence is accepted; or one state not in F if the sequence is negative. For this purpose we define the predicate $Accepted(I)$ that is $\top$ if the label $\bar{y}$ associated with I in the dataset is 1, and $\bot$ if the label is 0. We know that:

$$Accepted(I) \leftrightarrow \bigcup_{q_k \in F} State(q_k, l) \tag{5.4}$$

Let us notice that we use $\bigcup$ to denote logical disjunction in the previous two equations. We optimize the network so as to maximize the satisfiability of this last formula. In fact the truth value of formula 5.4 depends on the truth value of the last state, that in turn depends on the previous state and the previous classes and so on. Let $s(I_i, \bar{y_i}; \theta)$ be the truth value of Equation 5.4 for one particular sample $(I_i, \bar{y_i})$ in the dataset. The loss associated with that sample is $1 - s(I_i, \bar{y_i}; \theta)$ that is aggregated over all data in the dataset, since the formula must

be true $\forall x$.

$$L = \sum_{i=0}^{i=n} (1 - s(x_i, y_i; \theta)) \tag{5.5}$$

The loss L is backpropagated through the network and the classifier weights θ are updated with classical gradient descent. In particular, the loss depends on the last state prediction, which depends on symbol grounding in the last image and the second-last state, which in turn depends on the second-last image and third-last state, and so on and so forth. Therefore the loss minimization applies backpropagation *through time* as in classical recurrent neural networks. Let us notice the fuzzy automaton works as a recurrent neural network, encoding in its state the memory of the part of the sequence seen so far. For this reason, the classifier can process only one image at a time without needing to remember any latent dependency between the current symbol and the previous symbols, because these dependencies are encoded in the fuzzy automaton.

In summary, our knowledge base is composed of three logical axioms: (i) the initialization condition (Equation 5.2), (ii) the transition rule (Equation 5.3), (iii) the final condition (Equation 5.4). In particular, the initial condition only specifies the initial state and does not depend on the classifier predictions. The transition rule calculates the next state given the current automaton state and the symbol prediction over the current image. This rule is applied recursively as many times as many images compose the sequence. Finally, the final condition applies supervision over the last state exploiting the sequence labels present in the dataset.

We ground the truth value of each logical axiom by using the following fuzzy operators: the product t-norm T_P for conjunction, its dual t-conorm S_P for disjunction, standard negation N_S , and the Reichenbach implication I_R.

$$\begin{aligned} \neg &: N_S(a) = 1 - a \\ \wedge &: T_P(a, b) = a * b \\ \vee &: S_P(a, b) = a + b - a * b \\ \rightarrow &: I_R(a, b) = 1 - a + a * b \end{aligned}$$

Figure 5.2(b) shows the network behavior in case of perfect grounding. In this case, the classifier predicts all one-hot encodings, this represents an ideal situation where no uncertainty is present, and symbols are all perfectly true or perfectly false. Consequently, the output from the fuzzy transitions is also perfectly boolean, and the fuzzy automaton behaves exactly as the original DFA. The benefit of having a fuzzy automaton is that it can predict the sequence of states even with some uncertainty in the symbol grounding layer, while the original DFA cannot handle any uncertainty. Furthermore, transitions are differentiable, and we can back-propagate error through the model.

Intuitively, at the beginning of training, the classifier does not know how to map images into symbols; therefore, this grounding will initially be random and potentially incorrect, so the automaton states produced by taking this grounding as inputs. Although we do not know the ground truth sequence of states, the fuzzy automaton produces a sequence of probabilities over these states that are adjusted to be coherent with the sequence label, which in turn

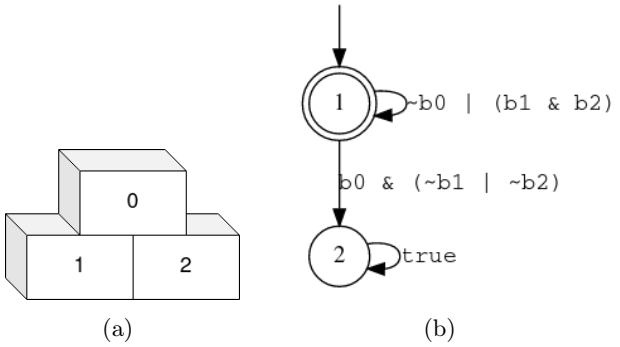


Figure 5.3: Pyramid example. (a) Drawing of the pyramid. (b) DFA corresponding to the instructions to build the pyramid. Bricks are not all groundable through the pyramid instructions, in fact brick 1 and 2 can be confused each other. Expected symbol grounding accuracy: 100% or 33%

adjusts the predicted labels over images. For example, if we know that the DFA accepts the sequence, the last state must be a final state. Therefore, the second-last must be linked to one final state; the third-last must be linked to the second-last, and so forth. All of this is automatically optimized through gradient descent.

5.3 Definition and Examples of Groundability through a Temporal Property

In this section, we formalize the concept of groundability of symbols through an LTLf formula.

We say a symbol $p \in P$ is groundable through an LTLf formula ϕ, where P is the formula alphabet, if p is distinguishable from all the other symbols in the formula alphabet.

Undistinguishability of two symbols: We define two propositional symbols $p_1, p_2 \in P$ *indistinguishable* from each other through the LTLf formula ϕ, defined over symbols in P, if and only if, given a generic finite trace ρ, and the trace $\tilde{\rho}$ obtained by ρ replacing truth values of p_1 with those of p_2 and truth values of p_2 with those of p_1, $\rho \vDash \phi \iff \tilde{\rho} \vDash \phi$, $\forall \rho \in (2^P)^*$.

Let us consider a generic 'rendering' function r that transforms the trace ρ in a sequence of non-symbolic observations $r(\rho)$. We denote the symbol grounding function as r^{-1}. Suppose we want to discover r^{-1} supervising non-symbolic traces $r(\rho)$ with the formula acceptance. Suppose p_1 and p_2 are indistinguishable. In that case, the symbol grounder will never receive an error different from 0 for grounding renderings of p_1 with p_2 and vice versa, and this *does not depend* on the rendering function or the grounding function, but only on the structure of the formula.

Let me further explain the concept with some examples in the Brick World domain. Consider an assembly process for which certain precedence constraints apply to the assembly of the various parts. A simple example of this is building a brick wall. We must impose that if a brick b_0 rests on other bricks $(b_1, b_2, ..., b_k)$, these must be placed on the wall *before* the brick b_0. During the construction process, we indicate with the propositional variable b_i whether the brick i is placed on the wall. Considering the wall in the figure 5.3(a), made up of 3 bricks,

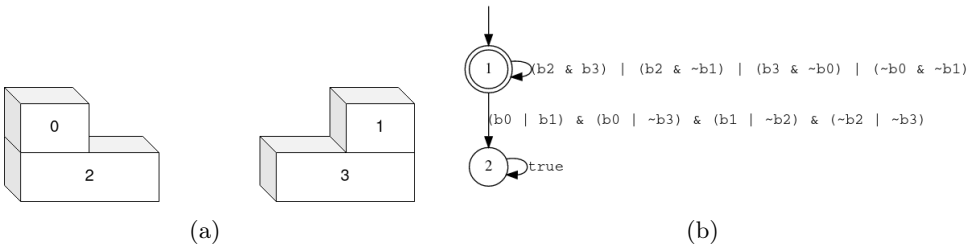


Figure 5.4: Gate example. (a) Drawing of the gate. (b) DFA corresponding to the instructions to build the gate. Bricks are all not groundable through the gate instructions, in fact brick 0 and 1 can be confused each other if bricks 2 and 3 are confused each other. Expected symbol grounding accuracy: 100% or 0%

the precedence constraint corresponds o the following LTLf formula.

$$\phi_{pyramid} = \Box(b_0 \Rightarrow (b_1 \wedge b_2)) \tag{5.6}$$

Let us suppose we want to ground the symbols b_i in observations of the wall, while it is under construction, supervising the grounding with compliance with the given assembly instruction $\phi_{pyramid}$. Since the formula accepts both the sequences in which bricks are placed in order b_2-b_1-b_0 and in order b_1-b_2-b_0, only symbol b_0 is 'groundable' through the formula, while b_1 and b_2 are not distinguishable from each other. We can infer this also by looking at the formula $\phi_{pyramid}$ or the DFA $A_{pyramid}$ shown in Figure 5.3(b). If we replace symbol b_1 with symbol b_2 and vice versa, the formula does not change. Equivalently, if we exchange the two symbols on all the arcs of the DFA, the automaton is not modified.

The given definition of indistinguishability is not strong enough to cover all the cases where symbols are ungroundable. Let me clarify why with another example in the Brick World domain. Consider the 'gate' example, shown in Figure 5.4, described by the formula

$$\phi_{gate} = (\Box(b_0 \Rightarrow b_2)) \wedge (\Box(b_1 \Rightarrow b_3)) \tag{5.7}$$

According to the previous definition, the bricks are all groundable in this example because there does not exist a couple of symbols satisfying the definition of indistinguishability of two symbols. However, in this example, none of the bricks is *really* groundable. Because bricks 0 and 1 can be confused by each other *if* bricks 2 and 3 are confused with each other. Therefore a symbol grounder is supposed to achieve either 100% or 0% accuracy, no matter the rendering function.

For this reason, we extend the definition to a *set* of symbols as follows.

Ungroundability of a set of symbols:
Given n couples of symbols $(p_{1,1}, p_{2,1}), (p_{1,2}, p_{2,2}), ..., (p_{1,n}, p_{2,n})$, with $p_{1,1}, p_{2,1}, ..., p_{1,n}, p_{2,n} \in P$, and $p_{1,1} \neq p_{2,1} \neq ... \neq p_{1,n} \neq p_{2,n}p$ (all considered symbols are different each other). We say the set of symbols $\{p_{1,1}, p_{2,1}, ..., p_{1,n}, p_{2,n}\}$ is ungroundable if and only if $\forall \rho \in (2^P)^*$, let $\tilde{\rho}$ be the trace constructed by replacing truth values of $p_{1,i}$ with those of $p_{2;i}$ fora all $1 \leq i \leq n$, $\rho \vDash \phi \iff \tilde{\rho} \vDash \phi$.

Consequently, we define a symbol as *goundable* when it does not exist any set of substitutions that makes it indistinguishable from another symbol.

The reader can verify on the gate formula or DFA, in Figure 5.4(b), that the specification remains the same if symbols b_0 and b_1 replace each other and symbols b_2 and b_3 replace each other, while it changes if we perform only one substitution of the two.

5.4 Experiments

In this section we report the experiments supporting our method. The implementation code is available online at `https://github.com/whitemech/grounding_LTLf_in_image_sequences`.

Since LTL can be used to specify innumerable constraints, we test our framework on a subset of formulas that is as complete as possible and, at the same time, useful for practical applications. We choose, therefore, to test it on the Declare constraints. Declare [130] is one of the prime languages of the declarative process modeling paradigm, and is composed of 20 types of activity constraints expressed as LTLf formulas. See Figure A.1 in the Appendix for the complete list of Declare formulas. Declare formulas assume that one and only one propositional symbol is true at each instant of time and the other are false, that is symbols are *mutually exclusive*. In our experiments, we also consider the opposite configuration: textitnon-mutually exclusive symbols, i.e., when in one image more than one class is predicted (multiple symbols set to true in a given instant of time) or no class is predicted (all symbols are false). For each Declare formula, we perform an LTLf evaluation experiment in three settings: (1) training on the complete dataset; (2) training on a restricted dataset; (3) training on the complete dataset by dropping the Declare assumption on mutually exclusive symbols (see the following section for more details about the dataset creation process).

We report the sequence classification accuracy, that is the ratio of correctly evaluated sequences, and the image classification accuracy, namely the ratio of correctly predicted symbolic interpretation in single images.

5.4.1 Dataset

The dataset is created by rendering symbolic configurations using images of zeros and ones from the MNIST dataset [114]. In these experiments, therefore, we used an alphabet composed of only two symbols. However, we can apply the framework to an alphabet of any size by changing the classifier output layer. For each formula, all the possible symbolic traces with length between 1 and 4 are created. The latter are randomly split in train traces and test traces, we denote as $p_{traces,train}$ the percentage of traces used for training. In the same way, images in the MNIST dataset are randomly divided in train and test images, we denote as $p_{images,train}$ the percentage of images used for training.

We construct the training dataset by rendering train traces with train images and the test dataset by rendering test traces with test images. In this way, the test contains symbolic traces never observed in the training, in which each symbolic interpretation is rendered with an image never observed during training.

We test our approach on three dataset: (i) complete, (ii) restricted, (iii) complete with non mutually exclusive symbols. The complete dataset is built as described above with parameters $p_{traces,train} = 50\%$ and $p_{images,train} = 85\%$. The restricted dataset is constructed by using parameters $p_{traces,train} = 40\%$ and $p_{images,train} = 15\%$. Achieving good perception

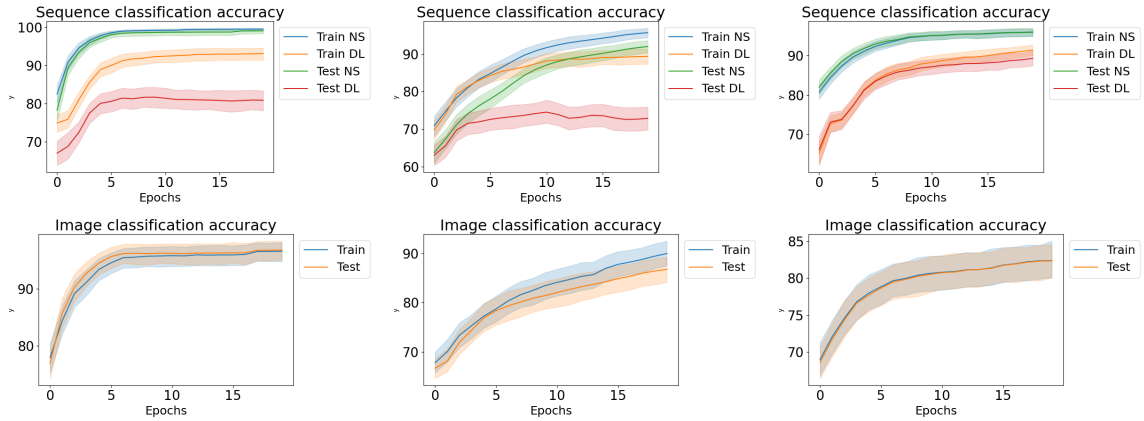


Figure 5.5: Experiments over 20 Declare formulas. In the first row: **sequence classification** accuracy, in the second row: **image classification** accuracy. They are obtained by training on three different datasets: (first columns) **complete dataset**, (second column) **restricted dataset**, (third column) complete dataset with **non-mutually exclusive symbols**. Solid lines represent mean values, shaded areas represent standard deviations.

performances on the restricted dataset is therefore more difficult, since a big percentage of possible renderings are not observed during training.

Images from MNIST dataset render only one digit at time (mutually exclusive symbols), however our framework can be tested also for multilabel classification, as needed when symbols are not mutually exclusive. For this purpose, we create also a dataset rendering interpretations non in MNIST: when all symbols are set to false (rendered as a black image), and when both symbols set to true (rendered as a 'zero' image and a 'one' image superimposed on each other). We create a dataset for multilabel classification by modifying MNIST images as described above and using the same parameters values used for the complete dataset, namely $p_{traces,train} = 50\%$ and $p_{images,train} = 85\%$.

5.4.2 Results on MNIST Dataset

We compare our neurosymbolic approach (NS) with a classical supervised deep learning approach (DL). We implement the latter with a convolutional neural network (the same used by NS) followed by an LSTM. For each approach, each formula, and each dataset, we perform 10 experiments with different seeds, and we keep the best 8 ones. Figure 5.5 show the mean results over the 20 different Declare formulas. In all the plots solid line is the mean, and the shaded area represents the standard deviation. In the sequence classification task, Figure 5.5 (first row), our approach outperforms the deep learning approach in all three datasets, even in the non-mutually exclusive symbol case, although Declare formulas are not designed for this kind of interpretation. This confirms our intuition that the LTLf knowledge can be exploited to simplify the learning process. The lstm-based approach struggles to reach the top accuracy on the test set, and this is even more evident in the experiment on the restricted image dataset. It also happens because in some formulas the lstm tends to overfit the training data, which is visible in the results on the single formulas.

In the image classification task, Figure 5.5 (second row), our approach reaches high accuracy on both the test and training sets without exploiting any image label. Following, we

report the results obtained on each single formula by training on the complete dataset, Figures 5.6 and 5.8; the restricted dataset, Figures 5.7 and 5.9; and the non-mutually exclusive symbols dataset, Figure 5.10 and 5.11.

5.4.3 Discussion on 'Groundability'

Our system does not need any single-data label to ground the symbols of a given formula into data, however, correctly grounding single data using only sequence labels and the formula is not always possible for any arbitrary formula as we have explained in Section 5.3. In particular, if there exists a redenomination of the symbols in the alphabet that maintains all the accepted traces still accepted and all the unaccepted traces still not accepted, multiple groundings are possible. When trained on these formulas, our system can still distinguish one class from the other, but can choose the wrong names for symbols. For example, in our experiment on MNIST digits, the classifier may assign all images of zeros the label 1 and all the images of ones the label 0. In this case, we observe that the accuracy on single image classification approaches 0% while the sequence classification accuracy still approaches 100%. In order to aggregate results from these formulas, which can do either 100% or 0%, we do not plot the value x of image classification accuracy, but the distance from 50%, that is (50 + |x - 50|)% in Figure 5.5 (second row) and in all plots showing image classification accuracy. This value goes to 100%, which means the system correctly clusters all images of zeros together and all images of ones together.

This happens for all the formulas except one: choice(c0,c1), that is $Fc0 \vee Fc1$ LTL, as it can be observed in the results obtained in the single formulas (Figures 5.7-5.11). This formula accepts any trace of mutually exclusive symbols, it is therefore not specific enough, even to cluster images in the correct way. In fact, this formula achieves the highest sequence classification accuracy and the lowest image classification accuracy in the two datasets with mutually exclusive symbols. We would obtain the same results with a tautology or an unsatisfiable formula.

We remark that this problem is only related to the formula and not to our implementation or the particular image classification task we chose. However, our system is compatible with the use of single image labels, which can be employed to ensure the assignment of the correct class names to the clusters in case the LTLf specification is not informative enough to infer them.

