City Research Online

City, University of London Institutional Repository

Citation: Borges, Rafael (2012). A neural-symbolic system for temporal reasoning with application to model verification and learning. (Unpublished Doctoral thesis, City University London)

This is the unspecified version of the paper.

This version of the publication may differ from the final published version.

Permanent repository link: https://openaccess.city.ac.uk/id/eprint/1303/

Link to published version:

Copyright: City Research Online aims to make research outputs of City, University of London available to a wider audience. Copyright and Moral Rights remain with the author(s) and/or copyright holders. URLs from City Research Online may be freely distributed and linked to.

Reuse: Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge. Provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.

City Research Online: http://openaccess.city.ac.uk/ publications@city.ac.uk

A Neural-Symbolic System for Temporal Reasoning with Application to Model Verification and Learning

Rafael Vergara Borges

Dr. Artur d'Avila Garcez (supervisor)

Dr. Luis C. Lamb (co-supervisor)

A thesis submitted for the degree of

Doctor of Philosophy (PhD) in Computer Science

of the City University London

CITY UNIVERSITY LONDON

SCHOOL OF INFORMATICS

DEPARTMENT OF COMPUTING

January, 2012

Contents

Abstract **21**

1 Introduction **23**

1.1 Objectives 26

1.2 Contributions 27

1.3 Related Work 29

1.4 Published Results 32

1.5 Organization 33

2 Background **35**

2.1 Challenges in Artificial Intelligence 35

2.2 Symbolic AI 39

2.3 Nonclassical logics 42

2.3.1 Temporal Logics 43

2.4 Logic Programming 48

2.5 Neural Networks 52

2.5.1 Recurrent Networks and Temporal Processing 56

2.5.2 Learning in neural networks 58

2.6 Neural-Symbolic Systems 61

3 A Neural-Symbolic Model for Temporal Reasoning and Learning 67

3.1 On Logic and Neural Networks 67

3.1.1 Knowledge Representation in CILP 74

3.1.2 Connectionist Modal Logics: CML and CTLK 79

3.2 The sequential logic 81

3.2.1 Semantics 82

3.3 SCTL - Sequential Connectionist Temporal Logic 86

3.3.1 Translating the immediate operators 86

3.3.2 Differences between recurrent connections 90

3.3.3 Full SCTL translation 92

3.3.4 Comparing the different modal approaches 94

4 Learning in SCTL 97

4.1 Temporal extensions of backpropagation 98

4.1.1 The SCTL learning algorithm 99

4.2 Integrating different information sources 103

4.2.1 Constraining the learning process 105

4.2.2 Integrating and treating conflicts 109

4.3 Extracting temporal knowledge 110

4.3.1 A simple pedagogical approach - State diagrams 115

4.3.2 Extracting logic programs 117

4.4 Discussion 118

5 Experimental Validation **121**

5.1 The temporal XOR 123

5.2 The Muddy Children Puzzle 127

5.3 The Dining Philosophers 133

5.3.1 Offline Learning 136

5.3.2 Online Learning 137

5.3.3 Constrained learning 139

5.3.4 Extracting learned knowledge 141

5.4 Discussion 143

6 The Verification and Adaptation Framework **145**

6.1 Formal methods and model checking 145

6.1.1 Integrating machine learning and verification 147

6.1.2 Our integrated verification/adaptation framework 149

6.2 Representing temporal models 150

6.2.1 Extending SCTL 152

6.2.2 Translating between representations 154

6.3 Extracting NuSMV specifications 157

7 Evaluation of the Framework **161**

7.1 Black Box Checking 161

7.1.1 Handling noise 162

7.1.2 Analysis on non-deterministic scenarios 164

7.2 Verification and learning of properties 166

8 Conclusion and Future Work **173**

List of Tables

2.1 Different meaning for modal operators [56] 43

3.1 Example of execution 75

4.1 Example of a temporal model 104

4.2 Input and output observations from an agent 105

4.3 Definition of target outputs according to constraints 108

4.4 Example of the different treatments for missing information 110

4.5 Transitions extracted from the example 117

5.1 Temporal XOR sequence 124

5.2 Logic program describing the reasoning of each agent 129

5.3 Confusion matrix of the Muddy Children example 132

5.4 An agent's temporal knowledge representation 135

5.5 Offline Learning Results 136

5.6 Constraints used in the learning experiments 139

5.7 Set of clauses extracted to infer *pickL* variable 142

6.1 The simplified NuSMV language 151

6.2 NuSMV description of the Pump System example 152

7.1 Extracted transitions in the case of scalar state 165

7.2 Illustration of counter-examples and the sequences to adaptation . . 167

7.3 Counter-example obtained 167

7.4 NuSMV description adapted according to the counter-example . . . 169

7.5 NuSMV description obtained in the end of the process 171

List of Figures

2.1 Relations between intervals [1] 46

2.2 Illustration of an acyclic logic program 50

2.3 Image illustrating the computation steps in a perceptron 54

2.4 Example of a feed forward network with one hidden neuron 55

2.5 Example of an Elman Network 57

2.6 Example of a NARX network 58

2.7 Illustration of the Backpropagation algorithm 60

2.8 Taxonomy of Neural-Symbolic Systems 63

2.9 Example of a KBANN network 65

3.1 Information flow in a neural-symbolic system 68

3.2 CILP translation algorithm 71

3.3 Analysis of $\mathcal{T}_{\mathcal{P}}$ of an accepable program 76

3.4 Illustration of CILP extension for propagation of input values to output 78

3.5 CILP algorithm extension for propagation of input values to output . 79

3.6 An example of CML program and equivalent neural networks 80

3.7 An example of CTLK program and equivalent neural networks . . . 81

3.8 Translation of ●-based programs 89

3.9 Example of the different kinds of recurrent links in the SCTL model 92

3.10 Logic treatment of different temporal operators 93

4.1 Illustration of an unfolded NARX network 100

4.2 Propagation of values in a SCTL network during a timepoint 102

4.3 Algorithm for learning from multiple sources of information 111

4.4 Incompleteness and unsoundness of decompositional methods . . . 113

4.5 Example of extraction procedure 117

5.1 SCTL Networks used for the temporal XOR case 125

5.2 Error of the networks for the experiments without noise 126

5.3 Error of the networks for the experiments with noise 126

5.4 Architectures used on the Muddy Children example 130

5.5 Evaluation of RMSE in learners on Muddy Children Puzzle 131

5.6 Network architectures used to perform the Dining Philosophers . . . 135

5.7 Offline Learning Error in Time 137

5.8 Online Error in Time 138

5.9 Resource allocation in time 138

5.10 Performance of the different approaches/possibilities 141

5.11 Extracted transitions from the network 142

6.1 Diagram of the framework for verification and adaptation 150

6.2 Algorithm that reads the variables declarations from NuSMV 159

6.3 Translation from NuSMV into SCTL clauses 160

7.1 Confidence of the learning according to the amount of noise 163

7.2 Transition diagrams representing effects of adapting to properties . . 168

7.3 Transition diagrams representing effects of adapting to properties . . 170

Acknowledgements

Whenever talking about this thesis, the words I use most are probably integration and learning. Those are words that define not only the subject of my work, but also the period of time deployed to achieve this goal. This PhD was marked by the integration between the dream of traveling the world, living in a world capital as London, and the strong connection with my home country and the people left there.

Living in London was an wonderful experience not only about the city, but especially for the amazing people I have met, people from all around the world from whom I have learned so much, and who also gave me the chance so share a little about my culture, my way of life. Thanks Ricardo, for being such a close friend, sharing so many moments of both happiness and frustration (which was inevitably overcome by the pints of Guinness). Thanks Tshiamo for trusting me to share so many things at both work and home. Thanks Rafael, Davide, Marco, Vanessa, Mark, Aravin, Olga and all the guys from City University, who made the working place much more pleasant. Thanks Giulia, Horacio, David, Tomoko, who were amazing people to share not only a home, but also part of the life. Thanks Michael, Carina, Ibz, Chris, Kieran, Hollie, Dani, Andy, and so many other friends, for being such a nice company, sharing relaxing moments of fun. You all have made my experience in London unforgettable.

I would also like to thank Evandro, Micheli, Leticia, Ana, Guilherme, Nanda,

Emanuela, Cristiano, Carlos, Americo, Edivania and all the other Brazilian people I have met on London, and who made me feel like having a small portion of home when traveling overseas. For bringing the same feeling, I also would like to thank the friends who visited me, or those who traveled with me during these years, like Moser, Débora, Carla, Lorenzo, Lucíola, Germán, Márcio, Adriel, Thaísa, Thomás, Fabiana, Fabrício, Milene e Clarissa - all of them being amazing company to brake the daily routine, which sometimes was almost unbearable because of the loneliness.

On the other hand, I thank a lot all the people who were in Brazil giving me full support even when I was so far. Let me thank my parents, that helped me so much even though being away from me was so painful for them. Also, thank my sister Raquel for the encouraging words and moments of laugh. Thanks also to all the other people from my family, especially the cousin Lucas and the aunt Maristela, and all the friends who kept contact and endorsed me in different manners: Mário, Luciane, Fátima, Marília. For the recent support and friendship, I would like to thank the people from my recent job, especially Tadeu, Lutz, Leite, Dalmir, Marcelo, Fernanda and Daniel.

And for all the lessons learned, and the support given, and all the trust invested on me, I am strongly thankful to my supervisors Luís Lamb and Artur Garcez. Also for the lessons, of joy from simple things in life, of inspiration and hope in a better future, I thank all the kids whom I hold so dear in my life: Bolívar, Mathias, Daniel and Pietro. In the end, all my love and gratitude to her, who has been the main source of dream and inspiration in the last years of my life: Flávia, this work would not be possible at all, without such an amazing woman like you by my side.

.

Declaration on consultation and copying

The following statement is included in accordance with the Regulations governing the 'Physical format, binding and retention of theses' of the City University London

I grant powers of discretion to the University Librarian to allow this thesis to be copied in whole or in part without further reference to me. This permission covers only single copies made for study purposes, subject to normal conditions of acknowledgement.

Rafael V. Borges

List of Abbreviations

ADALINE	ADAptive LInear NEuron
AI	Artificial Intelligence
ANN	Artificial Neural Networks
BDD	Binary Decision Diagram
BG	Background
BPTT	Backpropagation Through Time
CEGAR	Counter Example Guided Abstraction Refinement
CILP	Connectionist Inductive Learning and Logic Programming
CML	Connectionist Modal Logics
CTL	Computational Tree Logic
CTLK	Connectionist Temporal Logic of Knowledge
DNF	Discjunctive Normal Form
FK	Fully Knowledgeable
FOIL	First Order Inductive Learning

FOL	First Order Logic
ILP	Inductive Logic Programming
KBANN	Knowledge Based Artificial Neural Network
LTL	Linear Temporal Logic
MLP	Multi-Layer Perceptron
NARX	Nonlinear AutoRegressive with eXogenous inputs
NK	No prior Knowledge
NP	Nondeterministic Polinomial time
NuSMV	New Symbolic Model Verification
PK	Partially Knowledgeable
PROLOG	PROgramming in LOGic
PSSH	Physical Symbols System Hypothesis
RMSE	Root Mean Squared Error
RTRL	Real Time Recurrent Learning
SAT	SATisfiability of logic formula
SCTL	Sequential Connectionist Temporal Logics
SE	Software Engineering
SLD	Selective Linear Definite clause resolution
SMV	Symbolic Model Verification
SOM	Self-Organizing Maps

TDNN	Time-Delayed Neural Networks
XHAIL	eXtended Hybrid Abductive Inductive Learning
XOR	eXclusive Or

Abstract

The effective integration of knowledge representation, reasoning and learning into a robust computational model is one of the key challenges in Computer Science and Artificial Intelligence. In particular, temporal models have been fundamental in describing the behaviour of Computational and Neural-Symbolic Systems. Furthermore, knowledge acquisition of correct descriptions of the desired system's behaviour is a complex task in several domains. Several efforts have been directed towards the development of tools that are capable of learning, describing and evolving software models.

This thesis contributes to two major areas of Computer Science, namely Artificial Intelligence (AI) and Software Engineering. Under an AI perspective, we present a novel neural-symbolic computational model capable of representing and learning temporal knowledge in recurrent networks. The model works in integrated fashion. It enables the effective representation of temporal knowledge, the adaptation of temporal models to a set of desirable system properties and effective learning from examples, which in turn can lead to symbolic temporal knowledge extraction from the corresponding trained neural networks. The model is sound, from a theoretical standpoint, but is also tested in a number of case studies.

An extension to the framework is shown to tackle aspects of verification and adaptation under the SE perspective. As regards verification, we make use of established techniques for model checking, which allow the verification of properties described as temporal models and return counter-examples whenever the properties are not satisfied. Our neural-symbolic framework is then extended to deal with different sources of information. This includes the translation of model descriptions into the neural structure, the evolution of such descriptions by the application of learning of counter examples, and also the learning of new models from simple observation of their behaviour.

In summary, we believe the thesis describes a principled methodology for temporal knowledge representation, learning and extraction, shedding new light on predictive temporal models, not only from a theoretical standpoint, but also with respect to a potentially large number of applications in AI, Neural Computation and Software Engineering, where temporal knowledge plays a fundamental role.

Chapter 1

Introduction

This thesis contributes to two major areas of Computer Science, namely Artificial Intelligence (AI) and Software Engineering (SE). The work focuses on the development of methodologies and techniques of AI in the representation, learning and reasoning about temporal models. These methodologies are then applied to the verification and adaptation of models used for requirements and behaviour specifications in software engineering.

Artificial Intelligence has been developed by gathering contributions from different areas such of the natural sciences and engineering. Drawing inspiration from natural sciences and philosophy, AI can be defined as the search for artificial models of different human cognitive abilities such as reasoning, learning and adaptation [92]. Researchers in biology, neuroscience, psychology and philosophy have given important contributions to the field, improving the general understanding of different approaches to knowledge manipulation, as well as the actual cognitive aspects [51]. On the other hand, incorporating such aspects into computer tools can impact on a range of applications, especially in tasks where expert knowledge is considered essential to a system's performance [92].

More specifically, the integration of knowledge representation, reasoning and learning into a robust and effective computational model is one of the key challenges in Computer Science [43, 106]. Computational models of single cognitive tasks have been applied to several situations in different knowledge areas, but a general methodology for the integration several cognitive abilities remains an open problem [106]. One of the main issues to be tackled to achieve such integration is the specificity of many existing techniques. Systems based on logics, for instance, are very suitable and broadly used in representation and reasoning tasks, but are considered by several authors as too brittle to learning in certain domains [74].

A fruitful approach to the unification of cognitive abilities consists in integrating the representation structures and reasoning capacities from symbolic logics with numeric (quantitative) systems. Such systems are capable of numeric representation and manipulation, which will give the necessary flexibility that enables more plasticity to the learning task [74]. Examples of this integration can be found in fuzzy systems [114] and probabilistic logics [112]. Other approaches using numeric representation for such integration consist of probabilistic techniques which are based on graph structures such as Markov models and Bayesian networks [57].

Artificial Neural Networks (ANNs), also known as connectionist intelligent systems, are a good example of a graph structure for numeric data manipulation. They are biologically inspired in the functioning of the human brain. They have been used to model systems capable of flexible, robust learning in several domains. ANNs have been applied to domains where the available information is incomplete, noisy or incorrect [51]. They are broadly used in several domains for numeric learning and pattern recognition [59]. However, ANNs have as main disadvantage their limited knowledge representation capability. All the information learned by the networks is usually stored in the structure as numeric weights, and therefore may not be clear to the understanding of experts or to communication with other intelligent systems [53].

These limitations of ANNs have led to an intense research programme which aims at an effective integration of ANNs with symbolic reasoning and knowledge representation systems [19, 29, 53]. Besides unifying the plasticity and robustness of connectionist systems and the representation and reasoning performance of symbolic models into a single tool, integrated neural-symbolic systems also provide new insights into the capacity of connectionist structures to perform symbolic inference, bringing useful inspiration to research in neurosciences and reaffirming the importance of connectionism as a useful approach to Artificial Intelligence [19].

While these integrated neural-symbolic systems have shown considerable performance and applicability in propositional domains [53], the extension of these systems to deal with other knowledge domains remains an open problem. In the specific case of temporal domains, several techniques were proposed to extend existing neural network architectures to consider time as a dimension of knowledge to be learned [41, 95]. Also, logic systems which are capable to represent time have found a large number of applications in Computer Science since the pioneering works several authors, such as Pnueli [22, 44, 85]. The construction of neural-symbolic systems which are capable of dealing with temporal knowledge is therefore a natural step forward in AI research: the learning performance of (temporal) neural networks and the broad applicability of temporal logics make this integration a strong candidate to respond to the challenge of building robust intelligent tools which are applicable to a broad range of domains.

In Software Engineering, *formal methods* include a range of activities, including system specification, analysis and proof, transformational development and program verification [101]. The use of formal, logical systems in specifications allows for the application and use of representation and reasoning tools towards a more reliable process of software development, especially in the case of safety-critical systems [101, 56].

A successful example of the use of logic-based techniques in SE is model checking [22]. Model checking consists of the use of formal strategies to the verification of properties in models described through a formal language [56]. The formal verification of specifications not only brings more reliability to the developed products, but also allows the use of inference techniques to the identification of scenarios in which the system is subject to errors, and therefore speeding up the development process and reducing costs [101].

While formal specification and inference techniques are broadly applied in software engineering, a considerable research effort has recently targeted the application of machine learning techniques in SE. In particular, machine learning has been used to estimate external and internal aspects in a software project, as well as the analysis of features of a product [115]. More specifically, the proper integration of learning techniques and formal specification tools have been considered by several authors [83, 35, 23, 3] as a means to provide automated verification, refinement and adaptation of software models. This makes software specification a potential case study for the applicability of robust intelligent techniques, capable of integrating representation, reasoning and learning of temporal knowledge models.

1.1 Objectives

The main goal of our work is to construct a general framework for representing, learning and reasoning about temporal knowledge models. The framework consists of an integrated neural-symbolic system. Our work not only describes all the steps necessary to effect this integration, but also illustrates how the general framework can be adapted to applications in the verification and learning of software models. This can be broken in a number of sub-goals, as follows:

- Investigating and describing different languages for representing temporal knowledge.
- Analysing and extending existing techniques used in the literature for the integration between symbolic temporal languages with connectionist learning engines.
- Analysing learning strategies for neural networks, focusing on temporal knowledge learning.
- Specifying how to integrate the proposed neural-symbolic framework with techniques for the verification of properties in temporal models (more specifically, with model checking techniques)
- Evaluating the developed framework through the analysis of each individual framework step, as well as the framework as a whole.

1.2 Contributions

Due to its multidisciplinary nature, our work contributes to multiple fields. The main contribution of this thesis consists of a general, unified framework for representation, learning and reasoning about temporal models. This framework is capable of acquiring knowledge from different sources (such as the abstract description of a temporal model, and examples of its behaviour), reasoning about this model, learning and evolution according to newly acquired information, and representation in different languages for integration with other techniques. This allows the proper integration with tools for property verification in temporal models, therefore bringing new contributions to the area of software engineering. More specifically, we contribute to the integrated verification, adaptation an evolution of software models. These con-

tributions which are associated to the aforementioned objectives of our work, are as follows:

- We present different languages for symbolic representation of temporal knowledge, discuss different aspects regarding their syntax, semantics, applicability in different domains and interchangeability of knowledge among them. The considered representations include temporal logics, state diagrams and description languages for software specification.

- We investigate the integration between temporal symbolic knowledge and artificial neural networks, presenting and analysing different strategies to both translating a symbolic specification of a temporal model into a neural network architecture, and extracting the learned information from the corresponding neural networks.

- We investigate temporal learning strategies for neural networks, focusing on domains where the the learning stage is integrated to the representation of symbolic knowledge; we also investigate the possibility of using temporal dimensions to the represent of other forms of structured knowledge. This investigation led to the choice of the techniques and algorithms to be used for knowledge acquisition throughout the work.

- We show how to integrate the representation and learning framework with existing techniques of model checking, rendering an iterative process of verification and evolution of temporal models which is capable of integrating different sources of knowledge in order to generate an improved model description.

- We perform a comparative, theoretical and experimental evaluation of the proposed framework, through the analysis of each individual step in an isolated manner, as well as the analysis of the framework as a whole. The experimental

evaluation considers a broad range of domains, including traditional testbeds of temporal synchronization and software specification, as well as variations where the available knowledge is incomplete or incorrect, showing good results in aspects as learning performance, robustness and noise tolerance.

1.3 Related Work

Throughout this work, we start from the definitions of integrated methodologies for temporal reasoning and learning and validate the ideas by applying them into the cycle of verification and adaptation of software specifications. Integrating connectionist and symbolic methodologies for modeling intelligent behaviour has been, in the last decades, a widely investigated approach. However, this work is the first that applies the methodology to a significant number of testbeds in software engineering.

The gap between the knowledge representation of the symbolic and connectionist paradigms has been bridged through a broad range of works in the last decades. For instance, Smolensky [100], Towell and Shavlik [103], Holldobler and Kalinke [54] and d'Avila Garcez and Zaverucha [36] proposed different ways to represent symbolic knowledge in neural networks. Andrew at al [4], Craven and Shavlik [28] and d'Avila Garcez et al [37] investigated how to make the information learned by ANNs available in a symbolic manner. Browne and Sun [18, 19] survey these techniques, categorizing them into different aspects such as knowledge locality (also reviewed by Page [82]) and variable binding.

In our work, we give special attention to the temporal dimension under a discrete, linear point of view. The importance of temporal reasoning and learning systems has been highlighted by several works like those of Fisher et al [44] and Gabbay et al GaHoRe94 as an important objective to be pursued, helping to improve the applicability of these systems into real-life applications. The main influences of our work in

this regard come from temporal modal logics, especially imperative approaches that consider the future as a consequence of the past, as seen in the work of Barringer et al [6, 7]. Also, connectionist extensions that propagate information and allow learning through time can be found at the works of Elman [41], Lin et al [66] Siegelmann et al [95].

All these ideas collected from different areas of AI and integrated into a unique framework can then be applied to the representation, adaptation and evolution of temporal models. In the work of d'Avila Garcez et al [34, 35] we can find a general methodology where the use of verification and learning steps lead to improved temporal models. In this case, as in many others, the main focus is on automating the verification and refinement of software.

Temporal logic-based tools have been consistently used in the formalization of software models, and therefore in the automation of the verification process. Model checking [21, 22, 113] has been an important breakthrough in Software Engineering, providing formal languages for describing temporal models and also for the specification of properties over these models, as well as reasoning procedures to allow the verification of these given properties.

One of the issues to be tackled by model checking is the state explosion problem, that can occur in particular when the system performs parallel transitions. Although symbolic representation structures such as BDDs (Binary Decision Diagrams) have been used and improved to tackle such issues [71, 113], it still remains a considerable hurdle when applying model checking to large domains. An interesting approach to deal with the state explosion problem is to focus on an assumption of reachable states, instead of the overly pessimistic view of considering the whole state space, as proposed by [47]. This assume-guarantee approach is followed also by [78, 79], which integrate learning techniques to model checking in order to improve the definition of

such assumptions.

Abstraction techniques, which are based on removing or simplifying details which are not relevant to the property being verified, are considered general and flexible to handle the state explosion problem [10]. One of the approaches used to automate the process of abstraction is CEGAR - Counter example based abstraction refinement [23]. The idea of CEGAR consists in acquiring an initial abstraction of the model from its original description and then deploying an iterative process of verification and adaptation. Such process considers the use of lateral information from counter examples for the refinement of the abstraction.

Other approach that makes use of iterative cycles of verification and adaptation can be found in [2, 3]. In these works, the authors propose the use of inductive and abductive reasoning for the refinement of temporal models representing software requirements. The process also considers the use of a model checking tool for the verification of properties of the temporal models, and subsequent use of symbolic learning for the adaptation purposes.

Under the same perspective, the works of [104, 61] illustrate a rich application of verification and adaptation in the area of controller synthesis. Although there are differences between the areas, especially in the sense that synthesis is mainly concerned with complete models, similarities can be found in the objectives and algorithms used for synthesis and learning. These similarities are clear when considering the adaptation of existing models seeking to satisfy a given property or goal.

Two important, common features of these methodologies can be pointed out as limitations for certain domains. At first, they require an initial, abstract description of the temporal model to be considered, which may not be available in some cases. [49, 83] considered this as a major hurdle for the application of model checking techniques in already implemented system. To tackle this issue, the authors have

proposed the use of *black box checking*, which consists on the use of empirical learning techniques in order to acquire an original description of a temporal model from examples of its behaviour.

The second important feature to be highlighted is the exclusive use of adaptation for the refinement of models. This highlights one of the more significant differences of our framework when compared with the other methodologies for iterative verification and adaptation. Neural networks, by their distributed representation of knowledge, will naturally perform revision instead of refinement of the knowledge base, and therefore will be capable of performing changes on incorrect models, as well as filtering the effects of incorrect examples in noisy data used for empirical learning. This constitutes an important reason for our choice of connectionist structures for performing learning and adaptation.

1.4 Published Results

The work described in this thesis has been the subject of a number of publications in the recent years. In 2007, the theoretical foundations of our work have been published in conference proceedings such as IJCNN-07 [16] and AAAI-07 LaBodAg07. These works focus on the semantic definitions of the proposed temporal logics, their translation into neural networks and the experimental analysis of the learning performance.

The integration of learning abstract properties was the theme of a paper published in the ICANN'10 conference proceedings [15]. In that paper, we presented the first insights for using the co-called SCTL framework in more general applications for learning and adaptation of temporal models, allowing for the possibility of applications in Software Engineering.

This possibility was further investigated in the papers published at ASE 2010 [12] and at the NIER (New Ideas and Emerging Results) track at ICSE 2011 [14]. These papers present our neural-symbolic approach to learning and evolving software descriptions, and the integration of these techniques with model checking tools, in the development of the general framework of iterative verification and adaptation cycle.

Also, a general description of the developed work is accepted to appear in the journal IEEE Transactions on Neural Networks (special issue on white box non-linear prediction models) [13] in 2012. In that paper, several aspects from our framework is described, including the temporal logic representation, learning in the recurrent neural networks, extraction of knowledge and integration with other representation systems.

1.5 Organization

The remainder of the thesis divided into eight chapters.

In Chapter 2, we present a comprehensive description of background and related work which is relevant to the understanding of our work. The chapter starts with a general description of current challenges in AI which are relevant to our research; we then discuss some important techniques on both symbolic and connectionist AI approaches. In the end, we give a general idea of the features and advantages of existing approaches to the integration between the symbolic and connectionist approaches to AI.

In Chapter 3, we explain our proposal to integrate temporal knowledge with recurrent neural networks. At first, we present the temporal syntax we use for knowledge representation. We also explain a translation algorithm for propositional logic programs [36] that is used as foundation for our work, and after that, we propose the

temporal extension used by SCTL, discussing and comparing it with other existing techniques.

In Chapter 4, we focus on temporal learning in neural networks, focusing mainly on the case of SCTL, NARX-based networks. The chapter starts with an overview of temporal learning in neural networks, followed by the presentation of the algorithms and learning techniques proposed and used in the thesis. Moreover, we present and discuss elements related to the extraction of learned knowledge from neural networks, and other aspects related to learning.

In Chapter 5 we bring several experiments used in the analysis of the functionalities of our framework. We start by simple experiments illustrating the learning performance and how it is affected by the insertion of symbolic knowledge. Several experiments will allow an extended analysis, as well as an understanding of the extraction techniques. We also compare the results with other temporal neural-symbolic techniques.

In Chapters 6 and 7, we focus on applications to Software Engineering, illustrating all the steps necessary to the unification of a model checking tool with learning techniques. Chapter 6 will illustrate the languages used for representing software models properties and counter-examples, and how to allow the communication with SCTL. Chapter 7 will illustrate the iterative process of verification and adaptation, with the use of examples and experimental results to better explain the proposed techniques. In the end, Chapter 8 will enumerate the conclusions and possible paths for the continuation of this work.

Chapter 2

Background

2.1 Challenges in Artificial Intelligence

Since the conception of the term Artificial Intelligence, in the 1950s, the area has grown considerably, incorporating several techniques and approaches to tackle the ultimate problem of reproducing intelligence in computer systems. Nowadays, research in Artificial Intelligence is related not only to Computer Science, but also Biology, Neurosciences, Psychology, Philosophy and Social Sciences, among others [92]. Given this great variety of perspectives, the foundations of the area are defined differently, depending on the approach to be considered. In this section, we propose some basic definitions that will be used to guide the remaining of the work.

To guide our definitions, consider a basic agent as a entity capable to perceive information from an environment and take decisions based on the perceived information, acting back to the environment. Under this perspective, we can qualify intelligence as a property of an agent's decision-making process. Taking as reference the original description of Russell and Norvig [92], there are two main concepts that can be associated with intelligence. Rationality, which in a strict sense, is the process of

taking the logically correct decision. This concept can be softened and interpreted as the process of taking decisions that increase the chances of getting the best possible result. The second concept usually associated with intelligence is cognition, which we define here as the information processing faculties associated to an human agent.

Intelligent behaviour can also be analysed in terms of the inference capacities of an agent. Inference can be described as the process of using logical or probabilistic reasoning to derive consequences from existing information in a domain [92]. Usually, the term inference is associated with deductive inference, which consists in deriving a conclusion from a set of premises through the application of existing inference rules.

Computational implementations of deductive inference have been subject to research and implementation under many approaches. More specifically, the capacity to perform reasoning in a general case, independently of specific knowledge, is central to expert or knowledge based systems [98]. In these systems, while the actual implementation of inference techniques is fundamental, other aspects play an important role, such as the choice of a suitable language to represent the existing knowledge, the availability of this knowledge and the integration between the representation and reasoning.

Although the process of deductive reasoning is central to the establishment of intelligent behaviour, other forms of inference are also important to be considered. Inductive reasoning, the search from general rules from the presentation of its associated elements, is essential in many aspects when modeling intelligent and adaptable behaviour.

For the development of knowledge-based systems, one of the main challenges consists in obtaining and coding the necessary knowledge base for the implementation of intelligent systems [43]. In seeking solutions to this “Knowledge acquisition

bottleneck", intense research has been undertaken towards the automated acquisition of knowledge, also known as Machine Learning [72]

Learning is an essential component of intelligent behaviour [106], and can be defined as the capacity of an agent to adapt to stimuli from or changes in the environment, seeking for improvements on her performance regarding some objective [92]. The capacity of performing inferential reasoning is central to the process of learning: the generalization of rules from the observation of examples allows an agent to build or adapt a knowledge base according to the information perceived from the environment.

In the same way deduction can be described as inferring the conclusions (effects) from premises (causes) and inference rules, induction can be defined as inferring rules from a series of associated causes and effects. Under this analysis, a third kind of inference can be foreseen: given a set of rules, and the effects of the application of such rules, a set of possible causes can be inferred. This *abductive* reasoning has been considered in the AI community as a non-monotonic reasoning paradigm to address some of the limitations of deductive reasoning [38]. Developments in the area include the implementation of abductive reasoning in logic programming and its integration with inductive logic programming, providing tools with a broader range of alternatives to perform synthetic reasoning and learning [89].

This focus on the reasoning processes and the drive to develop rational tools produced a broad range of results in logic, statistics and correlated areas. The approach was dominant from the birth of AI, with important definitions from McCarthy [70], who gave important insights regarding the relation between philosophical and computational concepts, and Newell and Simon [80], which established the importance of the symbolic treatment of the information process.

In the work of Newell and Simon[80], they define the Physical Symbol System

Hypothesis (PSSH), which states that "A physical symbol system has the necessary and sufficient means for general intelligent action". A physical symbol system may be defined as a set of entities (symbols) which follow the basic rules of physics and are realizable by engineered systems. This means that, upon due analysis, any intelligent action can be rewritten under a symbolic structure, and performed by artificial means.

On the other hand, the inspiration drawn from the human cognitive capacities is also relevant to a great number of AI scientists. The complex nature of animal brains, and their capacity to perform several complex tasks at the same time fascinates and inspires new solutions to the development of artificial information-processing tools. The observation of human behaviour, and capacities such as analogy and language, were taken as reference for many works. A historical example of such inspiration is the Turing Test, devised by Alan Turing as the ultimate challenge to an artificial system that reproduces intelligence [105]. Turing suggests we should ask if the machine can win a game, called the "Imitation Game", where three participants are set in isolated rooms: a computer (which is being tested), a human subject, and a (human) judge. The human judge can converse with both the human and the computer, while both try to convince the judge that they are human. If the judge cannot consistently tell them apart, then the computer is considered to be able to "think", i.e., to demonstrate an actual intelligent behaviour.

The search for developing artificial systems to perform cognitive tasks also led to more literal interpretations, with different scientists seeking in the complex structure and massive connectivity of animal brains the best approach to perform tasks such as learning and adaptation. Smolensky [99], for instance, proposed the subsymbolic hypothesis as a response to the conceptual, symbolic approach to AI. This hypothesis states that "The intuitive processor is a subconceptual connectionist dynamic system that does not admit a precise formal conceptual level description", which he considers

as the cornerstone of the subsymbolic (connectionist) paradigm to AI.

Even though some challenges are common to the different approaches of AI, the complementary nature of the symbolic and subsymbolic paradigms makes their integration an interesting alternative to unify their individual advantages to tackle difficult problems [18]. More specifically, incorporating connectionist learning into the representation and reasoning structures from symbolic systems have been considered as an important approach to tackle the issue of robustness in the development of Intelligent Systems [107].

Throughout the remaining of this chapter, we will present a review of some ideas developed throughout the last decades in both paradigms of AI, as well as the integration between these areas. We will focus on those which are central in the definition of our work, as well as those necessary to the understanding of the new ideas presented in the remaining chapters.

2.2 Symbolic AI

In several areas of Computer Science and more specifically in the symbolic modeling of intelligent behaviour, logics play a fundamental role, which is comparable to the role played by calculus in physical sciences and traditional disciplines of Engineering. In the same way that differential equations can be used to model the behaviour of continuous systems, mathematical logic can be used as a non-ambiguous language to specify the structure and behaviour on discrete domains [50, 5]. Huth and Ryan [56] consider, as the main goal of logics in Computer Science, the development of languages to model the different situations, in such a way that formal reasoning can be applied to them.