5.5 Related works

Integrating Logical Knowledge and Neural Networks Integrating logical knowledge and deep learning is still an open problem, and many different approaches have been proposed. Some works propose embedding logical knowledge and symbolic data in the same feature space and inferring connections between the two using the distance in the feature space as a metric [173] [172]. In this case, the representation quality depends on the training, and obtaining the same exact behavior of the logical knowledge can be hard. Some other approaches use real-valued logic [16] [120], such as fuzzy logic or probabilistic logic, to integrate sub-symbolic perception and symbolic reasoning. The use of real-valued logic is compatible with gradient

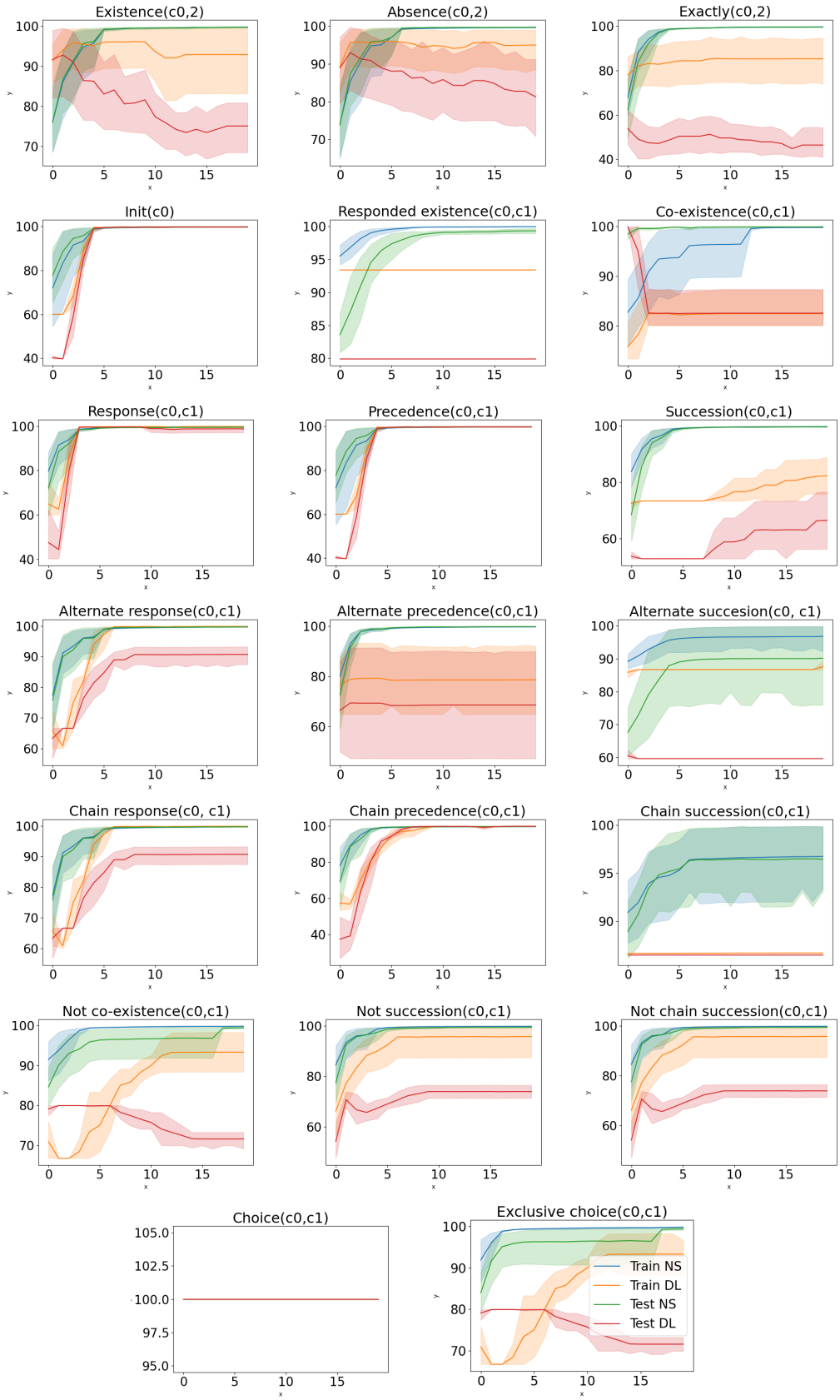


Figure 5.6: Experiments over 20 Declare constraints training on the **full dataset** in **mutually exclusive symbols**. On the y axis: **sequence classification** accuracy ; on the x axis: epochs of training. Solid lines represent mean values, shaded areas represent standard deviations.

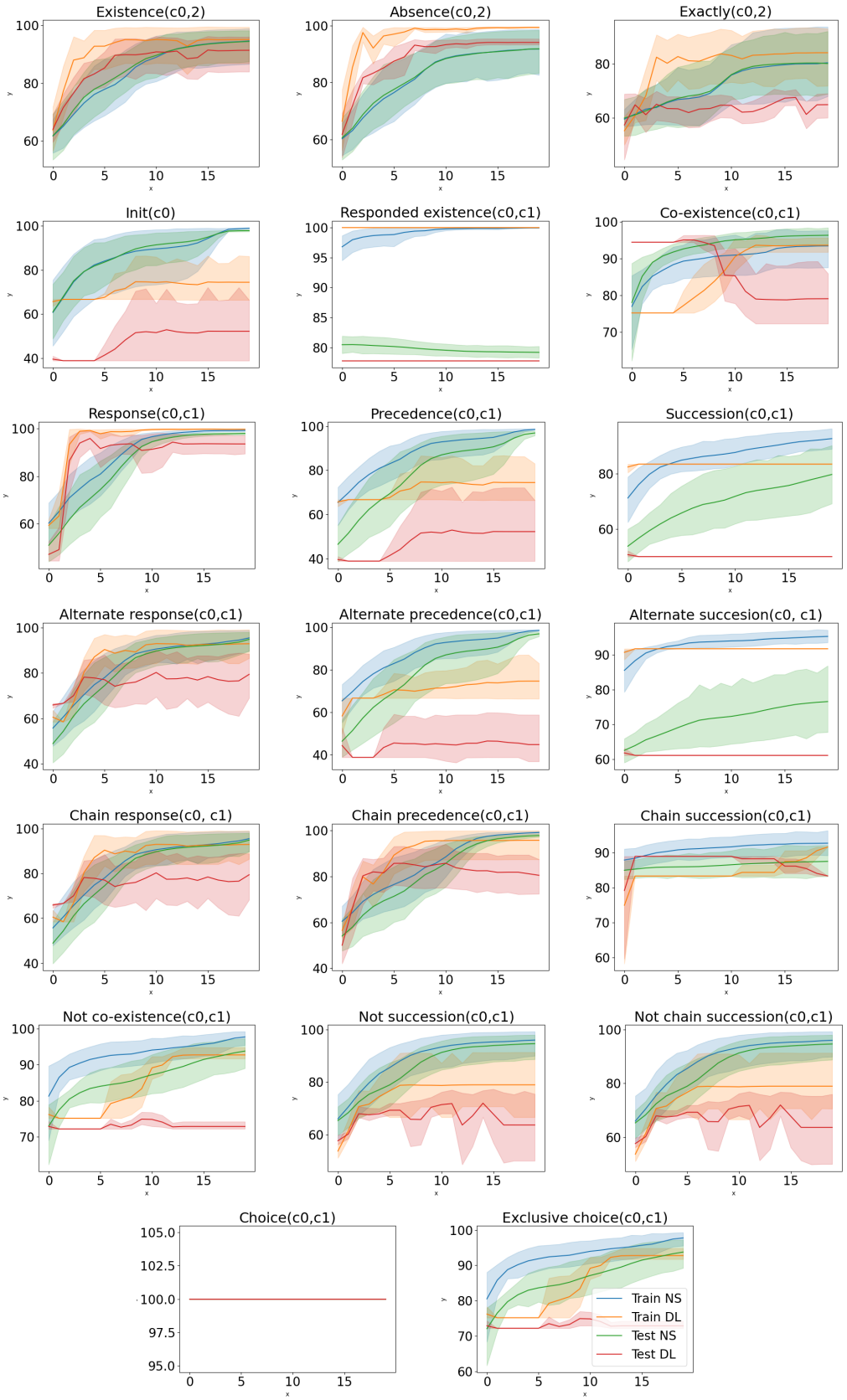


Figure 5.7: Experiments over 20 Declare constraints training on a **restricted dataset** in **mutually exclusive symbols** setting. On the y axis: **sequence classification** accuracy ; on the x axis: epochs of training. Solid lines represent mean values, shaded areas represent standard deviations.

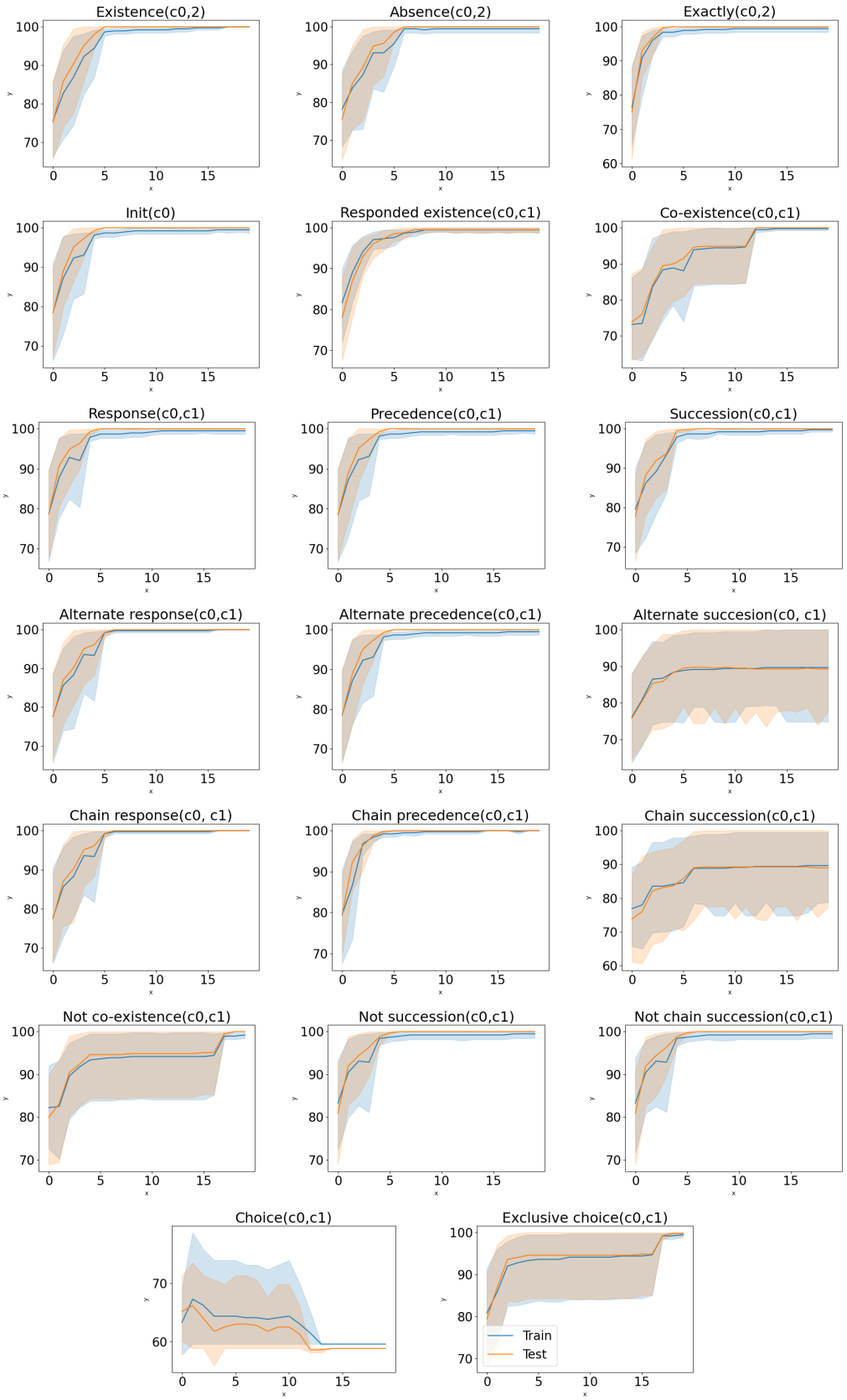


Figure 5.8: Experiments over 20 Declare constraints training on the **full dataset** in **non mutually exclusive symbols** setting. On the y axis: **image classification** accuracy ; on the x axis: epochs of training. Solid lines represent mean values, shaded areas represent standard deviations.

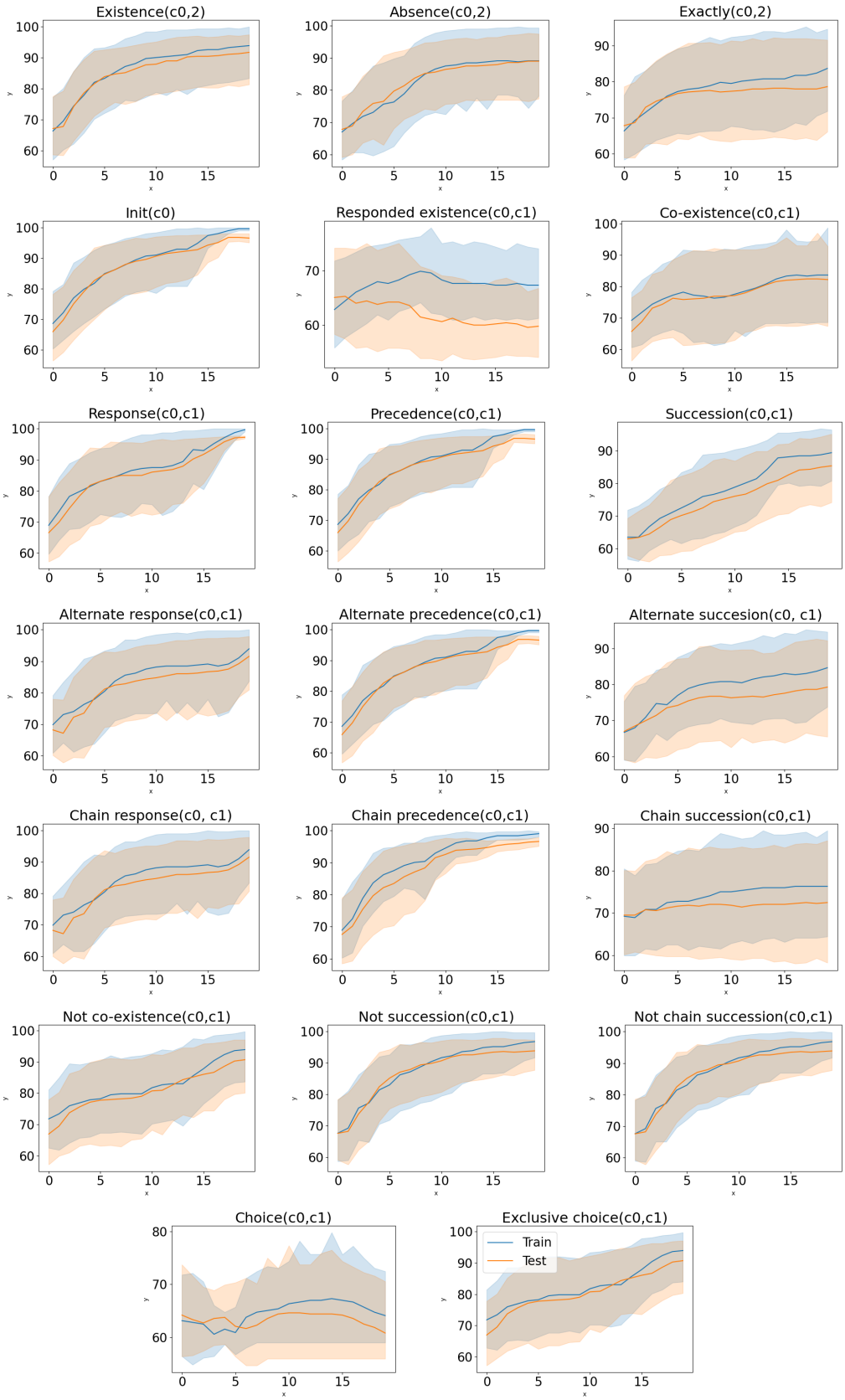


Figure 5.9: Experiments over 20 Declare constraints training on a **restricted dataset** in **mutually exclusive symbols** setting. On the y axis: **image classification** accuracy ; on the x axis: epochs of training. Solid lines represent mean values, shaded areas represent standard deviations.

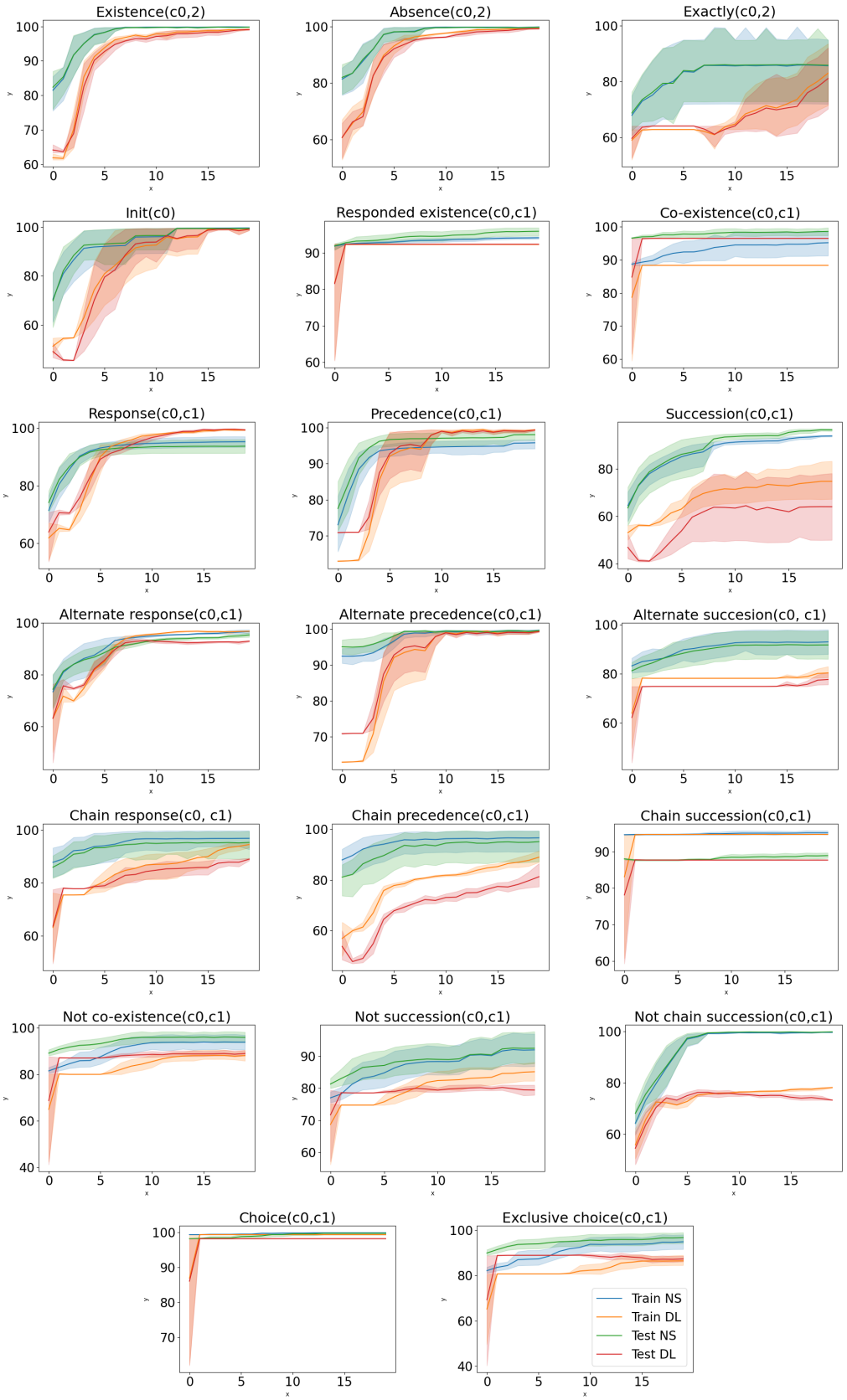


Figure 5.10: Experiments over 20 Declare constraints training on the **full dataset** in **non mutually exclusive symbols** setting. On the y axis: **sequence classification** accuracy ; on the x axis: epochs of training. Solid lines represent mean values, shaded areas represent standard deviations.

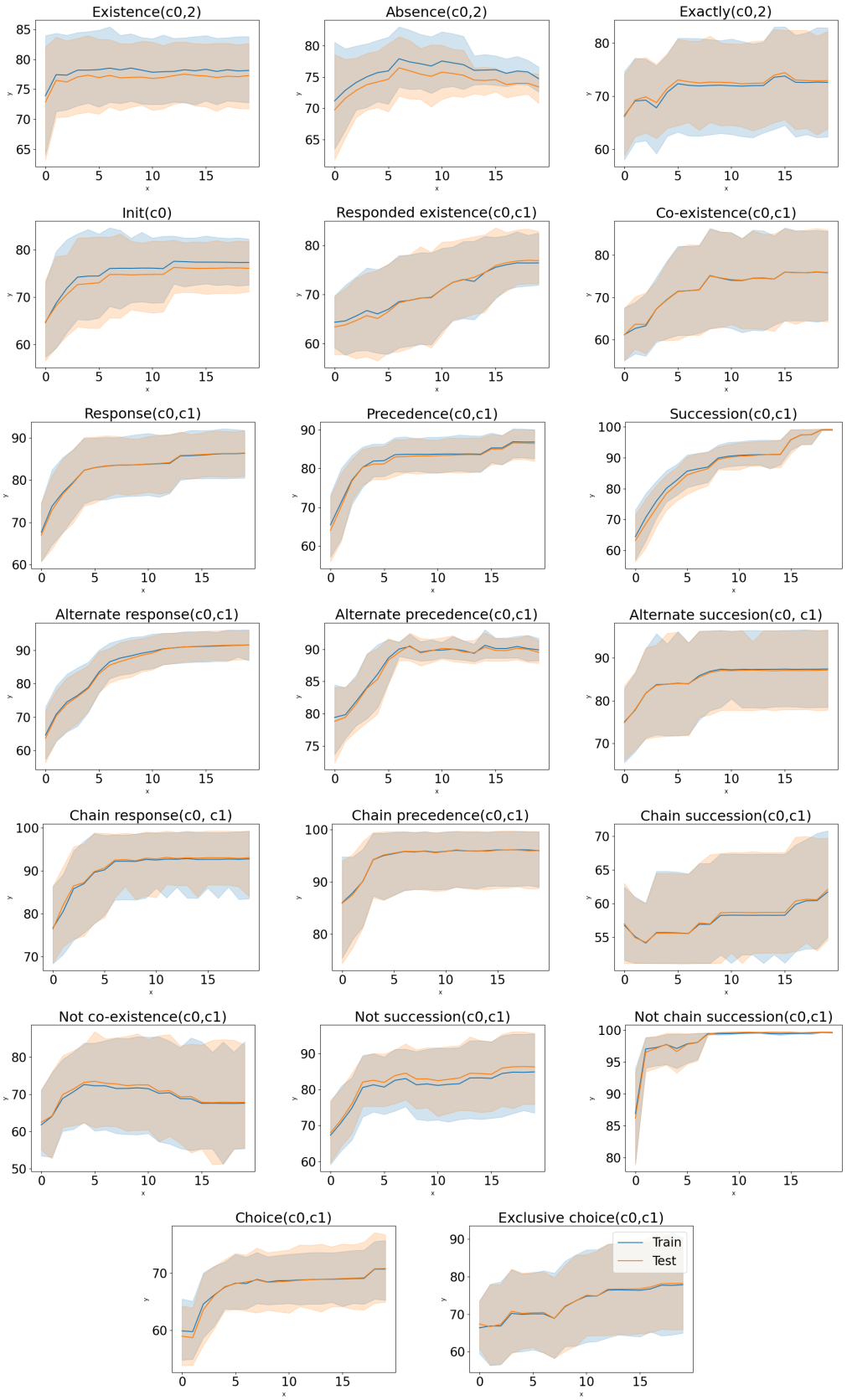


Figure 5.11: Experiments over 20 Declare constraints training on the **full dataset** in **non mutually exclusive symbols** setting. On the y axis: **image classification** accuracy ; on the x axis: epochs of training. Solid lines represent mean values, shaded areas represent standard deviations.

descent optimization that is at the base of neural network training. In this work, we use this second approach and in particular we focus on the use of LTLf knowledge.

Machine Learning and LTL Many works exploit the synergies between machine learning and LTL in a beneficial way. In reinforcement learning, LTL-based reward machines are used to simplify and automate the creation of reward functions for Markovian and non-Markovian decision processes [27][45]. However, they are applicable only in discrete-state environments or continuous problems for which a mapping between the continuous state and a symbolic interpretation is known, also known as labeled MDP [166]. Some works use neural networks to solve problems related to LTL, generally approached with combinatorial algorithms. Camacho and McIlraith [26] use deep learning to guide research in program synthesis and improve scalability. Walke et al. [165] use recurrent neural networks to learn LTLf formulas from a set of traces. However, these works use symbolic data and do not consider the problem of discovering latent symbols in the data, which is the problem we face in our work.

Exploiting High-Level Knowledge for Vision Tasks Previous works have shown that vision tasks can benefit from background knowledge. Stewart and Ermon [147] perform detecting and tracking objects, without any labels, by exploiting known laws of physics. Donadello et al. [57] exploit logical knowledge to increase robustness to noisy datasets with incorrect labels in semantic image interpretation tasks. In particular, our work focuses on classifying images using logical symbolic knowledge instead of image-class labels in a semi-supervised fashion.

Semisupervised Symbol Grounding A benchmark for semisupervised symbol grounding is the digit addition problem, where a system must learn to classify MNIST digits images by knowing only the result of their sum and how addition works. LTN [16] and DeepProbLog [120] show how their systems can benefit from knowing addition rules. However, they handle the problem only in two settings: single-digit and double-digit addition. Dai et al. [40] use logic abduction to correct the prediction of a CNN, by using a derivative-free optimization. They tested their framework on binary sums of digits, where the two binary numbers can have various lengths. We propose a similar experiment on MNIST digits, that does not concern addition and where we do not know in advance the input sequence length. In particular, we evaluate an LTLf formula over sequences of arbitrary lengths of digits by using a recurrent specification in the form of a fuzzy DFA. We use the same approach of LTN, by adapting it to LTLf formulas. To the best of our knowledge, it's the first time LTN has been used to incorporate temporal logic knowledge into neural networks.

5.6 Discussion

In conclusion, we propose a framework for exploiting high-level logical knowledge in the form of LTLf formulas in classification over sequences with neural networks. In particular, we use this knowledge to map images into a set of symbols with a known meaning without any image label. We have shown that discovering this mapping is possible using only sequence-level

labels and the logical knowledge. Furthermore, our approach outperforms the end-to-end approach based on recurrent neural networks in sequence classification: it is more general and can maintain high performances using fewer labels. These results confirm our intuition that the LTLf knowledge can be exploited to simplify tasks comprising classification over sequences, even if the knowledge is expressed in a set of symbols that is not grounded in the type of data composing the sequences.

We have also explained the possible drawbacks of this approach. The main limitation is that the LTLf formula has to be informative enough to supervise the symbol grounding process. Formulas that tend to be trivially positive or trivially negative are not supposed to give enough supervision. Tautologies and contradictions represent an extreme example of this. We also defined when two or more symbols are not distinguishable from each other because of the formula structure. In particular, we remark that, even if our experiments did not use any single data label, our framework could be combined with a supervised loss on symbol grounding, in case the formula alone does not provide enough supervision.

Furthermore, In this chapter, we only conducted experiments on formulas with a binary alphabet. Although experiments with recurrent neural networks showing the end-to-end approach struggle to solve these relatively simple tasks, more complicated settings can be investigated. In this regard, in Chapter 7, we will test symbol grounding through LTLf formulas in more challenging configurations related to non-markovian RL. In particular, we will test longer formulas defined on a bigger alphabet and limit the image sequences to be feasible trajectories observed in the environment. We anticipate that, to tackle this more challenging setting, we will have to supervise the output at each time step (instead of only the last step) and change the formula output to be *less sparse*, passing from DFAs to more general Moore Machines.

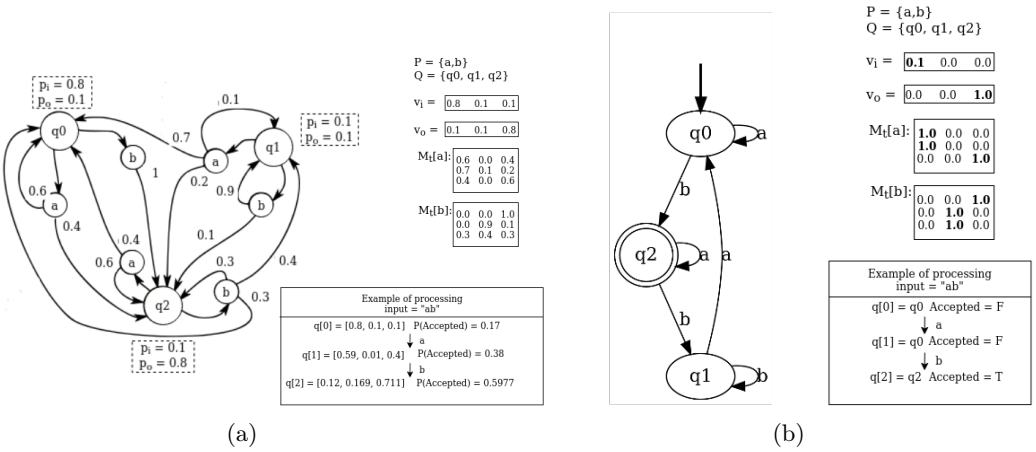
In the future, we want to apply this framework to a more realistic scenario in the area of BPM or natural language processing, and we want to investigate how it can perform in the case of nonperfect or partial symbolic knowledge.

Chapter 6

DFA Induction with Neural Networks

In this chapter we focus on learning the symbolic temporal property from symbolic data, in particular, we will focus on DFA induction. The results reported in this chapter are in the article under review for being published in an international conference [158].

The problem of identifying a deterministic finite state automaton (DFA) from labeled traces is one of the best-studied problems in grammatical inference. The latter sees applications in various areas, including machine learning, formal language theory, syntactic and structural pattern recognition, computational linguistics, computational biology, and speech recognition [47]. Both passive [92] and active [3] exact methods were proposed for DFA identification. These methods are guaranteed to succeed in theory. However, in practice, they require a notable amount of computation, and they are unable to handle errors in the training dataset, making them of limited applicability, especially to real applications and large DFAs. Unlike exact approaches, Recurrent Neural Networks (RNN) can tolerate errors in the training data, and they have proven highly effective at learning classifiers for sequential data [48]. DFAs and RNNs can both be used to match the language recognition task, which is essentially binary classification over strings. Many works highlight the similarities between RNNs and finite-state machines [141] [69]. The two differ in the transition representation: RNNs learn a parametrized transition function in a continuous hidden state space, while DFAs have a finite state space and completely transparent transitions. Furthermore, designing an RNN requires many choices: deciding the number of layers, the number of features of each hidden layer, and all the activation functions. Each of these decisions can affect the final performance and must be taken carefully. By contrast, DFA induction methods do not require nearly any hyperparameter fine-tuning. Many approaches have been proposed to extract a DFA from a pre-trained RNN [122] [168] [169], generally adapting techniques from the grammar induction literature and suggesting ways to discretize or cluster the continuous RNN hidden states in a finite structure. The purpose of these works is not DFA induction but rather to enhance the explainability of black-box sequence classifiers. However, they open the door to DFA induction through neural networks, and join two fields that are classicly kept separated.
In order to take the benefits from both worlds, grammar induction on one side and recurrent neural networks on the other side, we present in this chapter DeepDFA: a transparent neural network design specialized in learning DFAs from traces with gradient-based optimization. The model resembles a recurrent neural network, but, differently from RNNs, it is completely



(a) (b)

Figure 6.1: a) An example of PFA with three states and two symbols: graph describing the PFA, equivalent representation in matrix form, and produced states and acceptance probabilities while processing the string "ab". b) An exmple of DFA: graph describing the PFA, equivalent representation in matrix form, and produced states and acceptance probabilities while processing the string "ab". In particular, the DFA in (b) is obtained by the PFA in (a) approximing the matrix representation to the closest one-hot vectors.

transparent after training, as much as a DFA. Furthermore, when the number of symbols is smaller than 4, it uses fewer weights than the most commonly used recurrent architectures, such as LSTM and GRU, resulting in faster training and less memory consumption. Another benefit is that it only has one hyperparameter, significantly reducing the hyperparameter search. At the same time, since it is trained with back-propagation, it is able to learn DFAs in a significantly shorter time than grammar induction methods, even for large automatons; and it can tolerate errors in the training data. Our method is based on defining a neural network that behaves as a probabilistic finite automaton. We control how much the probabilities are close to categorical one-hot distribution through a temperature value. During training, we smoothly drive the network activations to be close to discrete 0/1 values by changing the temperature. When the gap between the discrete and actual activations is small enough, the network behaves precisely as a DFA.

We evaluate our method on the popular Tomita languages benchmark [152] and random DFAs of different sizes and different sets of symbols. Results show that DeepDFA is fast and accurate. It outperforms an exact SAT-based method [177] when the target DFAs is bigger than 20 states, or the training dataset is erroneous, and it reaches competitive results in the other cases. We also compared DeepDFA to DFA extraction from a pre-trained RNN [169], finding it reaches better accuracy and predicts DFA of size closer to the target DFA size. The remainder of this chapter is organized as follows: in section 6.1 we formulate the problem, we define the DeepDFA neural network model and the training procedure used; we report the experiments evaluating our approach in section 6.2; in section 6.3 we report related works; and finally we conclude and discuss directions for future work in section 6.4.

6.1 Method

We consider the problem of inferring a DFA from a training set of labeled strings $D = \{(x_1, \bar{y_1}), (x_2, \bar{y_2}), ..., (x_n, \bar{y_n})\}$, where x_i is a string of length l of symbols $x[0], x[1], ..., x[l-1]$, in the automaton alphabet P, $x[t] \in P$ with $0 \leq t < l$, and $\bar{y_i} \in \{0, 1\}$ is the associated label, denoting whether the string is accepted or not by the target automaton. Let us notice we denote the t-th symbol in the string as $x[t]$ to underlined it is the integer index of a symbol and not a continuous vector.

To infer the automaton, we define a recurrent neural network model that mimics the behavior of a PFA in a state space $\hat{Q}$ and action space P, where P is the target automaton alphabet. In contrast, $\hat{Q}$ is our hypothesis on the state space, which will generally differ from the true Q. For this reason, the size of $\hat{Q}$ is a hyperparameter of our model.

We cannot use gradient-based optimization to learn the DFA directly because of its non-differentiable transitions and output vector. The intuition behind our work is that PFAs are closely related to recurrent neural networks, since they calculate the next state and output using multiplications between continuous vectors and matrices, in the same way as RNNs do. However, DFAs can also be represented in matrix form, with the difference that their matrix representation is composed of only one-hot row vectors, as shown in Figure 6.1. The latter shows a PFA on the left and a DFA on the right. In particular, the DFA is obtained from the PFA approximating all the PFA matrix representation row vectors to the closest one-hot. It describes, therefore, only *the most probable* behavior of the same system deterministically.

6.1.1 DeepDFA Definition

Following this idea, we define DeepDFA as a parametric PFA in which we can drive the representation to be close to one-hot during training. We obtain this effect using an activation function that smoothly approximates discrete output. Many works in literature [97] [164] [107] use this technique, especially in neurosymbolic AI [20], to learn symbolic logic structures by using differentiable models such as neural networks. In particular, we use a modified version of the classical softmax and sigmoid activation functions, which we call $softmax\_with\_temp$ and $sigmoid\_with\_temp$. Given a function $f(x)$ we define $f\_with\_temp(x, \tau) = f(x/\tau)$, with tau being a positive temperature value.

Our RNN comprises two trainable modules: a transition function and an output function. The transition function $h_t(h_{t-1}, x[t-1]; \theta_h)$ has parameters θ_h, takes as input the probabilities over the previous state, $h_{t-1} \in [0, 1]^{|\hat{Q}|}$, and the prevuous symbol, $x[t-1] \in P$, and predicts the current state h_t. The output function $y(h_t; \theta_y)$ implements the classification module: given the current state estimation, h_t, predicts the probability of the current state to be an accepting state using its parameters θ_y. In particular, fixed a temperature value τ

$$
\begin{aligned}
&h_0 = [1, 0, ..., 0] \\
&h_t = h_{t-1} \times M_t[x[t-1]] \\
&y_t = h_t \times v_o \\
&M_t = softmax\_with\_temp(\theta_h, \tau) \\
&v_o = sigmoid\_with\_temp(\theta_y, \tau)
\end{aligned}
\tag{6.1}
$$

Where M_t is the PFA transition matrix, $M_t[x[t]]$ is the transition matrix for symbol $x[t]$, and v_o is the output vector, as defined in the background section 2.1.3. They are obtained by applying discrete activation functions on parameters $\theta_h \in \mathbb{R}^{|P|\times|\hat{Q}|\times|\hat{Q}|}$ and $\theta_y \in \mathbb{R}^{|Q|\times 1}$. In particular, the softmax activation applied to the third dimension of the matrix θ_h ensures $\sum_{q'} \delta_t(q, a, q') = 1$, and the sigmoid activation ensures values of v_o stay in $[0, 1]$.

In the forward pass, we calculate the probability of a string x in the dataset to be accepted by the RNN by applying Equation 6.1 recursively to the input sequence of symbols $x[0], x[1], ..., x[l-1]$. The final output y_l is compared with the ground truth label $\bar{y}$. We update the model weights with back-propagation by minimizing the binary cross-entropy between model predictions and the ground truth labels.

6.1.2 Temperature Annealing

Cold temperatures force the PFA to behave as a DFA, since the activation values become closer to boolean values as the temperature decreases. When the temperature is low enough, the model transforms into a DFA. Let us notice that using the classical activation functions and discretizing the model at the end is not guaranteed to work. Because, in that case, the discretized model differs from the trained one, and the two can have different performances. However, using a cold temperature from the start of training can prevent the system from learning the model properly. For this reason, we initialize τ to 1. In this way, the system starts the training using the normal softmax and activation functions. After that, we align the temperature to 0 by multiplying it for a positive constant $\lambda < 1$ at each epoch.

6.1.3 Model Minimization

Once the training is concluded, we read the DFA from the activations evaluated with the minimum temperature, and we use the Hopcroft algorithm [94] to minimize the number of states. We emphasize that automata minimization is a well-known problem for which many algorithms are available, as opposed to neural network compression, also known as knowledge distillation [93], which is still an open research problem. This represents another feature in which DeepDFA can take 'the best of both worlds.' We observe that, even if we set the state space size $|\hat{Q}|$ very big, the number of states after minimization tends to be close to the target DFA number of states, which suggests the model is robust to overfitting.