As an example of logic language broadly used across several fields like Philos-

ophy, Mathematics and Computer Science, consider first-order logic (FOL). These systems are very expressive, allowing for functions and relations over elements in the domain to be represented. They also allow for quantification over these elements, but are different from higher-order logics for not allowing quantification over the used predicates [56]. Although first-order logic is undecidable, several fragments are broadly used for automated reasoning in Computer Science [50].

As a specific case of computational implementation of inference in first-order logics, we may consider logic programming. Although the term has been used in a broader sense to designate different symbolic systems with a strong logic component [67], a logic program is nowadays considered in a more strict sense, usually referring to sets of Horn clauses. A Horn clause is defined as a logic disjunction, with at most one positive literal. For instance, a definite Horn clause can be written in the form $\alpha \vee \neg\beta_1 \vee \neg\beta_2 \vee ... \vee \beta_n$, and also can be rewritten as a logic implication $\beta_1 \wedge \beta_2 \wedge ... \wedge \beta_n \rightarrow \alpha$

In the next chapter, we give more details about logic programming systems limited to the propositional case, which will be used in this work. For inference in first-order representation, there are several implementations of Prolog (PROgramming in LOGic), which incorporate mechanisms such as SLD-resolution to solve queries expressed as a simple positive literal. Resolution consists in an iterative process of application of a resolution rule as a theorem proving technique based on refutation [67]. SLD-resolution (Selective Linear Definite clause resolution) is a refinement of the resolution procedure, which is sound and complete for refutation in Horn clauses.

While resolution and its refinements are used for deductive inference in logic domains, several techniques have been proposed also to incorporate inductive and abductive reasoning into logic programming systems. Inductive Logic Programming (ILP) involves a broad range of techniques to implement inductive learning into logic

programming, i.e., building new theories from background knowledge and positive and negative examples, using logic programs to represent all the knowledge involved in the process [77]. Existing ILP implementations include Progol, Golem and FOIL (First-Order Inductive Logic) [76, 87], which are used for applications such as pattern recognition and data mining [17]. Besides, extensions based on the SLD-resolution and integration of constraint specifications have also been proposed to perform abduction in logic programs [60]. Integration between both forms of reasoning have also been proposed under a logic programming structure, as in the eXtended Hybrid Abductive Inductive Learning (XHAIL) system [89].

One of the main reason sto restrict first-order logics to the case of conjunctions of Horn clauses is the undecidability of first-order logics [56]. This mean that, in order to allow a computational implementation of sound and complete inference systems, one needs to restrict the representation power of the used background logic. Other important approach to the case of deductive inference in logics consists in converting the domain problem into a propositional representation, and then using inference techniques for propositional logic.

The most common example of propositional inference regards satisfiability verification (SAT), in which a propositional formula is examined to determine whether if there exists at least one assignment of values to the variables that makes the formula true. Although SAT is a NP-complete problem [26], this problem is widely studied, and there are many techniques developed to reduce complexity in the average case [69].

Regarding logic domains, graph-based structures are also widely used for representation and inference purposes. Graphs allow for the representation of relational structures, while several techniques to transformation or simplification of such structures become an interesting alternative to perform inference tasks. One important

example is the use of decision diagrams or trees.

A decision tree is an acyclic, directed graph, in which each node represents a variable from the domain. The set of edges from a node represent a decision regarding the variables, i.e., the possible values of the variable. Each path on the graph, from the root to one of the leaves, will then represent an assignment of values to all the variables. For the representation of propositional logic formulae, decision trees might be compressed in such a way that the terminal nodes can be grouped to two nodes, representing whether the assignment to the variables makes the formula true or false. This compressed representation, called BDD (Binary Decision Diagram), allows for operations without the need of decompressing the whole tree, and are used for formal verification and logic synthesis of circuits [113].

Decision trees are also broadly used for applications such as data mining, which require empirical learning. In traditional algorithms such as ID3 [86, 88], examples from the domain are used to define, for each variable, its information gain: a metric which represents the influence of such variables to the definition of the output classes of the domain. This information is then used to define the structure of the decision trees, seeking for a small representation of the domain and therefore allowing the extraction of a simple explanation for the desired classification.

2.3 Nonclassical logics

Three main components can define the structure of a logic: the associated proof theory, its syntax and semantics. The syntax regards which expressions are well formed, whereas the proof theory provides syntactic proof rules (and axioms) to identify which formulae are theorems of the logical system. Semantics, on the other hand, refers to the meaning of the terms and the symbols composing them, as well as the interpretations, models and the validity of formulae [56].

Modal logics are among the most important examples of logic systems used in Computer Science, due to their simplicity and expressiveness, giving an internal, local perspective to the representation of relational structures [9]. Modal logics extend conventional logical systems by adding operators to express one or more truth modes [56]. In modal logics operators like $\Box$ and $\Diamond$ are usually defined to represent necessity and possibility, respectively. The semantics of these operators will vary according to the logic in which they are used, as shown in Figure 2.3.

Logic	$\Box\varphi$	$\Diamond\varphi$
Temporal	φ is always true	φ is sometimes true
Epistemic	The agent knows φ	The agent believes φ as possible
Alethic	φ is necessarily true	φ is possibly true
Deontic	φ is mandatory	φ is allowed

Table 2.1: Different meaning for modal operators [56]

One of the most traditional approaches to analyse the semantics of nonclassical logics consists in the possible world semantics. The approach was originally proposed by Saul Kripke in the 1950s, and considers a relational structure of possible worlds. Each propositional variable can have a different value assigned in each of the possible worlds, and the modal operators can be defined regarding the existing relation among such worlds. Usually, an expression $\Box A$ is considered as true in a world w_i if, and only if, A is true in every world w_j such that $R(w_i, w_j)$ is true, where R is the relation between worlds that is defined in the used relational structure [56].

2.3.1 Temporal Logics

Representation of time can be considered as a fundamental issue the development of knowledge-based systems [85, 1]. Representing and manipulating time can be seen as essential to several areas in Computer Science, such as Databases, Artificial Intelligence, Software Specification, Hardware Development, Real Time and Distributed

Systems, among others [45]. Several different approaches have been proposed to represent time into a logical framework, although the term temporal logic usually refers to the modal treatment proposed originally by Arthur Prior under the name of Tense Logic [46].

As an alternative approach, first-order logics can be used in the representation and manipulation of time-related knowledge, through the use of parameters that associates a temporal reference to the predicates of the domain. In this case, an explicit set of predicates and axioms are necessary to represent the time flow and its properties. This constitutes the main difference when compared with the modal approach, which has the time flow intrinsically represented into the structure.

Another important difference between the approaches is the perspective under which the time is considered. Modal systems provide a local perspective to the time flow, characterizing it in concepts such as present, past and future. On the other hand, the representation based on predicates consider an external perspective, considering only the relations (earlier or later) between the time points without taking any of them as a reference to the present [46].

Other aspects are also very important when defining a logical system applied to temporal domains. The relation between time flow and changes in the environment, for instance, is very important to define the primitive concepts of the representation language [45]. One of the possible approaches consists in defining the events and of the system as the primary concepts, with the time flow being considered as an effect of these events. Representing, for instance, moments in time as equivalence classes of states, would indicate that the position in the time flow is kept the same until some change happens in the environment.

One example of the first-order representation that takes the events and states as reference is the *event calculus*. The event calculus is a first-order language based on

a set of predefined predicates which define relations between events, fluents and time points [63]. For instance, a predicate $Happens(e,t)$ would indicate that an event e happens in a time point t, and $Initiates(e,f,t)$ represents that an event e, initiates the fluent f (makes it start happening), at the time point t. The event calculus is broadly used under a first-order logic programming syntax, allowing for traditional inference mechanisms to be applied in the temporal case [75, 3]

On the other hand, if the time flow is taken as primitive, two different primitives can be considered to represent time units: points or intervals. The use of intervals will consider that actions and events do not occur in specific instants, but actually are associated to a certain duration in time. Still, an interval can then be defined by its initial and final points, or by the set of instants (time points) that happen between the beginning and the end.

However, some other aspects need to be taken into consideration when intervals are used as primitives of representation. While comparing points requires only three relations (earlier, later, and at the same time), comparing intervals requires at least 13 different relations [1]: the seven shown in Figure 2.1 and the converse of the first six relations in the figure. Also, when working with intervals, the idea of homogeneity needs to be taken in consideration: an event e is considered homogeneous if, whenever it is true during an interval i, it is also true for any subinterval of i.

For the sake of simplicity and for a broader applicability of our system, we will consider a representation that takes the time flow as reference, using time points as primitives and under a modal perspective. Two important temporal logic frameworks with these features are LTL (linear temporal logic) and CTL (computational tree logic) [45]. While LTL considers a linear, deterministic approach to the time flow, CTL allows for the representation of different possible successors for each time point. This idea of branching time requires the use of quantifiers for temporal expres-

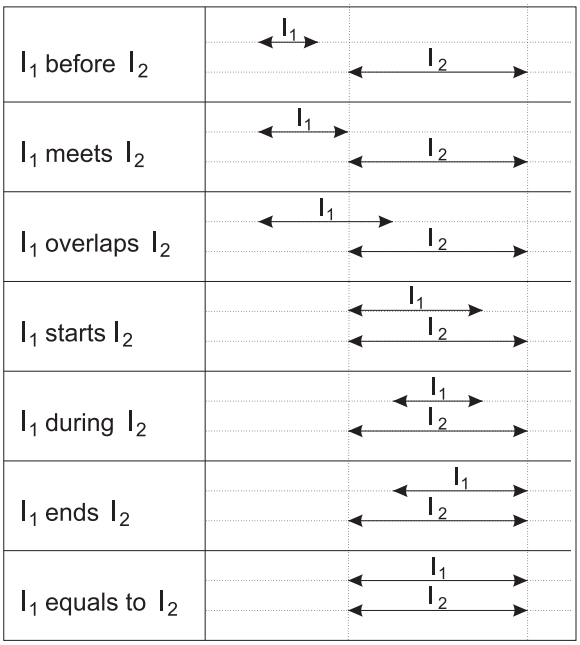


Figure 2.1: Relations between intervals [1]

sions, to represent if the expression is valid for at least one the possible ramifications of time (existential), or if it needs to be considered at every branch (universal) [65, 8].

In this work, we focus on the linear approach to temporal representation. More specifically we propose a logic programming language to the representation of linear, sequential aspects of time. In our proposed framework, the assignment of values to variables in the present can only be computed after the semantics of every time point in the past was already computed. More specifically, when assigning values to variables in a time point t, we consider that the information in the previous time point $t-1$ is already known an can be used for the computation. Besides the simplicity and the suitability to the representation of a broad range of domains, including models of reactive systems [7, 44], the choice for such language also takes into consideration its adequacy to the connectionist representation, as we will see in the next chapter.

For this language, which will be called SCTL (Sequential Connectionist Tem-

poral Logics), we consider a broad set of past and future temporal logic operators. The past operators includes the representation of the previous time point $\bullet$, always in past $\blacksquare$, sometimes in past $\blacklozenge$, and the weak and strong variations of the operator since, $\mathbb{Z}$ and $\mathbb{S}$ respectively. The complementary operators in the future are, respectively, next time point $\bigcirc$, always in the future $\Box$, sometimes in the future $\Diamond$, unless $\mathbb{W}$ and until $\mathbb{U}$. Therefore the syntax of a logic program can be kept as described before, but replacing the role of atom by the concept of temporal formulas:

Definition 1 *An expression α is defined as a temporal formula if, and only if, one of the following is true:*

- $\alpha = A$, *where A is a propositional variable;*
- $\alpha = \bullet\beta$, $\alpha = \blacksquare\beta$, $\alpha = \blacklozenge\beta$, $\alpha = \beta\mathbb{S}\gamma$ *or* $\alpha = \beta\mathbb{Z}\gamma$; *where β and γ are also temporal formulas;*
- $\alpha = \bigcirc\beta$, $\alpha = \Box\beta$, $\alpha = \Diamond\beta$, $\alpha = \beta\mathbb{U}\gamma$ *or* $\alpha = \beta\mathbb{W}\gamma$; *where β and γ are also temporal formulas;*

In our work, we consider a non-strict concept of past and future, i.e, they both include the present. In such a way, our operators are defined as follows:

- $\bullet\alpha$ is true at t iff α is true at $t - 1$
- $\blacksquare\alpha$ is true at t iff α is true at every time point $t' \leq t$
- $\blacklozenge\alpha$ is true at t iff α is true at some time point $t' \leq t$
- $\alpha\mathbb{S}\beta$ is true at t iff β is true at some time point $t' \leq t$, and α is true at every time point u such that $t' < u \leq t$
- $\alpha\mathbb{Z}\beta$ is true at t iff $\alpha\mathbb{S}\beta$ or $\blacksquare\alpha$ are true

- $\bigcirc\alpha$ is true at t iff α is true at $t + 1$
- $\Box\alpha$ is true at t iff α is true at every time point $t' \geq t$
- $\Diamond\alpha$ is true at t iff α is true at some time point $t' \geq t$
- $\alpha \mathbb{U} \beta$ is true at t iff β is true at some time point $t' \geq t$, and α is true at every time point u such that $t \leq u < t'$
- $\alpha \mathbb{W} \beta$ is true at t iff $\alpha \mathbb{U} \beta$ or $\Box\alpha$ are true

2.4 Logic Programming

The use of a symbolic representation of knowledge and the manipulation of such symbolic structures provides a powerful tool to perform reasoning. Logics provide a clear and non ambiguous language to represent knowledge, as well as the proof mechanisms to perform inference [56]. In a general sense, logic programming can be considered as any strategy that uses logic-based languages to represent the behaviour to be performed by a computer program. For instance, in one of the early influential works in AI, McCarthy [70] proposes such a logic-based language as adequate to the representation of complex computational tasks. In our work, however, we consider a more strict definition of logic programming, as follows:

Definition 2 *An atom A is a elementary formula, that cannot be broken into subformulae. A literal is an atom (A) or the negation of an atom (~ A).*

In our work, we will consider specifically the propositional case, where an atom is defined as a propositional variable. In first-order logics, an atom may also be a predicate (followed by the proper arguments) or a equality between terms.

Definition 3 *A logic program $\mathcal{P}$ is a set of Horn clauses. Each Horn clause is an expression in the form $A \leftarrow L_1, L_2, ..., L_n$, where A is an atom and L_i, for $1 \leq i \leq n$, are literals.*

Each clause might be considered as a logic implication, i.e., if the conjunction of the expressions in the right hand side of $\leftarrow$ is true, so the left hand side must also be true. For evaluating logic queries, expressed in the form of clauses without head, logic programming systems such as Prolog make use of inference systems based on resolution [68]. However, we are interested in evaluating the semantics of a program, as described below:

Definition 4 *An interpretation $I_{\mathcal{P}}$ of a program $\mathcal{P}$ is a mapping from the atoms of the program to truth values. A model is an interpretation $I_{\mathcal{P}}$ such that every clause in the program is satisfied, i.e, if when the conjunction of literals in the body of the clause is true, the atom in the head is also true.*

An interpretation $I_{\mathcal{P}}$ may also be represented through the set of atoms A such that $I_{\mathcal{P}}(A)$ is true. The semantics of logic programs may be defined, denotationally, w.r.t. the models of the program. More specifically, one may consider the denotational semantics of a program $\mathcal{P}$ as the minimum Herbrand model of $\mathcal{P}$.

Definition 5 *The minimum Herbrand model of a program $\mathcal{P}$ is a model $I_{\mathcal{P}}$ such that, for each other model $I'_{\mathcal{P}}$ of $\mathcal{P}$, $I_{\mathcal{P}}$ has, at most, the same number of elements of $I'_{\mathcal{P}}$ ($|I_{\mathcal{P}}| \leq |I'_{\mathcal{P}}|$).*

Before further definitions of the semantics of logic programs, we will describe different classes of programs that will be used throughout the work:

Definition 6 *A level mapping $|\ \ |$ of a logic program $\mathcal{P}$ is any mapping from the literals in $\mathcal{P}$ to natural numbers, such that $|A| = |\sim A|$. A program is called acceptable w.r.t. a model $I_{\mathcal{P}}$ and a level mapping $|\ \ |$ if, and only if, for each clause*

$A \leftarrow L_1, L_2, ..., L_n$ *in* $\mathcal{P}$*, the following is valid for* $1 \leq i \leq n$*: if* $M \vDash \bigwedge_{j=1}^{i-1} L_j$ *then* $|A| \geq |L_i|$*. A program* $\mathcal{P}$ *is called acyclic w.r.t. a level mapping* $|\ \ |$ *if, for every clause* $A \leftarrow L_1, L_2, ..., L_n$*,* $|A| \geq |L_i|$*, for* $1 \leq i \leq n$*.*

The class of acceptable programs consists in every program $\mathcal{P}$ that is acceptable with relation to, at least, one level mapping and one model. Also, a program is called acyclic if it is acyclic w.r.t., at least, one level mapping. One can easily verify that every acyclic program is also acceptable. For both classes of programs, a method for calculating the minimal Herbrand model is through the fixed point of a Immediate Consequence Operator $\mathcal{T}_\mathcal{P}$.

Definition 7 *The Immediate Consequence Operator* $\mathcal{T}_\mathcal{P}$ *is a mapping from an interpretation* $I_\mathcal{P}$ *of* $\mathcal{P}$ *to another, such that* $\mathcal{T}_\mathcal{P}(I_\mathcal{P})(A)$ *is true if, and only if, there is a clause* $A \leftarrow L_1, L_2, ..., L_n$ *such that* $\bigwedge_{i=1}^{n}(I_\mathcal{P}(L_i))$ *is true.*

For acceptable programs, we may find in [29] a proof that successive applications of $\mathcal{T}_\mathcal{P}$ converge to a fixed point, for acceptable programs $\mathcal{P}$. Also, such fixed point is a minimal Herbrand model of $\mathcal{P}$. Therefore, in order to perform a sound computation of the semantics of a program, one can develop a system that is capable of computing $\mathcal{T}_\mathcal{P}$ several times until the convergence. To define the number of $\mathcal{T}_\mathcal{P}$ executions necessary to reach the convergence, we will focus on the case of acyclic programs.

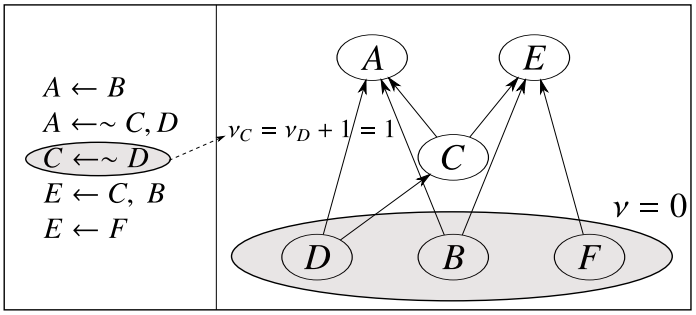


Figure 2.2: Illustration of an acyclic logic program

Any acyclic program can be represented in the form of a directed acyclic graph, where each vertex represents an atom and a directed arc from A to B represents that there is a clause with B as head and in which body A appears, either in positive or negated form. For the sake of clarity, we will refer to this relation by A "is body" of B. In Figure 2.2 we illustrate this relation between program and graph, as well as the assignment of a value ν to the variables according to the following definition.

Definition 8 *For each atom A in a logic program $\mathcal{P}$, we define a constant ν_A in such a way that $\nu_A = 0$ if A does not appear as head of any clause in $\mathcal{P}$. Otherwise, $\nu_A = max(\nu_B) + 1$, where $max(\nu_B)$ is the maximum value of ν_B among all atoms B that appear (in either positive or negative form) in the body of a clause where A is the head.*

One can notice that the assignment of a ν value to each variable of the program serves as a level mapping sufficient to meet the condition of the definition of acyclic program. Also, the graph representation helps to notice that, if a program $\mathcal{P}$ has a finite number of atoms A, it will also have a finite maximum value of ν_A, which is more specifically the size of the greater path between two vertices in the directed graph. This value we call $\nu_{\mathcal{P}}$, and it is important in the definition of the number of executions of the $\mathcal{T}_{\mathcal{P}}$ operator for reaching the convergence to the fixed point.

Lemma 9 *For any initial interpretation $I_{\mathcal{P}}$ of an acyclic program $\mathcal{P}$, a number $\nu_{\mathcal{P}}$ of successive executions of $\mathcal{T}_{\mathcal{P}}$ over $I_{\mathcal{P}}$ is sufficient to compute the fixed point of $\mathcal{T}_{\mathcal{P}}$, where $\nu_{\mathcal{P}}$ is defined as the greatest value of ν_A among all atoms in $\mathcal{P}$.*

Proof: We prove this lemma inductively. For each atom A such that $\nu_A = 1$, the first execution of $\mathcal{T}_{\mathcal{P}}$ over $I_{\mathcal{P}}$ will lead to a new interpretation that assigns A to *false*, because A is not head of any clause. Any new execution of $\mathcal{T}_{\mathcal{P}}$ will lead to the same interpretation. This means that $\nu_A = 1$ applications of $\mathcal{T}_{\mathcal{P}}$ guarantees a convergence point regarding A.

Now, when A appears as head of a clause, if we consider that a convergence point was reached regarding every atom B in the body of every clause which A is head, then one more execution of $\mathcal{T}_\mathcal{P}$ will lead to the convergence point regarding A, since it will assign values that will not change (according to the definition of $\mathcal{T}_\mathcal{P}$). In other words, if $max(\nu_B)$ executions of $\mathcal{T}_\mathcal{P}$ ensures a convergence point regarding all atoms B that are body of a clause with A as head, then $\nu_A = max(\nu_B)$ executions of $\mathcal{T}_\mathcal{P}$ will guarantee such convergence regarding A. Since we have that ν_A is a finite number for every atom A in an acyclic program $\mathcal{P}$, therefore $\nu_\mathcal{P}$ executions of $\mathcal{T}_\mathcal{P}$ will be enough to the computation of its fixed point. □

2.5 Neural Networks

The human brain processes information in a completely different manner than conventional digital computers. The connectionist approach to Artificial Intelligence takes the massively connected and parallel structure of the brain as inspiration to model intelligent behaviour [59].

> ANNs are massively distributed parallel processors, built upon simple processing units (artificial neurons), and having the natural propensity to store experimental knowledge and make it available to use. These networks are similar to the human brain in two aspects: the first regards the process of knowledge acquisition, which is performed from the environment through a learning process. The second aspect is related to the storage of the acquired knowledge, which happens through the connection strength between neurons, also called synaptic weights.

In the same work, we can find the description of several useful properties and capacities of the neural networks, among which one may cite:

- Capacity of non-linear computation
- Parallel processing to perform input-output mapping
- Adaptability, capacity of generalizing from the presented examples
- Contextualized information storage
- Tolerance to noisy or incomplete data
- Uniformity of analysis and project for similar networks.

The basic unit for the operation of neural networks is the artificial neuron as illustrated in Figure 2.3. As shown in the figure, its behaviour can be analysed in three different steps:

1. Each synapse (connection) represents the input of a neuron, being defined by a real-valued weight, in such a way that a value applied to the input is pondered (multiplied) by this weight. Besides these values received from external connections, the neuron also receives another constant value, called bias, which is also pondered by a variable weight;
2. A junction of the pondered inputs and the bias is performed, usually through the sum of these values, obtaining a new value v;
3. An activation function φ is applied to the value v obtained before, defining the output (activation) value of the neuron

Despite being based on a very simple processing unit, a neural network is capable of complex non-linear mapping from input to output, due to the use of non-linear activation functions, to the number of neurons and the connectivity structure among these neurons. These connectivity structures are called network architectures, which

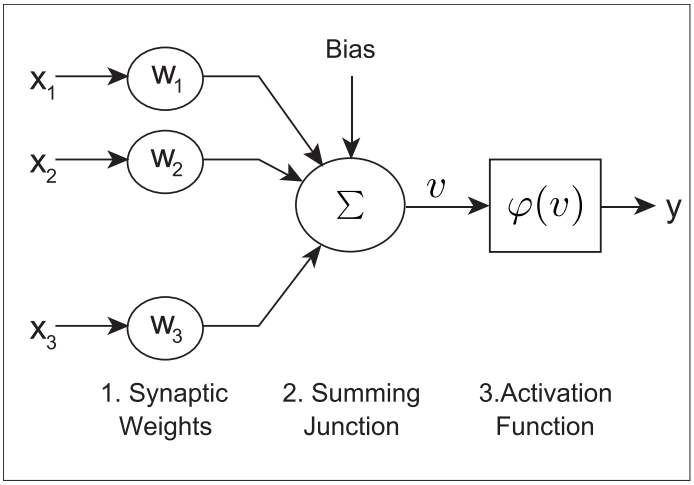


Figure 2.3: Image illustrating the computation steps in a perceptron

can be of three different kinds [51]: single-layered feedforward, multi-layered feedforward, and recurrent networks.

The simplest model of artificial neural network to be used to the task of pattern recognition is called perceptron, which was proposed in [90], and consists in a single neuron, with adaptable weights and bias and a threshold activation function (usually, the sign function). With this configuration, the perceptron will be activated if (and only if) the sum of the weighted values of input is greater than the negative value of the bias [1]. Other traditional model of single-layered network is the ADALINE (ADAptive LINEar neuron) [51], whose main difference with the perceptron consists in the activation function, which is an linear function in the ADALINE case.

The feedforward networks are those with acyclic architecture, usually organized in layers. The output value of the feedforward neurons of a layer is input to the input layer of the neurons in the following layer. In a feedforward network, connections between neurons in the same layer are not allowed, neither are connections from a neuron in a layer to a neuron in a previous layer. In the single-layered neural networks, there is only one neuron which makes the mapping from input to output.

[1] Several authors [51] analyse a neuron based on this negative bias, which is called the threshold.

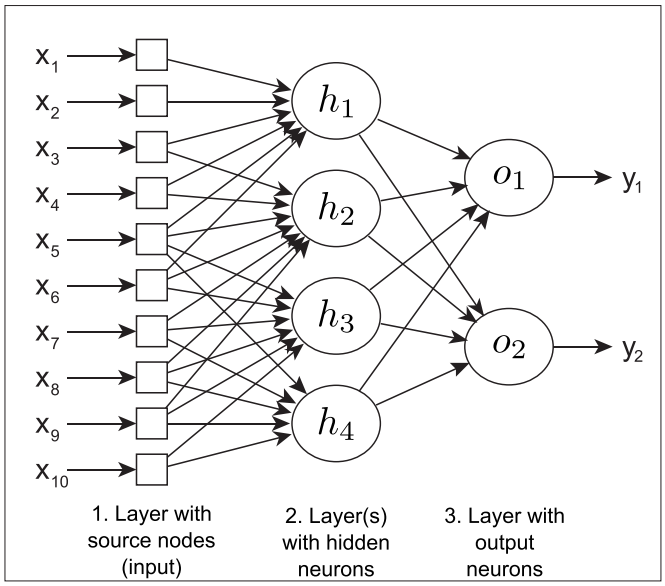


Figure 2.4: Example of a feed forward network with one hidden neuron

On the other hand, in multi-layered feedforward networks, some layers of neurons (called hidden layers) are inserted before the output layer, allowing the computation of more complex functions.

In general, the input layer of multi-layered networks is considered as a group of units that merely propagate the value received in the input into the hidden neurons, which will actually perform the operations detailed in Figure 2.3. Throughout the work, however, we change this pattern in some examples, seeking to improve the learning performance of the networks. Figure 2.4 shows an example of a multi-layered feedforward neural network, with one hidden layer.

Minsky and Papert [73], in their seminal paper, have proved that the simple perceptron is not capable of classifying patterns when they are not linearly separable. This had a strong impact in the research in connectionist AI during the 1970s. In the 1980s, several researchers resumed the work on ANNs, making use of multi-layered architectures - more specifically Multi-Layered Perceptrons (MLP). Hornik et al [55]

have then proved that feed-forward MLP networks, with a single hidden layer and a non-linear activation function for the neurons, are capable of approximating any Borel-measurable function (a class that includes all the continuous functions) to any desired degree of approximation, given enough neurons in the hidden layer.

2.5.1 Recurrent Networks and Temporal Processing

Haykin [51] considers that time is an essential component to the neural learning process, an ordered entity that is basic to several aspects when modeling intelligent behaviour, independently of having an implicit or explicit representation in the structure. In neural networks, two main resources are considered when extending traditional models to deal with temporal domains: delay units and recurrent links between neurons.

Delay units basically encapsulate short-term memory, returning as output the result of a function applied to the last values received as input. The simplest model of delay unit presents, as output, its input value which was presented in the previous time point. Several system were proposed taking as foundation the use of delay units to allow learning in temporal domains [108, 25].

The use of recurrent networks is based on a implicit representation of temporal aspects. In this case, the activation value in a neuron U will be applied to the input of a neuron in the same or in a previous layer. In such a way, this activation value of M will be used as information in a future computation of M or in another neuron in the same layer, influencing the behaviour of the network through time.

Among the traditional recurrent network models, an important example are the Elman networks, a recurrent MLP model which presents two different kinds of neurons in the first layer: the actual input layer, which receives values external to the network, and the context neurons, which receive the activation value from the hidden

neurons through recurrent links. For each neuron H in the single hidden layer, there is a context neuron C that will receive, through one of these links, the activation value of H. The remaining of the network is defined in the same way as a fully connected feedforward network, i.e. each neuron of the first layer is connected to all the neurons in the hidden layer, and each hidden neuron is connected to every neuron in the output layer. Figure 2.5 illustrates a simple Elman network, with one input, one output and two hidden and context neurons.

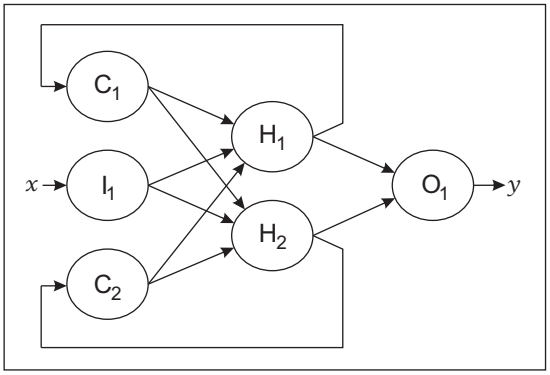


Figure 2.5: Example of an Elman Network

In our work, we will consider the NARX (Nonlinear AutoRegressive with eXogenous inputs) model [95]. This model makes use of simple delay units and recurrent links to allow the computation in a temporal dimension. In this architecture, each input neuron may receive as input a present value (applied directly) or a past value, applied to a chain of one or more delay units placed on the input of the network. Also, an input neuron can receive the activation value of an output unit, propagated through recurrent links with one or more delay units. The remaining architecture is similar to the MLP feedforward networks. Figure 2.6 shows an example of a NARX network, with the boxes marked with z^{-1} representing the delay units.

Siegelmann et al [95] have proved that NARX networks are able to emulate any other model of recurrent neural network, even considering the restrictions to the ar-

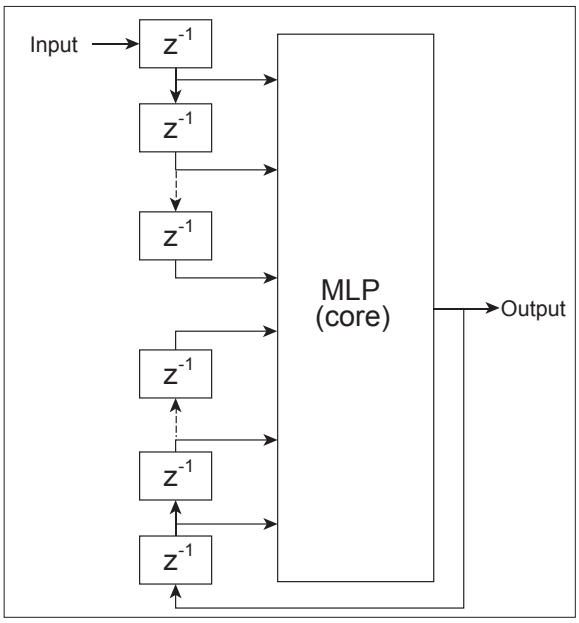


Figure 2.6: Example of a NARX network

chitecture imposed in the NARX model. On the other hand, the computational power of recurrent networks is equivalent to universal Turing Machines, as shown in different works. Siegelmann and Sontag [96], for instance, have proved that any Turing Machine can be simulated through the use of fully connected recurrent neural networks, with traditional neurons with sigmoid activation functions.

2.5.2 Learning in neural networks

Among the main features of the artificial neural networks, it is important to remark the natural propensity to adapt the structure according to the presentation of examples (empirical learning), through the use of learning algorithms. This learning task happens, usually, through the adaptation of the synaptic weights and bias values in the neurons according to the output error. Different network architectures allow for an empirical learning process according to different approaches: supervised, unsupervised, and reinforcement learning.

The main difference between supervised and unsupervised learning regards the existence of information about the output to be learned. While the task of supervised learning is based on the presentation of examples containing information regarding the relation between input and output, the unsupervised learning consists in finding structure patterns in the set of data, without specifying exactly which output should be assigned to each example [59]. A traditional case of connectionist system used to the unsupervised learning task is the Self Organizing Maps (SOM) [62], which produce a low-dimensional representation of an input space.

Reinforcement learning, on the other hand, consists in the process of learning through acting and observing the effects of those actions on the environment. The main goal in a reinforcement learning task consists in increasing the reward after an action or a sequence of actions, through a trial-and-error approach. Such a task can then be decomposed in several modules, in order to allow a proper balance between exploration and exploitation [102]. Neural networks are often applied in reinforcement learning, most commonly in the estimation or prediction of the reward associated with each action taken by an agent [27].

Considering again the case of supervised learning, the most popular algorithm for MLP networks is Backpropagation, proposed by Rumelhart et al [91]. This algorithm is based on the gradient descent approach to perform the correction of the synaptic weights on the input of a neuron according to the error obtained on its output. In the case of the hidden neurons, where no information about the output error is given explicitly, the algorithm is based on an error estimation backpropagated from the output layer.