6.2 Experimental Evaluation

6.2.1 Target DFAs

We test our approach on two different sets of DFAs. The first experiment is on the Tomita languages[152], which are a standard benchmark for grammar induction and DFA extraction from RNNs [169] [168]. The benchmark comprises seven formal languages of increasing difficulty defined on the binary alphabet $P = \{a, b\}$. Despite Tomita languages being a popular benchmark, they are represented by small DFAs with state size smaller than six. In order to test our approach on bigger DFAs, we conduct a second experiment on randomly generated DFAs with state size $|Q|$ between 10 and 30, and alphabet size $|P|$ between 2 and 3. For

each setting, we generate 5 random DFAs as described in [178]. $|Q|$ states are generated and enumerated between 1 and $|Q|$, we set 1 as the initial state, and each state is equiprobable to be accepting. We connect each state i with a random state in $[i+1, Q]$ with a random-labeled transition. In this way, we partially build an automaton where all states are reachable from the initial state. Finally, we complete the DFA with random transitions.

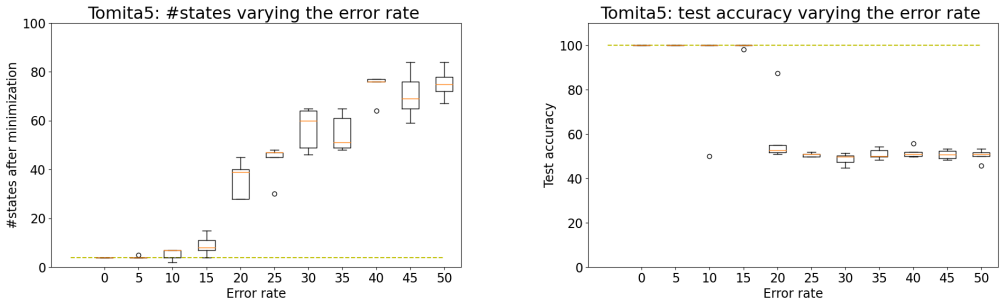
6.2.2 Dataset

For each target DFA, we create a train, a dev, and a test dataset by sampling strings of various lengths and labeling them with the target DFA. The training dataset contains strings with a length between 1 and len_{train}, the dev set is composed of strings of length len_{dev}, and the test set by sequences of length len_{test}. In order to test the model's capability to generalize to longer unseen sequences, for each dataset, we set $len_{train} < len_{dev} < len_{test}$. To prevent the models from learning degenerate solutions, all datasets are almost exactly balanced. We set $len_{train} = 30$, len_{dev}=60, and len_{test}=90 for all the Tomita datasets and the random DFAs datasets with $|Q| < 30$. For the random DFAs with state size of 30, we set len_{train}=50, len_{dev}=100, and len_{test}=150. To test the resiliency of different methods to errors in the training data, we also create a corrupted version of each training dataset by flipping 1% of the labels.

6.2.3 Baselines

Our approach is a hybrid between a recurrent neural network and a DFA. These two types of sequence acceptors are trained with very different methods and present different strengths and weaknesses. In order to better understand the benefits of having a hybrid method, we compare it with one state-of-the-art from the literature on grammar induction [177] and one state-of-the-art method to extract DFAs from recurrent neural networks [169]. The former is DFA-inductor, a SAT-based approach for exact DFA identification. In particular, we use the shared implementation code at [1], and we test with breadth-first search (BFS) symmetry breaking, shown to be the most effective in the paper. The other approach abstracts a finite state automaton from a pretrained RNN, starting from a predefined discretization of the hidden state space and applying the L* algorithm and abstraction refinement when required. To test this method, we first train a one-layer LSTM of hidden state size set to the same as our method. Then we extract the DFA from the trained network by using the code in the public repository [2] with 10 as the initial split depth, and the starting examples set composed of the shortest positive string and the shortest negative string in the train set, as suggested in the paper. We refer the reader to [177] and [169] for further details on the two comparison methods. For each approach, we report the accuracy obtained by querying the final DFA on the test set, the number of states of the predicted DFA $|\hat{Q}|$, and the execution time.

[1] https://github.com/ctlab/DFA-Inductor-py
[2] https://github.com/tech-srl/lstar_extraction



(a) (b)

Figure 6.2: Results on Tomita 5 with different error rates in the training dataset. a) Number of states of predicted DFAs b) Test accuracy

Table 6.1: Comparison between DeepDFA, L* extraction and DFA-inductor on the Tomita Languages. We report test accuracy, mean number of states $|\hat{Q}|$ and the number of parameters used $\#W$

	DeepDFA			L* extraction			DFA-inductor		
Lang	test accuracy	$\lvert\hat{Q}\rvert$	#W	test accuracy	$\lvert\hat{Q}\rvert$	#W	test accuracy	$\lvert\hat{Q}\rvert$	$\lvert Q\rvert$
T1	**100 ± 0**	**2 ± 0.0**	820	100.0 ± 0.0	2.0 ± 0.0	1880	100.0 ± 0.0	2	2
T2	**100 ± 0**	**3 ± 0.0**	820	100.0 ± 0.0	3 ± 0	1880	100.0 ± 0.0	3	3
T3	**100.0 ± 0**	**5 ± 0**	820	100.0 ± 0.0	5 ± 0	1880	100.0 ± 0.0	5	5
T4	**100.0 ± 0.0**	**4.0 ± 0.0**	820	100 ± 0	4 ± 0	1880	100.0 ± 0.0	4	4
T5	**100 ± 0**	**4.0 ± 0.0**	1830	67.77 ± 27.90	454.3 ± 384.2	4020	100.0 ± 0.0	4	4
T6	**100.0 ± 0.0**	**3.0 ± 0.0**	820	50.44 ± 2.67	524.0 ± 183.0	1880	100.0 ± 0.0	3	3
T7	**100.0 ± 0.0**	**5.0** ± 0.0	820	100.0 ± 0.0	5.0 ± 0.0	1880	100.0 ± 0.0	5	5

6.2.4 Training Details

We set the hidden state size as 20 for all the Tomita languages experiments except Tomita5, for which we used a hidden state size of 30. Regarding the experiments on the random DFAs, we set the hidden state size to 100 for learning the DFAs with state size < 30, and 200 for the random DFAs of size 30. As we discuss in section 6.2.6, the hidden state size seems to be at least three times the target DFA size to make the training accuracy able to increase. This motivates the choice of this hyperparameter for the various experiments. All the networks were trained on an Nvidia GPU, with a learning rate of 0.01, until training loss convergence, for a maximum of 200 epochs. In all the experiments we use $\lambda = 0.999$ and minimum temperature $= 10^{-5}$. All the experiments of DFA-inductor was performed on a Intel Core i7-10750H CPU, without any other process running at the same time.

Table 6.2: Comparison between DeepDFA and a DFA-inductor when the training set contains 1% of errors. Results on Tomita Languages.

Lan	DeepDFA test accuracy	DeepDFA $\lvert\hat{Q}\rvert$	DFA-ind test accuracy	DFA-ind $\lvert\hat{Q}\rvert$	$\lvert Q\rvert$
T1	**100 ± 0.0**	**2.0 ± 0.0**	95	12	2
T2	**100 ± 0.0**	**3.0 ± 0.0**	94	10	3
T3	**100.0 ± 0.0**	**5.0 ± 0**	0	32	5
T4	**100.0 ± 0.0**	**4.0 ± 0.0**	0	23	4
T5	**100.0 ± 0.0**	**4.0 ± 0.0**	0	27	4
T6	**100.0 ± 0.0**	**3.0 ± 0.0**	0	25	3
T7	**100.0 ± 0.0**	**5.0 ± 0.0**	0	41	5

Table 6.3: Comparison between DeepDFA and a DFA-inductor when the training set contains 1% of errors. Results on random DFAs.

$\lvert Q\rvert$	$\lvert P\rvert$	DeepDFA test accuracy	DeepDFA $\lvert\hat{Q}\rvert$	DFA-ind test accuracy	DFA-ind $\lvert\hat{Q}\rvert$
10	2	**98.38 ± 3.66**	**8.68 ± 2.10**	0 ± 0.0	-
10	3	**98.95 ± 2.74**	**10.92 ± 1.99**	0 ± 0.0	-
20	2	**95.80 ± 9.04**	**13.86 ± 1.97**	0 ± 0.0	-
20	3	**98.31 ± 3.16**	**20.14 ± 3.39**	0 ± 0.0	-
30	2	**98.56 ± 2.98**	**22.62 ± 1.56**	0 ± 0.0	-
30	3	**96.43 ± 11.48**	**42.02 ± 34.42**	± 0.00	-

6.2.5 Results

Results on Tomita Languages

For each Tomita language, we run 5 experiments with different seeds for DeepDFA and L* extraction and one for DFA-inductor since the latter has a deterministic behavior. We set the hidden state size to 20 for both DeepDFA and L extraction. This number is big enough to learn Tomita languages since each language is a DFA of at most 5 states. We found unoptimal solutions for Tomita5 with this setting, as described in section 6.2.6, so we used a bigger hidden state size of 30 only for it. We report the effect of changing the hidden state dimension on the Tomita5 performances in Figure 6.3. Table 6.1 shows the results.Our approach and DFA-inductor reach 100% test accuracy for all the languages, even for the more complex ones. Our approach outperforms L* extraction from the lstm, especially for Tomita 5 and 6. For these languages L* extracts extremely oversized DFAs of 400/500 states, while our approach correctly estimates DFAs of sizes 4 and 3. However the SAT-based method reaches top performances on a perfect dataset, we show that it is completely unable to handle errors in the training data. We tested our approach and DFA-inductor on the same training datasets after having flipped 1% of the labels; results are shown in table 6.2. With only 1% of errors, DFA-inductor is unable to provide a solution because the process is killed for exceeding the CPU capacity before finding a DFA consistent with training examples. In case of exceeding CPU, we put 0% as test accuracy, and we report in the $\lvert\hat{Q}\rvert$ column the DFA state size DFA-inductor was checking as a hypothesis at the process death time. The method crashes in all the Tomita languages except the first two, for which it loses around 5% of accuracy and overestimates the state space. By contrast, our method is completely

Table 6.4: Comparison between DeepDFA and L* extraction on randomly generated DFAs. The train set does not contain errors.

		DeepDFA				L* extraction			
$\|Q\|$	$\|P\|$	test accuracy	$\|\hat{Q}\|$	execution time	#W	test accuracy	$\|\hat{Q}\|$	execution time	#W
10	2	**98.31 ± 3.51**	**8.72 ± 2.23**	**52.51 ± 13.79**	**20100**	96.64 ± 11.13	7.7 ± 0.88	124.13 ± 21.28	41400
10	3	**99.42 +± 1.69**	**11.56 ± 3.68**	**43.26 ± 15.68**	**30100**	98.56 ± 6.82	132.06 ± 165.04	208.89 ± 42.68	41800
20	2	94.6 ± 9.75	**13.94 ± 2.86**	**49.23 ± 14.00**	**20100**	**99.63 ± 2.26**	39.26 ± 111.31	167.95 ± 37.72	41400
20	3	**96.80 ± 5.09**	**21.24 ± 4.19**	**55.65 ± 12.69**	**30100**	67.79 ± 4.11	314.04 +- 142.27	334.71 ± 52.36	41800
30	2	97.21 ± 4.47	**22.54 ± 1.60**	**988.46 ± 525.06**	**80200**	**99.78 ± 1.06**	42.52 ± 88.03	1216.29 ± 122.60	162800
30	3	**98.08 ± 8.28**	**42.06 ± 29.41**	**866.28 ± 447.92**	**120200**	67.33 ± 17.16	314.2 ± 142.30	2424.41 ± 684.99	163600

Table 6.5: Comparison between DeepDFA and a SAT-based DFAinductor. The train set does not contain errors. For each DFA we keep the experiment achieving best dev accuracy.

		DeepDFA		DFA-inductor		
$\|Q\|$	$\|P\|$	test accuracy	$\|\hat{Q}\|$	test accuracy	$\|\hat{Q}\|$	execution time
10	2	**100.0 ± 0.0**	**8.0 ± 0.0**	100.0 ± 0.0	7.8 ± 0.44	42.75 ± 5.31
10	3	100.0 ± 0.0	10.6 ± 0.54	**100.0 ± 0.0**	**10.0 ± 0.0**	129.43 ± 2.34
20	2	**100.0 ± 0.0**	**14.2 ± 1.92**	100.0 ± 0.0	14.0 ± 2.0	245.59 ± 117.62
20	3	99.4 ± 1.01	21.6 ± 3.57	**100.0 ± 0.0**	**18.4 ± 1.14**	857.58 ± 238.99
30	2	**100.0 ± 0.0**	**22.6 ± 1.51**	0 ± 0.0	-	-
30	3	**99.93 ± 0.14**	**29.0 ± 1.87**	0 ± 0.0	-	-

unaffected by little errors in the training dataset and maintains the same top performances achieved on the perfect dataset. Figure 6.2 shows the results obtained on Tomita 5 with error rate between 0 and 50%. For each configuration we do 5 experiments with the hidden-state size set to 200 and we plot the mean and the standard deviation. The figure shows that our model can tolerate error rates smaller or equal to 15%.

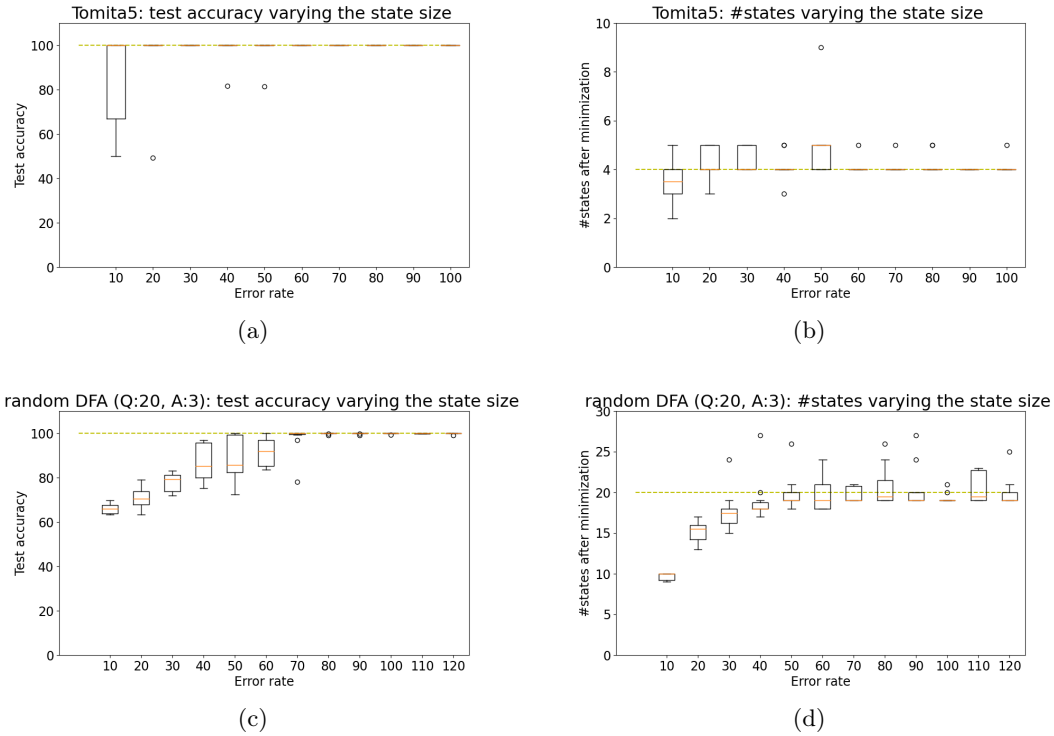


Figure 6.3: Results obtained varying the hidden state size hyperparameter with Tomita5 (a-b) and with with a random DFA of size 20 and alphabet size 3 (c-d). For each hidden state size we do 10 experiments.

Results on Randomly Generated DFAs

We generate random DFAs with different sizes and different alphabets using the method described in section 6.2.2. For each random DFA, we performed 10 tests for each stochastic method (DeepDFA and L* extraction) and one test with DFA-inductor. Table 6.4 reports comparing our approach with L* extraction. DeepDFA is more accurate and faster than L extraction. As in the Tomita experiments, the two methods especially differ in the number of states of the predicted DFA. L* extraction tends to greatly overestimate the number of states, while DeepDFA predicts DFAs with a number of states comparable to the ground truth size. Furthermore our model uses a significantly smaller number of weights $\#W$ and it is faster than L* extraction. Table 6.5 shows the results obtained with DFA-inductor. The method is able to reach top test accuracy when it does not crash for exceeding CPU capacity; this again happens for the biggest-size DFAs of size 30. Our method is competitive with DFA-inductor, especially if we consider the best run over each DFA instead of mean performances, shown in table 6.5. We repeat the experiments on a corrupted version of the training dataset, where % 1 of labels are erroneous. Results are shown in table 6.3. We found that DFA-inductor is unable to provide a solution even for the smallest size DFAs, while our method maintains nearly the same performances.

6.2.6 Ablation Study: the Effect of Changing the State Space Size

Recurrent neural networks have many hyperparameters and design choices that can affect performance: the model type, the number of layers, the number of features for each layer, and others. By contrast, our recurrent neural model is a simplified structure with only one hyperparameter: the hidden state size. In this section, we discuss this hyperparameter choice. Figure 6.3 shows the effect on the test accuracy and the predicted number of states of tuning this hyperparameter. We tested on Tomita5 language, with the hidden state size varying between 10 and 100; and we tested on a random DFA of size 20 and number of symbols equal to 3 with the hidden state size between 10 and 120. Results show that the model struggles to converge to the top test accuracy with a hidden state size equal to the ground truth number of states. This size has to be greater than the real number of states. Tomita5 starts to have nearly 100% test accuracy around 20 states, which is five times its real number of states. The random DFA reaches the top test accuracy at the state size of 70, which is 3.5 times its real number of states. However, the model maintains robust performances even if we highly overestimate the state size. In fact even with the largest state size the test accuracy remains high, and the number of states remains close to the ground truth. In other words, the model does not overfit; even if it is sized to represent many more states, it uses only a subset of them.

6.3 Related Work

Combinatorial Methods for Grammar Induction Many approaches were proposed to identify a target DFA from a set of positive and negative traces. The L* algorithm [3], is an exact active learning algorithm to learn a DFA from a minimally adequate teacher through membership and equivalence queries. Another approach is to apply the evidence-driven state-merging algorithm [109] [24], which is a greedy algorithm that is not guaranteed to converge to the global optimum. A more modern approach is to leverage highly-optimized SAT solvers by encoding the problem in SAT [92]. This approach is guaranteed to find the minimal DFA consistent with all the training examples, but suffers from scalability problems. In this regard, several symmetry-breaking predicates are proposed for the SAT encoding to reduce the search space [178] [177].

DFA Extraction from Recurrent Neural Networks Prior works focus on extracting a DFA from a pre-trained RNN, to explain the network behavior. Weiss et al. [169] adapt the L* algorithm From Angluin [3] to work with an RNN oracle. Other work uses k-means clustering on the RNN hidden states to extract a graph of states [168]. Merril et al. [122] extract a DFA from an RNN by first building a prefix tree and then merging states with close state representation in the RNN hidden space. These approaches train an RNN with the labeled strings and then extract an equivalent DFA from the trained model. Our approach differs from these since we directly train an RNN equivalent to a Probabilistic Finite Automaton (PFA), and we force the probabilities to become close to one-hot categorical distributions *during training*. In this way, our model *becomes a DFA*. In other words, there is no difference between the trained neural model and the automaton, and there is no risk that the abstraction

does not represent the network, as for previous works. For example, Wang et al. [168] cluster the RNN states, so the automaton depends on the clustering algorithm performances. Performances from [122] instead rely on a similarity threshold and, as the paper shows, also on the number of epochs the RNN is trained after convergence. Our work in this sense is more similar to [169], since it computes the abstraction *automatically*. The difference is in how the abstraction is computed: we use gradient descent optimization while [169] starts with a hand-crafted discretization that is automatically refined during automata learning with L*.

Learning Crispy Logical Models through Gradient Descent Classically, the induction of logic models, including DFAs, is not approached with gradient-descent optimization methods (es, deep learning methods) since their finite and 'crispy' nature prevents the gradient computation. However, recent works in neurosymbolic AI [68] propose techniques to reduce the gap between the induction of crispy models and that of continuous ones. Walke et al. [164] proposes a recurrent neural model with specialized filters to learn Linear Temporal Logic formulas from labeled traces. Kusters et al. [107] discovers logical rules describing patterns in sequential data using a differentiable model. Grachev et al. [80] proposes a neural network model similar to ours to learn DFA from traces. However, this model suffers the vanishing gradient problem for automatons larger than 6 states, while our method can effectively learn target automatons with up to 30 states.

6.4 Discussion

In conclusion, we propose DeepDFA: a hybrid between a DFA and a recurrent neural network, which can be trained from samples with backpropagation as usual deep learning models, but that is completely interpretable, as a DFA, after training. Our approach takes the best from the two worlds: grammar induction on one side and recurrent neural networks on the other side. It uses fewer weights and only requires setting one hyperparameter compared to recurrent neural nets. At the same time, it can tolerate errors in the training set and can be applied to large target DFAs, differently from exact methods.

Chapter 7

Visual Reward Machines

In the previous two chapters, we mainly concentrated on the following:

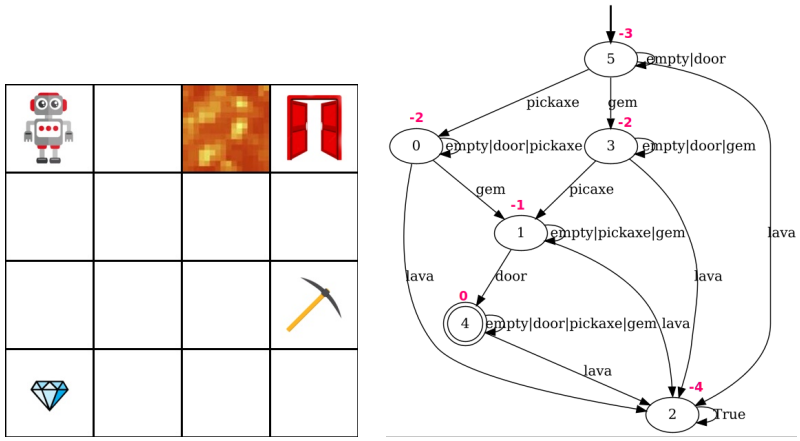
- symbol grounding exploiting prior LTLf specification, in chapter 5;
- temporal specification induction from labeled traces of symbols, in chapter 6.

As we underlined during the thesis, these two problems correspond to the two steps necessary to discover symbolic logical knowledge from non-symbolic domains. The frameworks described in Chapters 5 and 6 tackle these two problems separately. Here we describe some extensions that allow the integration of grounding and DFA induction in a single model. In particular, we will concentrate on the application to non-markovian RL environments with (non-symbolic) image states. For this kind of problems, we define Visual Reward Machines (VRM), an extension of classical Reward Machines to non-symbolic domains.

Visual Reward Machines consider the symbol grounding function a part of the system, instead of an external function, as most works in non-markovian RL do [45] [27] [67] [175] [135] [126]. Furthermore, the VRM formulation has a natural implementation in a neural network framework composed of convolutional and recurrent NNs. As a consequence, we can use it for both reasoning and learning. In particular, we can initialize the neural models implementing the VRM with a given task specification, and use it for computing markovian states and non markovian rewards for the task, as classical RMs. But we can also learn different system parts of the VRM *from data*, in case of missing or imperfect knowledge on some parts of the VRM specification.

We define VRM as a versatile framework for visual temporally-extended tasks, opening the door to many possible reasoning-learning combinations. In particular, we tested three different learning procedures: (i) Learning the symbol grounding exploiting prior knowledge of the symbolic machine, (ii)learning the machine structure using prior knowledge of a *imperfect* symbol grounding function, (iii) learning both the symbol grounding function and the machine structure from data. Results are very promising in the first two experiments, while further investigation are needed to overcome the third learning configuration.

The remainder of this chapter is organized as follows: we describe non-markovian RL tasks, and we illustrate the Visual Minecraft environment in Section 7.1; we give specifics on the desired structure and accepted input and outputs of VRMs in Section 7.2; In section 7.3, we describe how we extend DeepDFA to integrate it in the framework; In section 7.4 we



(a) (b)

Figure 7.1: a) Visual Minecraft environment. b) task specification as Reward Machine

give our definition of Visual Reward Machines; in Section 7.5, we describe the VRM implementation with neural networks and how to use the framework for learning and reasoning; we report experiments of different learning conditions in Section 7.6; we conclude with final considerations on results and directions for future work in Section 7.7. Results presented in this chapter produced the following publication [157]

7.1 Non Markovian RL Tasks

As described in Section 2.2.1, many tasks cannot be represented as MDP because the environment presents a non-markovian reward. In these cases, we can represent the reward function through a Reward Machine (RM) [27]. We formally define RMs in Section 2.2.1. An RM is basically a transducer converting sequences of states into sequences of rewards. The input alphabet of the machine is a set of symbols that must be *observable* in the environment states. In other words, we have to be able to recognize when these symbols are True or False during the interaction with the environment. This mapping is implemented by a *prior* labeling function. The RM monitors the symbols' truth values during time, and assigns rewards to the agent depending on the level of satisfaction of the temporal specification.

7.1.1 Minecraft Environment Example

In this section, we introduce the Visual Minecraft environment, which is an example of non-markovian task with non-symbolic states. A robotic agent can navigate a grid world environment through 4 movement actions $\{left, right, up, down\}$. Each cell of the grid can be empty or contain one of the following items: *pickaxe*, *gem*, *lava*, *door*

The task consists in collecting a pickaxe *and* a gem (it does not matter in which order the two items are collected) and then going to the door, while always avoiding the lava. The task specification corresponds to the DFA in figure 7.1. The specification is expressed in terms of 5 symbols

$$P = \{pickaxe, gem, lava, door, empty\} \tag{7.1}$$

One symbol for each item, that is True when the agent is in a cell containing that item, plus the symbol 'empty' that is True when no items are in the agent cell. We transform the DFA into an RM by defining a reward function on its states that is maximum on the final states. Many choices are possible here. In particular, we define the reward on state q_i as the opposite of the distance to the closest final state, when this distance is smaller than ∞, and equal to a negative constant C for states disconnected from the final states.

$$r(q_i) = max\{\min_{f \in F} dist(q_i, f), C\} \tag{7.2}$$

In this way way we obtain a quite dense reward function. Reward values are shown in red on the automaton in Figure 7.1(b). Figure 7.1(a) shows an example of image state from the environment.

7.2 Framework Specifics

Given a visual non-markovian environment, like the example described in 7.1.1, we want to define a neurosymbolic model that can capture the task in the image state space, where the environment is defined, while exposing the task structure in a finite symbolic space. This framework takes as input a sequence of images $I = i_0, i_1, ..., i_{l-1}$, where l is the sequence length, and returns as output a sequence of l probabilistic beliefs over a finite set of states and a finite set of rewards, we denote respectively as $h_1, h_2, ..., h_l$ and $r_1, r_2, ..., r_l$. We call the model Visual Reward Machine, since it works exactly as a Reward Machine, with the difference it can process non-symbolic sequences as input.

7.3 VRM as an Extension of DeepDFA

This section describes how the framework illustrated in Chapter 6, DeepDFA, can be extended to meet the VRM specifics.

7.3.1 From DeepDFA to DeepMooreMachine

First of all, DeepDFA can be easily modified to represent Moore Machines. We recall that Moore Machines, defined in Section 2.1.3, are DFAs augmented with an output function $\delta_o : Q \to O$, where O is a finite set alphabet of output symbols. A DFA is a Moore Machine with a binary output alphabet $O = \{Acc, Rej\}$ and output function.

$$\delta_o = \begin{cases} Acc & \text{if } q \in F \\ Rej & \text{if } q \notin F \end{cases} \tag{7.3}$$

Given a Moore Machine $(P, Q, O, q_0, \delta_t, \delta_o)$, we define its representation in matrix form as composed of:

- a transition matrix $M_t \in [0,1]^{|P|\times|Q|\times|Q|}$, where

$$M_t[p,q,q'] = \begin{cases} 1 & \text{if } \delta_t(q,p) = q' \\ 0 & \text{otherwise} \end{cases} \tag{7.4}$$

- an input vector $v_i \in [0,1]^{1\times|Q|}$, where

$$v_i[q] = \begin{cases} 1 & \text{if } q = q_0 \\ 0 & \text{otherwise} \end{cases} \tag{7.5}$$

- an output *matrix* $M_o \in [0,1]^{|Q|\times|O|}$, where

$$M_o[q,o] = \begin{cases} 1 & \text{if } \delta_o(q) = o \\ 0 & \text{otherwise} \end{cases} \tag{7.6}$$

Let us notice the representation is equal to that of PFA, with the only difference being that the output function is represented by a matrix M_O instead of a vector v_o. The rules for PFAs continue to hold; in particular, we can calculate states and output from input sequences as described in section 2.1.3, with the difference that the machine outputs the probability of having in output a particular symbol in the output alphabet, instead of probability of the sequence to be accepted.

Consequently, we define DeepMooreMachine similarly to DeepDFA

$$\begin{aligned} &h_0 = [1,0,...,0] \\ &h_t = h_{t-1} \times M_t[x[t]] \\ &y_t = h_t \times M_o \\ &M_t = softmax\_with\_temp(\theta_h, \tau) \\ &M_o = softmax\_with\_temp(\theta_y, \tau) \end{aligned} \tag{7.7}$$

Where M_t is the PFA transition matrix, $M_t[x[t]]$ is the transition matrix for symbol $x[t]$, and M_o is the output matrix, as defined above. An the network parameters are: $\theta_h \in \mathbb{R}^{|P|\times|\hat{Q}|\times|\hat{Q}|}$ and $\theta_y \in \mathbb{R}^{|Q|\times|O|}$. Let us notice θ_y as an increased size respect to DeepDFA.

7.3.2 Embedding Uncertainty over Symbols

DeepDFA (and DeepMooreMachine) allows us to represent uncertainty over both the transition and the output function, because it is based on representing the model as a probabilistic machine. However, let us notice the current symbol $x[i]$ is used to index the transition matrix in Equations 6.1 and 7.7. As a consequence, *symbols must be integers*. This contrasts with symbol grounding techniques based on neural networks, which usually predict a *probabilistic belief* on symbol truth values. For this reason, we extend the framework to be fully probabilistic, and consider probability values over symbols in the calculations. Given a sequence of inputs $x_1, x_2, ..., x_l$, where x_i is a probability vector over $|P|$ classes, we define the next state

and output of the network as follows

$$\begin{aligned} h_0 &= [1, 0, ..., 0] \\ h_t &= \sum_{p=0}^{p=|P|} x_{t-1,p}(h_{t-1} \times T_p) \\ y_t &= h_t \times M_o \\ M_t &= softmax\_with\_temp(\theta_h, \tau) \\ M_o &= softamx\_with\_temp(\theta_y, \tau) \end{aligned} \tag{7.8}$$

where we denote as $x_{t,p}$ the probability that x_t is symbol $p \in P$.

7.4 Visual Reward Machine Definition

We define a Visual Reward Machine as a tuple $VRM = (X, P, Q, R, q_0, \delta_{tp}, \delta_{rp}, sg)$, where

- X is the set of input data, possibly infinite and continuous, the machine can process;
- P is the finite machine alphabet;
- Q is the finite set of states;
- R is a finite set of rewards;
- q_0 is the initial state;
- $\delta_{tp} : Q \times P \times Q \to [0, 1]$ is the transition probability function;
- $\delta_{rp} : Q \times R \to [0, 1]$ is the reward probability function;
- $sg : X \times P \to [0, 1]$ is the *symbol grounding* probability function.

Similarly to an RM, a VRM produces a reward sequence by processing a data sequence. Unlike RM, however, these data do not need to be symbolic but can be of any type. The symbol grounding function maintains the link with the symbolic representation, assigning to a data instance $x \in X$ a probability value for each symbol in the alphabet P.

Given a sequence of states $s_0, s_1, ..., s_{l-1}$, where l is the sequence length, the VRM produces l state probabilities and reward probabilities, which depend on how much certain is the grounding of symbols and how much certain is the transition and reward function.

7.5 Visual Reward Machine Implementation with NN

Let us notice each VRM defined as above has a natural implementation with neural networks. The specification in the definition can be translated in matrix form using Equations 7.4, 7.5, 7.6 and then transformed in the recurrent neural network described by Equation 7.8. This network takes as input vectors x_i of probabilities. Each vector x_i corresponds to the probabilistic grounding sg of symbols into data s_i that can be implemented with a CNN, in case of image data, or another type of neural network, in case of different types of data. Figure 7.2 shows an example of implementation.

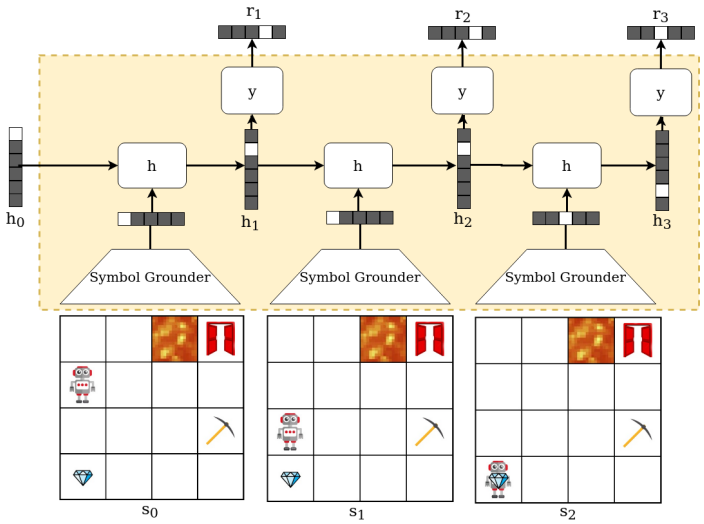


Figure 7.2: Implementation of a visual reward machine for the Visual Minecraft environment

7.5.1 Reasoning and Learning with VRM

In this section we investigate how we can exploit the framework for reasoning and learning in nonmarkovian RL tasks. In case, for a certain task, all the elements defining a VRM are perfectly known, the definition can be translated in a neural network and the latter can be used for reasoning in the same way as RM. In particular, it can be used to define a reward function and extract a Markov state from the specification. It is completely equivalent to an RM in the case of a deterministic task defined over a set of *perfectly grounded* symbols. However, VRMs allow us to represent also *probabilistic task specifications* and /or *probabilistic labeling functions*. This extends the range of specifiable problems significantly.

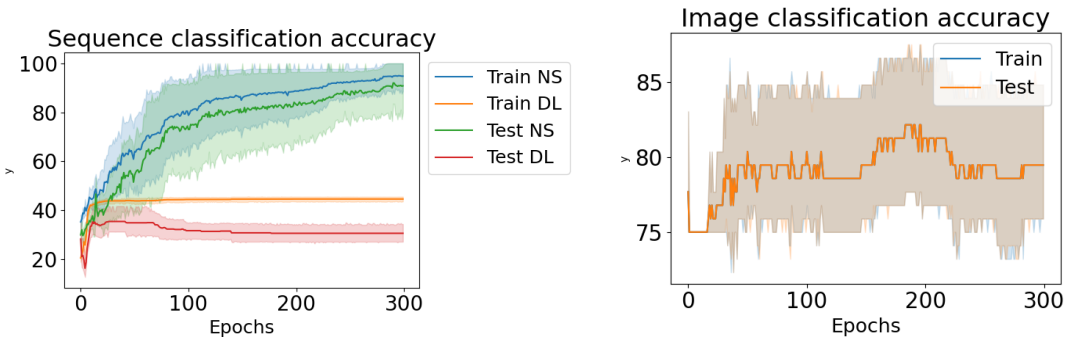
Furthermore, since the framework is all based on neural networks, in case of missing information in the VRM specifics, we can use the network implementation to *learn* the missing pieces from data, using backpropagation. In particular, we explored three different learning procedures:

- learning the *DFA structure* from sequences of imperfectly grounded symbols and output labels over these sequences
- learning the *symbol grounding function* by exploiting the Moore Machine structure and sequences of states-rewards
- learning both the grounding function and the temporal specification from sequences of visual states and rewards

In the next section, we report the preliminary results of these investigations.

7.6 Experiments

In this section we report some preliminary experiments validating the proposed framework. The implementation code can be found at `https://github.com/whitemech/VisualRewardMachine`.



(a) (b)

Figure 7.3: Results obtained by exploiting the reward machine structure to learn the symbol grounding function offline for the Visual Minecraft environment. a) Visual Reward Machine accuracy over sequences. b) Symbol grounding accuracy over single images

In particular, we explore many different learning procedures. In the first experiment we test *offline symbol grounding*, namely the capability to learn the *symbol grounding function* by exploiting the Moore Machine structure and sequences of states-rewards of an hand-crafted dataset. For this test, we initialize the Moore Machines parameters θ_{M_T} and θ_{M_O} with the given task specification and we train only the CNN weights by minimizing the crossentropy between the network predictions and the reward labels. In the second experiment we test a combination of *RL and online symbol grounding*. In this test, we ground the symbols of the task specifics *on the fly*, namely with sequences of image states and rewards collected by the agent while it is learning to perform the task. We use the learned symbol grounding to proceed on the automaton at each step and estimate the current automaton state. The latter is combined with the environment visual state to form a *markovian* state representation that is used by the RL algorithm to estimate the optimal policy.

In our experiments we mainly focus on learning the grounding function. However, many other learning-reasoning combinations are possible since the framework is very versatile. In particular, we test learning the DFA structure from traces of *imperfectly grounded symbols*, i.e., symbols for which we have only a probabilistic prediction instead of a strictly boolean interpretation. Although many techniques for DFA induction exist in the literature [3][92][109], and few extensions can handle noise in the output label [118] [155], to the best of our knowledge, there are no extensions to handle noise in the input symbols. In our experiments, we simulate the use of an imperfect pretrained classifier for symbol grounding by corrupting the input symbols with Gaussian noise, and we train the recurrent part of the VRM on this noisy dataset. We conduct our preliminary experiments in this direction on Tomita Languages, which are a popular benchmark for DFA induction. We let the integration of DFA induction with RL in non-markovian environments for future research.