The Backpropagation algorithm is described in the following steps, as depicted by Figure 2.7:

1. For each neuron k at the output layer, the value of δ is defined by product of the

derivative of the activation function $\varphi'(v_k)$ and the error in the output. The error for output neurons k is given by the difference between the activation value y_k and the expected value on the output z_k.

2. The weight w_{kj} for the connection between a hidden neuron j and an output neuron k is then corrected according to the value of δ_k. The old value of w_{kj} is incremented by the product $\eta \times y_j \times \delta_k$, where η is the learning rate.

3. In the hidden neurons j, the value of error is not available, therefore an estimation is used to calculate δ_j. This estimation is given by the sum $\sum_k \delta_k \times w_{kj}$, for every neuron k connected to the output of j. The definition of δ_j is then the product between $\varphi'(v_j)$ and the error estimation.

4. The weight correction for the connections to hidden neurons follows the same rule, by incrementing the current value by $\eta \times x_i \times \delta_k$. Considering that the input nodes have as output the same value as given by the input, the value of x_i is used directly.

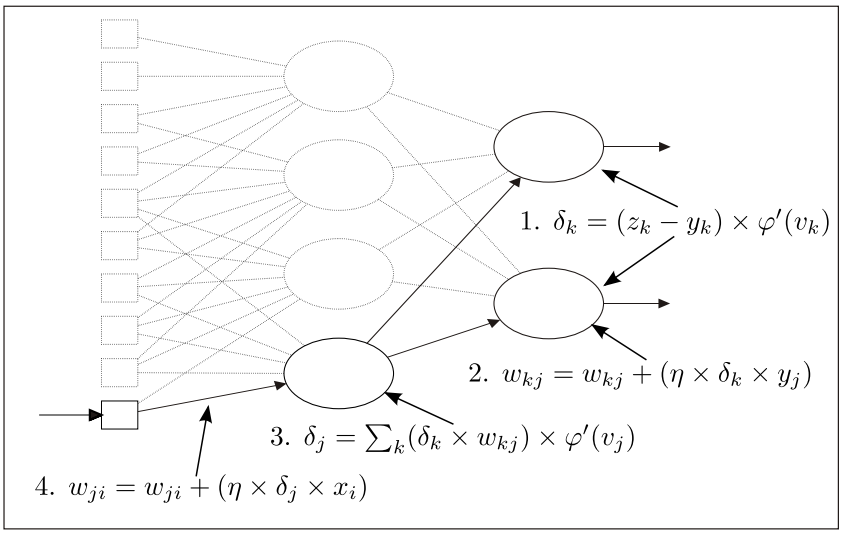


Figure 2.7: Illustration of the Backpropagation algorithm

The learning rate parameter η is a real positive number that defines how strong the

corrections to the weights will be. It is usually defined as a small number (below 0.1), and may be reduced through the training process, in order to allow the weights to be fine-tuned in the end of the process. Other constant that can be used in the training process is the momentum rate m. When momentum is considered, the weight's correction dw_{ji} in a time point t is given by $dw_{ji}(t) = (a \times dw_{ji}(t-1)) + ((1-a) \times \eta \times y_i \times \delta_j)$, where a is the momentum rate. This momentum rate is usually a value near 0, and it is used to avoid higher variation in the alteration of weights, making the learning process smoother. [51]

In order to perform temporal learning, the error information in the current time-point is not sufficient to ensure that the network will be capable to adapt to the presented series. Also, the error information calculated in the past or the future are required for such task. We return to the subject of learning in Chapter 5, where we analyse some traditional temporal learning algorithms and present some simplifications and enhancements that will be used throughout our work. Also, we illustrate the suitability of NARX models to temporal learning and knowledge integration through several experiments throughout the other chapters.

2.6 Neural-Symbolic Systems

Symbolic systems are very powerful for representing knowledge in a clear, non-ambiguous way, and also for reasoning over such representations [56]. On the other hand, in cases where there is no clear representation of the knowledge, analogy or generalization over a history of cases becomes a powerful tool. Neural (connectionist) systems are tailored for such cases, presenting the capacity to learn from examples, even in noisy or incomplete data sets, as their main advantage [51].

Several symbolic systems also are suitable to learning, however, the adaptation is too brittle, and usually the systems are not able to deal with the subtleties of the

learning task [52]. In the connectionist case, the distributed nature of the stored information, and the use of subtle adaptations of the numeric weights during the learning allows a balance between flexibility and noise tolerance [51].

The integration of connectionist and symbolic paradigms is therefore seen, by several authors [110, 53] as an important alternative to the development of intelligent systems. Their integration caters for the union of representation, reasoning and learning in a unique, robust framework. Also, they provide new insights to allow the understanding and the modeling of the human cognitive capacities[19].

It is possible to find in the literature different classifications for the neural-symbolic techniques. Hilario [52] considers two main classes of techniques: the unified approach, in which the symbolic reasoning and representation are incorporated into an connectionist architecture, and the hybrid approach, where the symbolic and neural tasks are performed by independent, communicating modules. Also, the unified approach might be divided in the connectionist symbolic processing, which uses the traditional neural networks on AI, and the neuronal symbolic processing, that uses architectures more inspired in the human brain model. Figure 2.8 illustrates this taxonomy in higher detail.

Wermter and Sun [110] consider a simpler taxonomy, following a similar strategy, in which three different classes are considered: The unified neural architectures, in which the neural networks are used for the whole task of representation and inference; the transformation architectures, which perform knowledge translation in both ways (symbolic $\leftrightarrow$ connectionist), and hybrid modular architectures. In our work, we will start focusing on transformation architectures, with the symbolic knowledge being translated into a neural network, where the learning process will occur. Later, for the studied application, we will consider a hybrid modular approach, where symbolic modules are integrated to perform model verification.

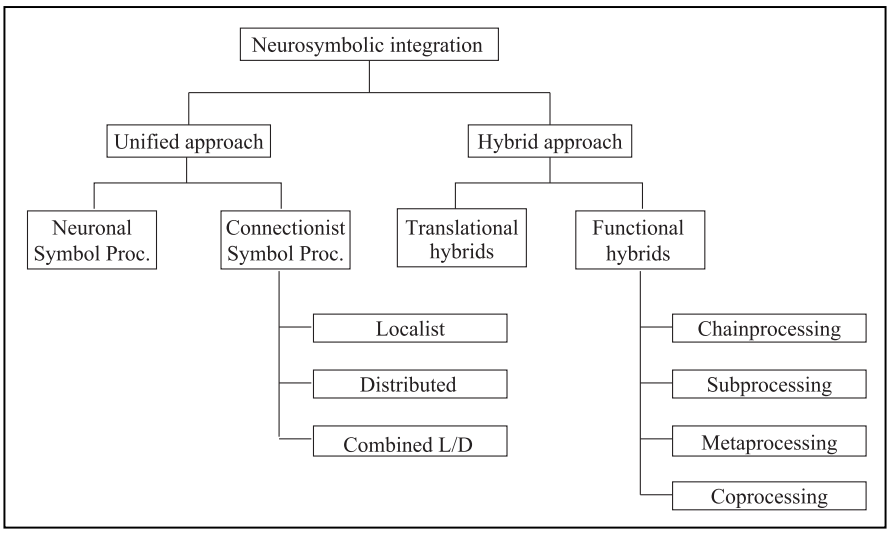


Figure 2.8: Taxonomy of Neural-Symbolic Systems

When analysing the representation of symbolic knowledge in neural networks, it is important to consider the association between the elements of the two representations. Two main strategies can be identified in this regard: the localist and the distributed representations [19]. A localist approach associates a single neuron (or group of neurons) to a symbol in the logic description, and therefore the activation of these individual units can be directly evaluated in a symbolic level, without necessity of knowledge regarding the remaining network [82].

On the other hand, several authors propose using a distributed representation of knowledge in neural networks. This representation can be defined through two properties: extension and superposition. While extension means that more than one unit (neuron) is used to represent a certain symbolic concept, superposition means that a unit is involved in the representation of more than one concept.

The main advantages of a localist approach are associated with the close relationship between the symbolic level and the connectionist architecture, when representing and manipulating knowledge. Such models are capable of realizing rule-based symbolic reasoning and replace hybrid systems with symbolic processing compo-

nents. This close relationship also allows an easier translation of knowledge between representations, and allows the manipulation and binding of variables. [18]

The idea of using a distributed representation seeks to overcome some issues that are pointed out as important flaws in the localist representation. Among then, the main issues are related to the brittleness and lack of generalization, due to the direct relation between symbols and neurons. Also, one may cite the difficulty to apply traditional connectionist learning algorithms, the high computational complexity due to the necessity of representation of all the used concepts and the lack of robustness to deal with noise.

However, the work of Page [82] argues that most of these pointed disadvantages are not actually true, due to the fact that localist representations may have a distributed component - since it is extended with units for the local representation of concepts. More specifically, if a symbol is represented locally in layer of a neural network, it is probably represented in a distributed fashion in a previous layer. Although this argument does not tackle the issue of computational complexity, several localist systems can be found to deal with plasticity and noise tolerance issues.

A good example of the localist approach is the KBANN (Knowledge-Based Artificial Neural Network) model [103], that consists of a MLP network generated through the application of a translation algorithm over a propositional logic program. Such an algorithm consists in generating, at least, one neuron for each atom that occurs in the program. In the case that this atom occurs as head of a single clause, the generated neuron will compute the conjunction of literals at the body of the clause. In the case that an atom occurs as head of multiple clauses, an extra atom is inserted as head of each clause, and the original atom will be represented, in the network, as a neuron computing the disjunction of such extra atoms. An example of such translation is shown in Figure 2.9.

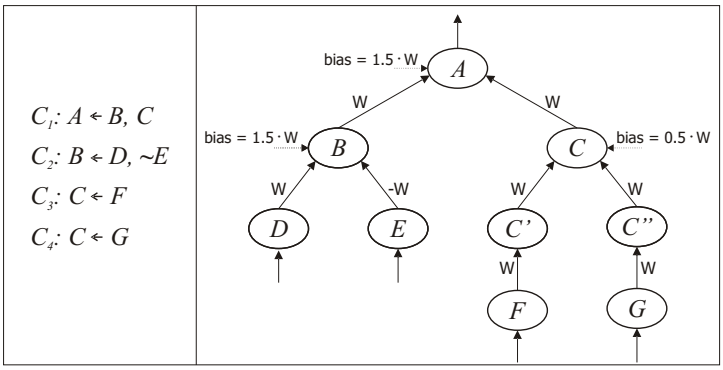


Figure 2.9: Example of a KBANN network

A different approach, considering a more formal definition under the logic perspective, is described by [54], that propose the use of three-layered MLP networks, extended with recurrent links, for representing the fixed point semantics of propositional logic programs, defined as function of the Immediate Consequence Operator $\mathcal{T}_{\mathcal{P}}$. The networks proposed by [54] use threshold activation functions for the neurons, in such a way that the hidden neurons can compute a conjunction of inputs and the output neurons compute a disjunction. In Chapter 4 we give further insights about this usage. Also, in Chapter 5 we provide more information about learning and representation of knowledge in neural-symbolic frameworks, focusing on our own system for temporal representation and learning.

Chapter 3

A Neural-Symbolic Model for Temporal Reasoning and Learning

3.1 On Logic and Neural Networks

As expressed in Chapter 2, a traditional approach to build unified reasoning and learning systems in an integrated neuro-symbolic fashion is by translating knowledge from one representation into another. For instance, initial knowledge represented by a symbolic language can be translated into a neural network that is semantically equivalent to the description. This target network can then be subject to learning with the presentation of examples. In turn, one can then explain the learned knowledge by extracting knowledge from the network thus explaining the system's behaviour [4]. In Figure 3.1, we illustrate how this process of knowledge integration takes place.

In Chapter 2, we also presented some approaches to translate propositional logic programs into feedforward neural networks capable of representing the same semantics. However, the mentioned systems present some structural flaws. KBANN uses unipolar activation functions for the neurons, and consequently unipolar representa-

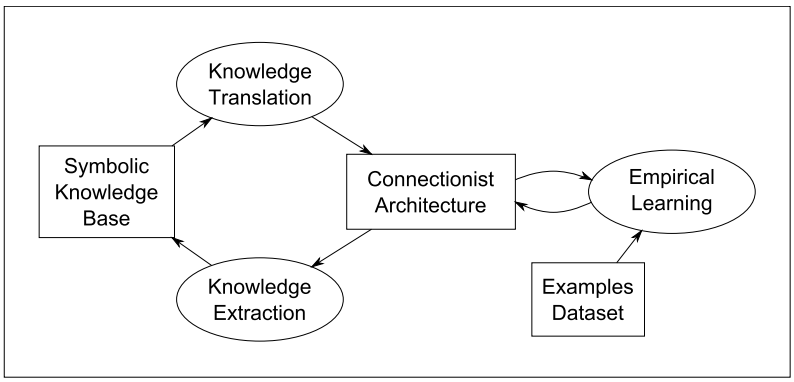


Figure 3.1: Information flow in a neural-symbolic system based on knowledge translation

tion of truth values (i.e. values near to 0 for *false* and values near to 1 for *true*). This may become a problem when a neuron has many inputs. In such cases, the sum of several values representing a *false* assignment (positive values close to 0) can lead to a high value, that can be interpreted as *true*. Also, this translation algorithm allows an unbounded growth of the number of layers of the network, depending on the original program. On the other hand, the work of [54] consider an activation function for the neurons that does not allow the application of a learning algorithm such as Backpropagation. The Connectionist Inductive Learning and Logic Programming (CILP) system, uses a translation algorithm proposed by d'Avila Garcez and Zaverucha [36], which addresses these issues and will be used as a foundation of our work.

CILP considers the same approach to represent basic logic operators as done by [54]. The great achievement of CILP consists in the use of a semilinear (sigmoid) activation function for the neurons, allowing the application of traditional learning algorithms that require the use of differentiable activation functions (such as Backpropagation [91]). Also, the activation function used is bipolar, ranging from -1 to 1, and therefore solving the problem of potential false positives in KBANN. CILP considers a bipolar representation of truth values, where the interval $[-1, -A_{min}]$ represents *false* and the interval $[A_{min}, 1]$ represents *true*. The value of A_{min}, as well

as the value the constant W, that defines the weights of connections, are defined in function of some features of the logic program:

- $k(l)$ and $\mu(l)$ are the number of literals in the body of a clause C_l and the number of clauses with the same head as C_l, respectively.
- $Max_{k\mu}$ is the maximum among the values of $k(l)$ and $\mu(l)$, and among every clause $C_l \in \mathcal{P}$.
- A_{min} is a positive constant, smaller than 1, arbitrarily defined in such a way that $\frac{1-Max_{k\mu}}{1+Max_{k\mu}} < A_{min} < 1$;
- $\phi(x)$ is the bipolar sigmoid function $\frac{2}{1+e^{-\beta x}} - 1$, where β is the parameter that defines the slope of the function. $\psi(x)$ is a linear function (identity);
- W is the weight of the positive connections, $-W$ is the weight of negative connections. W is defined by a value greater than $\frac{ln(1+A_{min})-ln(1-A_{min})}{Max_{k\mu}(A_{min}-1)+A_{min}+1} \cdot \frac{2}{\beta}$

As defined in the model of [54] the algorithm generates input and output neurons for representing each atom α. Afterwards, a hidden neuron is generated for each clause c, with connections from input neurons representing the body of c and connections to the output neuron representing the head of c. The values of biases are set in order to allow the hidden neurons to simulate a logic conjunction and the output neurons to compute a disjunction, and therefore to compute the $\mathcal{T}_\mathcal{P}$ operator. The values of W and A_{min} are defined to allow the proper computation of the desired semantics. The bias of the neurons are defined according to the number of inputs, in such a way that the hidden neurons represent properly a conjunction, and the output neurons represent a disjunction. These values are defined as $\frac{(1+A_{min})(k(l)-1)}{2}W$ and $\frac{(1+A_{min})(1-\mu(l))}{2}W$, respectively. These values are considered in order to allow the proper computation of conjunction and disjunction by the neurons, as describer in deeper mathematical

details in the works of d'Avila Garcez et al [36, 29]. Given such definitions, and the following notation, we present in the algorithm in Figure 3.2 the CILP algorithm for translating logic programs into neural networks.

- $Clauses(\mathcal{P})$: Set of clauses in a logic program $\mathcal{P}$;
- $Atoms(\mathcal{P})$: Set of atoms in the program $\mathcal{P}$;
- $Body(C_l)$: Set of literals in the body of a clause C_l;
- $Head(C_l)$: Atom in the head of a clause C_l;
- $Neurons(\mathcal{N})$: Set of neurons in a neural network $\mathcal{N}$;
- $InsertInputNeuron(\mathcal{N}, M)$ (respectively, $InsertHiddenNeuron$ and $InsertOutputNeuron$): Procedure to insert an input (respectively, hidden and output) neuron M in a network $\mathcal{N}$;
- $Activation(M)$: Activation Function of a neuron M;
- $Bias(M)$: Bias of a neuron M;
- $Connect(\mathcal{N}, M, M', W)$: Connects the neurons M and M' of network $\mathcal{N}$ with a weight W.

It is important to note that we consider a localist representation, where each neuron of the network n is associated to a logic formula α, in such a way that the activation value of n is between A_{min} and 1 if α is true, and between -1 and $-A_{min}$ if α is false. Considering the definition of activation function ϕ used in our work, we have that the value of v_n (the weighed sum of inputs and bias) needs to be higher than $ln(\frac{1+A_{min}}{1-A_min})/\beta$ to have a $\varphi(v_n)$ between A_{min} and 1 (the negative case is the same).

For the hidden neurons we have that, if all the positive literals in the body of a clause C_l are true, and all the negative literals are false, we have that the pondered

CILP_Translation($\mathcal{P}$)
  foreach $C_l \in Clauses(\mathcal{P})$ **do**
    $InsertHiddenNeuron(\mathcal{N}, h_l)$;
    foreach $A \in Body(C_l)$ **do**
      if $in_A \notin Neurons(\mathcal{N})$ **then**
        $InsertInputNeuron(\mathcal{N}, in_A)$;
        $Activation(in_A) \leftarrow \psi(x)$;
      $Connect(\mathcal{N}, in_A, h_l, W)$;
    end
    foreach $\sim A \in Body(C_l)$ **do**
      if $in_A \notin Neurons(\mathcal{N})$ **then**
        $InsertInputNeuron(\mathcal{N}, in_A)$;
        $Activation(in_A) \leftarrow \psi(x)$;
      $Connect(\mathcal{N}, in_A, h_l, -W)$;
    end
    if $out_{Head(C_l)} \notin Neurons(\mathcal{N})$ **then**
      $InsertOutputNeuron(\mathcal{N}, out_{Head(C_l)})$;
    $Connect(\mathcal{N}, h_l, out_{Head(C_l)}, W)$;
    $Bias(h_l) \leftarrow -\frac{(1+A_{min})(k(l)-1)}{2} W$;
    $Bias(out_{Head(C_l)}) \leftarrow -\frac{(1+A_{min})(1-\mu_l)}{2} W$;
    $Activation(h_l) \leftarrow \phi(x)$;
    $Activation(out_{Head(C_l)}) \leftarrow \phi(x)$;
  end
  foreach $A \in Atoms(\mathcal{P})$ **do**
    if $(in_A \in Neurons(\mathcal{N})) \wedge (out_A \in Atoms(\mathcal{N}))$ **then**
      $Connect(\mathcal{N}, out_A, in_A, 1)$
  end
  return $\mathcal{N}$;

Figure 3.2: CILP translation algorithm

sum of the k inputs is greater than $k \times A_{min} \times w$. Otherwise, the higher value for the weighed sum will be $(k-1) \times 1 \times w$ - $1 \times A_{min} \times w$ (remember that the higher value for a true input is 1, and the lower value is A_{min}). Therefore, analysing a hidden neuron h_l representing $C_l = \alpha \leftarrow \lambda_1, \lambda_2, ..., \lambda_k$, we have that, if $\bigwedge_{i=1}^{k}(\lambda_i)$ is true:

$$v_l > (k(l) \times A_{min} \times w) - \frac{(1 + A_{min})(k(l) - 1)}{2} w$$

$$v_l > \frac{2k(l)A_{min} - ((1 + A_{min})(k(l) - 1))}{2} w$$

Considering that $w > \frac{ln(1+A_{min}) - ln(1-A_{min})}{Max_{k\mu}(A_{min}-1)+A_{min}+1} \times \frac{2}{\beta}$ we have:

$$v_l > \frac{2k(l)A_{min} - (k(l)A_{min} + k(l) - A_{min} - 1)}{\beta} \times \frac{ln(1 + A_{min}) - ln(1 - A_{min})}{Max_{k\mu}(A_{min} - 1) + A_{min} + 1}$$

$$v_l > \frac{k(l)(A_{min} - 1) + A_{min} + 1}{Max_{k\mu}(A_{min} - 1) + A_{min} + 1} \times \frac{ln(1 + A_{min}) - ln(1 - A_{min})}{\beta}$$

By the definition of A_{min}, and knowing that $Max_{k\mu} \geq k(l)$ we can deduce that $k(l)(A_{min} - 1) \geq Max_{k\mu}(A_{min} - 1)$. Therefore, $v_l > \frac{ln(1+A_{min}) - ln(1-A_{min})}{\beta}$ and $\phi(v_l) > A_{min}$.

On the other hand, if $\bigwedge_{i=1}^{k}(\lambda_i)$ is false:

$$v_l < \frac{2((k(l) - 1) - A_{min}) - (1 + A_{min})(k(l) - 1)}{2} w$$

$$v_l < \frac{(2k(l) - 2A_{min} - 2) - (A_{min}k(l) + k(l) - A_{min} - 1)}{\beta} \times \frac{ln(1 + A_{min}) - ln(1 - A_{min})}{Max_{k\mu}(A_{min} - 1) + A_{min} + 1}$$

$$v_l < \frac{-A_{min}k(l) + k(l) - A_{min} - 1}{Max_{k\mu}(A_{min} - 1) + A_{min} + 1} \times \frac{ln(1 + A_{min}) - ln(1 - A_{min})}{\beta}$$

$$v_l < -\frac{A_{min}(k(l) - 1) + A_{min} + 1}{Max_{k\mu}(A_{min} - 1) + A_{min} + 1} \times \frac{ln(1 + A_{min}) - ln(1 - A_{min})}{\beta}$$

$$v_l < -\frac{ln(1 + A_{min}) - ln(1 - A_{min})}{\beta}$$

The same reasoning can be applied to the outputs, to show that the activation of an output neuron will be greater than A_{min} if the activation of at least one of the connected hidden neurons is greater than A_{min}, and will be less than $-A_{min}$ if the value from all the hidden neurons are less than $-A_{min}$. More details about the definition of the constants and analysis of the behaviour of CILP translation can be found in [36]. After verifying these numeric properties of the neurons, we can state the following:

Lemma 10 *Each hidden neuron h_l of the network $\mathcal{N}$ generated through the application of CILP translation over $\mathcal{P}$ ($CILP\_translation(\mathcal{P})$) computes the conjunction of the body literals in the body of the clause Cl_l of $\mathcal{P}$. Also, each output neuron representing an atom α computes a disjunction of the conjunctions related to the the clauses in which α is the head.*

Theorem 11 *Given an acyclic propositional temporal logic program $\mathcal{P}$, a neural network $\mathcal{N}$ generated by applying the CILP translation over $\mathcal{P}$ ($CILP\_translation(\mathcal{P})$) will compute the immediate consequence operator $\mathcal{T}_\mathcal{P}$ of $\mathcal{P}$.*

Proof: According to the definition, $\mathcal{T}_\mathcal{P}(I_\mathcal{P}(A))$ will be true if, and only if, A is head of a clause $A \leftarrow L_1, ..., L_n$ such that $\bigwedge_i L_i (1 \leq i \leq n)$ is true. Given the lemma above, we have that each of the hidden neurons will only return a positive value if all the connected inputs are also positive (or negative weighed by a negative weight). Given that the input of these neurons represent the literals in the body of the clause represented by the neuron, and the weights represent if the literal is an atom or its negation, we have that the output of a hidden neuron will be higher then A_{min} if the conjunction of literals in the body of the represented clause is also true (the output will be below

$-A_{min}$ otherwise). The translation algorithm also ensures that a neuron out_A will only receive connections from clauses where A appears as head, and therefore properly computing the disjunction and representing the semantic definition of $\mathcal{T}_{\mathcal{P}}$. □

3.1.1 Knowledge Representation in CILP

When representing logic programs as neural networks, one important question regarding the treatment of external inputs arises from the differences between both structures. When analyzing the structure of a logic program, all the information is expressed in the set of clauses. In this perspective, the information given as input to the network does not present a direct correspondence regarding the logic program. Two different approaches appear to be more adequate to deal with this information when calculating the fixed point of $\mathcal{T}_{\mathcal{P}}$: treating these external inputs as an initial interpretation of the atoms, or as a set of assumptions to be considered during all the steps necessary until reaching the convergence.

In the first approach, we consider the network as being semantically equivalent to the logic program, i.e. computing the fixed point of $\mathcal{T}_{\mathcal{P}}$ operator. In this case, the input vector will have a direct effect on the output of the first feedforward execution of the network. When considering several executions to reach the fixed point, the convergence property of the acyclic programs implies that the output of a network will always reach the same values, independently of the initial interpretation (initial input). On the other hand, keeping the information of the input vector as an assumption through all the steps of the fixed point calculation, the values at output will not be a function exclusively of the architecture, but will also regard the value applied on the input.

To illustrate the issues described above, consider the case of a simple program composed by the clauses $\{B \leftarrow A; C \leftarrow B\}$, where both atoms A an B can receive

an external assignment to their interpretation (as if a vector with two values was applied into the neurons representing A and B in the CILP-generated network). In the table below, we see the assignment of values to the atoms, when considering different inputs and the different treatments of this information. Notice, in the left columns, that if we consider the input as an initial interpretation, the program will converge to its fixed point (assigning false to every atom) independently of the input. On the other hand, the right columns shows how the program works when the initial assumptions are kept: if an atom is initially assigned to true by the assumption then its interpretation will remain positive, or else it will become positive if an execution of $\mathcal{T}_{\mathcal{P}}$ leads to it.

Input as:	initial interpretation				assumption			
Initial Input	$\emptyset$	$\{A\}$	$\{B\}$	$\{A, B\}$	$\emptyset$	$\{A\}$	$\{B\}$	$\{A, B\}$
First $\mathcal{T}_{\mathcal{P}}$	$\emptyset$	$\{B\}$	$\{C\}$	$\{B, C\}$	$\emptyset$	$\{A, B\}$	$\{B, C\}$	$\{A, B, C\}$
Second $\mathcal{T}_{\mathcal{P}}$	$\emptyset$	$\{C\}$	$\emptyset$	$\{C\}$	$\emptyset$	$\{A, B, C\}$	$\{B, C\}$	$\{A, B, C\}$
Following $\mathcal{T}_{\mathcal{P}}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{A, B, C\}$	$\{B, C\}$	$\{A, B, C\}$

Table 3.1: Example of execution

Given the intention of a system capable of delivering an output according to the presented input, especially considering the extension to represent the temporal semantics of SCTL (which uses the fixed point calculated in the past timepoints to define the present semantics), we will consider the treatment of inputs as assumptions for both the theoretical results and the applications shown as an example. In such a case, some important issues must be considered in the translation of the program into the network.

At this point, it is interesting to return to our discussion about the choice between acyclic and acceptable programs. In the last chapter, we restricted our framework to the case of acyclic programs, without clearly stating the reasons behind this choice. Now, when considering the CILP translation and the treatment of inputs as

assumptions, we can explain in further detail such options. Let us take as an example the following program $\mathcal{P}$ defined by the following clauses: $C \leftarrow B, \sim A$ and $A \leftarrow C, \sim A$. Considering a model $\varnothing$ (A, B and C assigned to false), and the level mapping $\{|B| = 1, |C| = 2, |A| = 3\}$, we can see that $\mathcal{P}$ is acceptable according to the definition 6 in chapter 2, and that $\mathcal{T}_{\mathcal{P}}$ will converge to a fixed point which is the same $\varnothing$ assignment.

However, if we consider as assumption that B is true, the program $\mathcal{P}$ will not converge to a fixed point. For a better understanding, notice that taking B as an assumption to $\mathcal{P}$ would be semantically equivalent to a have program $\mathcal{P}'$ given by the set of clauses in $\mathcal{P}$ incremented by a fact $B \leftarrow$ (i.e. $\mathcal{T}_{\mathcal{P}} = \mathcal{T}_{\mathcal{P}'}$). This program $\mathcal{P}'$, however, would not be acceptable anymore, and $\mathcal{T}_{\mathcal{P}'}$ would not converge, as we can see in Figure 3.3.

On the other hand, it is straightforward to verify that an acyclic program will not lose its acyclic nature by inserting more assumptions or facts. For this very reason, in order to treat input values to the network as assumptions during the computation of the fixed point semantics, we have limited our scope to the case of acyclic programs.

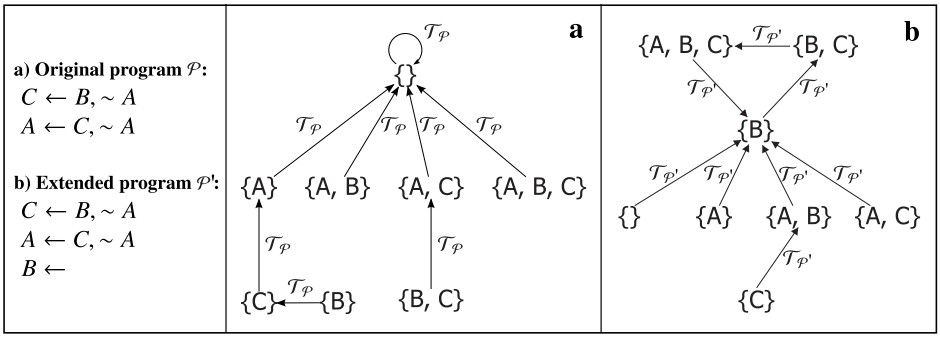


Figure 3.3: Analysis of the immediate consequence operator of an acceptable program

The first issue is about how to treat conflicting information on the input, in the case that the resulting interpretation of an atom, in the a previous execution of $\mathcal{T}_{\mathcal{P}}$

differs from the considered value applied in the input. In our approach, we keep the definition of default negation, i.e., a formula is false if there is no expressions assigning it to true. In this case, whenever one of two possible informations is true, the atom will be interpreted as true for the next execution of $\mathcal{T}_{\mathcal{P}}$. Otherwise the atom will be considered as false.

Other issue regards the possibility of information being lost in the propagation of values from input to output. Considering, for instance, the same example of Table 3.1.1, the application of the original CILP algorithm to the program would generate a network where the value applied in the input neuron A would not have any effect in the activation value of the output neuron representing A - i.e. there is no link between input and output neurons representing the same atom. If the value of A is necessary in the output of the system (as in the case of the temporal extension shown in the next section), this obtained value may be different from the applied input value.

To solve these issues, it is necessary to define clearly which atoms represent the input (i.e., which ones receive an external value as input), and which atoms represent the output of the program/network. Based on this information, we propose a treatment to the problem before the translation into a neural network. Such treatment consists in inserting a new atom A' for each atom A in which the conflict might occur. Such atom A' is then used to receive the external input information of the generated network, and a new clause $A \leftarrow A'$ is created so the information from the input can reach the output representing A. In Figure 3.4 we illustrate how this correction takes place in the logic program, and also how the translation into a neural network is effected.

More specifically, every time an atom A is specified as an input of the system, and it is either head of a clause or an output to the system, another variable A' will be created in such a way that a corresponding neuron $in_{A'}$ will exist in the network in

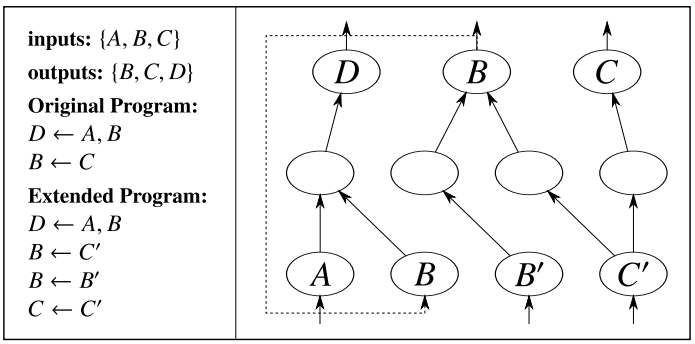


Figure 3.4: Illustration of the extension of CILP translation to allow the proper propagation of input values to output

order to receive the external input value. Also, a clause $A \leftarrow A'$, will be created in order to allow the value applied in $in_{A'}$ to be propagated to out_A, and therefore allow the proper value of A to appear in the output of the network. Moreover, in order to avoid unnecessary increments to the complexity of the network (more specifically to the value of ν of the program), we propose that if an atom A was not head of any clause in the original program, it should be replaced by B' in the body of every clause it appears. This happens, in this case, when the value of A and A' will always be the same during the process, but given that $\nu_A = nu_{A'} + 1$, the atom A' is preferable to be used in terms of numbers of $\mathcal{T}_\mathcal{P}$ executions needed to compute their convergence value. This situation is illustrated in Figure 3.4 by the atom C. In Figure 3.5 we describe properly the new algorithm to translate a propositional acyclic logic program into an equivalent neural network.