7.6.1 Offline symbol grounding

Dataset For the application of offline symbol grounding to the Minecraft environment, we create a dataset simulating 40 episodes in the environment and collecting the images rendered by the environment. Each episode lasts 30 steps; therefore, it produces a sequence of 30

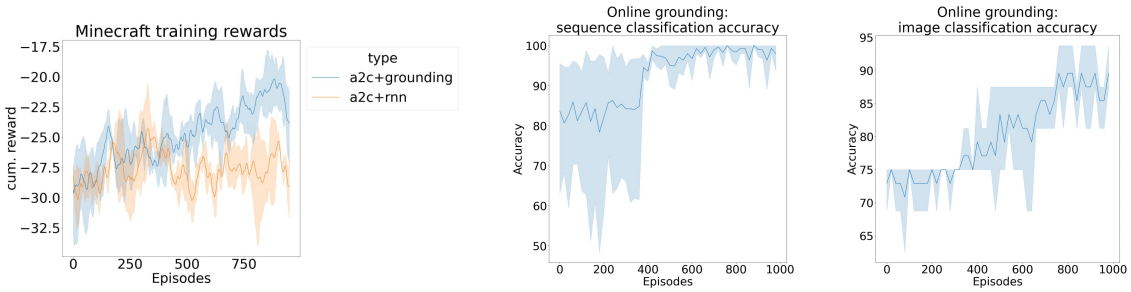
images corresponding to the environment states and a sequence of 30 reward values, that are used to label the image sequence *at each step*. Episodes are balanced between positive and negative examples. In 20 episodes, the agent correctly collects all the items and goes to the door avoiding the lava; in the other 20, it fails. We split the dataset in 80% for training and 20% for testing.

Results Figure 7.3 shows train and test sequence classification accuracy, figure (a), and symbol grounding accuracy on single images, figure (b). Let us notice the task specification has two ungroundable symbols: *gem* and *pickaxe*. Since the agent can collect these two items in any order, the framework does not receive enough supervision to distinguish between the two. Therefore image classification does not achieve 100% accuracy. However, sequence classification can still achieve top accuracy even if the symbol grounder confuses the gem for the pickaxe and vice-versa. We compare our approach with a pure deep-learning-based approach that learns to classify sequences end-to-end using a CNN and an LSTM. This approach performs very poorly in the task, obtaining only 40% of sequence accuracy. Investigating the reason for these poor performances, we found that it almost never predicts rewards of -1 and -2, corresponding to the scarcest reward labels in the dataset. In fact, even if we balanced the dataset between positive (reward = 0 in the last step) and negative (reward < 0 in the last step) episodes, the reward labels are not balanced within the episodes. Learning classification tasks from highly biased data with neural networks can be very hard, as it is shown in this experiment. However, our approach is unaffected by the label imbalance. Let us also notice that the environment highly biases the distribution of symbols and reward labels in sequences. For example, the 'empty' symbol is much more frequent than the others, and most possible symbolic traces are unfeasible in the environment and, therefore, never observed. This can complicate the image classification task in case this would be approached with supervised learning.

7.6.2 Reinforcement Learning and online grounding

In this second experiment, we perform symbol grounding *online*, and we exploit it to estimate the machine state. This information is used to speed up the learning of a policy. In particular, unlike the previous experiments, the dataset is not supposed to be balanced, because it is collected by the agent while exploring the environment. We construct a Markow state by concatenating features extracted by the image with a CNN with the estimated automaton state. We use Advantage Actor-Critic (A2C) [**?**] to learn a policy on this state representation. We compare this approach which exploits the ungrounded DFA specification with A2C using the state of an LSTM trained end-to-end as state representation.

Training settings We run three experiments with three random seeds for each approach and report the mean and standard deviation. We let each approach train for 1000 episodes before stopping the training. We use a learning rate of 0.0007 and an entropy coefficient of 0.0001 as hyperparameters for A2C. The symbol grounder is trained in the same fashion described in the previous section, with the newly acquired data, every 40 episodes. The baseline method uses a one-layer LSTM of hidden size 256. We tested with different hidden



(a) (b) (c)

Figure 7.4: Results obtained by exploiting the reward machine structure to learn the symbol grounding function for the Visual Minecraft environment. a) Training rewards Right) Symbol grounding accuracy over single images

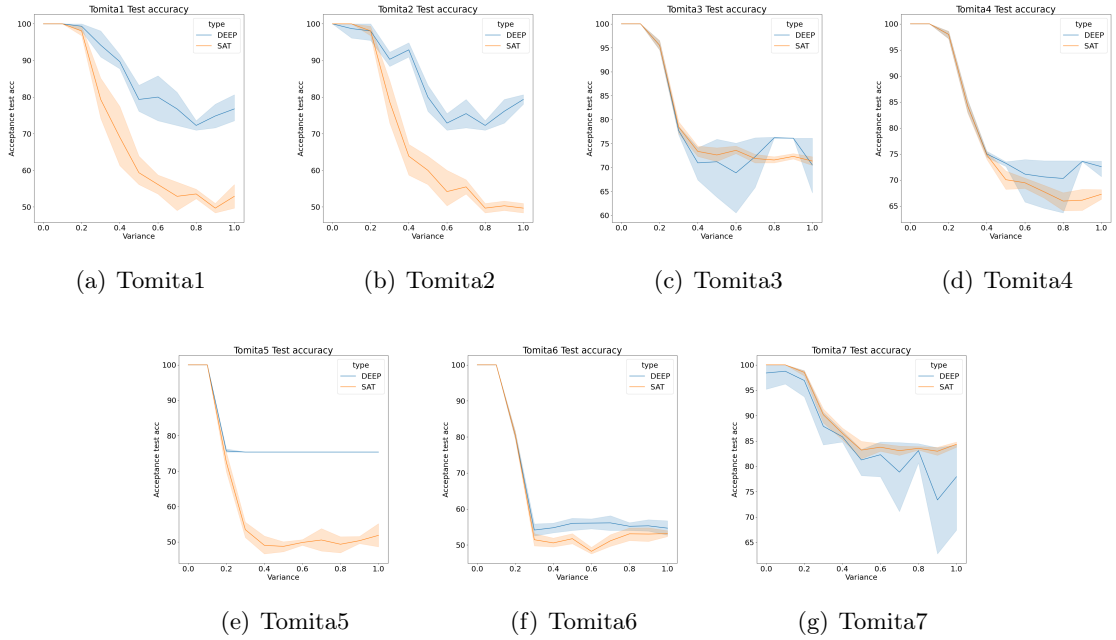
sizes for the LSTM (50 and 256) and two layers instead of one, finding that was the best configuration.

Results Figure 7.4 shows the results obtained in the experiment. Figure 7.4(a) shows training rewards for our method (A2C+grounding) and the baseline (A2C+RNN). The figure shows that our method outperforms the baseline. Figure 7.4(b) and (c) show the sequence classification and the symbol grounding classification accuracy of the Visual Reward Machine obtained during the training of the RL agent. The sequence classification accuracy tends to be quite high from the beginning of training, while the symbol grounding classification accuracy achieves top results only after 800 episodes. That is reasonable, since the VRM needs to observe some positive episodes to correctly ground all the symbols (e.g. the symbol *door* can be grounded only from a trajectory completing the task), and at the same time, positive episodes are observable only when the RL module has learned a good policy. Training rewards increase coherently with the increase in performance of the VRM. Despite these being only preliminary experiments, results are encouraging. In particular, they confirm our intuition that symbolic temporal specifications can also be exploited in visual tasks for which we do not know the symbol grounding function.

7.6.3 Comparisons with chapter 5

The task is similar to that of MNIST classification through Declare formulas described in Chapter 5, with some important differences. The first difference is that we have a sequence of labels for each sequence of images, represented by the rewards, instead of just one label on the last step. This gives more supervision to the framework. Apart from that, the Minecraft task is more challenging for the following reasons:

- the number of symbols to recognize is bigger, it is 5 in the Minecraft task versus 2 in the MNIST task;
- considered sequences are longer, we consider traces of length 20 in the Minecraft task and of length 4 in the MNISTtask;



(a) Tomita1 (b) Tomita2 (c) Tomita3 (d) Tomita4

(e) Tomita5 (f) Tomita6 (g) Tomita7

Figure 7.5: Learning DFA from traces composed of imperfectly grounded symbols: results on the 7 Tomita languages

- but most of all, the distribution of symbols in sequences is highly biased by the environment. For example, the 'empty' symbol is much more frequent than the others, and most possible symbolic traces are unfeasible in the environment and, therefore, never observed. For example, the agent cannot jump from one item to the other, but it has to walk and see many empty cells in between.

7.6.4 Learning the machine from imperfectly grounded symbols

In this experiment, we test the capability of our framework to learn DFA specifications from sequences of imperfectly grounded symbols. In particular, we trained only the recurrent module with traces of symbols and accepted-rejected labels generated by the Tomita languages [152]. We corrupt the one-hot representation of symbols in the train traces by adding Gaussian noise with zero mean and variable variance. Figure 7.5 compares our approach (Deep) and the SAT-based DFA-inductor (SAT) [177]. Since boolean logic induction methods like DFA-inductor cannot handle probabilistic truth values, we discretize the corrupted inputs to the closest one-hot vector before passing them to DFA-inductor. The figures show how the two methods test accuracy (on y-axis) is affected by different values of noise variance (x-axis). In particular, we observe that our method's test accuracy degrades less with increasing the noise variance and it is therefore more robust to noise in the symbol grounding with respect to DFA-inductor.

7.6.5 Learning All End-to-End

Since our framework is completely neural, in principle, we could use it to learn all the elements of the VRM from input-output examples in an end-to-end fashion. However, we tested on

both the Minecraft task and a visual Tomita induction task constructed with MNIST images, and the tests were unsuccessful.

In fact, If both the symbol grounding function and the Moore Machine are unknown and learnable, the system can oversimplify the grounding to oversimplify the machine. As a result, it is very prone to overfit training data. However, this is reasonable, given the very weak supervision imposed on the framework; and we need to define (similarly to how we do for groundability in Section 5.3) which conditions the task and the dataset must meet to ensure the correct (or incorrect) learning of the models.

7.7 Discussion

In conclusion, in this chapter we defined Visual Reward Machines: a neurosymbolic framework that produces non-markovian rewards for visual tasks. The framework can be used for both reasoning and learning. It is based on an extension of DeepDFA to MooreMachines and probabilistic input symbols. This second extension allows connecting DeepDFA with a neural module for symbol grounding and to define a fully neural but interpretable system. We tested learning from data different system parts. Results show that we can learn the machine from imperfectly grounded symbols and exploit the machine's prior knowledge to learn the grounding function. However, the system is defined to be very versatile, and many other reasoning-learning and transfer-learning settings can be explored, which are not considered in this thesis. Here we list different uses-cases that we let for future research.

- transfer learning between different tasks in the same environment: we remark that once we learned the grounding of a set of symbols in a certain environment, this is transferable to all possible task specifications defined on the same symbol set and executed in the same environment. Therefore, future experiments could exploit this property to transfer learning to different tasks.
- symbol grounding correction: another possibility consists in using prior knowledge on the machine and reward labels to fix an imperfect grounding function (instead of learning it from scratch as we have done in the experiments)
- machine transition and/or output correction: the same can be applied to the machine in case we know a perfect grounding but an imperfect Moore Machine.
- Machine induction with advice: in case we can access partial knowledge of the machine, as in [126], we could encode it in the recurrent network and then proceed to learn the rest of the machine from the data. In this case, however, we need to define how to train the machine to avoid catastrophic forgetting of the prior knowledge. This can be done, for example, by dividing the RNN weights into learnable and fixed weights, where the latter contain the prior knowledge while the former allow the adaptation to new data.

Part IV

Conclusions

Chapter 8

Conclusions

In this last chapter, we summarize the contributions described in this thesis and underline possible directions for future research.

8.1 Summary of Contributions

In summary, we have exploited different techniques to extract symbolic logical knowledge from nonsymbolic domains of various types.

Part II focused on extracting logical knowledge from markovian control domains. These problems are extremely complicated to convert in discrete logical domains, because the states are continuous feature vectors, and actions tend to be very low level. However, we were able to describe the environment dynamics and the action quality function in a finite planning model based on a symbolic latent representation of the state, and guide the agent interaction by using only planning with this model. We have observed that we can exploit the symbolic state representation for shaping rewards to reach different goals in the same environment, and that the symbolic representation gives some precious insights on the task. In particular, we observed that the symbol grounder trained to represent a symbolic model-invariant abstraction is far from having a homogeneous number of symbolic configurations in the ground state space. Symbols concentrate mostly on particular areas correlated with the action decision threshold. Resulting in a partition of the ground state space that would be very complicated to be encoded manually by human designers.

In Part III, I focused on different types of domains, namely non-markovian environments having visual observations. This type of problem presents other challenges with respect to those tackled in Part II. In particular, non-Markovianity causes observations to be processed in sequences. The tasks are more strategically complex, in the sense that only some specific, and potentially long, sequences of actions satisfy the specified task. These tasks are usually solved in symbolic environments, or continuous domains for which a mapping from states to symbols is already known, completely bypassing perception, which we consider a key skill in artificially intelligent systems instead. Therefore, the works described in Part III were mainly focused on removing this limiting assumption and bringing back perception and deep learning in the game.

We started by exploiting prior logical knowledge expressed in LTLf to classify sequences

of images in Chapter 5. We cast the problem as the learning of a symbol grounding function that classifies symbolic interpretations from images, that must maximize the satisfiability of the LTLf formula over a set of labeled training sequences. The LTLf formula satisfiability is calculated in fuzzy logic to allow integration with a neural classifier implementing the symbol grounder. Therefore, this work's main contribution was to encode the formula in fuzzy logic by first converting it into a Deterministic Finite Automaton and then translating the DFA into an equivalent recurrent Logic Tensor Network. We show that using the learned symbol grounder combined with the known LTLf specifications outperforms a classical end-to-end approach based on deep learning that cannot exploit the logic specification.

A second work on this line was DeepDFA, described in Chapter 6. The main contribution of this work is the definition of a recurrent neural network equivalent to a Probabilistic Finite Automaton that can be driven to approximate a DFA through temperature annealing. The latter is another technique used to discretize neural networks, particularly by increasing the steepness of classical activation functions. This architecture is very effective for automata induction from data. It benefits from tolerating a small percentage of errors in the training labels and being faster and more successful on big-size target automata than logic induction methods. However, it cannot handle probabilistic beliefs on symbols but only symbols represented as integers, and this complicates the connection with a symbol grounding function implemented with a neural network.

For this reason, in the last work, described in Chapter 7, we focus on extending DeepDFA to treat probabilistic symbols. Thanks to this extension, we can embed perception and temporal reasoning in a single model that we call Visual Reward Machine. The model is very similar to that described in Chapter 5, with the important difference that the temporal property is encoded in a *parametric* model that can be not only set from outside but also *learned* from data. We tested both learning the machine by exploiting a known but imperfect grounding, and learning the grounding function by using an available machine, and both tests were successful. In particular, we show that training our model on probabilistic symbols outperforms logic induction methods trained with the most probable boolean symbols.

This final framework is the one that best marries the philosophy behind neurosymbolic AI, and we now explain why. First, it embeds perception, which we stated is a fundamental feature for making reasoning *grounded* and applicable to realistic nonsymbolic problems. Second, but not less important, despite the logical symbolic model being implemented with a NN, it is finite and compact. We remind in particular that, even if big state sizes facilitate the training of DeepDFA, after training, we can minimize the DFA with classical DFA minimization techniques that tend to cut off most of the states and make the model even more compact than at training time. Third, the logical part can process probabilistic symbols, making it most robust to imperfect grounding. Finally, one of the best features of this framework is its *modularity*.

8.2 Future Research

Let us underline another crucial difference between the work described in Part II and those described in Part III: the environments tackled. Let us notice they are very different, not only

for the markovian property. The main difference is that environments tackled in Part II, such as Cartpole, Acrobot, and NAO, *are not constructed for symbolic logical knowledge discovery*. They are popular benchmark environments for RL without any connection with NeSy. While in Part III, we conduct experiments on environments constructed as a combination of a rendering function and a symbolic logical model; therefore, we know our *target symbol grounding function* and our *target logical model*. For this reason, we can consider Part II as an example of discovering logical knowledge 'in the wild', and Part III as testing the system's capacity to discover a latent logical structure that we encoded in the environment. Although the second scenario favors understanding results, I think it is crucial to move towards 'wild' scenarios. But it is essential to define metrics to really assess the capability of abstraction of NeSy systems in such scenarios. For our experiments, we considered satisfying the system was able to solve the environments planning in their abstractions, as it considered in other work [9]. Furthermore, we designed the network architecture to favor abstraction. However, this is a limited use of abstractions, and in the future, we would like to be able to successfully test on different but related environments. We think future research should focus on defining and designing environments that can be solved only by increasing the modularity of the representation, so as to move towards wild scenarios but always in a 'safe' and controlled way.

Bibliography

[1] D. Abel, D. E. Hershkowitz, and M. L. Littman. Near optimal behavior via approximate state abstraction, 2017.

[2] C. Allen, N. Parikh, O. Gottesman, and G. Konidaris. Learning markov state abstractions for deep reinforcement learning, 2021.

[3] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[4] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, 2017.

[5] M. Asai. Unsupervised grounding of plannable first-order logic representation from images, 2019.

[6] M. Asai. Unsupervised grounding of plannable first-order logic representation from images. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 29, pages 583–591, 2019.

[7] M. Asai and A. Fukunaga. Classical planning in deep latent space: Bridging the subsymbolic-symbolic boundary, 2017.

[8] M. Asai and A. Fukunaga. Classical planning in deep latent space: Bridging the subsymbolic-symbolic boundary. 2017.

[9] M. Asai and A. Fukunaga. Classical planning in deep latent space: Bridging the subsymbolic-symbolic boundary. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), Apr. 2018.

[10] M. Asai and H. Kajino. Towards stable symbol grounding with zero-suppressed state autoencoder, 2019.

[11] M. Asai and H. Kajino. Towards stable symbol grounding with zero-suppressed state autoencoder. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 29, pages 592–600, 2019.

[12] M. Asai and C. Muise. Learning neural-symbolic descriptive planning models via cube-space priors: The voyage home (to strips). In C. Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, pages

2676–2682. International Joint Conferences on Artificial Intelligence Organization, 7 2020. Main track.

[13] M. Asai and C. Muise. Learning neural-symbolic descriptive planning models via cube-space priors: The voyage home (to strips). 2020.

[14] E. Augustine, C. Pryor, C. Dickens, J. Pujara, W. Y. Wang, and L. Getoor. Visual sudoku puzzle classification: A suite of collective neuro-symbolic tasks. In A. S. d'Avila Garcez and E. Jiménez-Ruiz, editors, *Proceedings of the 16th International Workshop on Neural-Symbolic Learning and Reasoning as part of the 2nd International Joint Conference on Learning & Reasoning (IJCLR 2022), Cumberland Lodge, Windsor Great Park, UK, September 28-30, 2022*, volume 3212 of *CEUR Workshop Proceedings*, pages 15–29. CEUR-WS.org, 2022.

[15] F. Bacchus, C. Boutilier, and A. Grove. Rewarding behaviors. pages 1160–1167, Portland, OR, 1996.