Theorem 12 *Considering an acyclic logic program $\mathcal{P}$, a neural network $\mathcal{N}$ generated through the application of the algorithm in Figure 3.5 over $\mathcal{P}$ computes in $\nu_\mathcal{P}$ feedforward executions the fixed point semantics of $\mathcal{P}$.*

Proof: According to Theorem 11, the network $\mathcal{N}$ generated by $CILP\_translation(\mathcal{P})$ is capable of computing the $\mathcal{T}_\mathcal{P}$ operator of the program $\mathcal{P}$. Given that one feedforward step of the network $\mathcal{N}$ computes $\mathcal{T}_\mathcal{P}$, the recurrent links feed the input to a

```
Extended_Translation(P)
    foreach A ∈ Atoms(P) do
        if A ∈ Inputs(P) ∧ ((A is head in P) ∨ (A ∈ outputs(P))) then
            AddClause(A ← A′);
            DefineInput(A, in_A′);
    end
    define v_P;
    return CILP_Translation(P);
end
```

Figure 3.5: Extension of CILP translation algorithm to allow the proper propagation of input values to output

new computation of $\mathcal{T}_{\mathcal{P}}$ over the last output, the network will converge to the fixed point after $v_{\mathcal{P}}$ executions, as shown in Lemma 9. □

3.1.2 Connectionist Modal Logics: CML and CTLK

In order to overcome the propositional limitations of CILP, extended neural-symbolic systems based on CILP were proposed. These systems are based on labeled modal logics, which extend the representation power of propositional systems, but avoid the complexity of (full) first-order logics. In [30], we find the Connectionist Modal Logics (CML), a system that uses a ensemble of CILP networks for representation of the semantics of labeled modal logic programs. Each clause of such programs is associated with a possible world, identified by a label. When translating the program, the set of clauses associated with each individual world is translated into a different CILP network.

As shown in section 2.3, modal operators can be defined in terms of possible worlds and the relation between them. For instance, an operation $\Box A$ is true in a world w_1 iff A is true in every possible world w_i such that $R(w_1, w_i)$. According to such definition, hidden and output neurons are inserted at each individual network, and connections among them are defined, whenever necessary, in order to represent the modal operators that appear in the logic program to be translated. In Figure 3.6

we show an example of a CML translation.

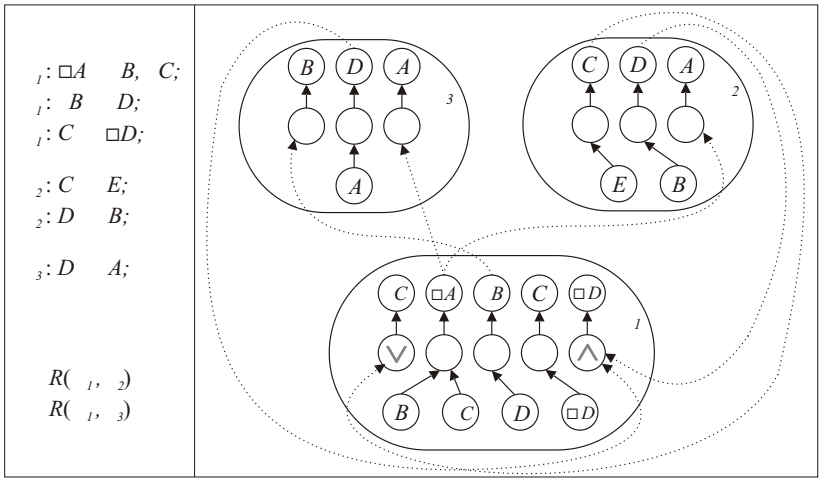


Figure 3.6: An example of CML program and equivalent neural networks

For the case of temporal reasoning, [33] proposed the Connectionist Temporal Logic of Knowledge (CTLK), an extension of CILP that deals with time and knowledge through modal operators K and $\bigcirc$. The same idea used on CML of having a different network to represent each possible world is also used on CTLK. The system assumes a discrete, linear temporal representation, and therefore the time sequence needs to be taken in consideration. In this case, each possible world is associated to a time point t_i, and a sequence $t_1, t_2, ..., t_n$ is defined, in such a way that t_2 is the time point immediately after t_1, t_3 after t_2 and so on. The temporal representation in this model is limited to the operator of *next time* $\bigcirc$.

The approach considered for translation is similar to the one used by CML. Each clause is labeled according to the time point to which it is associated, and a different network is generated for representing each individual time point. For representing formulas in the form $\bigcirc\alpha$, hidden neurons are generated, and connected with the networks representing other time points. Figure 3.7 depicts an example of CTLK translation, analysing only the temporal modal representation of the language: CTLK also

presents modalities to the representation of epistemic logic, which relates not only about truth and falsity of statements, but also to the knowledge of these statements by an agent. In the example shown in the figure, we will consider one of the increments to CILP described in previous section, by inserting a hidden neuron (dotted neuron in t_2 network) to allow the propagation of the input value of A to the output. This is a specific case where the missing link between input and output would affect the result of the whole ensemble.

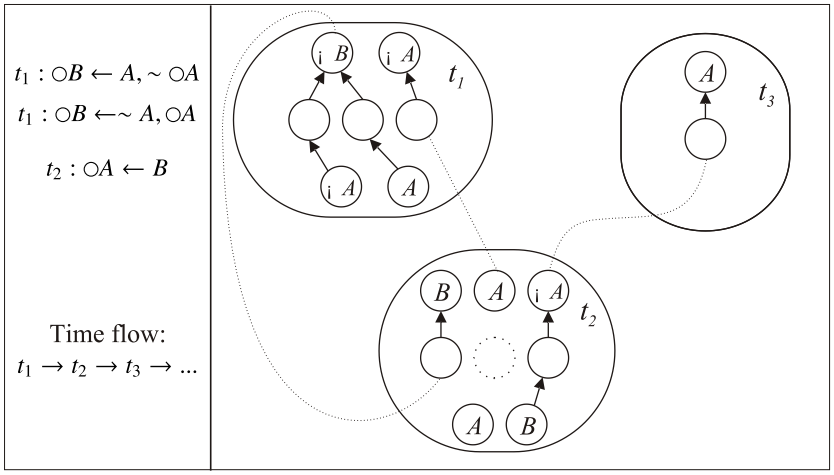


Figure 3.7: An example of CTLK program and equivalent neural networks

3.2 The sequential logic

In order to have an integrated framework for the manipulation and acquisition of temporal knowledge, our work requires the choice of a symbolic language capable of representing the desired models. To achieve such a goal, we consider a propositional logic programming syntax, extended with modal operators to represent temporal relations. Throughout the work, we call this language Sequential Connectionist Temporal Logic, or SCTL logic in reference to the proposed framework. The syntax of this language extends the concepts of clause and program in order to consider the definition

of temporal expressions.

Definition 13 *A logic formula α is called an temporal expression if, and only if, one of the following conditions is true:*

- *α is a propositional variable;*
- *α is of the form $\bigcirc\beta$, $\bullet\beta$, $\Box\beta$, $\blacksquare\beta$, $\Diamond\beta$ or $\blacklozenge\beta$, and β is also an temporal expression;*
- *α is of the form $\beta\mathbb{U}\gamma$, $\beta\mathbb{S}\gamma$, $\beta\mathbb{W}\gamma$ or $\beta\mathbb{Z}\gamma$, and both β and γ are temporal expressions.*

Definition 14 *A temporal clause is an expression of the form $\alpha \leftarrow \lambda_1, ..., \lambda_n$, where α is a temporal expression and λ_i for $1 \leq i \leq n$ are temporal literals, i.e., temporal expressions α_i or their negated form $\sim \alpha_i$.*

We also extend the definition of temporal program to include the declaration of propositional variables involved in the program, specifying which of them are tagged as input and output. The importance of defining the sets of input and output variables become clear in the next chapter, when the translation into neural networks is presented.

Definition 15 *A SCTL logic program $\mathcal{P}$ is given by a tuple $\mathcal{P} = \{Cl_\mathcal{P}, Var_\mathcal{P}, In_\mathcal{P}, Out_\mathcal{P}\}$, where $Cl_\mathcal{P}$ is a set of temporal clauses, $Var_\mathcal{P}$ is the set of propositional variables used in the clauses of $\mathcal{P}$, and $In_\mathcal{P} \subseteq Var_\mathcal{P}$ and $Out_\mathcal{P} \subseteq Var_\mathcal{P}$ are, respectively, the set of input and output variables of $\mathcal{P}$.*

3.2.1 Semantics

We consider a fixed point approach to define the semantics of the SCTL language, based on a sequential treatment of the time flow. We consider that some inference

over the present is only performed after the establishment of the past knowledge. In other words, the interpretation of a formula α at a point t on the time flow is processed by the system after the definition of the interpretation of every formula in the program at time points $t' < t$. This convention is based on the work of [7], where temporal logics are used in the imperative definition of a system, based on the use of consequence relations with premises about the present to infer something about the future. The main argument for the use of this approach is the direct relation with the execution flow of a system, presenting great representation power (as in the case of modeling reactive systems, for instance).

To define the semantics of the SCTL programs we will consider, at first, a simplified fixed point semantics, that attributes some meaning only to the $\bullet$ operator. We then define an immediate consequence operator $\bullet\mathcal{T}_\mathcal{P}$, that will be used as an auxiliary structure in the system (as we will describe afterwards). The definition of this operator in a time point t ($\bullet\mathcal{T}_\mathcal{P}$) is given as a function of the fixed point calculated for the prior time point $t-1$ ($\mathcal{F}_\mathcal{P}^{t-1}$). In a time flow beginning in a point $t = 1$, we consider a virtual point $t = 0$, such that $\mathcal{F}_\mathcal{P}^0$ is defined as true for every formula in the forms $\blacksquare\alpha$ and $\alpha\mathbb{Z}\beta$, and defined as false for every other formulas.

Definition 16 *The immediate consequence operator $\bullet\mathcal{T}_\mathcal{P}$ of a temporal program $\mathcal{P}$ is a transformation over interpretations of $\mathcal{P}$. The application of $\bullet\mathcal{T}_\mathcal{P}$ over an interpretation $I_\mathcal{P}^t$ at a time point t results in a new interpretation at t ($\bullet\mathcal{T}_\mathcal{P}(I_\mathcal{P}^t)$) that assigns true to every temporal atom α such that:*

a. α is head of a clause of the form $\alpha \leftarrow \lambda_1, \lambda_2, ..., \lambda_n$ and $I_\mathcal{P}^t(\lambda_1 \wedge \lambda_2 \wedge ... \wedge \lambda_n)$ is true.

b. α is an atom of the form $\bullet\beta$, and $\mathcal{F}_\mathcal{P}^{t-1}(\beta)$ is true.

Lemma 17 *For every acyclic temporal program $\mathcal{P}$, successive applications of the*

immediate consequence operator $\bullet\mathcal{T}_\mathcal{P}$ *at a time point* t *converge to a unique fixed point* $\mathcal{F}^t_\mathcal{P}$.

Proof: The convergence of $\bullet\mathcal{T}_\mathcal{P}$ in the case of acyclic programs can be verified in the same way as the convergence of the traditional $\mathcal{T}_\mathcal{P}$ for classic logic acyclic programs. One can easily verify that the rule "a." of the $\bullet\mathcal{T}_\mathcal{P}$ definition is similar to the definition of $\mathcal{T}_\mathcal{P}$. On the other hand, the associations made by rule "b" are directly defined before the first execution of $\bullet\mathcal{T}_\mathcal{P}$, and is kept constant during all the other executions (by our sequential definition of time), and therefore do not affect the convergence. □

To define the semantics regarding the other past operators, considering the sequential approach, we will use a recursive definition of such operators w.r.t. the prior and present time points. For instance, in the case of the ■ operator, we consider that a formula $\blacksquare\alpha$ is true at $t = 1$ if α is true at t, and $\blacksquare\alpha$ will be true at the time points $t > 1$ if α is true in t and $\blacksquare\alpha$ was true in $t - 1$. In such a way, when we define the arbitrary interpretation for the atoms at virtual time point $t = 0$, we can define an immediate consequence operator that is based on the present and prior time points, as in the items "b." to "f." of definition 18.

Following the same sequential approach, the idea of imperative semantics for the future operators is related to the commitment of an agent, and the actions taken to fulfill such commitment, as in [7]. In our work, we also consider such idea of commitments, but in a declarative approach. In such approach, the future operator declares the information that should be true at the next time points, without defining the necessary steps to define such goal.

In a similar way as before, we can define the future operators recursively, but at this time, regarding the next time point. For instance if $\Box\alpha$ is true at t, we have that α should be true at t (due to the non-strict definition of future), that $\Box\alpha$ should be true

at $t + 1$, and that these statements represent the complete definition of $\Box$. The other future operators can be defined as follows, taking as reference the $\bigcirc$ operator:

- $\Diamond\alpha \equiv \alpha \vee \bigcirc\Diamond\alpha$
- $\alpha\mathbb{U}\beta \equiv \beta \vee (\alpha \wedge \bigcirc(\alpha\mathbb{U}\beta)) \equiv (\beta \vee \alpha) \wedge (\beta \vee \bigcirc(\alpha\mathbb{U}\beta))$
- $\alpha\mathbb{W}\beta \equiv \beta \vee (\alpha \wedge \bigcirc(\alpha\mathbb{W}\beta)) \equiv (\beta \vee \alpha) \wedge (\beta \vee \bigcirc(\alpha\mathbb{W}\beta))$

In the case of $\Box$, our definition is based on a conjunction, i.e, when $\Box\alpha$ is true, both α and $\bigcirc\alpha$ are true. On the other hand, the other operators are defined using disjunctions, so a detailed analysis is necessary to define a consequence relation to represent them. Our option is to rewrite such disjunctions, avoiding the case where both are false (using the equivalences $(\alpha \vee \beta) \leftrightarrow (\neg\alpha \rightarrow \beta)$ and $(\alpha \vee \beta) \leftrightarrow (\neg\beta \rightarrow \alpha)$).

Again, we will consider our sequential approach (past $\rightarrow$ future) to define $\Diamond\alpha$, that will assume the form: $\neg\alpha \rightarrow \bigcirc\Diamond\alpha$. For the other operators, the only disjunction in the definition is $\alpha \vee \beta$, that allows two representations. We will use each representation to characterize a different operator, in such a way that $\alpha\mathbb{U}\beta$ is defined through $\neg\alpha \rightarrow \beta$ and, conversely, $\alpha\mathbb{W}\beta$ uses $\neg\beta \rightarrow \alpha$. This choice is because the $\mathbb{W}$ operator accepts, by definition, infinite sequences of α, i.e., unless β becomes; true, α will always be true.

Considering these observations, and the direct association between $\bigcirc$ and $\bullet$ ($\bullet\bigcirc \alpha \equiv \alpha$), we will define the immediate consequence operator, for the future temporal operators, also w.r.t. the present and the prior time points. These are described in the items "g." to "n" definition 18.

Definition 18 *The application of the immediate consequence operator $\mathcal{T}_\mathcal{P}$ of a temporal program $\mathcal{P}$ over an interpretation $I^t_\mathcal{P}$ at a time point t results in a new interpretation at t ($\mathcal{T}_\mathcal{P}(I^t_\mathcal{P})$) that assigns true to every temporal atom α such that one of the following conditions is true:*

a. α *is head of a clause of the form* $\alpha \leftarrow \lambda_1, \lambda_2, ..., \lambda_n$ *and* $I^t_{\mathcal{P}}(\lambda_1 \wedge \lambda_2 \wedge ... \wedge \lambda_n)$ *is true;*

b. $\alpha = \bullet\beta$*, and* $\mathcal{F}^{t-1}_{\mathcal{P}}(\beta)$ *is true;*

c. $\alpha = \blacksquare\beta$ *and both* $\mathcal{F}^{t-1}_{\mathcal{P}}(\blacksquare\beta)$ *and* $I^t_{\mathcal{P}}(\beta)$ *are true;*

d. $\alpha = \blacklozenge\beta$ *and either* $\mathcal{F}^{t-1}_{\mathcal{P}}(\blacklozenge\beta)$ *or* $I^t_{\mathcal{P}}(\beta)$ *are true;*

e. $\alpha = \beta\mathbb{S}\gamma$ *and either* $I^t_{\mathcal{P}}(\gamma)$ *is true or both* $\mathcal{F}^{t-1}_{\mathcal{P}}(\beta\mathbb{S}\gamma)$ *and* $I^t_{\mathcal{P}}(\beta)$ *are true;*

f. $\alpha = \beta\mathbb{Z}\gamma$ *and either* $I^t_{\mathcal{P}}(\gamma)$ *is true or both* $\mathcal{F}^{t-1}_{\mathcal{P}}(\beta\mathbb{Z}\gamma)$ *and* $I^t_{\mathcal{P}}(\beta)$ *are true;*

g. $\mathcal{F}^{t-1}_{\mathcal{P}}(\bigcirc\alpha)$ *is true;*

h. $I^t_{\mathcal{P}}(\Box\alpha)$ *is true;*

i. $\alpha = \Box\beta$ *and* $\mathcal{F}^{t-1}_{\mathcal{P}}(\Box\beta)$ *is true;*

j. $\alpha = \Diamond\beta$*,* $\mathcal{F}^{t-1}_{\mathcal{P}}(\Diamond\beta)$ *is true and* $\mathcal{F}^{t-1}_{\mathcal{P}}\beta$ *is false;*

k. There exists some formula β *such that* $I^t_{\mathcal{P}}(\beta\mathbb{U}\alpha)$ *is true and* $I^t_{\mathcal{P}}(\beta)$ *is false;*

l. $\alpha = \beta\mathbb{U}\gamma$*,* $\mathcal{F}^{t-1}_{\mathcal{P}}(\beta\mathbb{U}\gamma)$ *is true and* $I^t_{\mathcal{P}}(\gamma)$ *is false;*

m. There exists some formula β *such that* $I^t_{\mathcal{P}}(\alpha\mathbb{W}\beta)$ *is true and* $I^t_{\mathcal{P}}(\beta)$ *is false;*

n. $\alpha = \beta\mathbb{W}\gamma$*,* $\mathcal{F}^{t-1}_{\mathcal{P}}(\beta\mathbb{W}\gamma)$ *is true and* $I^t_{\mathcal{P}}(\gamma)$ *is false;*

3.3 SCTL - Sequential Connectionist Temporal Logic

3.3.1 Translating the immediate operators

As we stated in Chapter 2, we use a temporal representation based on a sequential approach, where the knowledge about the past is used to infer new information about

the future. Following this approach, our strategy to represent temporal knowledge is based on the propagation of values through the time flow, from a time point $t-1$ to its subsequent one t. Since the adopted semantics for our logic programs follows strictly this idea, we will consider how to represent this delayed propagation of information in the neural networks.

We have chosen the NARX models to extend the MLP networks with temporal processing. As described in Chapter 2, NARX networks use recurrent links and delay units to allow the propagation of information through time. In our concept chosen connectionist architecture, each time point is defined as the application of a different input vector to the network, and all the computation runs until the output values are returned at the output. The use of delay units (also incorporated in the NARX recurrent links) caters for the proper representation of the propagation from a value at time $t-1$ to the next time t. Therefore, this allows that a neuron representing a formula $\bullet\alpha$ receives the value of α in the previous time, and therefore provides a correct representation of the $\bullet$ operator. The representation of the complementary operator $\circ$, the situation is similar: the delay units allow the value associated to $\circ\alpha$ in $t-1$ to be propagated to a neuron to represent α at t also providing the semantically correct value. Throughout this section, we illustrate and analyse the case of propagation from α to $\bullet\alpha$, omitting the complementary $\circ\alpha \rightarrow \alpha$ case for the sake of clarity.

We considered different ways to perform this representation. The first idea consists of using only (delayed) recurrent links, carrying the value of an output neuron representing a formula α into the input neuron representing $\bullet\alpha$. When α appears as head of a clause in the translated program $\mathcal{P}$, the CILP translation algorithm will generate the output neuron to represent it. Otherwise, the network will not have the correct information about α in the output, and therefore the value of $\bullet\alpha$ will also not be properly assigned (as explained earlier in this chapter). Our proposed solution extends the insertion of clauses shown in the algorithm of Figure 3.10 to propagate the

values from input to output, by also inserting a clause $\alpha \leftarrow \alpha'$ every time a formula $\bullet\alpha$ appears in a program $\mathcal{P}$ where α is not head of any clause. In this case, the value of $v_{\mathcal{P}}$ might be incremented, due to the insertion of a new clause.

Our second idea is similar to the first one, except that for each inserted clause $\alpha \leftarrow \alpha'$ (for a formula α that does not appear as head of $\mathcal{P}$), we replace all the occurrences of α in the body of other clauses by α'. In such a way, even inserting clauses, the head of these inserted clauses will not appear as body of any other clause, so the value of v_P will not be incremented. On the other hand, since α' and α are semantically equivalent, the translation keeps the correct value associated to the variable.

The third idea makes a better use of the available resources of NARX architectures in order to reduce the size of the network. This approach considers the insertion of delay units before the input units, to apply the value of $\bullet\alpha$ every time that α does not appear as head in the program. In such a way, we avoid the clauses inserted on the other approaches, whose only purpose was the temporal propagation. For each formula of the form $\bullet^n\alpha$, we insert the delay units as follows:

- If a formula $\bullet^i\alpha$ appears as head of a clause in $\mathcal{P}$, where $0 \leq i < n$, we create a recurrent link from the output neuron representing $\bullet^{max(i)}\alpha$ to the input neuron representing $\bullet^n\alpha$, such that this link will have $n - max(i)$ delay units.

- If no formula $\bullet^i\alpha$ appears as head in $\mathcal{P}$, add n delay units before the input neuron representing $\bullet^n\alpha$, so this neuron will receive the value of α applied at time point $t - n$.

In Figure 3.8 we present the translation algorithm that performs the translation from temporal logic programs into recurrent neural networks, according to the third approach. Below, we also give more details about the correctness of our approach,

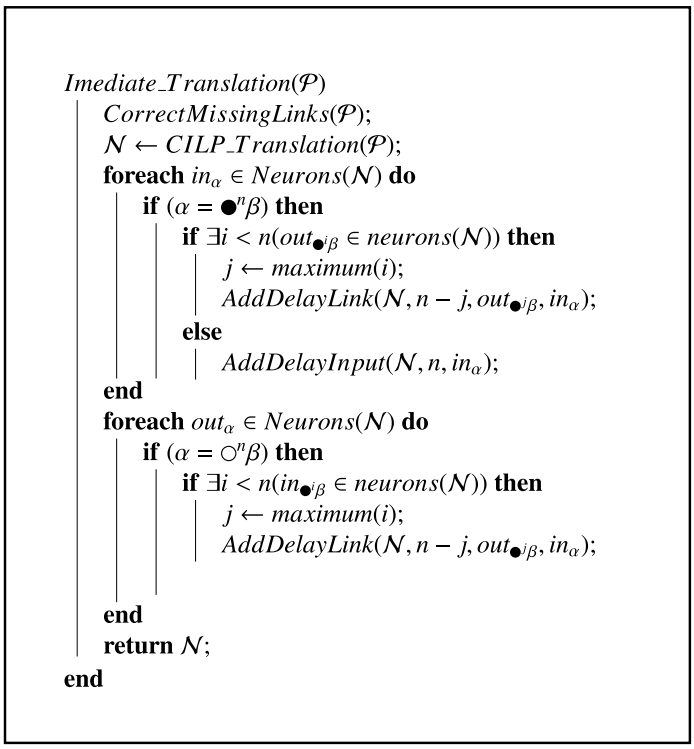


Figure 3.8: Translation of ●-based programs

in terms of the semantic equivalence between the original program and the generated network.

Lemma 19 *A neural network $\mathcal{N}$ generated through the application of the translation algorithm ●-based_Translation over an acyclic temporal program $\mathcal{P}$ computes the immediate consequence operator $\bullet\mathcal{T}_\mathcal{P}$ of $\mathcal{P}$.*

Proof: We will use an inductive proof. For the first time point ($t = 1$) given the initial values for the $\bullet\alpha$ formulas (the values defined for $t = 0$), we have that the computation of the operator $\bullet\mathcal{T}_\mathcal{P}$ is similar to the $\mathcal{T}_\mathcal{P}$ operator of the classic propositional case, and also converges to the proper fixed point. For the inductive step, we may consider that, for every time points t' such that $t' < t$, either the network $\mathcal{N}$ presents an output neuron out_α properly representing $\mathcal{F}^{t'}_\mathcal{P}(\alpha)$ or the value of α is given as input. Therefore, for each formula $\bullet^n\alpha$ in t, if the value of $\bullet^i\alpha(i < n)$ is at the output of the

network, the recurrent link with $n - i$ delay units will apply the correct value to the value $in_{\bullet^n\alpha}$. On the other hand, if $\bullet^i\alpha$ is not represented as output for any $i < n$, the input value of the neuron $in_{\bullet^n\alpha}$ is correctly given by the chain of n delay units, that receive the value of α as input. Given the convergence of $\bullet\mathcal{T}_\mathcal{P}$, we have that the network reaches the desired fixed point, and so the procedure can be repeated for all the other time points. □

3.3.2 Differences between recurrent connections

In Section 3.1, we showed that in order to compute the fixed point semantics of a propositional logic program, CILP makes use of recurrent connections. Each connection is used to propagate the result of a $\mathcal{T}_\mathcal{P}$ execution, related to an atom A (represented in a neuron out_A) to the input of the network (in a neuron in_A), in such a way that a new computation of $\mathcal{T}_\mathcal{P}$ is performed over this assignment to A. The process is repeated until the fixed point $\mathcal{F}_\mathcal{P}$ of the program $\mathcal{P}$ is reached (i.e. after $\nu_\mathcal{P}$ computations).

When considering SCTL, the translation performed by CILP remains the same, except by the insertion of delay units and (delayed) recurrent connections for the computation of the temporal operators. Differently from the recurrent connections described above, these SCTL recurrent connections will be used to propagate the value of out_α at a time point $t-1$ (which represents $\mathcal{F}_\mathcal{P}^{t-1}(\alpha)$ into a neuron in_α at time t [1], allowing for the correct computation of the semantics at a new time point.

For a proper understanding of the connectionist representation, it is important to make clear the distinction between these different kinds of recurrent connections, and also how does the system behaves regarding the multiple feedforward executions and the propagation of information, in order to perform a *correct* computation of the logic

[1] The connections are also used for the propagation of values from a neuron $out_{\bigcirc\alpha}$ at $t-1$ into a neuron in_α at t.

program given as background knowledge. In the next chapter, when we explain the backpropagation of values for learning purposes, this difference becomes even more important to allow the proper integration of reasoning and learning.

To explain this process, it is important to emphasize that each input neuron of the network receives one unique value as input. Moreover, in SCTL, each input neuron can receive a value from any of three different sources:

a External inputs of the system

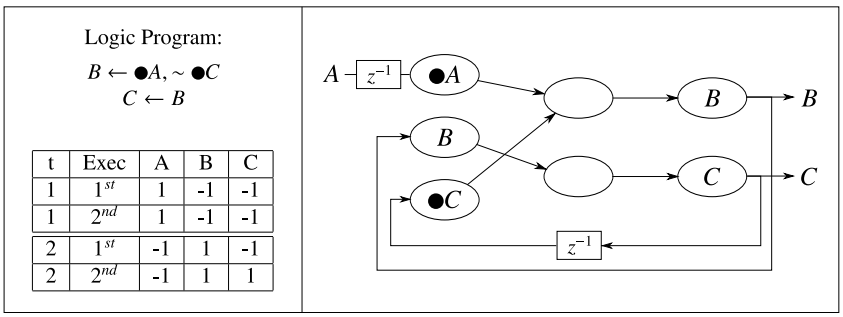
b SCTL recurrent links

c CILP recurrent links

The computation of each time point t in SCTL comprehends the whole process to reach $\mathcal{F}^t_{\mathcal{P}}$, i.e., $\nu_{\mathcal{P}}$ computations of the $\mathcal{T}_{\mathcal{P}}$ operator, which is represented by one feedforward execution of the network. In order to calculate $\mathcal{F}^t_{\mathcal{P}}$, the system must be presented with the assumptions in t represented by the external inputs and the result of the computation of $\mathcal{F}^{t-1}_{\mathcal{P}}$. Therefore, for the first feedforward execution at each time point, these values will be applied at the input of the neurons in groups a and b. For the neurons in group c, which represent the intermediate assignments to atoms during fixed point calculation, the convergence theorem 17 states that their initial value does not affect $\mathcal{F}^t_{\mathcal{P}}$, and therefore any value can be assigned (in our case, we arbitrarily assign false, represented in the network by -1).

For the following feedforward steps until the $\mathcal{F}^t_{\mathcal{P}}$ is reached, the applied values for the neurons in group c are exactly the values received by the CILP recurrent connection, i.e. a neuron in_α will receive in t the obtained value of an output neuron out_α in $t-1$. For the other groups a and b, we define the representation of the inputs by considering them as assumptions regarding the whole time point, as described in the previous section. Therefore, the values in the input of these neurons will be kept

the same during all the ν feedforward executions performed until the fixed point is achieved. Then, for the next time point, new values will be presented. Figure 3.3.2 illustrates this process in a simple example.



t	Exec	A	B	C
1	1^{st}	1	-1	-1
1	2^{nd}	1	-1	-1
2	1^{st}	-1	1	-1
2	2^{nd}	-1	1	1

Figure 3.9: Example of the different kinds of recurrent links in the SCTL model

3.3.3 Full SCTL translation

After defining the strategy to compute the $\bullet\mathcal{T}_{\mathcal{P}}$ using a recurrent neural network, we will now extend this system to allow the representation of the semantics regarding the other temporal operators. Our strategy is based on inserting clauses in the original program, in such a way that every formula containing temporal operators has its semantics defined through these clauses, that will consider only the present or formulas in the form $\bullet\alpha$ or $\circ\alpha$. The insertion of clauses is described in Algorithm 3.10.

Lemma 20 *Consider $\mathcal{P}_1$ as a temporal logic program, generated through the translation of clauses for each temporal operator over an original program $\mathcal{P}$, according to the Algorithm in Figure 3.10. For each formula α in $\mathcal{P}$, $\mathcal{T}_{\mathcal{P}}(I^t_{\mathcal{P}})(\alpha)$ is true if, and only if $\bullet\mathcal{T}_{\mathcal{P}1}(I^t_{\mathcal{P}})(\alpha)$ is also true.*

Proof: The algorithm inserts clauses representing exactly the semantic definitions of the operators. We can verify this by analyzing individually all the inserted clauses. Take, for instance, the case of the $\mathbb{S}$ operator. The first inserted clause ($\beta\mathbb{S}\gamma \leftarrow \gamma$) represents exactly the first definition of the item *e.* in definition 18. Also, $\bullet\alpha$ correctly

```
Logic_Treatment(P)
    foreach α ∈ atoms(P) do
        if α = ■β then
            AddClause(P, ■β ← β, ●■β);
        end
        if α = ♦β then
            AddClause(P, ♦β ← β);
            AddClause(P, ♦β ← ●♦β);
        end
        if α = β𝕊γ then
            AddClause(P, β𝕊γ ← γ);
            AddClause(P, β𝕊γ ← β, ●(β𝕊γ));
        end
        if α = β𝕫γ then
            AddClause(P, β𝕫γ ← γ);
            AddClause(P, β𝕫γ ← β, ●(β𝕫γ));
        end
        if α = □β then
            AddClause(P, β ← □β);
            AddClause(P, □β ← ●□β);
        end
        if α = ◊β then
            AddClause(P, ◊β ← ●◊β, ~ ●β);
        end
        if α = β𝕌γ then
            AddClause(P, β𝕌γ ← ●(β𝕌γ), ~ ●(γ));
            AddClause(P, γ ← β𝕌γ, ~ β);
        end
        if α = β𝕎γ then
            AddClause(P, β𝕎γ ← ●(β𝕎γ), ~ ●(γ));
            AddClause(P, β ← β𝕎γ, ~ γ);
        end
    end
end
```

Figure 3.10: Logic treatment of different temporal operators

represents the information about the formula α at time point $t-1$, so the clause $\beta \mathbb{S} \gamma \leftarrow \beta, \bullet(\beta \mathbb{S} \gamma)$ represents properly the second option of the formal definition of $\mathbb{S}$. The remaining of the proof is as follows:

($\rightarrow$) Assuming that $\mathcal{T}_{\mathcal{P}}(I^t_{\mathcal{P}})(\alpha)$ is true, we have two possibilities. If $\bullet\mathcal{T}_{\mathcal{P}}(I^t_{\mathcal{P}})(\alpha)$ is true, then the inserted clauses do not change it, so $\bullet\mathcal{T}_{\mathcal{P}_1}(I^t_{\mathcal{P}})(\alpha)$ will also be true. Otherwise, the positive interpretation of α is given by the semantic rule of a temporal operator, and then a clause representing this operator will be inserted by the algorithm, and the interpretation of the conjunction of the literals in the body of this clause will be true, therefore $\bullet\mathcal{T}_{\mathcal{P}_1}(I^t_{\mathcal{P}})(\alpha)$ will also be true.

($\leftarrow$) If $\mathcal{T}_{\mathcal{P}}(I^t_{\mathcal{P}})(\alpha)$ is false, then α is not positively interpreted due to the semantics of any temporal operator. Therefore, none of the clauses inserted by the algorithm will change the interpretation of α, and $\bullet\mathcal{T}_{\mathcal{P}_1}(I^t_{\mathcal{P}})(\alpha)$ will also be false. □

Theorem 21 *Considering a temporal program $\mathcal{P}$, and a temporal program $\mathcal{P}_1$ generated through the application of the algorithm in Figure 3.10 over $\mathcal{P}$, we have that the following statement is true: If $\mathcal{P}_1$ is acyclic, then the recurrent neural network $\mathcal{N}$ = $\bullet$-based_Translation($\mathcal{P}_1$) computes, in $v_{\mathcal{P}_1} - 1$ executions, the fixed point of $\mathcal{T}_{\mathcal{P}}$ of $\mathcal{P}$ for each time point t.*

Proof: The proof follows from lemmas 9, 17, 19 and 20. □

3.3.4 Comparing the different modal approaches

Both SCTL and CTLK consider the modal approach to temporal representation, but there are important differences between the systems. First, CTLK and SCTL present major differences regarding the range of temporal operators that can be represented. However, some aspects are important to be considered regarding such representation. At first, CTLK is based on the representation of the $\bigcirc$ operator, and SCTL also defines the representation of the $\bullet$ operator. Since such operators are complementary,

adapting CTLK to a direct representation of both operators would be straightforward. In our SCTL translation of the example in the next Chapters, we can see how straightforward this kind of adaptation is, by using only the previous time operator: we consider that the information of output neurons representing $\bigcirc Q_2$ and $\bigcirc Q_3$ are recurrently propagated to the atoms Q_2 and Q_3, keeping the same semantics as if the network had been translated through the original definition. Also, the full SCTL algorithm performs the translation of the remaining operators through the insertion of clauses in the logic program. Since this operation is done only in the level of the logic program, it could also be applied to CTLK. Therefore, through simple adaptations, both systems would be able to operate with the same range of temporal operators.

However, a significant, relevant difference between the approaches is actually the structure used to represent each time point and the temporal relations among then. CTLK considers a distributed representation of time, i.e. the computation of different time points is represented by different neural networks, that may run in parallel. The clear advantage of this approach is the possibility to represent information (clauses) that are specific to a time point. In our example, again, we clearly see an example of this situation, with every clause used in only one network. A solution to represent this kind of information in SCTL is through the use of an atom to represent each time point. In the specific case of our example, the information about the minimum number of agents known as muddy (Q_i) is directly related with the current round of the game (time point), and therefore it was used in this sense. This will be made more clear in the next chapters.

On the other hand, SCTL is based on a serial approach, with a unique network operating during all the time points, and following the time flow for the computation at each time point. This allows a reduction of the size (number of neurons) of the network, due to the use of the same neurons to represent a clause (or set of clauses) that refers to different time points. For a learning perspective, this use of the same neurons

presents also the advantage of learning some information that is independent of the considered time point, leading to improved conditions regarding generalization. In Chapter 5 we present an extensive case study, where we adapt the example in [33] to compare CTLK and SCTL regarding both representation and learning perspectives.

In this chapter, we have discussed the CILP strategy to represent propositional logic programs into neural networks. We have also presented an extended translation which allows temporal resources of connectionist networks (delay units and recurrent connections) to represent the modal operators used in a linear time temporal logic. We also have compared our approach with other existing works that aim at performing the same tasks, but that clearly use a different structure for knowledge representation. In the next chapter, we will explain how temporal learning can be performed in the proposed neural-symbolic networks, and we will also discuss extensions and constraints to the learning performance in order to allow the incorporation of new sources of information.

Chapter 4

Learning in SCTL

So far, we have been focused on the representation of temporal knowledge, as well as the reasoning associated with the language and its semantic aspects. However, as important as the representation of knowledge is the acquisition of such knowledge. This capacity of acquiring as well as adapting knowledge is a crucial aspect in the development of robust intelligent systems.

The work of Michalsky [72] defines machine learning as the process of automatically acquiring or adapting a knowledge base, considering information that can come from different sources, such as: direct implementation, instruction, deduction, analogy, examples and discovery. While the use of examples in learning is strongly investigated by connectionist systems [51], the integration of different sources of information is a much broader domain.

In this chapter, we will explain the numerical processes involved in the learning and generalization from examples in temporal networks. Moreover, we consider how the system can acquire information from different sources, in such a way that abstract descriptions of constraints in the problem domain can be incorporated during learning. This composite process of knowledge acquisition is not only innovative

within neural-network learning, but allows a better applicability of our framework in symbolic domains, as detailed in what follows.

4.1 Temporal extensions of backpropagation

When modeling cognitive behaviour, time is a fundamental aspect to be considered in the task of learning. Under the biological reference of the animal brain, learning is a gradual process of consolidating a knowledge base, and therefore time is a crucial aspect that pervades all the process [59]. However, the task of learning a knowledgebase of temporal sequences and delayed influences between events is much more specific. Several tools, which are capable of learning static knowledge, have been adapted to the case of temporal domains, as in the case of adaptations of ILP focusing on learning event calculus programs [75].

For connectionist systems, we have seen in Chapter 2 that there are several models capable of dealing with temporal knowledge. These systems often require specific resources for modeling, explicitly or implicitly, memory of recent sequences of information, in order to capture the temporal aspects of the knowledge to be learned. The learning algorithms in these cases, then, need to be able to deal with such resources [51].

In the specific case of extending feedforward networks, adaptations have been proposed to the traditional backpropagation algorithm in order to deal with temporal aspects. In the work of Waibel [108], an algorithm called Temporal Backpropagation is used to learn TDNNs - Time Delayed Neural Networks. These networks contain delay units distributed through the network, and the learning algorithm makes use of an array to keep a history of the previous values of δ for each neuron, in order to combine the input and the error estimation when calculating the weight correction.

When delay units are only applied before the input of the network, there is no delay applied to the error information, and this treatment is not necessary.

For recurrent neural networks, two algorithms deserve attention: the Backpropagation Through Time (BPTT) [109] and the Real Time Recurrent Learning (RTRL) [111]. The BPTT is based on the idea of unfolding the recurrent network into a feedforward one, in such a way that the traditional backpropagation algorithm can be applied. Since the connections of the original network are duplicated in the unfolded network, all the error information applied to the different copies of the same connection is considered after a backpropagation step.

In figure 4.1 we illustrate the process of unfolding a NARX network. The upper part of the figure (a) shows an original NARX network, and the part below shows how the unfolded network would be, if two time points (t and $t-1$) were considered. Notice that, for the sake of simplicity, we did not illustrate the duplication of the source (input) nodes. The number of time points to consider depends on the duration of the time sequences to be learned, and the desired accuracy.

While the BPTT algorithm usually requires an entire sequence to be presented to the network before the correction of weights can take place, the RTRL proposes an alternative whereby this correction happens through the execution of the network. In the RTRL algorithm, a unique layer of computational nodes is used, with a subset of the nodes in this layer considered as the output. The error correction is defined by making use of a full matrix representing the different configurations of the weights, outputs and error estimates through time, as detailed in [111].

4.1.1 The SCTL learning algorithm

Taking as reference the different algorithms used for temporal learning in neural networks, here we propose a new algorithm for learning in the specific networks used

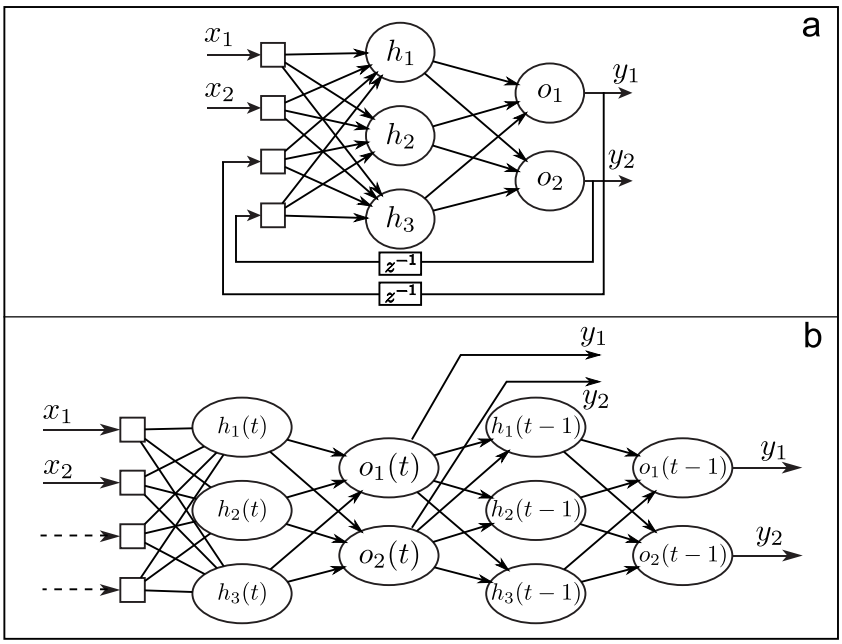


Figure 4.1: Illustration of an unfolded NARX network

by CILP and SCTL. Our proposed algorithm seeks to satisfy a number of requirements for symbolic computation (our target application), given the structure of the networks.

- Localized processing: one of the main features of the CILP translation is the localized nature of the representation: each neuron is responsible for representing a logical operation on the original knowledge description. In the learning algorithm, keeping the error estimation and weight correction associated with the individual neurons allows isolating certain parts of the network for fine-tuning the learning process.

- Handling different kinds of recurrent links: recall that SCTL networks contain two different kinds of recurrent links. When defining the learning algorithm, this needs to be taken into consideration so that a proper supervised learning from examples can take place as expected.

- Adequacy to the representation of symbolic knowledge: For numerical learning tasks, usually associated with neural networks, neurons operating in the middle region of the activation function, with higher derivatives, might be preferred in order to achieve a quicker convergence. However, the CILP algorithm sets the weights of the neurons to operate away from this middle region, which might cause problems when the network is trying to explore new solutions when learning from examples. The learning algorithm must then cope with this issue, allowing values to be in the middle region for hidden neurons, but enforcing the input and output values to be near the ends.

- Real time learning: Some of the applications we will describe in this work assume that the observed knowledge needs to be incorporated into the network on-the-fly, in such a way that the changes to the network need to be promptly realized and made available.

- Balance between efficiency and learning capacity: As we mentioned for BPTT, the unfolding of the network is directly associated with the accuracy of the learning algorithm. On the other hand, increasing the size of the unfolded network will also increase the computational complexity of the system. In our applications, efficiency is crucial, particularly in the case of real-time learning when the network is required to be integrated with other tools as part of an iterative process.

In the algorithm of Figure 4.2, we depict the entire proposed learning process for an SCTL network, from when a new vector is provided as input (at a time point t), through to the process of error backpropagation including the recurrent links, and subsequent adjustment of weights.

SCTL network $\mathcal{N}$'s execution
  foreach *timepoint t* **do**
    foreach $in_\alpha \in InputNeurons(\mathcal{N})$ **do**
      if $isInput(\alpha)$ **then** $x(in_\alpha) = netInput(\alpha)$;
      if $isCILPLink(\alpha)$ **then** $x(in_\alpha) = -1$;
      if $isSCTLLink(\alpha)$ **then** $x_(in_\alpha) = y(t-1)(out_\alpha)$;
    end
    $Feedforward(\mathcal{N})$;
    repeat
      foreach $in_\alpha \in InputNeurons(\mathcal{N})$ **do**
        if $isInput(\alpha)$ **then** $x(in_\alpha) = netInput(\alpha)$;
        if $isCILPLink(\alpha)$ **then** $x(in_\alpha) = y(out_\alpha)$;
        if $isSCTLLink(\alpha)$ **then** $x_(in_\alpha) = y(t-1)(out_\alpha)$;
      end
      $Feedforward(\mathcal{N})$;
    until $(\nu_\mathcal{P} - 1)$ *times*;
    foreach $out_\alpha \in OutputNeurons(\mathcal{N})$ **do**
      if $isOutput(\alpha)$ **then** $Error(O) = desired(\alpha) - y(out_\alpha)$;
      if $isCILPLink(\alpha)$ **then** $Error(O) = 0$;
      if $isSCTLLink(\alpha)$ **then** $Error(O) = \delta(t-1)(in_\alpha)$;
    end
    $Backprop(\mathcal{N})$;
    repeat
      foreach $out_\alpha \in OutputNeurons(\mathcal{N})$ **do**
        if $isOutput(\alpha)$ **then** $Error(O) = desired(\alpha) - y(out_\alpha)$;
        if $isCILPLink(\alpha)$ **then** $Error(O) = 0$;
        if $isSCTLLink(\alpha)$ **then** $Error(O) = \delta(t-1)(in_\alpha)$;
      end
      $Backprop(\mathcal{N})$;
    until $(\nu_\mathcal{P} - 1)$ *times*;
  end
end

Figure 4.2: Illustration of the propagation of values in a SCTL network during a timepoint

4.2 Integrating different information sources

So far, we have seen how to integrate two sources of information: an initial symbolic description, given as a temporal logic program, and a set of examples, given as a temporal sequence. This allows our system to generalize a knowledge base and, hopefully, learn about the influence of time in a given problem domain.

Regarding the initial symbolic knowledge, the correctness of SCTL translation algorithm ensures a semantic equivalence between the logic program and the translated neural network. Given this equivalence, we can refer to both representations interchangeably as a temporal model. Moreover, for the purposes of learning temporal models, we will consider that two kinds of variables can be defined: *input variables*, whose values are informed to the model by an external entity, and *state variables*, which can have their values modified by the model through the computation of logical consequences given the input values and previous information about other states.

Definition 22 *A **temporal model**, when given by a logic program $\mathcal{P}$, is defined by the tuple $\mathcal{P} = \left\{St^{\mathcal{P}}, In^{\mathcal{P}}, C^{\mathcal{P}}\right\}$, where $St^{\mathcal{P}}$ is the set of state variables α, $In^{\mathcal{P}}$ is the set of input variables β, and $C^{\mathcal{P}}$ is a set of clauses in the form $\bigcirc\alpha \leftarrow \alpha_1, ..., \alpha_n, \beta_1, ..., \beta_m$.*

In the above temporal model, a training example can be defined as a sequence of input values and desired output values. The desired outputs are a subset of the state variables. Each observed example has values assigned to all the input variables and to a subset of the state variables. Thus, let us redefine training examples more specifically in terms of input and state variables.

Definition 23 *An **observed example** E at timepoint t is a tuple $E_t = \langle I_t, D_{t+1}\rangle$, where the mapping $I_t : In \rightarrow \{-1, 1\}$ makes an assignment of values to the input variables and $D_{t+1} : St \rightarrow \{-1, 0, 1\}$ makes an assignment of the desired values for the state*

variables at the next timepoint, where 0 *denotes that no information is available about the corresponding variable.*

Notice that the network is not supposed to have its weights changed in the absence of information, such as in the case of non-observable state variables. Hence, the error on the output neurons representing those non-observable variables is defined as null. In terms of learning in neural networks, this can be implemented as follows: for each output neuron where no information is given as the desired output for back-propagation, consider the desired value to be equal to the actual activation value for the neuron obtained in the feedforward step.

In order to illustrate the representation of temporal models and training examples, consider a monitor of a resource that is supposed to allocate such resource to different processes. Each process communicates with this monitor through a signal to request the resource (*Req*) and a signal to release it (*Rel*). These signals are input variables to the monitor, which also has a state variable for each process, to denote that the resource is allocated to the process. Consider now a simple temporal model, describing how such monitor should work for one process A:

$In^{\mathcal{P}} =$	$\{Req_A, Rel_A\}$
$St^{\mathcal{P}} =$	$\{A\}$
$Cl^{\mathcal{P}} =$	$\{A \leftarrow Req_A$
	$A \leftarrow \bullet A, \sim Rel_A$

Table 4.1: Example of a temporal model

Let us now extend this example to deal with two processes. However, instead of having rules stating how an agent should deal with a second process, let us use our system to learn from observing how an existing monitor handles this. In this case, we extend our model $\mathcal{P}$ to include the variables that are necessary to handle the second process B, i.e., $In^{\mathcal{P}} = \{Req_A, Rel_A, Req_B, Rel_B\}$ and $St_{\mathcal{P}} = \{A, B\}$. We assume that the only observable output of the existing monitor is the variable B, which indicates that

the resource is allocated to the second process - notice that the columns regarding the variable A are filled with zeros below. The idea is that the examples derived from observation of an existing system can be used to train a network, extending it to deal with two or more processes. In this case, the network would become a partial model of the existing system.

Timepoint	Req_A	Rel_A	Req_B	Rel_B	A	B
1	1	-1	-1	-1	0	-1
2	-1	1	-1	-1	0	-1
3	-1	-1	1	-1	0	1
4	1	-1	-1	-1	0	1
4	-1	-1	-1	1	0	-1

Table 4.2: Input and output observations from an agent

4.2.1 Constraining the learning process

One of the main issues of the integration between an initial knowledge and some extra data regards the relative influence of the two knowledge sources on the result. In ILP, for instance, the set of examples is usually used to add to the existing knowledge, i.e. the initial knowledge, in the form of clauses, remains the same while new clauses are inserted to explain the examples [76].

When negation is allowed in logic programs, we have a situation of non-monotonic reasoning. Without negation, monotonicity holds, i.e. if from a set of clauses S one can infer variable A ($S \models A$), the insertion of extra clauses will not affect the evaluation of A ($S \cup S' \models A$). However, the same is not true for clauses with negation by failure. If S contains negation, it is possible that $S...A$ and yet we cannot ensure $S \cup S' \models A$. When refining a knowledge base expressed as logic programs with negation, the above needs to be taken into consideration. We will handle this by assuming that if an example contradicts the inference of a positive variable, the example will not affect the result of the learning process.

Differently from ILP, in systems like CILP and SCTL, the original rules are translated into a numerical representation, which is subject to the weights' adaptation using backpropagation. In the case of SCTL, the network is normally assumed deterministic. The examples that are used to adapt the existing knowledge provide new state transitions to this deterministic network, and can cause considerable changes to the overall network behaviour. Therefore, we say that the learning process in SCTL is more of a revision process, rather than refinement, where the information provided by the examples can change the existing knowledge and has priority over it, creating a new set of rules.

The above aspect of SCTL, whilst very important when the original knowledge is incorrect, may cause undesirable changes to the original knowledge when it is largely correct. To prevent this situation, we propose to add certain constraints into the learning process, in order to weaken the effect of certain examples when they may cause undesirable changes. These constraints are not encoded directly into the weights of the network, but are part of the examples in the learning process, as follows.

Definition 24 *A **constraint** $\mathcal{X}$ is defined by a tuple $\mathcal{X} = \{S_0, I, S_n\}$, where S_0 is the initial state condition, S_n is the final state condition, and I is a sequence of input conditions $I_0, ..., I_{n-1}$ with $S_k : St \rightarrow \{-1, 0, 1\}$ and $I_k : In \rightarrow \{-1, 0, 1\}$.*

Definition 25 *A **value assignment** to state variables S_t is said to correspond to a state condition S_k if for every $\alpha \in St$, $S_t(\alpha) = S_k(\alpha)$ or $S_k(\alpha) = 0$. The definition is analogous for input conditions.*

The constraint $\mathcal{X}$ states that if the current state of the system at timepoint t corresponds to S_0, the input applied to the system corresponds to I_0, and thereafter each input applied to the system at timepoint $t + k$ corresponds to I_k until k is equal to a predefined size n, then the new state of the system must correspond to S_n. When

a value of *zero* is assigned by a state (or input) condition to a variable $\alpha \in St$ (or $\beta \in In$), then no constraint is imposed on the value of α (or β).

This definition of constraints is not only applicable for weakening the effect that individual examples have in the learned model, but also allows the reinforcement of the initial knowledge that happens to satisfy the constraints. This is very useful as a flexible way of incorporating knowledge during learning: while examples simply define the inputs and outputs for each time point during an observed interval, constraints allow us to represent more general knowledge, including long term relations involving assignments to input and state variables.

Let us see in more detail how the use of the above constraints is implemented in our system. Consider the case where examples and constraints are given simultaneously. We keep a record of *active constraints* (initially empty) as well as an index k for each active constraint. At each timepoint t, if the current state corresponds to the initial state condition S_0 of a constraint $\mathcal{X}$ then $\mathcal{X}$ becomes active and k is set to 0. When an input is applied to the network, the system verifies if the input corresponds to the current position In_k of each active constraint $\mathcal{X}$, eliminating those not satisfying this condition. When a constraint reaches the end of the input sequence, the assignments to the state variables given by the final state condition S_n are used to define the desired output values as part of the learning process. The full algorithm for this kind of learning is shown at the end of this section.

Let us illustrate, with an example, how constraints are used. Suppose we want to add some constraints to the monitor example used before. In that example, we have not given to the network any information about how to avoid two processes accessing the resource at the same time. In Table 4.3, we illustrate two simple constraints dealing with the case where this conflict between processes appears. The table also shows how the active constraints are stored and used to define the desired output

values used for training.

Constraints		
	$\mathcal{X}1$	$\mathcal{X}2$
S_0	$\neg B$	$\neg A$
I_0	$ReqA$	$ReqB$
I_1	$ReqB, \neg RelA$	$ReqA, \neg RelB$
S_2	$\neg B$	$\neg A$

Exec	State		Inputs				Active	Desired
Count	A	B	ReqA	RelA	ReqB	RelB	Const..	Output
1	−1	−1	1	−1	−1	−1	$\{\mathcal{X}_1(1)\}$	$\emptyset$
2	1	−1	−1	1	−1	−1	$\emptyset$	$\emptyset$
3	−1	−1	−1	−1	1	−1	$\{\mathcal{X}_2(1)\}$	$\emptyset$
4	−1	1	1	−1	−1	−1	$\{\mathcal{X}_2(2)\}$	$\{\neg A\}$

Table 4.3: Definition of target output values (right) according to specified constrains (left)

The system also deals with the case where examples are not available. In this case, just the constraints must be used. As before, they are used to define the desired output values of the network, but now also to select the inputs to be applied at each time step. In this case, for each timepoint, a random constraint in the active list is selected, and the current input condition associated with such constraint is used to define the input to be applied - i.e. if a selected random constraint $\mathcal{X}$ has the current input condition assigning a specific value to a variable β (e.g. $In_k^{\mathcal{X}}(\beta) = 1$), the input value of β to be applied to the network should be the same (in the example, in_β should receive value 1 as input at that time point). Whenever no value is assigned to an input at that time point (i.e. $In_k^{\mathcal{X}}(\beta) = 0$), either 1 or -1 can be applied to the neuron, and this choice is made randomly.

Finally, in order to allow some exploration, the system also allows the selection of a random input value independently of the constraints. This is defined through the use of a parameter p, which indicates the probability of the system choosing a different input than that prescribed by the set of active constraints. When this happens, the input values for all the variables are defined randomly within the set containing 1 and -1. Figure 4.3, at the end of this section, gives the algorithm for the entire process.

4.2.2 Integrating and treating conflicts

When the system sets desired output values as done above, two special cases deserve closer attention. The first is when no information is given about the desired value of a state variable α. In this case, our first option is to assume that the value obtained by the network is the desired value, i.e. $D_{t+1} = S_{t+1}$. In this way, the error will be null for that neuron and it will not affect the weight correction in the network (since there is no information on the associated state variable).

Treating the lack of information in the output as a null error seeks to keep, rather than change, the existing knowledge in the network. Over time, however, this approach may lead to output values near zero within the -1,1 spectrum. This reduces the confidence in the system (a measure of confidence will be defined in the next chapter) and is undesirable from the point of view of the underlying logic in the network, where 1 represents true and -1 false. The system caters, therefore, for other ways of setting the desired output values in the network when information is not provided, as follows:

- Reinforce current output: under this approach, we also attempt to preserve the existing knowledge of the network, but we assume that positive values denote true and negative values false. Thus, we make the desired value of an outputs +1 if the obtained value is greater than 0, and -1 if it is less than 0.

- Reinforce previous state: this approach assumes that the system should be taught to keep its current state unless stated otherwise. Hence, the desired value for an output neuron representing a state variable α is set as the same value given to α at the previous timepoint.

- Fixed default value: the easiest way of defining the desired output value; if no information is given about the desired value of an output, it should be false by

default. Thus, the desired output is set to -1. This approach follows the concept of default negation that something is false if it cannot be proven true.

	Previous State	1	-1	1	-1	1
	Network output	0.1	0.8	-0.4	0.3	1
	Desired output	-1	0	0	0	1
1	Null Error	-1	0.8	-0.4	0.3	1
2	Reinforced Output	-1	1	-1	1	-1
3	Previous State	-1	-1	1	-1	1
4	False as default	-1	-1	-1	-1	1

Table 4.4: Example of the different treatments for missing information

The other important situation that deserves further attention is when there is a conflict between constraints (or between a constraint and an example). In this case, we do not assign any priorities; instead we add all the value assignments given by the constraints and by the examples into a variable *sum*, and take $D_{t+1} = 1$ if $sum > 0$, $D_{t+1} = -1$ if $sum < 0$ and $D_{t+1} = 0$ otherwise. Figure 4.3 contains the entire algorithm for performing learning with missing information and multiple sources of information in our system.

4.3 Extracting temporal knowledge

Extraction of knowledge from neural networks may be defined as the task of representing, in a symbolic, intelligible form, the knowledge learned by a network, which is coded in the form of the weights and the architecture of connections and neurons. This was considered an important challenge, mainly in the beginning of the 2000s, as the main way of allowing the application of connectionist systems in traditional symbolic domains, which required an explanation for the reasoning made by the intelligent system. Even though several interesting approaches were proposed [37, 94, 93], striking a perfect balance between accuracy and computational complexity is still an open issue, very dependent on the application domain.

Learning algorithm
- $t \leftarrow 0$ **while** *not finish* **do**
  - **foreach** $\mathcal{X} \in Constraints$ **do**
    - **if** $S_t^{\mathcal{N}}$ *is according to* $S_0^{\mathcal{X}}$ **then**
      - $k^{\mathcal{X}} \leftarrow 0$;
      - add $\mathcal{X}$ to *activeConstraints*
  - $E \leftarrow nextExample$;
  - **if** E *exists* **then**
    - **for** $1 \leq j \leq |In_{\mathcal{N}}|$ **do**
      - $I_t^{\mathcal{N}}(j) \leftarrow I^E(j)$
  - **else if** $random[0..1] \leq \rho$ **then**
    - **for** $1 \leq j \leq |In_{\mathcal{N}}|$ **do**
      - $I_t^{\mathcal{N}}(j) \leftarrow random\{-1, 1\}$
  - **else**
    - $\mathcal{X} \leftarrow selectRandomConstraint$;
    - **for** $1 \leq j \leq |Inp_{\mathcal{N}}|$ **do**
      - **if** $I_k^{\mathcal{X}}(j) = 0$ **then**
        - $I_t^{\mathcal{N}}(j) \leftarrow random\{-1, 1\}$
      - **else**
        - $I_t^{\mathcal{N}}(j) \leftarrow I_k^{\mathcal{X}}(j)$
  - **foreach** $\mathcal{X} \in activeConstraint$ **do**
    - **if** IN_t *is according to* $I_k^{\mathcal{X}}$ **then**
      - $k^{\mathcal{X}} \leftarrow k^{\mathcal{X}} + 1$;
    - **else**
      - remove $\mathcal{X}$ from *activeConstraint*
  - **for** $1 \leq i \leq |St_{\mathcal{N}}|$ **do**
    - **if** E *exists* **then**
      - $D_{t+1}^{\mathcal{N}} = D_t^E + 1$
    - **else**
      - $D_{t+1}^{\mathcal{N}} = 0$
  - **foreach** $\mathcal{X} \in activeConstraint$ **do**
    - **if** $k^{\mathcal{X}} = n^{\mathcal{X}}$ **then**
      - **for** $1 \leq i \leq |St_{\mathcal{N}}|$ **do**
        - $D_{t+1}^{\mathcal{N}} = D_{t+1}^{\mathcal{N}} + S_n^{\mathcal{X}}$
      - remove X from *activeConstraint*
  - **for** $1 \leq i \leq |St_{\mathcal{N}}|$ **do**
    - **if** $D_{t+1}^{\mathcal{N}} >= 1$ **then** $D_{t+1}^{\mathcal{N}} \leftarrow 1$
    - **else if** $D_{t+1}^{\mathcal{N}} <= -1$ **then** $D_{t+1}^{\mathcal{N}} \leftarrow -1$
    - **else** $D_{t+1}^{\mathcal{N}} \leftarrow S_{t+1}^{\mathcal{N}}$

end

Figure 4.3: Algorithm depicting the full process of learning from multiple sources of information

In an important survey of the area, Andrew et al [4] have considered different dimensions to classify the existing approaches to knowledge extraction. They include: (a) the expressive power of the extracted rules, (b) the translucency of the rule extraction technique w.r.t. the neural network, whether black-box or white-box, (c) the extent to which the underlying network uses specialized training regimes, (d) the quality of the extracted rules as measured in terms of rule readability, accuracy and fidelity to the network, and (e) the algorithmic complexity of the rule extraction technique.

Features (a) and (d) regard the set of extracted rules: while the expressive power of the extracted rules is related to the question of the languages that represent neural networks, the quality of the extracted rules is associated with the specific parameters of the knowledge extraction process and its soundness. Features (c) and (e) regard the algorithm used for extraction, respectively, to do with the range of neural architectures and training processes to which the algorithm is applicable, and the complexity of performing the extraction. The remaining feature (b) regards how much internal information from the network's structure is used by the extraction algorithm to build the symbolic representation of the knowledge. Two main categories of algorithms can be considered here: *pedagogical* approaches, where internal information about the network is not available, i.e. the algorithm must extract information given input and output patterns obtained from querying the network, and *decompositional* approaches, which can make use of all the values of synaptic weights and connection structures inside the network to accomplish their purpose.

Pedagogical approaches face two major (and related) challenges: The choice of a set of examples to be used, and the combinatorial explosion of the number of input-output patterns necessary to have a good sample of the domain. One of the proposed solutions consists in using the same examples as used for training (when available). Some argue, however, that if the same data is used for training and extraction, an-

other method like, for example, a decision tree could be used directly on the original data, making the neural network dispensable. It is the generalization obtained by the network training that the extraction algorithms should try to explain.

Decompositional techniques also have considerable challenges. One of them regards the complex nature of a connectionist architecture: the behaviour of the whole system cannot be described as the sum of its parts. Therefore, the extracted knowledge can be incomplete, or even incorrect, as illustrated by an example in [37], which we adapt below for the case of bipolar networks. In Figure 4.4, we show a simple feedforward network with two inputs a and b, two hidden neurons n_1 and n_2 and an output neuron x, where the activation function of the hidden and output neurons is the same as in the CILP networks ($\phi = \frac{2}{1+e^{-\beta x}} - 1, \beta = 0$), and the bias is zero for all the neurons.

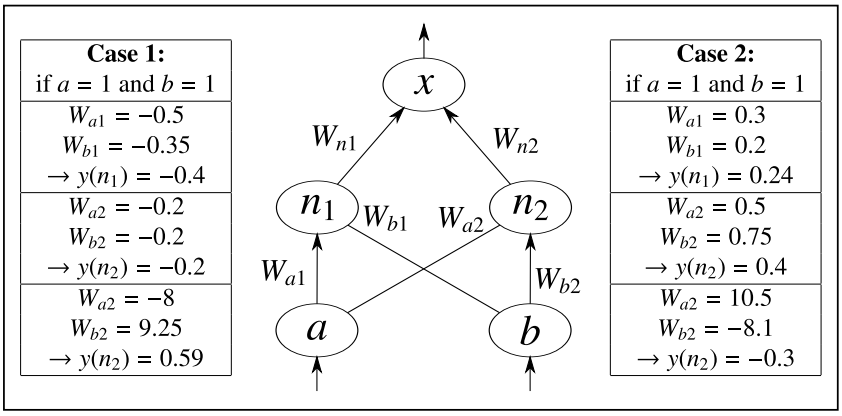


Figure 4.4: Example of incompleteness and unsoundness of decompositional methods

For analysing the example, let's consider a simple, decomposition analysis of the network, by analysing the behaviour of each neuron individually. This analysis consists in applying either -1 or 1 to the inputs of the network, and verifying the value obtained on the output. If this value is positive, the output should be interpreted as true, otherwise as false.

For the case 1, we can see an example of incomplete extraction of knowledge. Assigning positive values to the input of the network will result in a false interpretation for hidden neurons n_1 and n_2. However, when analysing individually the neuron x, if negative input -1 is applied to both its inputs, it will also not be positively activated ($y(x) = -0.55$). Therefore, the composition of these results would not be capable of inferring the actual mapping performed by the network, which is $ab \rightarrow x$.

We can even get an unsound result from a similar decompositional extraction. Notice that, in case 2, the combination of positive inputs would activate both n_1 and n_2, and that an analysis of x with both n_1 and n_2 being positive (assigned to 1) would lead to the activation of x (with an output of 0.83). Therefore, the decompositional extraction would conclude the undesired rule $ab \rightarrow x$.

Another criticism of decompositional approaches consists in the lack of generality: a technique built for extracting knowledge from a specific type of neural network will probably not be applicable to other networks (e.g. a system that is capable of extracting knowledge from a multilayer perceptron will probably not be applicable to radial-basis function networks).

Below, we propose a simple pedagogical extraction method for SCTL networks. We focus our discussion on extending existing pedagogical techniques to allow their application in the temporal case, and on the use of the background knowledge to propose heuristics to improve the extraction process, in particular the selection of examples for pedagogical extraction. Because of the generality issues and the above serious criticisms regarding soundness, we do not consider decompositional methods further.

4.3.1 A simple pedagogical approach - State diagrams

We will consider a pedagogical approach to represent the temporal knowledge learned by SCTL networks in a symbolic temporal language. For pedagogical extraction, one needs to generate a set of examples (input vectors) to be applied to the network. This set must be large enough to be a good representative of the domain, but not so large that the extraction process become computationally intractable. Different approaches trying to strike this balance can be found in the literature. In [37], for example, a partial ordering is imposed on the set of input vectors according to the structure of the network so that certain input vectors become preferred over certain others for querying the network and rule generation. Outside the area of symbolic extraction, other approaches to the automatic generation of examples can be found, like in the case of the generation of negative examples in order to complement positive examples as part of a learning process [48].

We use a simple pedagogical approach that turns out to be sufficient for our purposes. We consider (i) performing a random and exhaustive assignment of the possible input vectors, when possible, and (ii) splitting the dataset into two groups, one for learning and validation and one for extraction, otherwise.

First, we will focus on networks where input information is applied directly to the neurons (without delay units on the input) and the temporal recurrent links are delayed only by one time point. With these restrictions, at each time point we can associate the input vector I applied to the network to the temporal formulas represented by such input neurons. We then run the network once to obtain activation values for the output neurons and, through the recurrent connections, new values for some of the input neurons. Such input neurons that receive information from the output are known as context units. It is useful to distinguish input units (those associated with input vector I) and context units (the values of which define a new state given I).

Below, we extract symbolic knowledge from a recurrent network by creating a state transition diagram mapping the state of the context units to a new state given the input, according to the following definition.

Definition 26 *A **transition** $\mathcal{T}$ is a tuple $\{S_0, I, S_f, w, count\}$ containing a **source** state S_0 and a **target** state S_f given **input** I. w and count are auxiliary information representing a **weight** and the number of occurrences, respectively.*

For each time point, a new transition $\mathcal{T}$ is stored: I represents the input vector applied to the network, S_0 contains the values of the context units and S_f contains the values of the output units. We assign truth-value true (value 1) to positive values in S_f and false (value -1) otherwise, but we use the auxiliary weight w, calculated as a function of the absolute values obtained in the network's output, to calculate a confidence interval on the assignment of truth-values.