[16] S. Badreddine, A. d'Avila Garcez, L. Serafini, and M. Spranger. Logic tensor networks. *Artificial Intelligence*, 303:103649, 2022.

[17] W. Bandres, B. Bonet, and H. Geffner. Planning with pixels in (almost) real time. In S. A. McIlraith and K. Q. Weinberger, editors, *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 6102–6109. AAAI Press, 2018.

[18] S. Bansal, Y. Li, L. M. Tabajara, and M. Y. Vardi. Hybrid compositional reasoning for reactive synthesis from finite-horizon specifications. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 9766–9774. AAAI Press, 2020.

[19] Y. Bengio, N. Léonard, and A. Courville. Estimating or propagating gradients through stochastic neurons for conditional computation, 2013.

[20] T. R. Besold, A. d'Avila Garcez, S. Bader, H. Bowman, P. Domingos, P. Hitzler, K. Kühnberger, L. C. Lamb, P. M. V. Lima, L. de Penning, G. Pinkas, H. Poon, and G. Zaverucha. Neural-symbolic learning and reasoning: A survey and interpretation. In P. Hitzler and M. K. Sarker, editors, *Neuro-Symbolic Artificial Intelligence: The State of the Art*, volume 342 of *Frontiers in Artificial Intelligence and Applications*, pages 1–51. IOS Press, 2021.

[21] T. R. Besold, A. S. d'Avila Garcez, S. Bader, H. Bowman, P. M. Domingos, P. Hitzler, K.-U. Kühnberger, L. Lamb, D. Lowd, P. M. V. Lima, L. de Penning, G. Pinkas, H. Poon, and G. Zaverucha. Neural-symbolic learning and reasoning: A survey and interpretation. In *Neuro-Symbolic Artificial Intelligence*, 2017.

[22] B. Bonet and H. Geffner. Learning first-order symbolic representations for planning from the structure of the state space. *arXiv*, pages arXiv–1909, 2019.

[23] M. Bruynooghe, T. Mantadelis, A. Kimmig, B. Gutmann, J. Vennekens, G. Janssens, and L. De Raedt. Problog technology for inference in a probabilistic first order logic. In *Proceedings of the 2010 Conference on ECAI 2010: 19th European Conference on Artificial Intelligence*, page 719–724, NLD, 2010. IOS Press.

[24] M. Bugalho and A. L. Oliveira. Inference of regular languages using state merging algorithms with search. *Pattern Recogn.*, 38(9):1457–1467, sep 2005.

[25] M. Cai, S. Xiao, B. Li, Z. Li, and Z. Kan. Reinforcement learning based temporal logic control with maximum probabilistic satisfaction. *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 806–812, 2020.

[26] A. Camacho and S. A. McIlraith. Towards neural-guided program synthesis of Linear Temporal Logic specifications. In *Workshop on Knowledge Representation and Reasoning Meets Machine Learning (KR2ML) at NeurIPS*, 2019.

[27] A. Camacho, R. Toro Icarte, T. Q. Klassen, R. Valenzano, and S. A. McIlraith. Ltl and beyond: Formal languages for reward function specification in reinforcement learning. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 6065–6073. International Joint Conferences on Artificial Intelligence Organization, 7 2019.

[28] M. Caron, H. Touvron, I. Misra, H. Jégou, J. Mairal, P. Bojanowski, and A. Joulin. Emerging properties in self-supervised vision transformers. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 9650–9660, October 2021.

[29] P. S. Castro. Scalable methods for computing state similarity in deterministic markov decision processes. In *Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence (AAAI-20)*, 2020.

[30] O. Chang, L. Flokas, H. Lipson, and M. Spranger. Assessing satnet's ability to solve the symbol grounding problem. In H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1428–1439. Curran Associates, Inc., 2020.

[31] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton. A simple framework for contrastive learning of visual representations. In H. D. III and A. Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 1597–1607. PMLR, 13–18 Jul 2020.

[32] X. Chen, Y. Duan, R. Houthooft, J. Schulman, I. Sutskever, and P. Abbeel. Infogan: Interpretable representation learning by information maximizing generative adversarial nets. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, page 2180–2188, Red Hook, NY, USA, 2016. Curran Associates Inc.

[33] F. Chollet. On the measure of intelligence, 2019.

[34] K. Chua, R. Calandra, R. McAllister, and S. Levine. Deep reinforcement learning in a handful of trials using probabilistic dynamics models, 2018.

[35] D. Corneil, W. Gerstner, and J. Brea. Efficient model-based deep reinforcement learning with variational state tabulation. *arXiv preprint arXiv:1802.04325*, 2018.

[36] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1, 2016.

[37] A. Creswell, T. White, V. Dumoulin, K. Arulkumaran, B. Sengupta, and A. A. Bharath. Generative adversarial networks: An overview. *IEEE Signal Processing Magazine*, 35(1):53–65, 2018.

[38] A. Cropper and S. Dumancic. Inductive logic programming at 30: A new introduction. *J. Artif. Intell. Res.*, 74:765–850, 2022.

[39] W.-Z. Dai, Q. Xu, Y. Yu, and Z.-H. Zhou. Bridging machine learning and logical reasoning by abductive learning. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[40] W.-Z. Dai, Q. Xu, Y. Yu, and Z.-H. Zhou. Bridging machine learning and logical reasoning by abductive learning. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[41] A. Daniele, T. Campari, S. Malhotra, and L. Serafini. Deep symbolic learning: Discovering symbols and rules from perceptions. *CoRR*, abs/2208.11561, 2022.

[42] A. Darwiche. Sdd: A new canonical representation of propositional knowledge bases. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Two*, IJCAI'11, page 819–826. AAAI Press, 2011.

[43] A. d'Avila Garcez and L. C. Lamb. Neurosymbolic ai: The 3rd wave, 2020.

[44] P. Dayan and G. E. Hinton. Feudal reinforcement learning. In S. Hanson, J. Cowan, and C. Giles, editors, *Advances in Neural Information Processing Systems*, volume 5. Morgan-Kaufmann, 1993.

[45] G. De Giacomo, L. Iocchi, M. Favorito, and F. Patrizi. Foundations for restraining bolts: Reinforcement learning with ltlf/ldlf restraining specifications. *Proceedings of the International Conference on Automated Planning and Scheduling*, 29(1):128–136, May 2021.

[46] G. De Giacomo and M. Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI '13, page 854–860. AAAI Press, 2013.

[47] C. de la Higuera. A bibliographical study of grammatical inference. *Pattern Recognition*, 38(9):1332–1348, 2005. Grammatical Inference.

[48] W. De Mulder, S. Bethard, and M.-F. Moens. A survey on the application of recurrent neural networks to statistical language modeling. *Computer Speech Language*, 30(1):61–98, 2015.

[49] J. De Smedt, S. vanden Broucke, J. Weerdt, and J. Vanthienen. A full r/i-net construct lexicon for declare constraints, 02 2015.

[50] T. L. Dean and R. Givan. Model minimization in markov decision processes. In *AAAI/IAAI*, 1997.

[51] M. P. Deisenroth and C. E. Rasmussen. Pilco: A model-based and data-efficient approach to policy search. ICML'11, page 465–472, Madison, WI, USA, 2011. Omnipress.

[52] P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, and P. Zhokhov. Openai baselines. `https://github.com/openai/baselines`, 2017.

[53] T. G. Dietterich. An overview of maxq hierarchical reinforcement learning. In B. Y. Choueiry and T. Walsh, editors, *Abstraction, Reformulation, and Approximation*, pages 26–44, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[54] M. Diligenti, M. Gori, and C. Saccà. Semantic-based regularization for learning and inference. *Artificial Intelligence*, 244:143–165, 2017. Combining Constraint Solving with Mining and Learning.

[55] A. Dittadi, F. K. Drachmann, and T. Bolander. Planning from pixels in atari with learned symbolic representations. 2020.

[56] A. Dittadi, F. K. Drachmann, and T. Bolander. Planning from pixels in atari with learned symbolic representations. In *AAAI*, 2021.

[57] I. Donadello, L. Serafini, and A. d'Avila Garcez. Logic tensor networks for semantic image interpretation. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 1596–1602, 2017.

[58] S. Du, A. Krishnamurthy, N. Jiang, A. Agarwal, M. Dudik, and J. Langford. Provably efficient RL with rich observations via latent state decoding. In K. Chaudhuri and R. Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 1665–1674. PMLR, 09–15 Jun 2019.

[59] J. L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990.

[60] N. Ferns, P. Panangaden, and D. Precup. Bisimulation metrics for continuous markov decision processes. *SIAM J. Comput.*, 40(6):1662–1714, dec 2011.

[61] N. Ferns and D. Precup. Bisimulation Metrics are Optimal Value Functions. In N. L. Zhang and J. Tian, editors, *The 30th Conference on Uncertainty in Artificial Intelligence*, page 10, Quebec City, Canada, July 2014. Ann Nicholson, AUAI Press.

[62] R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3):189 – 208, 1971.

[63] V. François-Lavet, Y. Bengio, D. Precup, and J. Pineau. Combined reinforcement learning via abstract representations. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 3582–3589, 2019.

[64] V. François-Lavet, Y. Bengio, D. Precup, and J. Pineau. Combined reinforcement learning via abstract representations, 2018.

[65] L. Gabora and A. Russon. The evolution of intelligence. In *The Cambridge Handbook of Intelligence*, pages 328–350. Cambridge University Press, may 2011.

[66] Y. Gal, R. McAllister, and C. E. Rasmussen. Improving PILCO with Bayesian neural network dynamics models. In *Data-Efficient Machine Learning workshop, International Conference on Machine Learning*, 2016.

[67] M. Gaon and R. Brafman. Reinforcement learning with non-markovian rewards. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(04):3980–3987, Apr. 2020.

[68] A. d. Garcez and L. C. Lamb. Neurosymbolic ai: The 3rd wave, 2020.

[69] A. S. d. Garcez and L. C. Lamb. Reasoning about time and knowledge in neural-symbolic learning systems. In *Proceedings of the 16th International Conference on Neural Information Processing Systems*, NIPS'03, page 921–928, Cambridge, MA, USA, 2003. MIT Press.

[70] M. Garnelo, K. Arulkumaran, and M. Shanahan. Towards deep symbolic reinforcement learning. 2016.

[71] M. Ghallab, C. Knoblock, D. Wilkins, A. Barrett, D. Christianson, M. Friedman, C. Kwok, K. Golden, S. Penberthy, D. Smith, Y. Sun, and D. Weld. Pddl - the planning domain definition language. 08 1998.

[72] M. Ghallab, D. Nau, and P. Traverso. *Automated planning and acting*. Cambridge University Press, 2016.

[73] G. D. Giacomo and M. Favorito. Compositional approach to translate ltlf/ldlf into deterministic finite automata. In S. Biundo, M. Do, R. Goldman, M. Katz, Q. Yang, and H. H. Zhuo, editors, *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling, ICAPS 2021, Guangzhou, China (virtual), August 2-13, 2021*, pages 122–130. AAAI Press, 2021.

[74] G. D. Giacomo, L. Iocchi, M. Favorito, and F. Patrizi. Foundations for restraining bolts: Reinforcement learning with ltlf/ldlf restraining specifications, 2019.

[75] G. D. Giacomo, R. D. Masellis, M. Grasso, F. M. Maggi, and M. Montali. Monitoring business metaconstraints based on ltl and ldl for finite traces. In *BPM*, 2014.

[76] E. Giunchiglia, M. C. Stoian, and T. Lukasiewicz. Deep learning with logical constraints. In L. D. Raedt, editor, *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, pages 5478–5485. International Joint Conferences on Artificial Intelligence Organization, 7 2022. Survey Track.

[77] F. Giunchiglia and T. Walsh. A theory of abstraction. *Artificial Intelligence*, 57(2-3):323–389, 1992.

[78] R. Givan, T. Dean, and M. Greig. Equivalence notions and model minimization in markov decision processes, 2003.

[79] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT press, 2016.

[80] P. Grachev, I. S. Lobanov, I. Smetannikov, and A. Filchenkov. Neural network for synthesizing deterministic finite automata. *Procedia Computer Science*, 119:73–82, 2017.

[81] J.-B. Grill, F. Strub, F. Altché, C. Tallec, P. H. Richemond, E. Buchatskaya, C. Doersch, B. A. Pires, Z. D. Guo, M. G. Azar, B. Piot, K. Kavukcuoglu, R. Munos, and M. Valko. Bootstrap your own latent: A new approach to self-supervised learning, 2020.

[82] J.-B. Grill, F. Strub, F. Altché, C. Tallec, P. H. Richemond, E. Buchatskaya, C. Doersch, B. Ávila Pires, Z. Guo, M. G. Azar, B. Piot, K. Kavukcuoglu, R. Munos, and M. Valko. Bootstrap your own latent - a new approach to self-supervised learning. In *NeurIPS*, 2020.

[83] E. Gumbel. *Statistical Theory of Extreme Values and Some Practical Applications: A Series of Lectures*. Applied mathematics series. U.S. Government Printing Office, 1954.

[84] D. Ha and J. Schmidhuber. Recurrent world models facilitate policy evolution. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[85] D. Ha and J. Schmidhuber. World models. 2018.

[86] D. Hafner, T. Lillicrap, J. Ba, and M. Norouzi. Dream to control: Learning behaviors by latent imagination, 2020.

[87] D. Hafner, T. Lillicrap, J. Ba, and M. Norouzi. Dream to control: Learning behaviors by latent imagination. In *International Conference on Learning Representations*, 2020.

[88] D. Hafner, T. P. Lillicrap, M. Norouzi, and J. Ba. Mastering atari with discrete world models. In *International Conference on Learning Representations*, 2021.

[89] P. Hájek. *Metamathematics of Fuzzy Logic*. Trends in Logic. Springer Netherlands, 2001.

[90] K. He, A. M. Wells, L. E. Kavraki, and M. Y. Vardi. Efficient symbolic reactive synthesis for finite-horizon tasks. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 8993–8999, 2019.

[91] N. Heess, J. J. Hunt, T. P. Lillicrap, and D. Silver. Memory-based control with recurrent neural networks. *CoRR*, abs/1512.04455, 2015.

[92] M. J. H. Heule and S. Verwer. Exact dfa identification using sat solvers. In *Proceedings of the 10th International Colloquium Conference on Grammatical Inference: Theoretical Results and Applications*, ICGI'10, page 66–79, Berlin, Heidelberg, 2010. Springer-Verlag.

[93] G. E. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network. *CoRR*, abs/1503.02531, 2015.

[94] J. E. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. 1971.

[95] Y.-X. Huang, W.-Z. Dai, L.-W. Cai, S. H. Muggleton, and Y. Jiang. Fast abductive learning by similarity-based consistency optimization. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 26574–26584. Curran Associates, Inc., 2021.

[96] B. Jang, M. Kim, G. Harerimana, and J. Kim. Q-learning algorithms: A comprehensive classification and applications. *IEEE Access*, PP:1–1, 09 2019.

[97] E. Jang, S. Gu, and B. Poole. Categorical reparameterization with gumbel-softmax. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[98] E. Jang, S. Gu, and B. Poole. Categorical reparameterization with gumbel-softmax. 2017.

[99] E. Jang, S. Gu, and B. Poole. Categorical reparameterization with gumbel-softmax. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[100] S. Kapturowski, G. Ostrovski, W. Dabney, J. Quan, and R. Munos. Recurrent experience replay in distributed reinforcement learning. In *International Conference on Learning Representations*, 2019.

[101] G. Konidaris. On the necessity of abstraction. *Current Opinion in Behavioral Sciences*, 29:1–7, 2019. Artificial Intelligence.

[102] G. Konidaris, L. Kaelbling, and T. Lozano-Perez. From skills to symbols: Learning symbolic representations for abstract high-level planning. *J. Artif. Intell. Res.*, 61:215–289, 2018.

[103] W. Kratsch, F. König, and M. Röglinger. Shedding light on blind spots – developing a reference architecture to leverage video data for process mining. *Decision Support Systems*, 158:113794, 2022.

[104] Y. Kuo, B. Katz, and A. Barbu. Encoding formulas as deep networks: Reinforcement learning for zero-shot execution of LTL formulas. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2020, Las Vegas, NV, USA, October 24, 2020 - January 24, 2021*, pages 5604–5610, 2020.

[105] T. Kurutach, A. Tamar, G. Yang, S. J. Russell, and P. Abbeel. Learning plannable representations with causal infogan. In *Advances in Neural Information Processing Systems*, pages 8733–8744, 2018.

[106] T. Kurutach, A. Tamar, G. Yang, S. J. Russell, and P. Abbeel. Learning plannable representations with causal infogan. In *Advances in Neural Information Processing Systems*, pages 8733–8744, 2018.

[107] R. Kusters, Y. Kim, M. Collery, C. de Sainte Marie, and S. Gupta. Differentiable rule induction with learned relational features. In A. d'Avila Garcez and E. Jiménez-Ruiz, editors, *Proceedings of the 16th International Workshop on Neural-Symbolic Learning and Reasoning as part of the 2nd International Joint Conference on Learning & Reasoning (IJCLR 2022), Cumberland Lodge, Windsor Great Park, UK, September 28-30, 2022*, volume 3212 of *CEUR Workshop Proceedings*, pages 30–44. CEUR-WS.org, 2022.

[108] L. Lamanna, A. E. Gerevini, A. Saetti, L. Serafini, and P. Traverso. On-line learning of planning domains from sensor data in pal: Scaling up to large state spaces. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(13):11862–11869, May 2021.

[109] K. J. Lang, B. A. Pearlmutter, and R. Price. Results of the abbadingo one dfa learning competition and a new evidence-driven state merging algorithm. In *ICGI 1998: Grammatical Inference*, number 1433 in Lecture Notes in Computer Science book series (LNCS), pages 1–12. Springer, 1998. Cite as: Lang K.J., Pearlmutter B.A., Price R.A. (1998) Results of the Abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. In: Honavar V., Slutzki G. (eds) Grammatical Inference. ICGI 1998. Lecture Notes in Computer Science, vol 1433. Springer, Berlin, Heidelberg.

[110] S. Lange and M. Riedmiller. Deep auto-encoder neural networks in reinforcement learning. In *The 2010 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2010.

[111] S. Lange, M. Riedmiller, and A. Voigtländer. Autonomous reinforcement learning on raw visual input data in a real world application. In *The 2012 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2012.

[112] M. Laskin, A. Srinivas, and P. Abbeel. CURL: Contrastive unsupervised representations for reinforcement learning. In H. D. III and A. Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 5639–5650. PMLR, 13–18 Jul 2020.

[113] Y. Lecun, Y. Bengio, and G. Hinton. Deep learning. *Nature Cell Biology*, 521(7553):436–444, May 2015.

[114] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[115] A. C. Li, Z. Chen, P. Vaezipoor, T. Q. Klassen, R. T. Icarte, and S. A. McIlraith. Noisy symbolic abstractions for deep RL: A case study with reward machines. *CoRR*, abs/2211.10902, 2022.

[116] L. Li, T. J. Walsh, and M. L. Littman. Towards a unified theory of state abstraction for mdps. In *In Proceedings of the Ninth International Symposium on Artificial Intelligence and Mathematics*, pages 531–539, 2006.

[117] M. L. Littman, U. Topcu, J. Fu, C. L. I. Jr., M. Wen, and J. MacGlashan. Environment-independent task specifications via GLTL. *CoRR*, abs/1704.04341, 2017.