After we apply a set of examples, all the occurrences of transition $\mathcal{T}$ with the same S_0, I and S_f are grouped into a single transition $\mathcal{T}'$, where $w^{\mathcal{T}'}$ is the sum of the individual weights and $count^{\mathcal{T}'}$ is the number of transitions grouped. This information is then used to generate a transition diagram that will visually indicate the behaviour of the network.

As an example, consider a simple case where an input (*Inc*) is used to increment the value of a counter, an input (*Dec*) is used to decrement this value, and the output of the system identifies if the value is greater than zero. Assume that this counter is capable of counting from 0 to 2, and therefore a state variable is needed to record if the value is greater than 1. Figure 4.5 shows a network that represents this example and a set of executions used for the extraction of knowledge with their associated transitions.

In the left hand side of Figure 4.5, we illustrate the configuration of input and output neurons in a network representing this example. In order to perform the ex-

	Inputs		State		Outputs		
t	Inc	Dec	> 0	> 1	$\bigcirc(> 0)$	$\bigcirc(> 0)$	$\mathcal{T}$
1	-1	1	-1	-1	-1	-1	$S_0 = \emptyset, I = \{Inc\}, S_f = \emptyset$
2	1	-1	-1	-1	1	-1	$S_0 = \emptyset, I = \{Inc\}, S_f = \{\bigcirc(> 0)\}$
3	-1	-1	1	-1	1	-1	$S_0 = \{> 0\}, I = \{Inc\}, S_f = \{\bigcirc(> 0)\}$
4	1	-1	1	-1	1	1	$S_0 = \{> 0\}, I = \{Inc\}, S_f = \{\bigcirc(> 0), \bigcirc(> 1)\}$
5	-1	-1	1	1	1	1	$S_0 = \{> 0, > 1\}, I = \{Inc\}, S_f = \{\bigcirc(> 0), \bigcirc(> 1)\}$
6	1	-1	1	1	1	1	$S_0 = \{> 0, > 1\}, I = \{Inc\}, S_f = \{\bigcirc(> 0), \bigcirc(> 1)\}$
7	-1	1	1	1	1	-1	$S_0 = \{> 0, > 1\}, I = \{Inc\}, S_f = \{\bigcirc(> 0)\}$
8	-1	-1	1	-1	1	-1	$S_0 = \{> 0\}, I = \{Inc\}, S_f = \{\bigcirc(> 0)\}$
9	-1	1	1	-1	-1	-1	$S_0 = \{> 0\}, I = \{Inc\}, S_f = \emptyset$
10	-1	-1	-1	-1	-1	-1	$S_0 = \emptyset, I = \{Inc\}, S_f = \emptyset$

Table 4.5: Transitions extracted from the example

traction, a set of executions is used as shown in Table 4.5. The table also shows each of the transitions $\mathcal{T}$ obtained. When grouping the transitions, those in time points $t = 3$ and $t = 8$ will be grouped into a transition $\mathcal{T}$, while the others remain the same - these transitions are illustrated in the diagram on the right side of Figure 4.5.

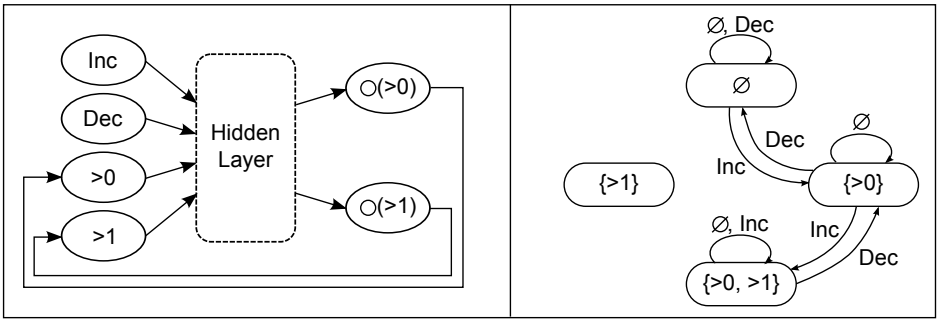


Figure 4.5: Example of extraction procedure

4.3.2 Extracting logic programs

Besides generating the above diagrams, we can also represent the extracted knowledge as a temporal logic program. In order to do so, SCTL tries to identify the most important group of transitions. The auxiliary weight and count parameters are used for this. Transitions below a certain threshold of occurrences or a desired confidence can be removed from the diagram. Each remaining transition $\mathcal{T}'$ is then rewritten as

a set of clauses - one clause for each output variable. The body of each clause will contain all the input and state variables either in positive or negative form according to the assignments of values to S_0 and I. The head of each clause will be one of the output variables: either $\bigcirc\alpha$, if $S_f(\alpha) = 1$, or $\bigcirc\neg\alpha$, if $S_f^{\mathcal{T}'}(\alpha) = -1$. To allow a better understanding of the rule set, the rules obtained from different transitions can also be simplified. We use Karnaugh maps [58] to do so, in such a way that complementary literals can be removed from the body of rules with otherwise the same body and the same head, e.g.: $a \leftarrow b, c$ and $a \leftarrow b, \sim c$ can be simplified into a single rule $a \leftarrow b$.

The above extraction process can be extended to the case with larger delays in the network. For delay units inserted in the input of the network, the expression associated with that input neuron will contain a $\bullet$ temporal operator for each delay unit in the network. Thus, if information about an expression α is associated with an input neuron through two delay units, the expression $\bullet^2\alpha$ will be used when defining the input I for each transition $\mathcal{T}$. The same process can be used for extra delays in a recurrent link, with a slight difference when the expression is of the form $\bigcirc\alpha$. In this case, each delay unit will be associated with one $\bigcirc$ operator, i.e. if an expression $\bigcirc\bigcirc\alpha$ is represented by an output neuron *out*, and this neuron is connected through two delay units to an input neuron *in*, the expression α should be added to S_0 when extracting a transition $\mathcal{T}$ for neuron *in*.

4.4 Discussion

In this chapter, we have introduced a series of structures and procedures allowing the learning, evolution or adaptation of temporal models under a neural-symbolic perspective. We have shown how the natural propensity of neural networks for empirical learning can be applied to temporal logic domains. Following this, we have extended the model to incorporate, in the learning stage, abstract information in the

form of temporal constraints, extending the possibilities of application of the system. We have also presented a simple pedagogical extraction method that allows us to extract state transition diagrams and associated temporal logic programs from trained recurrent networks.

In the next chapter, we will present a series of experiments, analyse the performance of our proposed learning strategies, as well as how they interact with other modules as part of a complete neural-symbolic framework for representation, learning and reasoning of temporal models. The results will provide important insights into the applicability of this framework to different temporal scenarios. Much work in the literature point out that learning and adaptation play a crucial role in the creation of robust intelligence. Either as a strategy to overcome the knowledge-acquisition bottleneck, or to make agent-based systems deal better with unforeseen scenarios, a capacity for building new knowledge bases (or adapting existing ones) from experience and observation is crucial for the execution of complex, intelligent tasks.

Chapter 5

Experimental Validation

In this chapter, we apply a number of testbeds to evaluate the techniques proposed so far. We focus on the performance of the techniques regarding representation and learning of temporal knowledge. Among the possible ways of analysing the behaviour of our neural-symbolic system, we will mainly make use of learning curves, cross-validation and behavioural analysis.

Learning curves are used to illustrate the convergence of neural networks. They show the evolution of the output error calculated in the network during the training phase. For this purpose, a 2-dimensional chart is used, where the X axis represents the number of training epochs and the Y axis represents the error measurement. An epoch is defined as the presentation of a number of examples to the network. In general, we will use the normalized RMSE (root mean squared error) as an error measurement. For each output A, we have the output o_A obtained after the execution of the network, and a desired value d_A, which defines the target value that should be learned by the network for that output. In order to calculate the normalized RMSE, we first normalize the output domain for both o_A and d_A to the interval between 0 and 1. After that, we calculate $(d_A - o_A)^2$ for every output A, and the error will be the

square root of the sum of these values, i.e. $RMSE = \sqrt{\sum_A (d_A - o_A)^2}$

The validation of a learning system is usually done in two phases. During the training phase, a set of examples is presented and the learning system is allowed to adapt to these examples as the above error is monitored. Then, in a test phase, another set of examples is presented to the system to provide an estimate of the behaviour of the learner on new examples, i.e. no adaptation of weights is performed during testing. In order to obtain a more robust measure of how the network generalizes to new examples, especially in the case when a small number of examples is available, one may use the cross-validation process. In order to perform an n-fold cross-validation, we split an original set of examples into n groups, and then perform n different validations with a set of $n - 1$ groups used for training purposes and the remaining group used for testing. The chosen test group is different for each of the n validations. An average test-set error is then taken. A specific case of cross-validation is called "leave-one-out", used mainly for small datasets, and which consists of a n-fold cross-validation for a data set with n examples.

In order to evaluate SCTL when a predefined set of examples is not available for the execution of cross-validation, we also perform some analyses of the network behaviour. This analysis can be performed by comparing the behaviour of the network with some existing temporal system, or by extracting a symbolic representation from the trained network.

The remainder of this chapter contains the results and analysis of a range of experiments, which illustrate the different aspects of the system. The experiments performed can be summarized as follows:

- **Temporal XOR**: Simple experiment used to illustrate the representation of the ● operator in the SCTL network, and also showing that the learning performance of SCTL is comparable to that of other architectures used for the same

temporal learning purposes (in this case, the Elman network [41]).

- **The Muddy Children Puzzle**: Testbed used by other authors to explain the use of the Connectionist Modal and Temporal Logic architecture [32, 33], is used here to compare SCTL with that other architecture on both representation and learning of temporal knowledge.

- **The Dining Philosophers**: Testbed used to illustrate the representation of different temporal operators, and to evaluate the learning performance in situations where an agent has to learn while embedded in an environment (online learning). This testbed is also used to exemplify the task of constrained learning.

5.1 The temporal XOR

To provide a better illustration of our approach to representation, we consider a simple example: the temporal XOR, proposed by Elman [41], a temporal version of a problem traditionally used in the learning and validation of non-linearly separable classes. We also present some preliminary learning results, seeking to provide empirical justification for the use of NARX networks.

The temporal XOR problem consists of a system that receives one bit as input, according to a predefined sequence, and should return as output a prediction of the next bit in the given sequence. The sequence of bits is defined by the XOR operation as follows: after the presentation of a sequence of two bits A and B as input, the next bit will be positive if A and B are different, and negative if they are the same. In Table 5.1, we illustrate a possible sequence of bits for the temporal XOR experiment (upper line), and the expected output for the desired predictive behaviour. The highlighted values are the actual result of an

t	1	2	3	4	5	6	7	8	9	10	11	12
in(α)	0	0	**0**	1	0	**1**	1	1	**0**	0	1	**1**
out(β)	0	**0**	1	0	**1**	1	1	**0**	0	1	**1**	?

Table 5.1: Temporal XOR sequence

It is important to notice that the system needs to be able to predict correctly the third bits of the sequence (i.e. produce the correct value in the output of the network at the second time point). Therefore, the evaluation of success of our experiment will consider these third bits in the sequence. For the purpose of supervised learning, we will consider two different scenarios: in the first scenario, the network will only perform a backpropagation step for the prediction of the third bit, not performing any change of weight in the other time steps. In a second scenario, we use all the outputs for learning with the random bits considered as noise in the task of learning the XOR operation.

In addition to the learning of the XOR above, we also evaluate the effect of translating background knowledge into the network's initial structure. To perform this evaluation, we compare different networks with the same architecture, but with different initial values for their weights. As usual, the weights of a network created by the SCTL translation are given by its translation algorithm. The weights of a network without background knowledge (BK) are initialized with small random values.

The program that represents the XOR sequence has two clauses: $\beta \leftarrow \alpha, \sim \bullet\alpha$ and $\beta \leftarrow\sim \alpha, \bullet\alpha$, where α is the atom representing the input and β represents the output. To represent atom $\bullet\alpha$, we use a delay unit directly in the input, as shown in Fig. 5.1(Net_1). Also, if we consider that α is required as output, we create a delayed recurrent link from an output neuron representing α to the input neuron representing $\bullet\alpha$. This also requires the insertion of a clause $\alpha \leftarrow \alpha'$ for propagating the information from input to output, as discussed in section 3.1. This architecture is shown in

Fig. 5.1(Net_2). Notice that the insertion of this new clause increases the value of $v_{\mathcal{P}}$ to 2. Connections with weight 0 were inserted in order to achieve a fully-connected network. Results are depicted in Figs. 5.3 and 5.2.

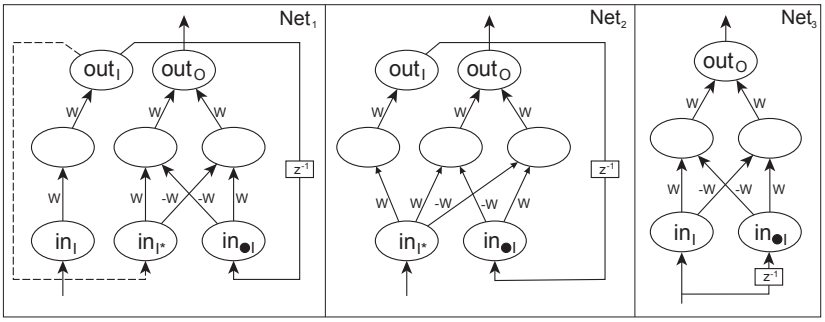


Figure 5.1: SCTL Networks used for the temporal XOR case

The results compare the networks without BK, one of the above clauses as BK and both clauses as BK. In the case of networks representing partial knowledge (one clause in this example), we translate that partial knowledge and insert as many extra hidden neurons as necessary to have the same network architecture for comparison. These neurons receive connections from every input neuron and are connected to every output neuron, with the same random initialization of weights as for the network without BK. This procedure will be used for all the experiments in this thesis.

In Figs. 5.3 and 5.2, the difference between Net_2A and Net_2B is that, in Net_2A the weights for the neurons corresponding to $\alpha \leftarrow \alpha*$ were randomly initialized. Network Net_3 presents a model similar to Net_2, but using $in_{\alpha*}$ to compute the XOR operator, and not in_α. Both architectures correctly compute the program.

For each architecture, three combinations of clauses were used to define the initial weights of the connections. In Fig. 5.3, the left column shows the result for a network generated without knowledge, i.e. with all weights randomly initialized. The middle bar chart represents a network that uses a neuron to represent $\beta \leftarrow\sim \alpha, \bullet\alpha$ and one with random weights. The right column represents the experiments with full

knowledge. We have run the same learning process for all the networks.

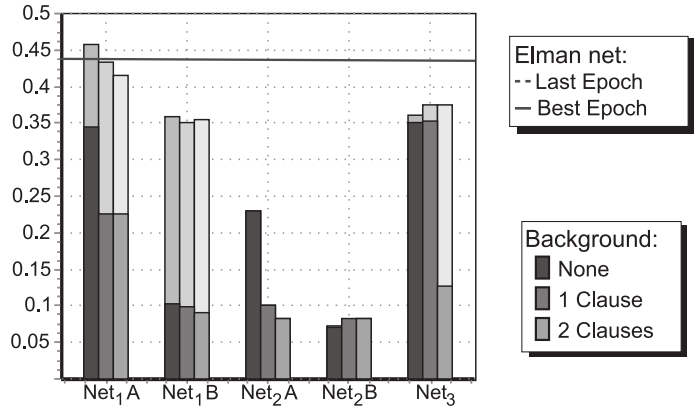


Figure 5.2: Error of the networks for the experiments without noise

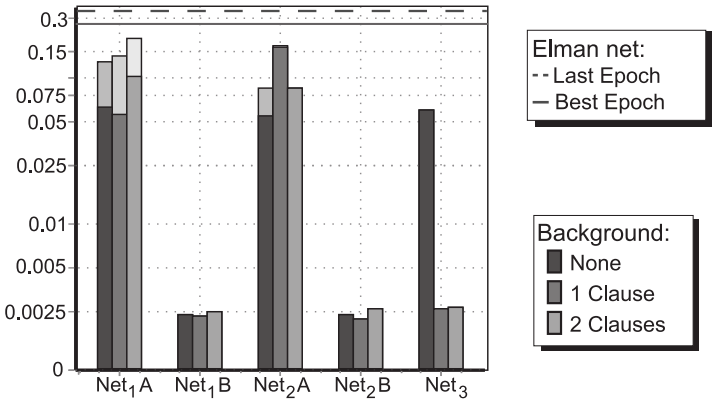


Figure 5.3: Error of the networks for the experiments with noise

Instead of a single presentation of data as done in [41], we performed a 10-fold cross-validation. This consists of splitting the dataset into 10 groups, performing 10 learning experiments, using 9 of the groups for training and a different group for testing the network. The learning process used backpropagation applied for 500 epochs with a learning rate of 0.3. All networks were submitted twice to the process of 10-fold cross-validation on a dataset containing 3000 patterns, i.e. each network was trained 20 times with 2700 patterns, and each trained network was tested on a set of 300 patterns. To calculate the error we used the RMSE (root mean square

error) averaged over 20 validations. We also repeated the learning process for Elman networks as described in [41], and the error for such experiment is depicted as the horizontal lines in the charts. The upper region of each column, represented in light gray, shows the error on the training set after 500 epochs, whilst the dark region shows the smallest error obtained during the 500 epochs.

It can be observed in the charts that the insertion of background knowledge had an effect on learning performance. It can be noted that the way in which the value of α at time point $t-1$ is propagated to the neuron representing $in \bullet I$ at time point t is very important. An adequate representation causes a considerable reduction of the error in the recognition of the temporal sequences, improving convergence and noise tolerance. Also, in certain cases, the insertion of clauses helped the performance of the network, especially in the case of the Net_1 architecture.

5.2 The Muddy Children Puzzle

Next we consider the *Muddy Children Puzzle*. This example has been used by several authors as a testbed to illustrate the representation of knowledge evolution in time. In particular, it has been used in [31] to illustrate a neural-symbolic system based on modal and temporal logics, called CTLK (Connectionist Temporal Logic of Knowledge). In this section, we analyse the differences between SCTL and CTLK, two approaches for the representation and learning of temporal knowledge. The *Muddy Children Puzzle* can be defined as follows:

A number n of (truthful and intelligent) children are playing in a garden. A certain number of children $k (k \leq n)$ have mud on their faces. Each child can see if the others are muddy but not himself or herself. Now, consider the following situation: a caretaker announces that at least one child is muddy ($k \geq 1$) and asks "Does any of you know if you have mud on your own face?". To help understand the puzzle,

let us consider the cases in which $k = 1$, $k = 2$, and $k = 3$. If $k = 1$ (only one child is muddy), the muddy child answers yes at the first instance since she cannot see any other muddy child. All the other children answer no at the first instance. If $k = 2$, suppose children 1 and 2 are muddy. In the first instance, all children can only answer no. This allows 1 to reason as follows: if 2 had said yes the first time, she would have been the only muddy child. Since 2 said no, she must be seeing someone else muddy; and since I cannot see anyone else muddy apart from 2, I myself must be muddy! Child 2 can reason analogously and also answers yes the second time. If $k = 3$, suppose children 1, 2, and 3 are muddy. Every children can only answer no the first two times. Again, this allows child 1 to reason as follows: if 2 or 3 had said yes the second time, they would have been the only two muddy children. Thus, there must be a third person with mud. Since I can see only 2 and 3 with mud, this third person must be me! Children 2 and 3 can reason analogously to conclude as well that yes, they are muddy.

In our work, the decision making process of each "intelligent and truthful" child is described in the form of temporal logic programs, as expressed in Table 5.2. The program on the left hand side of the table is written for CTLK, while the one on the right is written for SCTL. In both representations, Q_i as an auxiliary atom, related to the sequence of rounds, representing that the agent knows that at least i children are muddy. Atom K_iP_i represents that agent i knows that agent j is muddy. In both representations, we highlight a distinction between two groups of clauses: the top clauses are responsible for reasoning about whether a child is muddy: if the known number of muddy children is greater than the number of children that an agent can see with mud on their faces, then she must conclude that she herself is muddy. The other group of clauses, at the bottom, is responsible for the inference about how many children are actually muddy: if the agent knew that at least n children were muddy in the previous round, and no children discovered that they are muddy, then

the agent must know that there are at least $n+1$ muddy children at the current round. The architectures representing these programs, generated by the respective translation algorithms, are shown in Figure 5.4. For practical reasons, some simplifications were made, allowing the reduction of the number of neurons without compromising the represented semantics. Notice that the main difference between CTLK and SCTL is that CTLK requires an explicit labeling of time points, so that each clause holds at a time point and each time point is represented by a separate network. SCTL, on the other hand, uses a more standard recurrent network architecture with time points encoded implicitly in the delay units of the network.

$t_1 : K_1P_1 \leftarrow Q_1, K_1NP_2, K_1NP_3$ $t_2 : K_1P_1 \leftarrow Q_2, K_1NP_2$ $t_2 : K_1P_1 \leftarrow Q_2, K_1NP_3$ $t_3 : K_1P_1 \leftarrow Q_3$	$K_1P_1 \leftarrow Q_1, K_1NP_2, K_1NP_3$ $K_1P_1 \leftarrow Q_2, K_1NP_2$ $K_1P_1 \leftarrow Q_2, K_1NP_3$ $K_1P_1 \leftarrow Q_3$
$t_1 : \bigcirc Q_2 \leftarrow\sim K_1P_1, \sim K_2P_2, \sim K_3P_3$ $t_2 : \bigcirc Q_3 \leftarrow\sim K_1P_1, \sim K_2P_2, \sim K_3P_3$	$\bigcirc Q_2 \leftarrow\sim K_1P_1, \sim K_2P_2, \sim K_3P_3, Q_1$ $\bigcirc Q_3 \leftarrow\sim K_1P_1, \sim K_2P_2, \sim K_3P_3, Q_2$

Table 5.2: Logic program describing the reasoning of each agent

To evaluate the learning performance of the networks on the Muddy Children Puzzle, we used a learner agent based on a neural network [64]. Two different scenarios were used: in online learning, the agent was put into an environment, interacting with two agents which presented the "intelligent and truthful" behaviour described above. In this case, the environment was responsible for informing the agent what the desired output for learning was in each case. In the offline learning scenario, three "intelligent and truthful" agents played, and the learner observed the behaviour of one of those players, using the inputs and outputs from that agent as information for learning.

Four levels of background knowledge were provided to the learner, based on different combinations of the two subsets of rules shown in Table 5.2. Network Net_1 consists of the system without background knowledge, Net_2 contains only the knowl-

edge from the top of the set of rules, Net_3 contains the knowledge from the bottom part of the set of rules, and Net_4 is generated by the translation of the entire logic program (top and bottom part) in Table 5.2, one network for each set: SCTL and CTLK).

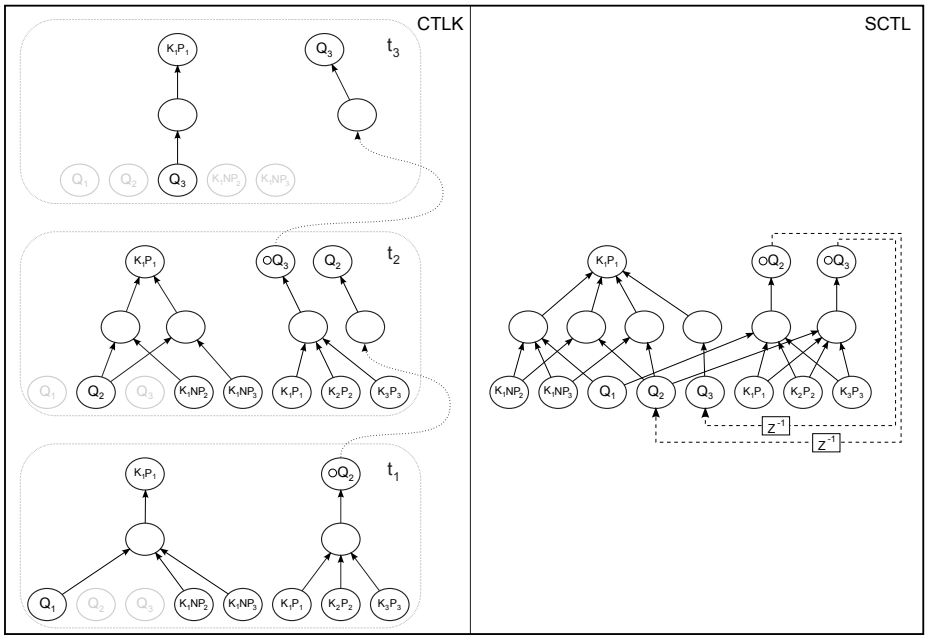


Figure 5.4: Architectures used on the Muddy Children example

Each learning process consisted of 50,000 games played by the agents. The configuration of each game was defined randomly, and it was played until one agent have answered correctly that she was muddy, in a maximum of three rounds. The network used for the learner consisted of the architectures described above for SCTL and CTLK, using backpropagation with a fixed learning rate of 0.3 and without momentum. To measure the evolution of performance, these 50,000 games were grouped into 250 sets of 200 consecutive games. The performances of the learner in online and offline scenarios did not present considerable difference, and therefore we chose to present the results of one execution in offline mode.

The first measurement taken was the evolution of error during training. This is

measured as the difference between the actual output of the network and the expected value. In Figure 5.5, we show the evolution of Root Mean Square Error (RMSE) for the different architectures. In the figure, each chart shows the average error at each round from left to right. The charts use a logarithmic scale in the vertical axis to improve visibility.

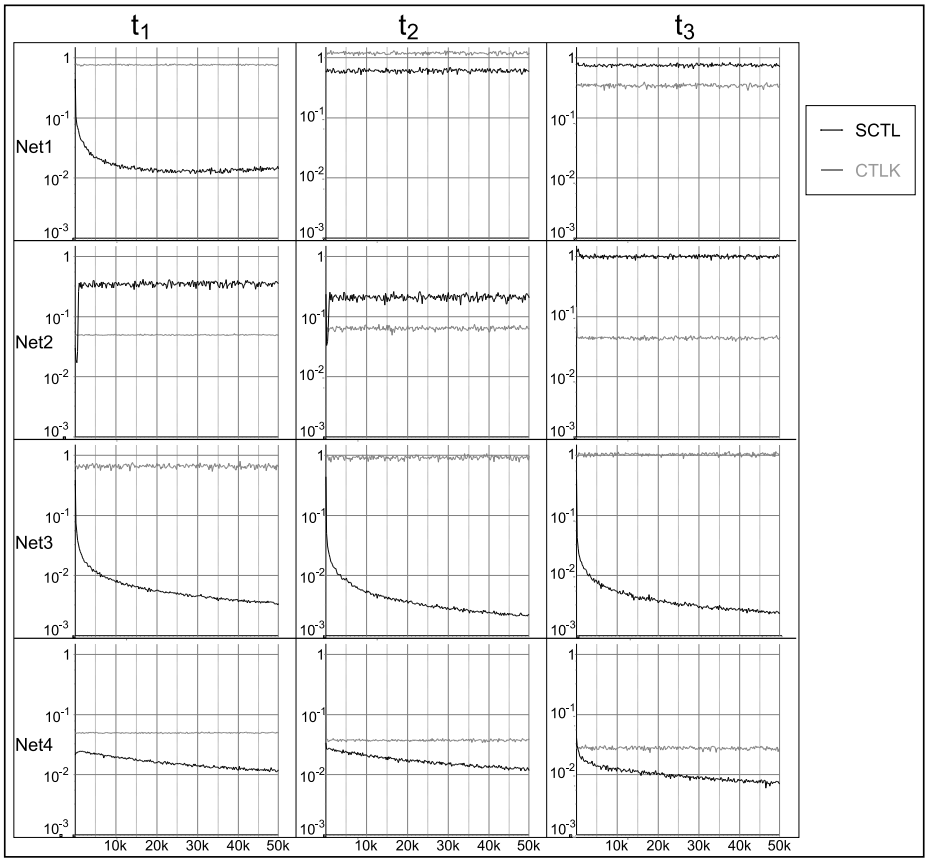


Figure 5.5: Evaluation of RMSE in both SCTL and CTLK learners on Muddy Children Puzzle. Each line refers to a different level of background knowledge considered, and each column regards a different round of the games.

In order to analyze the generalization performance, we also produced a confusion matrix for the last group of 200 games for the different networks. These matrices shown the percentage of items in each configuration of expected output (false at top row and true at bottom row) and obtained output (false at left column and true at the

right column). The results are shown in Table 5.3. We considered two interpretations for the true/false values in the network. In the first, values higher than zero were assumed to represent *true* and values lower than zero as representing *false*. In a more strict analysis of the obtained results, only the values in the intervals $[A_{min}, 1]$ and $[-1, -A_{min}]$ were considered as representing *true* and *false*, respectively, with A_{min} defined by the CILP translation algorithm, as usual.

CTLK	t_1	t_2	t_3
Net1:	87.5%(0%) 0% 12.5%(0%) 0%	0% 75.6% 0% 24.4%(19.3%)	0% 0% 0% 100%(0%)
Net2:	86% 0% 0% 14%	72.8% 0% 0% 27.8%	0% 0% 0% 100%
Net3:	90.5%(0%) 0% 0% 9.5(0%)%	62.8% 0% 37.2% 0%	0% 0% 100% 0%
Net4:	87.5% 0% 0% 12.5%	63.7% 0% 0% 36.3%	0% 0% 0% 100%
SCTL	t_1	t_2	t_3
Net1:	90.5% 0% 0% 9.5%	38.2%(0%) 22.1%(0%) 0% 39.7%	0% 0% 100%(0%) 0%
Net2:	93.5%(4%) 0% 0% 6.5%	63.5%(4.4%) 0% 0% 36.5%	0% 0% 100%(10.3%) 0%
Net3:	90% 0% 0% 10%	68.4% 0% 0% 1.6%	0% 0% 0% 100%
Net4:	86.5% 0% 0% 13.5%	68.5% 0% 0% 13.5%	0% 0% 0% 100%

Table 5.3: Confusion matrix of the Muddy Children example. The values in parenthesis show the classification for a stricter definition of true/false ($x <= -A_{min}$ for false and $x >= A_{min}$ for true). These values are only shown when they differ from the soft definition ($x < 0$ for false and $x > 0$ for true).

The results show how background knowledge influences the performance of the networks. Notably, the use of the first set of rules in Net_2 did not improve the performance of SCTL, indicating that the specific, temporal nature of the information provided by this rule set is difficult to generalize in these networks. CTLK presented a better performance in this case, confirming that the organization of networks into deep structures, depending on the domain, can improve learning results. This is in-

teresting in relation to the ideas of massive modularity as put forward by Pinker [84], and should be further investigated.

On the other hand, SCTL presented a better performance than CTLK on the learning of the first set of rules in the first rounds with Net_1 and all rounds with Net_3. This seems to indicate that the static nature of the first set of rules is more easily learned by this network than the temporal recurrent values. Finally, SCTL presented a better approximation of the desired 1,-1 values than CTLK, having less activations close to zero.

5.3 The Dining Philosophers

In this section, we intend to analyse the system at learning the internal states necessary to take decisions in a temporal domain associated with synchronization of processes. For such task, we have chosen a well-known testbed for distributed and communicating computing systems: the *dining philosophers problem*, originally described in [40]. The scenario is as follows: *n philosophers sit at a table, spending their time thinking and eating. In the centre of the table, there is a plate of noodles, and each philosopher around the table needs two chopsticks (forks) to eat it. The number of forks on the table is the same as the number of philosophers. One fork is placed between each pair of philosophers and they will only use the forks to their immediate right and left, and never share a fork at the same time. They never talk to each other, so they cannot negotiate a protocol of synchronization, which creates the possibility of deadlock and starvation.*

In order to create a simple, deadlock-free environment, we defined the following policy regarding the behaviour of each philosopher (or agent): from the moment an agent i perceives that she is hungry ($hungry_i$), she must start to try and get the necessary forks, in an arbitrary order (in our case, we stipulate it is from the left). When an

agent has both forks, she will eat until she perceives she is sated ($sated_i$). Each agent i interacts directly with the environment through five different actions: eat_i, representing that the agent is either eating or trying to, $dropL_i$ and $dropR_i$, indicating that the agent is returning a fork to the table (the one on her left or right, respectively), $pickL_i$ and $pickR_i$, indicating that the agent is trying to obtain one of the left or right forks, respectively. Since a fork may be unavailable, the environment responds to any request made at time t at the next time point $t + 1$ through a message $gotL_i$ or $gotR_i$, respectively, if the allocation was successful. It is important to highlight that an agent does not receive any external information about her state: she only receives information about events at an individual time point, and needs to find an internal representation of her states according to these events.

We can describe the desired behaviour of each agent as a temporal logic program, as shown in Table 5.4. Considering that each agent behaves according to such specification, our environment is defined to menage the allocation of forks in such a way that the execution is free of deadlocks. Given that our goal is to assess learning performance (i.e. how the agents learn the above-mentioned internal representation), the environment also acts as a supervisor of each agent's learning, where actions should be taken according to an expected behaviour.

Table 5.4, defining the agents' expected behaviour, is divided into two parts: the upper part contains the actual description of the behaviour, and the lower part contains rules inserted by the SCTL translation algorithm of Figure 3.10. During the experiments, we consider three kinds of agents, each of them having a different neural network as core decision process. A fully knowledgeable (FK) agent will have a neural network built by the SCTL translation of the entire set of rules in Table 5.4. A partially knowledgeable (PK) agent will have a network built from the rules in the lower part of Figure 5.4, and an agent with no prior knowledge (NK) will have a network with random weights. In order to provide equal conditions for the learning

$pickL_1 \mathbb{W} gotL_1 \leftarrow hungry_1$
$pickR_1 \mathbb{W} gotR_1 \leftarrow gotL_1$
$eat_1 \mathbb{W} sated_1 \leftarrow gotR_1$
$dropL_1 \leftarrow sated_1$
$dropR_1 \leftarrow sated_1$
$sated_1 \leftarrow sated'_1$
$GotL_1 \leftarrow GotL'_1$
$GotR_1 \leftarrow GotR'_1$
$pickL_1 \mathbb{W} gotL_1 \leftarrow \bullet(pickL_1 \mathbb{W} got_{1,A}), \sim \bullet gotL_1$
$pickL_1 \leftarrow pickL_1 \mathbb{W} gotL_1, \sim gotL_1$
$pickR_1 \mathbb{W} gotR_1 \leftarrow \bullet(pickR_1 \mathbb{W} gotR_1), \sim \bullet gotR_1$
$pickR_1 \leftarrow pickR_1 \mathbb{W} gotR_1, \sim gotR_1$
$eat_1 \mathbb{W} sated_1 \leftarrow \bullet(eat_1 \mathbb{W} sated_1), \sim \bullet sated_1$
$eat_1 \leftarrow eat_1 \mathbb{W} sated_1, \sim sated_1$

Table 5.4: An agent's temporal knowledge representation

of the different agents, the neural networks are created with the same numbers of neurons: the FK networks have their number of hidden neurons defined by the SCTL algorithm, the PK networks have some neurons inserted into the hidden layer with random weights, and the NK networks use the same architectures as FK and PK, but with all the weights initialized randomly. The figure below shows the architectures of the networks.

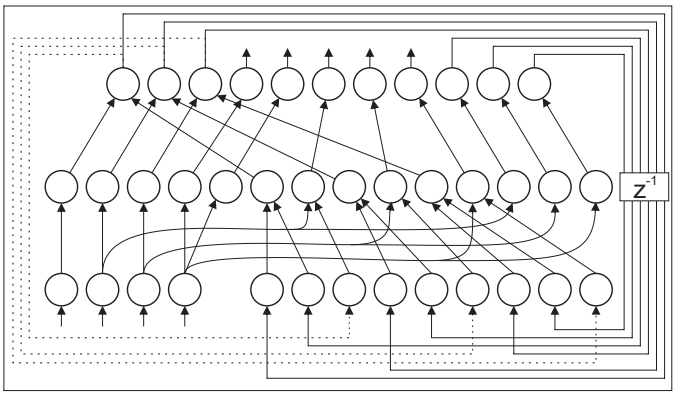


Figure 5.6: Network architectures used to perform the Dining Philosophers

5.3.1 Offline Learning

Having defined the agents, the first scenario we consider is *offline learning*, where three FK agents (networks) interact with the environment, and a fourth, learner agent is put alongside them to observe one of the agents at a time, and learn its behaviour according to such observations. At first, we used a NK learner agent, i.e. it needs to learn the complete behaviour from observation. We then considered the PK and FK learner agents for completeness.

Figure 5.7 shows the training set performance of the FK, PK and NK networks in the offline learning setting. The figure shows the root mean squared error (RMSE) for each network over 500 epochs, each epoch consisting of the application of 200 consecutive observations.

As expected, the FK network seems to offer a baseline for learning, with the PK network converging faster than the NK network. Then, the NK network seems to converge to the behaviour of the PK and FK networks, but its error shoots up again near the end of the 500 epochs. Table 5.5 gives a more detailed view of the offline experiments, showing results averaged over eight applications of the learning process. The first two lines in the table show how many epochs are needed for each FK, PK and NK network to achieve RMSE below 0.2 and 0.1, respectively. Line 3 shows the average of the lowest RMSE obtained during training, and line 4 shows the average of the RMSE at the end of the process. Having a difference between the two errors illustrates the case where the network does not converge.

	FK	HK	NK
RMSE ≤ 0.2	0	73	137.88
RMSE ≤ 0.1	0	80	155.63
Lower Error	0.0032	0.016	0.082
Last Error	0.0032	0.07	0.79

Table 5.5: Offline Learning Results

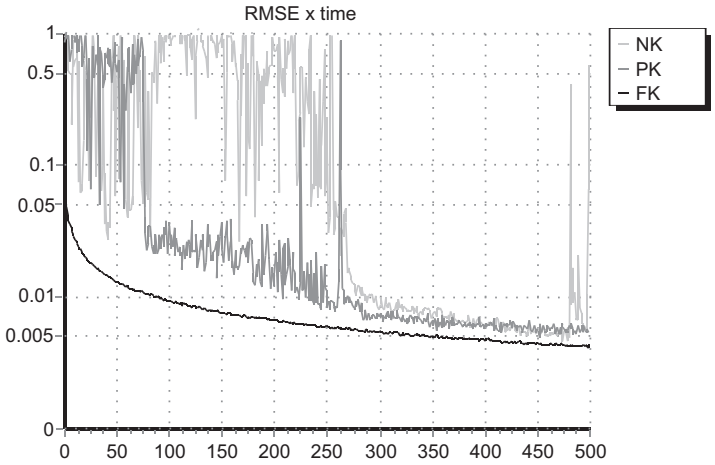


Figure 5.7: Offline Learning Error in Time

5.3.2 Online Learning

We then carried out *online learning* experiments, i.e. with the agent acting in the environment as it learns. We used an environment with three agents: agent 1 was the learning agent using either the FK, PK or NK network; agents 2 and 3 were fully knowledgeable and not learning. We have run these experiments for 100,000 time points. Figure 5.8 shows the RMSE on the training set again for 500 epochs. It illustrates well the differences in learning performance between the networks with different levels of knowledge. The figure clearly indicates that the use of background knowledge as encoded by SCTL can improve convergence and training performance.

In this experiment, we also analysed the behaviour of the system as a whole by measuring the allocation of forks to agents through the relationship between the number of agents wishing to eat and the number of agents actually eating. This is shown in Figure 5.9, where the FK network shows, as expected, a rather stable behaviour from the beginning, and the PK and NK networks seem to converge to that stable behaviour after about 150 epochs. However, just after 250 epochs, the NK network got hold of resources, preventing other agents from eating and therefore

reducing the resource allocation rate. But soon afterwards, it was able to *learn its way out* of this situation.

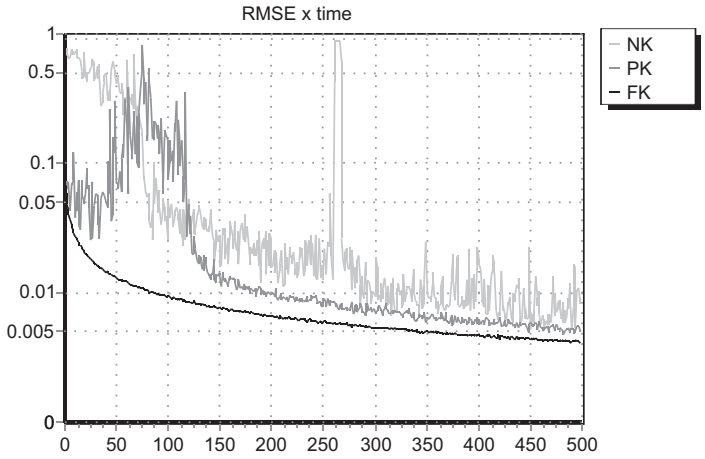


Figure 5.8: Online Error in Time

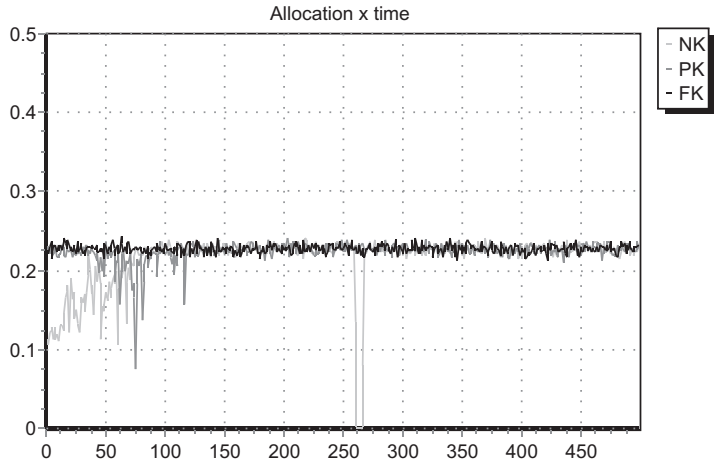


Figure 5.9: Resource allocation in time

If we compare Figures 5.8 and 5.9, it becomes clear that the decrease in the error is directly related to the agents' synchronization and proper use of resources. The results obtained in these experiments indicate that this line of research, combining temporal knowledge and connectionism, is worth pursuing. It provides supporting evidence for the integration of symbolic knowledge and neural networks in the case

Source St.	Input	Target St.
{}	$\{Hungry\}$	$\{PickL\}$
{}	$\{GotL\}$	$\{\neg PickL, PickR\}$
{}	$\{GotR\}$	$\{\neg PickR, Eat\}$
{}	$\{Sated\}$	$\{\neg Eat, DropL, DropR\}$
$\{PickL\}$	$\{\neg GotL\}$	$\{PickL\}$
$\{PickR\}$	$\{\neg GotR\}$	$\{PickR\}$
$\{DropL\}$	{}	$\{\neg DropL\}$
$\{DropR\}$	{}	$\{\neg DropR\}$
$\{Eat\}$	$\{\neg Sated\}$	$\{Eat\}$

Table 5.6: Constraints used in the learning experiments

of temporal knowledge.

5.3.3 Constrained learning

Next, we redefine the testbed to illustrate our idea of *constrained learning* to incorporate abstract symbolic knowledge into the neural network. In our experiments, we consider only neural networks without background knowledge, learning from a predefined set of constraints, shown in Table 5.6. The table is divided into two parts: the top part contains the trigger constraints, which define the conditions that trigger the state variables, and the bottom part presents the persistence constraints, which represent when the states should be kept.

Notice that each cell in the table represents the conditions which should be verified in the initial state, input and target state for each constraint. This means, for instance, that any state will match the first constraint. Therefore, if the current input assigns *Hungry* to true, the next state should assign *PickL* to true. Notice that the constraints give a considerable degree of freedom to the system, since it accepts any value for the variables that are not defined by the constraints. In our first experiment, we will verify the effects of this on the learning performance and the learned behaviour.

We consider offline learning, where a learner is not an actor in the environment, but is an observer of the behaviour of an agent. However, although the input data is given by the observed environment, the learned output will not take into account the behaviour of the agents in the environment: learning will happen by considering only the defined set of constraints. The training process will run for 100 epochs, where an epoch is the entire set of observations containing 100,000 input vectors.

We have mentioned earlier that special care is needed when neither examples nor constraints provide information about the desired value of a state variable. In this example, we will consider 4 different default values for such variables, and explore how they affect the learning process and the extracted knowledge. Table 4.4 contains an example of the value assignment illustrating the different possibilities.

In all cases, the network was capable of learning the desired constraints. The RMSE obtained for the set of inputs was always below 0.01% at the end of the training process. However, the constraints gave only general information about the desired behaviour, allowing the learning process to use different solutions to adapt to them. In the first two cases, the original knowledge of the network was kept when no information was provided. This did not happen in the other cases, as detailed below.

In order to evaluate the behaviour of the network, we compared the behaviour of each trained network with an optimal agent which behaves according to the expected policy (as defined by the entire rule set). Figure 5.11 shows the percentage of right decisions taken by the networks in comparison with the optimal agent. As shown in the figure, keeping or reinforcing the existing knowledge of the network did not produce a good result. This happens because the constraints do not have enough conditions to guide the learning process to the desired behaviour, i.e. the system can learn a different behaviour which satisfies the constraints. On the other hand, the use of arbitrary default values seems to have created a more focused learning process,

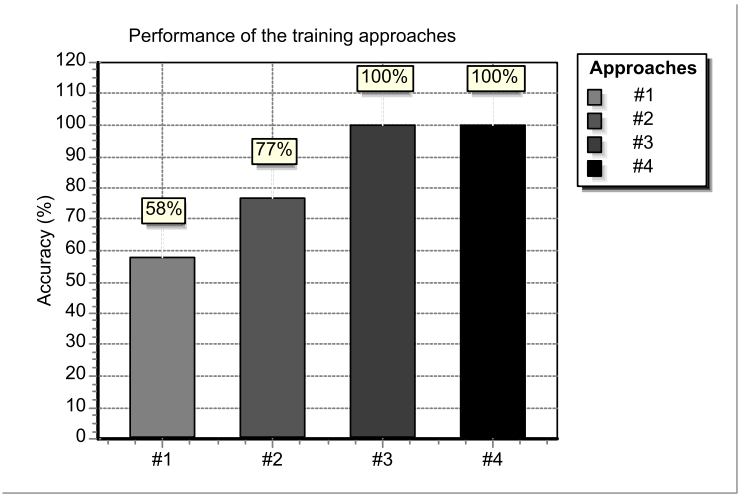


Figure 5.10: Performance of the different approaches/possibilities

according to the results obtained here.

5.3.4 Extracting learned knowledge

The transition diagram in Figure 5.10 was extracted from the network trained by following the fourth approach/possibility, using a pedagogical extraction applied to the training data. The figure illustrates all the possible states in the system, with the darker ones being reachable by the agent. The state variables *PickL*, *PickR*, *DropL*, *DropR*, *Eat* are represented, respectively, by A, B, C, D, E. Most of the states do not follow the "correct" or desired behaviour of the system, therefore more freedom can cause unpredictable effects on the system. On the other hand, limiting the behaviour through the definition of a default option whenever the constrains are not applied, has helped the system focus on a limited set of states.

The above results indicate the need for a quantitative as well as a qualitative analysis of our system. Take, for example, a comparison with the work of [64]. Both systems are capable of learning abstract, temporal knowledge in neural networks. However, SCTL was capable of learning with only 6 hidden neurons, while [64]

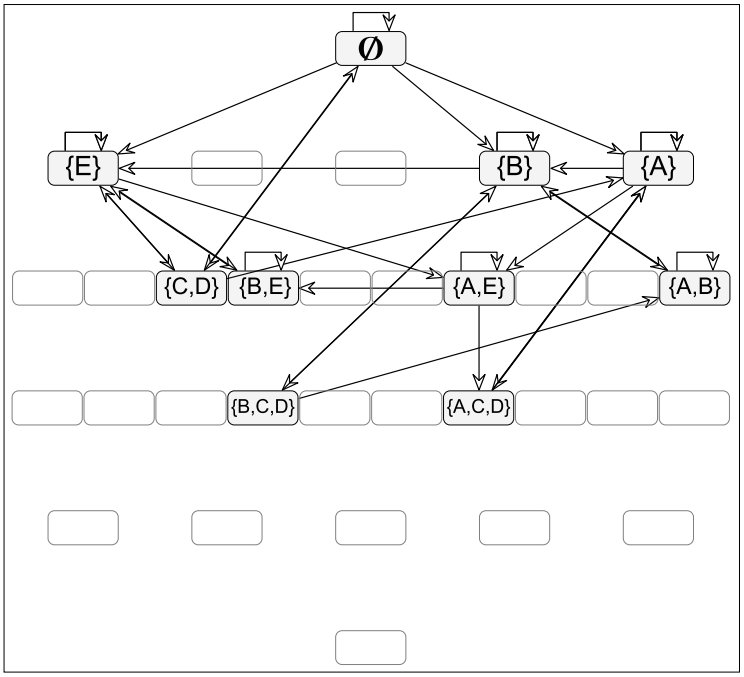


Figure 5.11: Extracted transitions from the network

$$
\begin{array}{c}
\bigcirc pickL \leftarrow Hungry \\
\bigcirc pickL \leftarrow PickL, Sated \\
\bigcirc pickL \leftarrow PickL, GotR \\
\bigcirc pickL \leftarrow PickL, \sim GotL \\
\bigcirc pickL \leftarrow PickL, DropR \\
\bigcirc pickL \leftarrow PickL, DropR
\end{array}
$$

Table 5.7: Set of clauses extracted to infer *pickL* variable

required 11 hidden neurons to represent each of the clauses of the program and the extra context neurons to represent internal states.

On the other hand, if we extract the learned knowledge in the form of a logic program from the SCTL networks, we will have a considerably larger number of clauses in the extracted knowledge than [64]. Table 5.7 shows the simplified version of the clauses in which the atom $\bigcirc PickL$ is in the head. Although our extraction algorithm is not optimized for readability, we can see that by incorporating the knowledge through learning from constraints, thus allowing the network to build a distributed internal

representation, can create difficulties for extraction. The work of [64], however, focused on a localist representation, taking the logic program and creating, instead, an individual unit for the representation of the symbols and clauses. We argue therefore, that [64] seems more adequate under a symbolic perspective, while SCTL follows a more connectionist perspective.

5.4 Discussion

An effective inductive learning technique, when applied to a specific task, must be able to satisfy two requirements, which we will call identification and generalization. Identification consists of building an internal representation that properly describes the information presented through examples. Generalization regards the adequacy of this internal representation when applied to unseen examples.

In the literature, one can find mathematical proofs of the convergence of several training algorithms for connectionist systems [91, 95]. While these theoretical results are relevant to the task of identification, generalization performance can only be estimated by experimental analysis. A good generalization performance depends on the learning technique and architecture (i.e. representation), but also on whether the examples are a good representative of the problem, and how adequate the learning system is to the application domain.

Throughout this chapter, we analysed the application of the temporal connectionist learning techniques proposed earlier in the thesis in a number of domains. Although the results cannot offer a definitive answer as to which representation is most suitable, they contribute to the ongoing scientific discussion about the need for representation in learning. The results indicate that SCTL can be suitable as a technique for robust temporal learning using neural networks.

Also, in the last two chapters, we have extended our system to allow the integration of more abstract information during the learning process. By working with constraints, we allow the learning process to be driven by a desired set of conditions, therefore expanding the applicability of the system to areas that require this abstract notion of property learning as part of an adaptive process. In the remainder of the thesis, we will explore this idea and use SCTL as part of a verification and adaptation framework, integrating SCTL with a model checker and using it to adapt and evolve software system models.

Chapter 6

The Verification and Adaptation Framework

In this chapter, we integrate the SCTL system with the NuSMV model checker, offering a new, robust way of adapting and verifying software system specifications with respect to system properties.

6.1 Formal methods and model checking

While engineering disciplines are in general based on strong mathematical formalism, the concept of Software Engineering is usually considered under a less strict perspective. Until recently, the development of computational tools was seen as being closer to an art than an actual engineering discipline. While several processes, abstractions and tools have been incorporated into the task in the last decades, most of the process still relies on human creativity and expertise to do most of the critical work[101].

Take, for instance, the task of ensuring that a developed software accomplishes

the tasks it was specified to do. Validation and verification are two important areas in software engineering. The former consists in analysing a developed software to define if its project attends the requirements. Software verification, on the other hand, is the process to verify if a developed software is working according to the specified project, for instance, by testing the software [56]. Several businesses are dependent on dynamic verification (testing), which consists in the verification performed during the execution of the software to ensure the quality of the product, with several processes and templates used to facilitate the task of testing. However, in the case of complex systems, where the software is subject to a large number of different scenarios, testing all the different possibilities consists in a difficult, expensive and sometimes impossible task.

However, several temporal logic formalisms have been proposed for the representation of requirements, behaviour and properties of software systems, and are being used to automate certain tasks in software engineering, reducing the dependency on human participation and improving the reliability of the deployed processes and the developed products [22], also reducing the time and costs of software projects.

Software verification is, therefore, a very important application of temporal logic in software engineering. Model-based verification consists of verifying if a model $\mathcal{M}$, which describes the system, satisfies a formula ϕ ($\mathcal{M} \vDash \phi$), where ϕ specifies the desired property of the system. The term model checking is used to describe the group of automatic formal methods for verifying properties of a system. The original ideas date from the beginning of 1980's with the work of [42, 24, 97]. Model checking has been widely used in industry, including applications in hardware, software and artificial intelligence systems [10, 11, 22, 39].

Model checking tools can be described as comprising three different entities: A *description language* used to represent the model, a *specification language* to repre-

sent the properties that should be satisfied by the system, and the *verification engine* that will perform the actual verification. Among the several existing tools available for Model Checking, a widely used tool is the New Symbolic Model Verification (NuSMV) [20, 21]. NuSMV gathers several different functionalities in an unique tool. Besides having an expressive description language, it allows the verification of liveliness and safety properties expressed in both CTL and LTL. Also, NuSMV allows the engineer to choose between two different approaches to perform the verification. One is based on the use of binary decision diagrams (BDD) [113] to represent and perform inference over the models, and the other considers writing the models as propositional logic expressions, and using propositional satisfiability verification (SAT) for verification [21]. We will use NuSMV in the rest of this thesis.

6.1.1 Integrating machine learning and verification

The application of automated verification techniques has brought more reliability to the process of software development. However, a number of issues remain open. One of the main issues regards the need for an abstract model of the system, described in the specific language used by the model checker, in order to allow the verification process to take place. Also, the size and complexity of the model to be verified is an issue, given that computational complexity is critical to the task [101].

In the verification of abstract models, one of the main issues is how to discover the necessary changes to be made to the model to make it comply with the given properties. While model checking tools allow the automatic verification of such properties, they only provide hints about what is wrong with the model, in the form of counter-examples. Recent work has proposed the use of machine learning in the refinement or revision of models taking into consideration the information provided by counter-examples. One interesting approach to refinement in software models consists of

CEGAR - Counter-example Guided Abstraction Refinement [23]. CEGAR uses an iterative approach to improve an existing abstract model, making use of the counter-examples obtained from the verification process. This iterative process of refinement limits the scope of the refinement according to the level of abstraction.

Abductive and deductive inference was used by [2] in the refinement of models described in event calculus, according to positive or negative examples of desired behaviour. Also, in the work of [3], the approach is extended to deal with abstract description of properties such as goal models. It is claimed that the application of such techniques may improve different processes in the software cycle, such as the evolution of requirement descriptions according to examples given by stakeholders.

In more general terms, the iterative cycle of analysis and revision to evolve requirements specifications was explored in [34], where not only the refinement, but revision of an existing model was considered. In this work, the authors also highlight the importance of allowing changes to incorrect knowledge existing in the original model.

Anther approach to learning in the process of verification of software consists of the Black Box checking [83], which uses learning techniques to build a model describing the behaviour of a system, through the observation of such behaviour in real cases. Such process, useful as a strategy of reverse engineering, caters for the construction of a high level model of a software already implemented, allowing the verification without a previous description of the model. This process can also be done with the use of an incorrect model of the system [49], where the model gives an initial approximation of the system, and the adaptation process allows not only the verification of the system, but also the reduction of the discrepancies between the model and the actual system.

6.1.2 Our integrated verification/adaptation framework

In figure 6.1, we depict all the modules involved in our framework. The round blocks illustrate the knowledge repositories, while the rectangles illustrate the main modules used by the framework. These modules are four:

1. The **NuSMV model checker** is responsible for the verification tasks in the system. It receives, as input, a symbolic description of the model to be verified, as well as a set of properties (expressed in LTL or CTL). If the model satisfies the given properties, it will inform it to the user, otherwise it will return a set of counter-examples, i.e., specific sequences of inputs/states in which the property is violated;

2. The **SCTL translation** is responsible for converting the symbolic description of the system into a neural network architecture. If the system is described as a logic program, it runs the SCTL translation algorithm as described in the Chapter 4 to create the network. If the knowledge is expressed as a NuSMV description, it will convert it into a logic program, as will be described later in this chapter;

3. The **Learning supervisor** has the algorithms that allow the observed examples of an existing system and the counter-examples from the NuSMV model checker to be integrated into the existing knowledge in the network, through the use of backpropagation. Moreover, if a partial description of the model is not given, the network can be trained directly from a set of examples;

4. The **SCTL extraction**, in the end, is capable of returning a revised model of symbolic knowledge, by extracting the knowledge from the neural network using a pedagogical approach. This extracted knowledge can be expressed in

two different manners: as a state transition diagram, or as a logic program or NuSMV description.

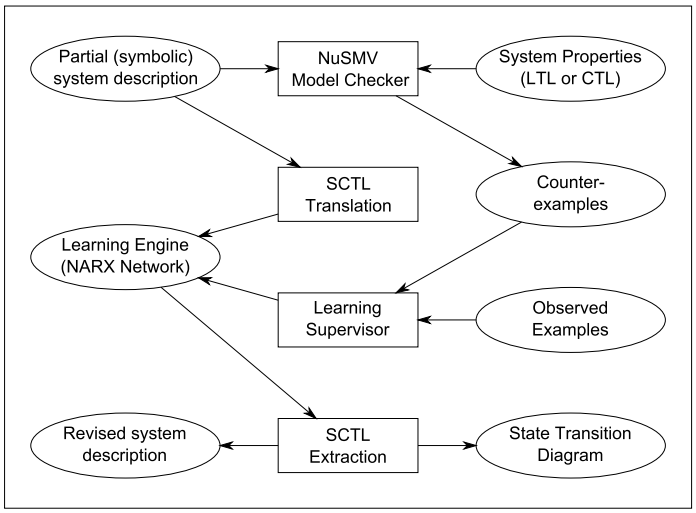


Figure 6.1: Diagram of the framework for iterative verification and adaptation of software specifications

6.2 Representing temporal models

Throughout the work, we will focus on a fragment of the NuSMV language for the description of models. For the sake of simplicity, we consider, at first, the representation of deterministic models. We will relax this restriction later, thus increasing the applicability of the framework to nondeterministic scenarios.

In Table 6.1, we describe the syntax of the NuSMV fragment that will be used throughout this work. Notice that two data types are allowed: boolean and scalar (i.e. variables whose domain is described by a finite enumeration). In practice, the NuSMV model checker converts any description into a propositional model before performing the actual verification, by converting every variable into a boolean.

As a running example, let us consider the Pump System case study used by [2, 3]

NuSMV Program ::
“MODULE” ModuleName “
“IVAR” VarDeclaration “;”
“VAR” VarDeclaration “;”
“ASSIGN”
InitBlock “;”
NextBlock “;”
VarDeclaration ::
VarId “:” VarType
— VarDeclaration “;” VarDeclaration
InitBlock ::
“init(” VarId “)=” ConstValue
— InitBlock “;” InitBlock
NextBlock
“Next(” VarId “)=” SimpleExp
— NextBlock “;” NextBlock
SimpleExpr ::
atom
— VarId
— BoolExpr
— CaseExpr
CaseExpr ::
“case” CondExpr “;” “esac”
CondExpr ::
BoolExpr “:” SimpleExpr
— CondExpr “;” CondExpr

Table 6.1: The simplified NuSMV language

to evaluate symbolic strategies to adapt software requirements according to properties. The Pump System monitors and controls the levels of water in a mine, to avoid risk of overflow, through the use of three state variables: $CrMeth$ indicating that the level of methane is critical, $HiWater$ indicating a high level of water, and $PumpOn$, indicating that the pump is turned on. In order to turn on and off such indicators, six different input signals are considered: $sCMOn$, $sCMOff$, $sHiW$, $sLoW$, $TurnPOn$ and $TurnPOff$. In table 6.2 we illustrate an initial description of the system in NuSMV.

MODULE PumpSystem
IVAR
s : {sCMOn, sCMOff, sHiW, sLoW, , TurnPOff};
VAR
CrMeth : **boolean**;
HiWat : **boolean**;
PumpOn : **boolean**;
ASSIGN
init(CrMeth) := **FALSE**;
init(HiWat) := **FALSE**;
init(PumpOn):= **FALSE**;
next(CrMeth) :=
case
s = sCMOn : **TRUE**;
s = sCMOff : **FALSE**;
esac;
next(HiWat) :=
case
s = sHiW : **TRUE**;
s = sLoW : **FALSE**;
esac;
next(PumpOn) :=
case
s = TurnPOn : **TRUE**;
s = TurnPOff : **FALSE**;
esac;

Table 6.2: NuSMV description of the Pump System example

6.2.1 Extending SCTL

We have adapted SCTL to work as the learning core of our framework. SCTL allows the use of a symbolic logic representation to describe the models in a neural networks, and therefore allowing the learning task to be performed in noisy or incomplete datasets. In order to allow a better representation of the variable types of NuSMV, we use the following definition of variable groups:

Definition 27 *An state variable group $g_{st}^{\mathcal{P}}$ in a SCTL program $\mathcal{P}$ is given by a set of state variables $\alpha_i \in St^{\mathcal{P}}$, in such a way that, for each time point t, at least one*

variable $\alpha_i \in g_{st}$ is associated to true, and if a variable $\alpha_i \in g_{st}$ is associated to true at t, all the other variables $\alpha_{i'} \in g_{st}$ are associated to false, i.e. one and only one variable $\alpha_i \in g_{st}$ is associated to true at each time point t. The same definition can be extended to input variable groups $g_{in}^{\mathcal{P}}$ of input variables $\beta_i \in In^{\mathcal{P}}$.

It can be noticed that this concept of variable groups is useful for the representation of NuSMV scalar variables, where each group of n propositional variables in SCTL is capable of representing a scalar variable with n different possibilities of value in NuSMV. When translating the SCTL program $\mathcal{P}$ into a neural network, and using this network for learning, these groups need to be taken into consideration. For input groups, for instance, the numeric vectors applied into the input of the network need to be able to comply with the definition of the group, assigning 1 to one of the inputs and -1 to the remaining ones.

For the output neurons and the interpretation of the state groups, we keep the network free to adapt the weights and calculate the values according to the weights without information about groups. However, as we explain below, the definition of the groups is used when we extract the knowledge from the network in the form of a NuSMV description. Having defined the variable groups, we can define a temporal model in SCTL as follows:

Definition 28 *A **SCTL temporal model** $\mathcal{P}$ is defined by the tuple $\mathcal{P} = \{St^{\mathcal{P}}, In^{\mathcal{P}}, Init^{\mathcal{P}}, C^{\mathcal{P}}, Gr^{\mathcal{P}}\}$, where $St^{\mathcal{P}}$ is the set of state variables α, $In^{\mathcal{P}}$ is the set of input variables β, $Init^{\mathcal{P}}$ is the initial state, defined by a mapping from $In^{\mathcal{P}} \cup St^{\mathcal{P}}$ to $\{true, false\}$, $C^{\mathcal{P}}$ is a set of clauses of the form $\bigcirc\alpha \leftarrow \alpha_1, ..., \alpha_n, \beta_1, ..., \beta_m$ and $Gr^{\mathcal{P}}$ is a set of input groups g_{in} and state groups g_{st}.*

The set of clauses will define the assignment of values to the state variables at the next time point, in the same way as defined before for SCTL: a state variable

α will be true at a time point $t + 1$ if, and only if, it is head of at least one clause $\circ\alpha \leftarrow \alpha_1, ..., \alpha_n, \beta_1, ..., \beta_m$ where all the input variables $\beta_1, ..., \beta_m$ and state variables $\alpha_1, ..., \alpha_n$ are true at time t.

6.2.2 Translating between representations

The algorithm in Figure 6.2 describes how the NuSMV headers can be used to generate the SCTL variables. It is important to notice that the set of variables is constant for an application, i.e. the future learning steps are not capable of creating, change the type or removing any variable in the system.

In order to translate *NextBlock* into SCTL, we need to make sure that scalar variables defined in the NuSMV model fit the structure of SCTL networks. All the comparisons regarding scalar variables (found in the body of the case expressions of the NuSMV language in Table 6.1) can be directly treated as a boolean variable in the SCTL network, by considering expressions like $VarId = ConstValue$ as variables, and converting $VarId != ConstValue$ into $\sim (VarId = ConstValue)$. When comparing different variables, all the possible instances of the variables need to be considered, thus building more complex boolean expressions and networks.

However, when dealing with state variables, this instantiation is more complicated, due to the assignment of values form a variable to another. In this case, to describe an expression $VarId_1 := VarId_2$ in the network, one needs to consider a conditional statement, with a new condition to represent each possible value. The NuSMV model checker already performs these kinds of conversions before verifying the model [20], and here we use the same mechanisms.

Once the above treatment of scalar comparisons is complete, the clauses for the SCTL network can be created. Our strategy consists of grouping all the conditions that lead to a boolean variable $VarId$ being mapped to true in a single expression

ϕ. Then, ϕ is rewritten as a ϕ' expression in disjunctive normal form (DNF) ($\phi' = \alpha_1 \vee \alpha_2 \vee ... \vee \alpha_n$, where each α_i is an expression $\beta_{i1} \wedge \beta_{i2} \wedge ... \wedge \beta_{im}$). Then, a clause is created for each conjunction α_i in ϕ': this clause will be of the form $\bigcirc(var = value) \leftarrow \alpha_1, \alpha_2, ..., \alpha_n$, denoting that variable *var* will have a certain value at time point $t + 1$ if $\alpha_1, \alpha_2, ..., \alpha_n$ all hold true at time point t.

Another important difference between NuSMV and SCTL regards the interpretation of a list of statements. In NuSMV, a conditional statement ($CaseBlock$) assumes that the conditions are read in a sequence, and when a condition is satisfied, the following ones are not read. In a neural-symbolic framework, knowledge is processed in parallel, meaning that two conditions (bodies of clauses) can be satisfied at the same time. Parallelism is a main attribute of a neural network, and one of the main reasons for its efficiency and robustness. Hence, we assume here that any such parallel conditions in the same $CaseBlock$ of a NuSMV model are mutually exclusive. An alternative to this assumption would be to include constraints explicitly as part of the derived clauses.

Our assumption of mutual exclusion above needs to be considered further in the case when no conditions of a $CaseBlock$ is true. Instead of implementing an "$Else$" definition, NuSMV uses a "TRUE" or "1" condition at the end of a block to indicate that if all the conditions before are false, the assignment given by this condition should be true instead. In SCTL, we do not use this interpretation, and if all the conditions in a $CaseBlock$ are false, no new value will be assigned to the related variable (i.e. the variable will keep its previous value). We claim that this approach is more appropriate in a temporal model of learning. SCTL works with "default negation", i.e. a variable is interpreted as false unless there is a clause assigning it to true. Thus, we group all the conditions that lead a variable $VarId$ to false in a boolean expression ψ. To represent the NuSMV cases when a previous variable value should be kept, we expand the ϕ expression defined above to $\phi \vee (VarId \wedge \neg\psi)$ before converting to DNF.

This means that clauses will be inserted to deal with all conditions leading $VarId$ to true, and also the situation where $VarId$ was previously true and there are no conditions leading $VarId$ to false. The algorithm accounting for the above and translating NuSMV into SCTL is shown in Figure 6.3.