[118] S. Lucas and T. Reynolds. Learning deterministic finite automata with a smart state labeling evolutionary algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(7):1063–1074, 2005.

[119] C. J. Maddison, D. Tarlow, and T. Minka. A* sampling, 2015.

[120] R. Manhaeve, S. Dumancic, A. Kimmig, T. Demeester, and L. De Raedt. Deepproblog: Neural probabilistic logic programming. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[121] G. Marra, F. Giannini, M. Diligenti, and M. Gori. Lyrics: A general interface layer to integrate logic inference and deep learning. In *ECML/PKDD*, 2019.

[122] W. Merrill and N. Tsilivis. Extracting finite automata from rnns using state merging, 2022.

[123] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In M. F. Balcan and K. Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1928–1937, New York, New York, USA, 20–22 Jun 2016. PMLR.

[124] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, Feb. 2015.

[125] T. M. Moerland, J. Broekens, and C. M. Jonker. Model-based reinforcement learning: A survey, 2020.

[126] D. Neider, J.-R. Gaglione, I. Gavran, U. Topcu, B. Wu, and Z. Xu. Advice-guided reinforcement learning in a non-markovian environment. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(10):9073–9080, May 2021.

[127] A. Y. Ng, D. Harada, and S. Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *In Proceedings of the Sixteenth International Conference on Machine Learning*, pages 278–287. Morgan Kaufmann, 1999.

[128] T. Nguyen, T. M. Luu, T. Vu, and C. D. Yoo. Sample-efficient reinforcement learning representation learning with curiosity contrastive forward dynamics model, 2021.

[129] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell. Curiosity-driven exploration by self-supervised prediction. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 2778–2787. JMLR.org, 2017.

[130] M. Pesic, H. Schonenberg, and W. M. van der Aalst. Declare: Full support for loosely-structured processes. In *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007)*, pages 287–287, 2007.

[131] M. Pesic and W. M. van der Aalst. A declarative approach for flexible business processes management. In *Business Process Management Workshops*, 2006.

[132] S. Pitis. Beyond binary: Ternary and one-hot neurons. 2017.

[133] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977.

[134] I. D. Rodriguez, B. Bonet, J. Romero, and H. Geffner. Learning first-order representations for planning from black box states: New results. In M. Bienvenu, G. Lakemeyer, and E. Erdem, editors, *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning, KR 2021, Online event, November 3-12, 2021*, pages 539–548, 2021.

[135] A. Ronca, G. P. Licks, and G. D. Giacomo. Markov abstractions for PAC reinforcement learning in non-markov decision processes. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*, pages 3408–3415, 2022.

[136] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65 6:386–408, 1958.

[137] S. J. Russell and P. Norvig. *Artificial Intelligence: a modern approach*. Pearson, 3 edition, 2009.

[138] L. Saitta and J.-D. Zucker. *Abstraction in Artificial Intelligence and Complex Systems*. Springer Publishing Company, Incorporated, 2015.

[139] P. Sen, B. W. S. R. de Carvalho, R. Riegel, and A. G. Gray. Neuro-symbolic inductive logic programming with logical neural networks. *CoRR*, abs/2112.03324, 2021.

[140] M. Shanahan and M. Mitchell. Abstraction for deep reinforcement learning. In L. D. Raedt, editor, *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, pages 5588–5596. International Joint Conferences on Artificial Intelligence Organization, 7 2022. Survey Track.

[141] H. T. Siegelmann and E. D. Sontag. On the computational power of neural nets. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, COLT '92, page 440–449, New York, NY, USA, 1992. Association for Computing Machinery.

[142] T. Silver, A. Athalye, J. B. Tenenbaum, T. Lozano-Perez, and L. P. Kaelbling. Learning neuro-symbolic skills for bilevel planning. In *Conference on Robot Learning (CoRL)*, 2022.

[143] T. Silver, R. Chitnis, N. Kumar, W. McClinton, T. Lozano-Perez, L. P. Kaelbling, and J. Tenenbaum. Predicate invention for bilevel planning. In *AAAI Conference on Artificial Intelligence (AAAI)*, 2023.

[144] S. Srivastava, C. Li, M. Lingelbach, R. Martín-Martín, F. Xia, K. Vainio, Z. Lian, C. Gokmen, S. Buch, C. K. Liu, S. Savarese, H. Gweon, J. Wu, and L. Fei-Fei. Behavior: Benchmark for everyday household activities in virtual, interactive, and ecological environments, 2021.

[145] L. Steels. The symbol grounding problem has been solved. so what's next? *Symbols, embodiment and meaning. Academic Press, New Haven*, 2008.

[146] L. L. Steels. The symbol grounding problem has been solved, so what's next? 2008.

[147] R. Stewart and S. Ermon. Label-free supervision of neural networks with physics and domain knowledge. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[148] M. Stolle and D. Precup. Learning options in reinforcement learning. In S. Koenig and R. C. Holte, editors, *Abstraction, Reformulation, and Approximation*, pages 212–223, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[149] A. Suárez-Hernández, J. Segovia-Aguas, C. Torras, and G. Alenyà. Strips action discovery. 2020.

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

[151] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[152] M. Tomita. Dynamic construction of finite automata from examples using hill-climbing. In *Proceedings of the Fourth Annual Conference of the Cognitive Science Society*, pages 105–108, Ann Arbor, Michigan, 1982.

[153] S. Topan, D. Rolnick, and X. Si. Techniques for symbol grounding with SATNet. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, editors, *Advances in Neural Information Processing Systems*, 2021.

[154] E. Tsamoura, T. Hospedales, and L. Michael. Neural-symbolic integration: A compositional perspective. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(6):5051–5060, May 2021.

[155] V. I. Ulyantsev, I. Zakirzyanov, and A. A. Shalyto. Bfs-based symmetry breaking predicates for dfa identification. In *Language and Automata Theory and Applications*, 2015.

[156] E. Umili, E. Antonioni, F. Riccio, R. Capobianco, D. Nardi, and G. De Giacomo. Learning a symbolic planning domain through the interaction with continuous environments. In *Workshop on Bridging the Gap Between AI Planning and Reinforcement Learning (PRL)*, 2021.

[157] E. Umili, F. Argenziano, A. Barbin, and R. Capobianco. Visual reward machines. In *Proceedings of the 17th International Workshop on Neural-Symbolic Learning and Reasoning, La Certosa di Pontignano, Siena, Italy, July 3-5, 2023*, pages 255–267, 2023.

[158] E. Umili and R. Capobianco. Deepdfa: a transparent neural network design for dfa induction. 04 2023.

[159] E. Umili, R. Capobianco, and G. D. Giacomo. Grounding ltlf specifications in images. In *Proceedings of the 16th International Workshop on Neural-Symbolic Learning and Reasoning as part of the 2nd International Joint Conference on Learning & Reasoning (IJCLR 2022), Cumberland Lodge, Windsor Great Park, UK, September 28-30, 2022*, pages 45–63, 2022.

[160] A. van den Oord, Y. Li, and O. Vinyals. Representation learning with contrastive predictive coding. *CoRR*, abs/1807.03748, 2018.

[161] E. van Krieken, E. Acar, and F. van Harmelen. Analyzing Differentiable Fuzzy Implications. In *Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning*, pages 893–903, 9 2020.

[162] C. K. Verginis, C. Köprülü, S. Chinchali, and U. Topcu. Joint learning of reward machines and policies in environments with partially known semantics. *CoRR*, abs/2204.11833, 2022.

[163] A. S. Vezhnevets, S. Osindero, T. Schaul, N. Heess, M. Jaderberg, D. Silver, and K. Kavukcuoglu. Feudal networks for hierarchical reinforcement learning, 2017.

[164] H. Walke, D. Ritter, C. Trimbach, and M. Littman. Learning finite linear temporal logic specifications with a specialized neural operator, 2021.

[165] H. Walke, D. Ritter, C. Trimbach, and M. Littman. Learning finite linear temporal logic specifications with a specialized neural operator, 2021.

[166] C. Wang, Y. Li, S. L. Smith, and J. Liu. Continuous motion planning with temporal logic specifications using deep neural networks, 2020.

[167] P. Wang, P. L. Donti, B. Wilder, and J. Z. Kolter. Satnet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver. In K. Chaudhuri and R. Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 6545–6554. PMLR, 2019.

[168] Q. Wang, K. Zhang, A. Ororbia, X. Xing, X. Liu, and C. Giles. An empirical evaluation of rule extraction from recurrent neural networks. *Neural Computation*, 30(9):2568–2591, Sept. 2018. Publisher Copyright: © 2018 Massachusetts Institute of Technology.

[169] G. Weiss, Y. Goldberg, and E. Yahav. Extracting automata from recurrent neural networks using queries and counterexamples. In J. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 5247–5256. PMLR, 10–15 Jul 2018.

[170] M. Westergaard. Better algorithms for analyzing and enacting declarative workflow languages using ltl. In S. Rinderle-Ma, F. Toumani, and K. Wolf, editors, *Business Process Management*, pages 83–98, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[171] T. Winters, G. Marra, R. Manhaeve, and L. D. Raedt. Deepstochlog: Neural stochastic logic programming. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36(9):10090–10100, Jun. 2022.

[172] Y. Xie, Z. Xu, M. S. Kankanhalli, K. S. Meel, and H. Soh. Embedding symbolic knowledge into deep networks. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[173] Y. Xie, F. Zhou, and H. Soh. Embedding symbolic temporal knowledge into deep sequential models, 2021.

[174] J. Xu, Z. Zhang, T. Friedman, Y. Liang, and G. Van den Broeck. A semantic loss function for deep learning with symbolic knowledge. In J. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 5502–5511. PMLR, 10–15 Jul 2018.

[175] Z. Xu, B. Wu, A. Ojha, D. Neider, and U. Topcu. Active finite reward automaton inference and reinforcement learning using queries and counterexamples. In *Machine Learning and Knowledge Extraction - 5th IFIP TC 5, TC 12, WG 8.4, WG 8.9, WG 12.9 International Cross-Domain Conference, CD-MAKE 2021, Virtual Event, August 17-20, 2021, Proceedings*, pages 115–135, 2021.

[176] Z. Yang, A. Ishay, and J. Lee. Neurasp: Embracing neural networks into answer set programming. In C. Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, pages 1755–1762. International Joint Conferences on Artificial Intelligence Organization, 7 2020. Main track.

[177] I. Zakirzyanov, A. Morgado, A. Ignatiev, V. I. Ulyantsev, and J. Marques-Silva. Efficient symmetry breaking for sat-based minimum dfa inference. In *LATA*, 2019.

[178] I. Zakirzyanov, A. A. Shalyto, and V. I. Ulyantsev. Finding all minimum-size dfa consistent with given examples: Sat-based approach. In *SEFM Workshops*, 2017.

[179] A. Zhang, R. McAllister, R. Calandra, Y. Gal, and S. Levine. Learning invariant representations for reinforcement learning without reconstruction, 2021.

[180] S. Zhu, L. M. Tabajara, J. Li, G. Pu, and M. Y. Vardi. Symbolic ltlf synthesis. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 1362–1369, 2017.

[181] Z. Zhu, K. Lin, and J. Zhou. Transfer learning in deep reinforcement learning: A survey, 2020.

Appendix A

Appendix

Template	LTL Formula [29]	Regular Expression [38]	R/I-Net Constructs	Description
Existence(A,n)	$\Diamond(A \wedge \bigcirc(existence(n-1,A)))$	.*(A.*){n}	$F = \{(A,p_1),(p_1,t_{sink})\}, W(p_1,t_{sink}) = n$	Activity A happens at least n times.
Absence(A,n)	$\neg existence(n,A)$	[^A]*(A?[^A]*){n-1}	$F = \{(t_{source},p_1),(p_1,t_A)\}, W(t_{source},p_1) = n-1$	Activity A happens at most n times.
Exactly(A,n)	$existence(n,A) \wedge absence(n+1,A)$	[^A]*(A[^A]*){n}	$F = \{(t_{source},p_1),(p_1,t_A)\}, W(t_{source},p_1) = n, I(t_{sink}) = p_1$	Activity A happens exactly n times.
Init(A)	A	(A.*)?	$F = \{(t_{source},p_1)\}, \forall t \in T \setminus t_A, I(t) = p_1$	Each instance has to start with activity A.
Last(A)	$\Box(A \implies \neg X \neg A)$	.*A	$F = \{(t_A,p_1),(p_1,t_{sink})\}, \forall t \in T \setminus t_A, R(t) = p_1$	Each instance has to end with activity A.
Responded existence(A,B)	$\Diamond A \implies \Diamond B$	[^A]*((A.*B.*) |(B.*A.*))?	$F = \{(t_A,p_1),(t_B,p_2),(p_2,t_\wedge),(t_\wedge,p_3)\}, I(t_A) = p_3, I(t_B) = p_3, I(t_{sink}) = p_1, R(t_\wedge) = \{p_1,p_2\}$	If A happens at least once then B has to happen or happened before A.
Co-existence(A,B)	$\Diamond A \Longleftrightarrow \Diamond B$	[^AB]*((A.*B.*) |(B.*A.*))?	$F = \{(t_A,p_1),(t_A,p_4),(t_B,p_2),(t_B,p_4),(p_1,t_\wedge),(p_2,t_\wedge),(t_\wedge,p_3)\}, I(t_A) = p_3, I(t_B) = p_3, I(t_{sink}) = p_4, R(t_\wedge) = \{p_1,p_2,p_4\}$	If A happens then B has to happen or happened after A, and vice versa.
Response(A,B)	$\Box(A \implies \Diamond B)$	[^A]*(A.*B)*[^A]*	$F = \{(t_A,p_1)\}, I(t_{sink}) = p_1, R(t_B) = p_1$	Whenever activity A happens, activity B has to happen eventually afterward.
Precedence(A,B)	$(\neg B\ U\ A) \vee \Box(\neg B)$	[^B]*(A.*B)*[^B]*	$F = \{(t_A,p_1),(t_B,p_1),(p_1,t_B)\}$	Whenever activity B happens, activity A has to have happened before it.
Succession(A,B)	$response(A,B) \wedge precedence(A,B)$	[^AB]*(A.*B)*[^AB]*	$F = \{(t_A,p_1),(t_B,p_1),(p_1,t_B),(t_A,p_2)\}, I(t_{sink}) = p_2, R(t_B) = p_2$	Both Response(A,B) and Precedence(A,B) hold.
Alternate response(A,B)	$\Box(A \implies \bigcirc(\neg A\ U\ B))$	[^A]*(A[^A]*B[^A]*)*	$F = \{(t_{source},p_1),(p_1,t_A),(t_A,p_2),(t_B,p_1)\}, I(t_{sink}) = p_2, R(t_A) = p_1, R(t_B) = p_2$	After each activity A, at least one activity B is executed. A following activity A can be executed again only after the first occurrence of activity B.
Alternate precedence(A,B)	$precedence(A,B) \wedge \Box(B \implies \bigcirc(precedence(A,B)))$	[^B]*(A[^B]*B[^B]*)*	$F = \{(t_A,p_1),(p_1,t_B)\}, R(t_B) = p_1$	Before each activity B, at least one activity A is executed. A following activity B can be executed again only after the first next occurrence of activity A.
Alternate succession(A,B)	$altresponse(A,B) \wedge precedence(A,B)$	[^AB]*(A[^AB]*B[^AB]*)*	$F = \{(t_{source},p_1),(p_1,t_A),(t_A,p_2),(p_2,t_B),(t_B,p_1)\}, I(t_{sink}) = p_2$	Both alternative response(A,B) and alternate precedence(A,B) hold.
Chain response(A,B)	$\Box(A \implies \bigcirc B)$	[^A]*(AB[^A]*)*	$F = \{(t_A,p_1)\}, I(t_A) = p_1, I(t_C) = p_1, R(t_B) = p_1$	Every time activity A happens, it must be directly followed by activity B (activity B can also follow other activities).
Chain precedence(A,B)	$\Box(\bigcirc B \implies A)$	[^B]*(AB[^B]*)*	$F = \{(t_{source},p_1),(t_B,p_1),(t_C,p_1)\}, I(t_B) = p_1, R(t_A) = p_1$	Every time activity B happens, it must be directly preceded by activity A (activity A can also precede other activities).
Chain succession(A,B)	$\Box(A \Longleftrightarrow \bigcirc B)$	[^AB]*(AB[^AB]*)*	$F = \{(t_{source},p_2),(t_A,p_1),(t_B,p_2)\}, I(t_A) = p_1, I(t_B) = p_2, I(t_C) = p_1, R(t_A) = p_2, R(t_B) = p_1$	Activities A and B can only happen directly following each other.
Not co-existence(A,B)	$\neg(\Diamond A \wedge \Diamond B)$	[^AB]*((A[^B]*) |(B[^A]*))?	$F = \{(t_A,p_1),(t_B,p_2)\}, I(t_A) = p_2, I(t_B) = p_1$	Either activity A or B can happen, but not both.
Not succession(A,B)	$\Box(A \implies \neg(\Diamond B))$	[^A]*(A[^B]*)*	$F = \{(t_A,p_1)\}, I(t_B) = p_1$	Activity A cannot be followed by activity B, and activity B cannot be preceded by activity A.
Not chain succession(A,B)	$\Box(A \implies \neg(\bigcirc B))$	[^A]*(A+[^AB][^A]*)*A*	$F = \{(t_A,p_1)\}, I(t_B) = p_1, R(t_C) = p_1$	Activities A and B can never directly follow each other.
Choice(A,B)	$\Diamond A \vee \Diamond B$	.*[AB].*	$F = \{(t_{source},p_1)\}, I(t_{sink}) = p_1, R(t_A) = p_1, R(t_B) = p_1$	Activity A or activity B has to happen at least once, possibly both.
Exclusive choice(A,B)	$(\Diamond A \vee \Diamond B) \wedge \neg(\Diamond A \wedge \Diamond B)$	([^B]*A[^B]*) |(.*[AB].*([^A]*B[^A]*))	$F = \{(t_{source},p_2),(t_A,p_1),(t_B,p_3)\}, I(t_A) = p_3, I(t_B) = p_1, I(t_{sink}) = p_2, R(t_A) = p_2, R(t_B) = p_2$	Activity A or activity B has to happen at least once, but not both.

Figure A.1: List of Declare formulas as in [49]. We tested on all except last(a). Meaning of modal operators symbols:◯=X, ◊=F, □=G

Acknowledgements

It may sound cliché, but these last three years have taught me so much about who I am and who I want to become in the future, and for that, I owe a special thanks to everyone who has been a part of it. First of all, I would like to thank my supervisor, Professor De Giacomo, and my co-supervisor, Professor Roberto Capobianco, for believing in me until the very end, and for guiding me on this crazy journey that was my Ph.D. I thank the people of the Whitemech group, for being my travel companions, for the research chats, and also for the less serious conversations over a cup of coffee or a plate of Cantonese rice. I thank my boyfriend, who has always been close to me, even and especially during my difficult times. I thank my family and my lifelong friends, without whom I wouldn't be the person that I am today. Last but not least, I thank my dad, who has passed away, and to whom I dedicate this thesis. Thank you for always having encouraged me to follow my dreams despite everything. I hope that looking at me from the sky you can be proud of the woman I have become.