When applying the algorithms to translate the NuSMV description of the Pump System into a logic program, we should obtain a description $\mathcal{P} = \left\{St^{\mathcal{P}}, In^{\mathcal{P}}, Init^{\mathcal{P}}, C^{\mathcal{P}}\right\}$ where the information in $C^{\mathcal{P}}$ is read from the $NextBlock$ in the description, and the remaining information is given as follow, where $\sim$ stands for negation (we denote by $Gr^{\mathcal{P}}$ the groups of variables used to represent the scalar input variable).

- $St^{\mathcal{P}} = \{CrMeth, HiWat, PumpOn\}$
- $In^{\mathcal{P}} = \{s = sCMOn, s = sCMOff, s = sHiW, s = sLoW, s = TurnPOn, s = TurnPOff\}$
- $Init^{\mathcal{P}} = \sim CrMeth, \sim HiWat, \sim PumpOn$
- $Gr^{\mathcal{P}} = \{\{s = sCMOn, s = sCMOff, s = sHiW, s = sLoW, s = TurnPOn, s = TurnPOff\}\}$.
- $C^{\mathcal{P}} = \{$
  - $\bigcirc CrMeth \leftarrow s : sCMOn$
  - $\bigcirc CrMeth \leftarrow CrMeth, \sim s : sCMOff$
  - $\bigcirc HiWat \leftarrow s : sHiW$
  - $\bigcirc HiWat \leftarrow CrMeth, \sim s : sLoW$
  - $\bigcirc PumpOn \leftarrow s : TurnPOn$
  - $\bigcirc PumpOn \leftarrow CrMeth, \sim s : TurnPOff$

  $\}$

6.3 Extracting NuSMV specifications

As we have seen, we are can convert the simplified NuSMV specifications into SCTL programs, which can then be translated into NARX neural networks and then subject to learning and adaptation algorithms. Moreover, as described in the last chapter, the learned knowledge can be extracted in the form of state transition diagrams. In order to obtain a new NuSMV specification from the knowledge learned by the network and close the verification and adaptation cycle, we now propose a simple strategy for obtaining this specification from the defined set of transitions.

In previous chapters, we have described how we filter the extracted transitions and then convert them into a set of logic clauses: the initial state S_0 and the input I information from the transition are used to define the body of the clauses, while each variable α in the target state S_f is chosen as head of a different clause: if positive, the head will be α, otherwise it will be $\neg\alpha$.

The same process can be defined for extracting the NuSMV "NextBlock" for boolean variables. For each variable α, each transition (or group of transitions) in which the variable appear as positive in the target state, will be converted into an expression $conj : TRUE$ in the next block of the variable α, where $conj$ is the conjunction of atoms that represent the source state and input in the transition. The same is applied to the transitions where the variable α appears as negated in the target state, generating an expression $conj : FALSE$.

When considering the scalar variables, a different treatment is necessary. Given that only one of the values can be true at the same time, simultaneous activation of different neurons representing $\alpha = value_1$, $\alpha = value_2$, ... $\alpha = value_n$ for the same variable α need not be considered.

A simple way to deal with such situations is to impose a restriction in the learn-

ing process, in such a way that assigning the same values to the same scalar variable is avoided. This kind of solution would have to change the learning algorithm considerably. We prefer to leave the learning algorithm unconstrained and constrain the extraction process instead.The extracted transition diagram will then have to consider that the system can have both values assigned to the variable, which could be considered a non-deterministic view of the learned model. Given that each transition is also associated to a weight information (given as a function of the output value of the neurons), this numerical information could be used to measure the confidence of our system to assign a variable to a specific value. These ideas will become clearer in the next chapter as we apply the framework to an entire cycle of learning, extraction and verification, until a property is satisfied in the pump system example.

This chapter was focused on the representation and translation of knowledge from the different modules of our framework (namely, the NuSMV model checker and the SCTL learning engine). The next chapter will illustrate the role played by machine learning in the framework, illustrating how the information from observed examples and NuSMV counter examples can be used during the process to allow the evolution of existing models. A series of examples will then be used to illustrate all the different cases, allowing a deeper discussion of some of the aspects introduced here.

```
HeaderTranslation:
    switch Declaration do
        case "IVAR" VarDeclaration ";"
            foreach VarId" :" VarType in VarDeclaration do
                if VarType = "boolean" then
                    In^P ← In^P ∪ {VarId}
                else
                    foreach v ∈ possibleValue(VarType) do
                        In^P ← In^P ∪ {VarId = v}
                    end
            end
        case "VAR" VarDeclaration ";"
            foreach VarId" :" VarType in VarDeclaration do
                if VarType = "boolean" then
                    St^P ← St^P ∪ {VarId}
                else
                    foreach v ∈ possibleValue(VarType) do
                        St^P ← St^P ∪ {VarId = v}
                    end
            end
        case "init(" VarId ")":=" ConstValue
            if VarId is boolean then
                if ConstValue = "TRUE" then
                    Init^P ← Init^P ∪ {VarId}
                else if ConstValue = "TRUE" then
                    Init^P ← Init^P ∪ {~ VarId}
            else
                Init^P ← Init^P ∪ {VarId = ConstValue};
                foreach u ∈ possibleValue(x) do
                    if VarId ≠ k then
                        Init^P ← Init^P ∪ {~ (VarId = ConstValue)}
                    end
    endsw
end
```

Figure 6.2: Algorithm that reads the variables declarations from NuSMV

GenerateClauses:
Propositionalization;
foreach *declaration VarId " := " exp in NextBlock* **do**
if *exp is caseExp* **then**
foreach *declaration BoolExpr* : v **do**
if v = *"TRUE"* **then**
$\phi \leftarrow \phi \vee BoolExpr$
else if v = *"FALSE"* **then**
$\psi \leftarrow \psi \vee BoolExpr$
else
$\phi \leftarrow \phi \vee (BoolExpr \wedge v)$
end
else if *exp is VarId* **then**
$\phi \leftarrow exp$
$\phi' \leftarrow DNF(\phi \vee (VarId \wedge \sim psi))$
foreach $\alpha_i = (\beta_1 \wedge ... \wedge \beta_m)$ *in* $\phi' = (\alpha_1 \vee ... \vee \alpha_n)$ **do**
$Cl^P \leftarrow Cl^P \cup \{\bigcirc VarId \leftarrow beta_1, ..., beta_n\}$
end
end
end

Figure 6.3: Translation from NuSMV into SCTL clauses

Chapter 7

Evaluation of the Framework

Throughout this chapter, we illustrate the verification and adaptation framework. The framework incorporates learning, adaptation and revision of software descriptions, integrated with a model checking tool and the description languages introduced in the last section. More specifically, it uses the NuSMV model checker for the verification of properties, but also allow a NuSMV description to be created or adapted according to observed examples of system behaviour. Also, below we explain how NuSMV counter-examples can be used in our framework in the revision of an existing model, in such a way that the process results in an evolved model which satisfies the property to be verifying.

7.1 Black Box Checking

In our neural-symbolic framework, the process of learning from examples of an observed behaviour is straightforward. Each example of the observed system will have a value assigned to all the input variables, and a desired value assigned to a subset of the state variables. The feedforward process will consist of applying the input given

by the example to the network, and obtaining a value for the next state. Information about state variables will be used to define the desired state to be informed in the backpropagation step. The flexibility of using a subset of state variables allows the use of learning from examples when some of the state variables are not observable, such as the case where this subset of states represents the actual outputs of the observed system. In this case, the backpropagation process will consider the information obtained in the feedforward process to play the role of the missing information in the example, allowing the system to keep or reinforce the existing knowledge regarding that variable.

In our first experiments, we will focus on the generation of abstract descriptions from existing examples. Differently from Chapter 5, where the analyses were mainly focused on the learning performance, here we try to evaluate how this experimental learning can be useful in the specific application of learning software specifications for further verification. Besides, we also evaluate how a connectionist learning engine handles certain issues like noise and non-determinism, which were not considered before.

7.1.1 Handling noise

Our first experiments involving adaptation consider the idea of learning the description of the model from the observed behaviour of an existing system. Considering the Pump System testbed, we apply our model to learn the behaviour described in Table 6.2. To accomplish such task, we considered an actual implementation of the system according to the given behaviour, and kept record of the inputs and states for 1000 timepoints. This set of observed examples were presented then to the framework, without any specification of the relations between inputs and states, in such a way that a description for such behaviour is built from scratch in our system.

Afterwards, we considered noisy sets of examples in the same task, i.e., we used sets of examples that do not reflect the actual desired behaviour of the system. This correspond to situations that can be caused by errors on the original observed system, or by problems when recording the observations, for instance. In order to represent that, we have changed the observed system in order to (sometimes) perform transitions to a random state, instead of the desired one, according to a defined probability.

In order to evaluate the effect of the noise into the learning process, we considered two different aspects: a qualitative analysis of the extracted model, comparing it to the expected model to be learned, and a quantitative analysis of the extracted transitions. This quantitative analysis is given by the average of the weights of the obtained transitions - i.e., the higher the value, the higher is the confidence of the model regarding the extracted model.

In Figure 7.1.1, we describe the evaluation of both metrics. The line chart shows the variation of the confidence related to the increment in the noise, considering the noise rate from 0% (without noise) to 50% (i.e. with half of the transitions leading to a random state). The grey region depicts the experiments where the extracted model was according to the original description.

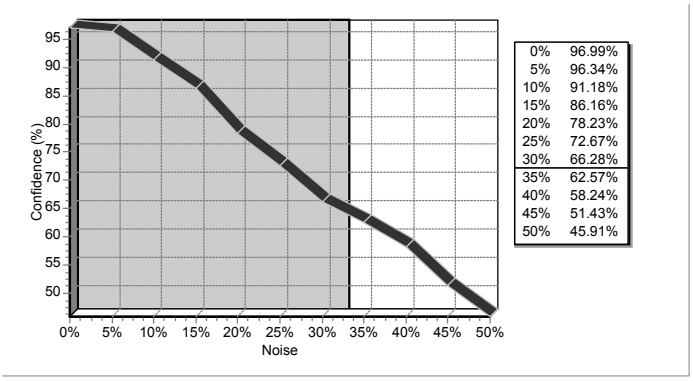


Figure 7.1: Confidence of the learning according to the amount of noise

These results clearly illustrate the noise-tolerance capability of our approach,

given that the system was able to correctly learn the desired knowledge until the noise reached 30%. The degradation in the learning performance in noisy databases can be perceived when we analyse the confidence of the system: as expected, increments in the noise led to lower confidence of the system: in this case, confidence below 65% were assigned to the models which were not learned properly. This indicates that the confidence metric might be an interesting and very relevant information when evaluating the quality of the learned model.

7.1.2 Analysis on non-deterministic scenarios

For an analysis of the behaviour of our framework when handling non-deterministic situations, we will consider a different experiment based on the previous discussion about representation and learning of scalar variables. Consider a simple system which can assume three different states: left, centre and right. The system also has a "switch": an input that, when activated, change the state to a neighbouring position, i.e. if the system is at the left it cannot go directly to the right, and vice-versa. If the system is at centre position, it can go both left or right - we consider p as the probability of going left from centre.

For this experiment, we considered the observation of systems with different values of p, intending to analyse how this would affect the learning/adaptation in our framework. In Table 7.1.2, we show the extracted transitions after the learning phase takes place, over a model where the input variable is switch, and the state variables are $state = left$, $state = centre$ and $state = right$ (represented in the table by L, C and R, respectively). The adaptation task were performed without an initial description of the model, only through the presentation of examples of the observed systems.

As we can see, in the deterministic cases ($p = 0$ and $p = 100\%$), the system behaves as expected, with high confidence in all of the transitions. In the case where

	Obtained weight in the case where p =...				
Transition	0%	25%	50%	75%	100%
$L \rightarrow\sim switch \rightarrow L$		99%	94%	99%	100%
$C \rightarrow\sim switch \rightarrow C$	100%	100%	98%	99%	100%
$R \rightarrow\sim switch \rightarrow R$	100%	100%	95%	99%	
$L \rightarrow switch \rightarrow C$		100%	98%	99%	99%
$R \rightarrow switch \rightarrow C$	100%	100%	97%	99%	
$C \rightarrow switch \rightarrow L$			43%	79%	100%
$C \rightarrow switch \rightarrow R$	100%	84%	43%		
$C \rightarrow switch \rightarrow LR$			44%		
$C \rightarrow switch \rightarrow \emptyset$			44%		

Table 7.1: Extracted transitions in the case of scalar state

$p = 25\%$ (resp. $p = 75\%$), the system treats the variation as noise, assuming the transition from centre to left (resp. right) but with a lower confidence around 84% (resp. 79%).

In the table, we show four different transitions regarding the case where *switch* is positive and $state = centre$. The target of these transitions, represented by L, R, LR and $\emptyset$ in the table, refer to $state = left$, $state = right$, both variables being positive and none of them being positive, respectively. All of these transitions showed comparable confidence (around 44%).

The result of this simple experiment brings a new element to the discussion started in the last section of the previous chapter. There, we considered different options when treating a situations like this. A proposed solution regarded imposing stronger restrictions to the adaptation process, in such a way that transitions like $C \rightarrow\sim switch \rightarrow LR$ and $C \rightarrow\sim switch \rightarrow \emptyset$ should not be allowed. As shown by these experiments, the noise tolerance of SCTL networks was sufficient to handle the situations where one of the options was preferable to the others (e.g. $p = 25\%$ and $p = 75\%$). On the other hand, in the case of maximum uncertainty, the extracted knowledge regarding both options shows the same confidence level, and even a numerical analysis of the behaviour of the network would give better information about

which solution to choose.

On the other hand, our second proposed solution considered tackling this uncertainty as an expression of the non-determinism associated with the learned knowledge (in this case, about the observed system). Even though the framework was uncertain about which variable ($state = left$ or $state = right$) should be considered true in the conflicting case, it was able to dismiss the case where $state = centre$, which is also false in the observed system. This is evidence about the capacity of the framework in capturing the non-deterministic nature of the observed system.

7.2 Verification and learning of properties

For the next experiment, we will illustrate the integration between verification and adaptation. For this purpose, we will consider the same description of the Pump System, together with a safety property expressed in LTL as $G\neg(CrMeth \wedge HiWat \wedge PumpOn)$ meaning that the pump should not be on when the level of methane is critical and the water is high at the same time. In table 7.2, we show the counter-example retrieved after the verification of the model by the NuSMV model checker.

To exemplify this idea, we have given the original pump model above to NuSMV, together with a safety property expressed in LTL as $G\neg(CrMeth \wedge HiWat \wedge PumpOn)$ meaning that the pump should not be on when the level of methane is critical and the water is high at the same time. In table 7.2, we show the counter-example retrieved in this case.

From this counter example, we can express a property $\mathcal{X}$, such that $S_0^{\mathcal{X}} = \{\neg CrMeth, \neg HiWat, \neg PumpOn\}$, $I_0^{\mathcal{X}} = \{s = sCMon\}$, $I_1^{\mathcal{X}} = \{s = sHiW\}$, $I_1^{\mathcal{X}} = \{s = turnPOn\}$ and $S_n^{\mathcal{X}} = \{\neg PumpOn\}$, with $n = 2$. Notice that $\mathcal{X}$ keeps all the information of the initial state and the sequence of inputs given by the system, and alters the final state in order to reduce the

Counter-example obtained		
t	State	Input
1	$\emptyset$	$s = sCMon$
2	$\{CrMeth\}$	$s = sHiW$
3	$\{CrMeth, HiWat\}$	$s = turnPon$
4	$\{CrMeth, HiWat, PumpOn\}$	$***$

Table 7.2: Illustration of counter-examples and the sequences to adaptation

constraint on the variable that regulates the actual state of the pump in this case.

To exemplify the process, consider the sequence $\mathcal{X}$ used in last section: suppose that in a time t, the current state assigns the three state variables to false, and therefore according to the initial state condition $S_0^{\mathcal{X}}$. Then, a copy of $\mathcal{X}$ would be added to the active sequences list, and then could be randomly selected to define the next input according to $I_0^{\mathcal{X}}$. If the input applied to the model in t equals to $\{s = sCMon\}$ ($I_0^{\mathcal{X}}$), the input in $t + 1$ is $\{s = sHiW\}$ and in $t + 2$ it is $\{s = turnPon\}$, this will define the desired value for the state in $t + 3$ to be according to $S_n^{\mathcal{X}}$ (i.e., assigning $false$ to the variable $PumpOn$). On the other hand, if any of the inputs is different, the sequence is removed from the active list.

Counter-example obtained		
t	State	Input
1	$\emptyset$	$s = sCMon$
2	$\{CrMeth\}$	$s = sHiW$
3	$\{CrMeth, HiWat\}$	$s = turnPon$
4	$\{CrMeth, HiWat, PumpOn\}$	$***$

Table 7.3: Counter-example obtained

For this experiment, we relaxed the restrictions to the initial state, represented then by the sequence: $\{\} \rightarrow s = sCMOn \rightarrow s = sHiW \rightarrow s = turnPon \rightarrow \{\neg PumpOn\}$. This represents that any configuration can be considered as the initial state of the sequence. Having the model description translated, and the sequences obtained from the counter-examples, we can apply our adaptation process. It is interesting to remark that this algorithm consists of a numeric manipulation of internal

parameters of the engine. These alterations are independent of the original syntactic structure of the model, therefore a complete different structure could be generated in order to unify the original knowledge to the learned one.

In the execution of the experiment, we considered both the network obtained by the learning from examples, and the network generated by the translation. Both systems were submitted to different configurations of the learning process, and two hypothesis for the new model were learned. In Figure 7.2, we show the transition diagram extracted in both cases. Notice that in case *a* of the diagram, the impact of the adaptation was stronger: the only situation where the pump switched from off to *on* was when both $CrMeth$ and $HiWat$ were false. In the case *b*, the only change happened to the case were both variables $CrMeth$ and $HiWat$ were true. In both cases, anyway, the counter-example was learned.

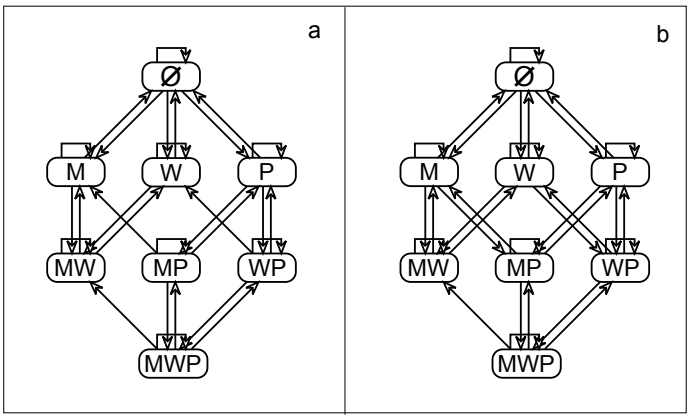


Figure 7.2: Transition diagrams representing effects of adapting to properties

Considering the case *b* to continue our analysis, we can represent the adapted model in the form of a new NuSMV program. The system adapted to the counter-example by inserting a new condition to turn the pump on. This learned condition ignores the input telling to turn the pump on when the water is high and the methane is in a critical level - being therefore general enough to deal with different sequences than the one presented by the counter-example. However, the system still does not

next(CrMeth) :=
case
s = sCMOn : **TRUE**;
s = sCMOff : **FALSE**;
esac;
next(HiWat) :=
case
s = sHiW : **TRUE**;
s = sLoW : **FALSE**;
esac;
next(PumpOn) :=
case
!CrMeth & (s = TurnPOn) : **TRUE**;
!HiWat & (s = TurnPOn) : **TRUE**;
s = TurnPOff : **FALSE**;
esac;

Table 7.4: NuSMV description adapted according to the counter-example

deal with the case where the pump needs to be turned off because a new input leads to an undesired state.

According to our proposal and the work of [34], the cycle of verification and adaptation can be repeated until the property is satisfied. Therefore, we apply the model checking tool to verify the same property in the model described by table 7.2, obtaining the counter-example described below:

- time = 0: State = $\{\sim CrMeth, \sim HighWater \sim PumpOn\}$ Input = $\{s = sCMOn\}$
- time = 1: State = $\{CrMeth, \sim HighWater, \sim PumpOn\}$ Input = $\{s = turnPOn\}$
- time = 2: State = $\{CrMeth, \sim HighWater, PumpOn\}$ Input = $\{s = sHiW\}$
- time = 3: State = $\{CrMeth, HighWater, PumpOn\}$

From the counter-example, we define the new sequence to be presented to the system in order to perform the adaptation: $\{\} \rightarrow s = sCMOn \rightarrow s = sHiW \rightarrow$

$s = turnPon \rightarrow \{\sim PumpOn\}$. Again, different initial configuration for the engine can be considered in this case: The translation of the model in table 7.2 or the original version obtained in the last adaptation process. As in the other case, we considered different configurations of the adaptation process in both cases, and the diagram shown in Figure 7.3 (a) illustrated one of the possible models learned.

Again, one can perceive that the original LTL property is still not satisfied. After expressing the model in the NuSMV language, and perform the verification again, we obtain a new counter-example as described below. After adapting to this counter-example, the obtained model is shown in 7.3(b), and expressed in the form of a model in table 7.2. When applying the model checker into this new description, the property is actually satisfied (as the state diagram makes clear).

- time = 0: State = $\{\sim CrMeth, \sim HighWater \sim PumpOn\}$ Input = $\{s = sHiW\}$
- time = 1: State = $\{\sim CrMeth, HighWater, \sim PumpOn\}$ Input = $\{s = turnPOn\}$
- time = 2: State = $\{\sim CrMeth, HighWater, PumpOn\}$ Input = $\{s = sCrMeth\}$
- time = 3: State = $\{CrMeth, HighWater, PumpOn\}$

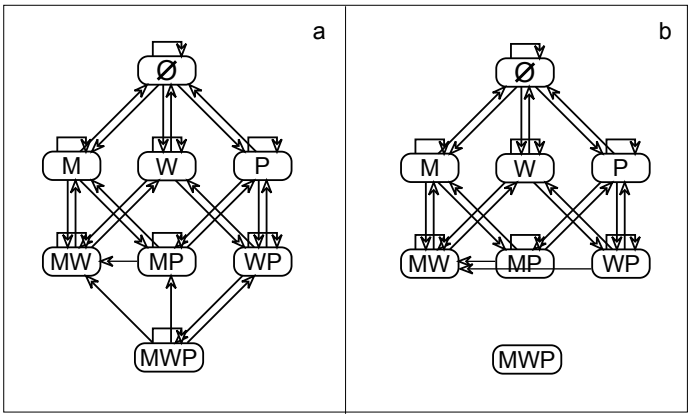


Figure 7.3: Transition diagrams representing effects of adapting to properties

```
next(CrMeth) :=
case
s = sCMOn : TRUE;
s = sCMOff : FALSE;
esac;
next(HiWat) :=
case
s = sHiW : TRUE;
s = sLoW : FALSE;
esac;
next(PumpOn) :=
case
!CrMeth & (s = TurnPOn) : TRUE;
!HiWat & (s = TurnPOn) : TRUE;
CrMeth & PumpOn & (s = sHiW) : FALSE;
HiWat & PumpOn & (s = sCMon) : FALSE;
s = TurnPOff : FALSE;
esac;
```

Table 7.5: NuSMV description obtained in the end of the process

Chapter 8

Conclusion and Future Work

We have described a set of investigations about learning, symbolic knowledge representation and reasoning in temporal models. Further, we have also illustrated the application of these ideas. As described in our introduction, this our work has brought contributions to both Artificial Intelligence and Software Engineering.

Under the perspective of Artificial Intelligence, we have described the Sequential Connectionist Temporal Logic, a novel foundational framework, that serves as an umbrella for a broad range of functionalities, as it includes:

- Symbolic language for temporal representation, which is based on a propositional modal approach which gives a broad syntax and well defined semantics for the representation of (sequential) symbolic knowledge;
- Supervised learning processes, based on traditional connectionist structures and algorithms, which brings flexibility and noise-tolerance to the task, and therefore is applicable to a range of domains;
- Integration between different sources of information, such as temporal logic programs, observed examples (through supervised learning) and sequences of

propositional conditions, which can be incorporated to the supervised learning process. These sequences can also be incorporated directly to an existing knowledge base through an adaptation process also based on connectionist algorithms;

- Full communication between the different representation structures, allowing symbolic knowledge to be incorporated into the connectionist architecture, as well as extraction of the knowledge learned by the neural network, in the form of a new symbolic representation.

While the symbolic structures have been formalized, we also presented a broad set of experiments to illustrate the accuracy and performance of the connectionist learning processes. Besides, such examples provide strong evidences of the capacities of the system in more specific applications.

Following this line, we then combine the proposed neural-symbolic system with software engineering tools, more specifically as a support tool for automatizing the specification and refinement of requirements. Integrating the SCTL's functionalities with a model checking tool led to a platform capable of performing verification and adaptation of software models, with the following features:

- A uniform cycle of verification and adaptation, where a model specified in NuSMV is subject to the verification of the properties by the model checker; the resulting counter-examples (when existing) are used to generate an improved model. This improved model can then be represented as a NuSMV description, and can be subject to new cycles of verification and adaptation.

- Adaptation through revision of the existing model, instead of refinement, which allows the substitution of incorrect declarations in the model by new expres-

sions according to the counter-examples. This extends the applicability of the framework to deal with incorrect knowledge models.

- Implementation of supervised learning to perform black box checking, i.e. the verification of observed systems even without the existence of an abstract model. This acquisition of knowledge from observed models has the properties of robustness and noise-tolerance intrinsic to the connectionist architecture used for learning.

In summary, we believe the thesis has described a rich methodology for temporal knowledge representation, learning and extraction, shedding new light on predictive temporal models not only from a theoretical standpoint, but also with respect to a potentially large number of applications in Computational Intelligence, Neural Computation and Cognitive Science, where temporal knowledge plays a fundamental role. The use of the proposed methodology should also be useful in Product-Focused Software Process Improvement [81], as well as other approaches that contribute to the automation of different phases of the process of software engineering.

Future Work

Throughout this work, we have explored thoroughly our proposed architecture for integration of the neural and symbolic AI paradigms. However, several points are still open for investigation regarding this architecture, such as the extraction of the learned knowledge in a symbolic representation, and the scalability of the architecture; in particular, if one considers a large number of variables.

Extraction is generally perceived as the bottleneck of the neural-symbolic methodologies and this is no exception in this work. Perhaps this is more evident in the case

of recurrent networks. Nevertheless, the extraction and validation of partial models has been possible, with the visualization through transition diagrams being very helpful to the understanding of the learning knowledge. Also, a number of immediate possible actions for improving the performance have been articulated, e.g. the use of rule simplification. This opens up a number of research possibilities in the area of rule extraction from recurrent networks, but also one could consider more sophisticated methods for generating positive examples from counter-examples. This may lead to a range of new applications, as suggested in [48].

Moreover, taking the SCTL structure as the foundation of our work has led us to focus on a limited range of aspects, specially in what regards the illustrated applications. Under the perspective of Software Engineering, extensions to our work can be drawn out from these limitations:

- The limitation on handling deterministic models was important for the use of SCTL learning engine as-is, but may become a hurdle in some applications. Analysing and comparing different learning algorithms under a similar structure might prove valuable to overcome this obstacle and increase the applicability of the framework.

- Some aspects are still missing to fully-automate the entire process. In particular, the conversion of counter-examples into useful training sequences for learning requires direct intervention of an expert. Information-theoretic approaches can be used to reduce this need, and therefore may lead to improvements in the efficiency of the framework when applied to a software development process.

Bibliography

[1] James F. Allen. Mantaining knowledge about temporal intervals. *Communications of the ACM*, 26(3):832–843, 1983.

[2] D. Alrajeh, J. Kramer, A. Russo, and S. Uchitel. Learning operational requirements from goal models. In *ICSE '09: Intl. Conf. Softw. Engineering*, pages 265–275. IEEE, 2009.

[3] D. Alrajeh, O. Ray, A. Russo, and S. Uchitel. Using abduction and induction for operational requirements elaboration. *Journal of Applied Logic*, 7(3):275–288, 2009.

[4] R. Andrews, J. Diederich, and A. B. Tickle. A survey and critique of techniques for extracting rules from trained artificial neural networks. *Knowledge-based Systems*, 8(6):373–389, 1995.

[5] Konstantine Arkoudas. Specification, abduction, and proof. In Farn Wang, editor, *ATVA*, volume 3299 of *Lecture Notes in Computer Science*, pages 294–309. Springer, 2004.

[6] H. Barringer, M. Fisher, D. Gabbay, G. Gough, and R. Owens. Metatem: a framework for programming in temporal logic. In *REX workshop: Proc. on Stepwise refinement of distributed systems: models, formalisms, correctness*, pages 94–129. Springer, 1990.

[7] Howard Barringer, Michael Fisher, Dov M. Gabbay, Graham Gough, and Richard Owens. Metatem: An introduction. *Formal Asp. Comput.*, 7(5):533–549, 1995.

[8] M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. In *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 164–176, New York, NY, USA, 1981. ACM Press.

[9] Patrick Blackburn, Maarten de Rijke, and Yde Venema. *Modal Logic*. Cambridge University Press, Cambridge, UK, 2001.

[10] M. G. Bobaru, C. S. Pasareanu, and D. Giannakopoulou. Automated assume-guarantee reasoning by abstraction refinement. In *CAV*, pages 135–148, 2008.

[11] R. H. Bordini, L. A. Dennis, B. Farwer, and M. Fisher. Automated verification of multi-agent programs. In *ASE*, pages 69–78, 2008.

[12] Rafael V. Borges, Artur d'Avila Garcez, and Luis C. Lamb. Integrating model verification and self-adaptation. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 317–320, New York, NY, USA, 2010. ACM.

[13] Rafael V. Borges, Artur d'Avila Garcez, and Luis C. Lamb. Learning and representing temporal knowledge in recurrent networks. *IEEE Transactions on Neural Networks*, 22:2409–2421, 2011.

[14] Rafael V. Borges, Artur d'Avila Garcez, Luis C. Lamb, and Bashar Nuseibeh. Learning to adapt requirements specifications of evolving systems: (nier track). In *Proceeding of the 33rd international conference on Software engineering*, ICSE '11, pages 856–859, New York, NY, USA, 2011. ACM.

[15] Rafael V. Borges, Artur d'Avila Garcez, and Luis C. Lamb. Representing, learning and extracting temporal knowledge from neural networks: a case study. In *Proceedings of the 20th international conference on Artificial neural networks: Part II*, ICANN'10, pages 104–113, Berlin, Heidelberg, 2010. Springer-Verlag.

[16] Rafael V. Borges, Luis C. Lamb, and A. S. d'Avila Garcez. Reasoning and learning about past temporal knowledge in connectionist models. In *Proc. of Intl Joint Conference on Neural Networks (IJCNN 2007)*. IEEE Press, 2007.

[17] Ivan Bratko and Stephen Muggleton. Applications of inductive logic programming. *Commun. ACM*, 38:65–70, November 1995.

[18] A. Browne and R. Sun. Connectionist variable binding. In S. Wermter and R. Sun, editors, *Hybrid Neural Systems*. Springer Verlag, Heidelberg, 2000.

[19] A. Browne and R. Sun. Connectionist inference models. *Neural Networks*, 14(10):1331–1355, 2001.

[20] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: A new symbolic model verifier. In *CAV '99: Proc. of the 11th Intl. Conf. on Computer Aided Verification*, pages 495–499, London, UK, 1999. Springer-Verlag.

[21] A. Cimatti, M. Pistore, M. Roveri, and R. Sebastiani. Improving the Encoding of LTL Model Checking into SAT. In *VMCAI'02*, volume 2294 of *LNCS*. Springer, 2002.

[22] E. Clarke, E. Emerson, and J. Sifakis. Model checking: algorithmic verification and debugging. *Commun. ACM*, 52(11):74–84, 2009.

[23] E. Clarke, O Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[24] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[25] D. S. Clouse, C. L. Giles, B. G. Horne, and G. W. Cottrell. Time-delay neural networks: Representation and induction of finite-state machines. *IEEE Transactions on Neural Networks*, 8(5):1065–1070, 1997.

[26] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[27] Rémi Coulom. Feedforward neural networks in reinforcement learning applied to high-dimensional motor control. In *ALT '02: Proceedings of the 13th International Conference on Algorithmic Learning Theory*, pages 403–414, London, UK, 2002. Springer-Verlag.

[28] Mark W. Craven and Jude W. Shavlik. Extracting tree-structured representations of trained networks. In David S. Touretzky, Michael C. Mozer, and Michael E. Hasselmo, editors, *Advances in Neural Information Processing Systems*, volume 8, pages 24–30. The MIT Press, 1996.

[29] A. S. d'Avila Garcez, K. Broda, and D. M. Gabbay. *Neural-Symbolic Learning Systems: Foundations and Applications*. Perspectives in Neural Computing. Springer-Verlag, 2002.

[30] A. S. d'Avila Garcez, L. C. Lamb, and D. M. Gabbay. A connectionist inductive learning system for modal logic programming. In *Proceedings of IEEE International Conference on Neural Information Processing ICONIP'02*, Singapore, 2002.

[31] A. S. d'Avila Garcez, L. C. Lamb, and D. M. Gabbay. *Neural-Symbolic Cognitive Reasoning*. Springer, 2009.

[32] A. S. d'Avila Garcez and Luís C. Lamb. Neural-symbolic systems and the case for non-classical reasoning. In Sergei N. Artëmov, Howard Barringer, Artur S. d'Avila Garcez, Luís C. Lamb, and John Woods, editors, *We Will Show Them! Essays in honour of Dov Gabbay*, pages 469–488. College Publications, 2005.

[33] A. S. d'Avila Garcez and Luis C. Lamb. A connectionist computational model for epistemic and temporal reasoning. *Neural Computation*, 18(7):1711–1738, 2006.

[34] A. S. d'Avila Garcez, A. Russo, B. Nuseibeh, and J. Kramer. An analysis-revision cycle to evolve requirements specifications. In *ASE*, pages 354–358, 2001.

[35] A. S. d'Avila Garcez, A. Russo, B. Nuseibeh, and J. Kramer. Combining abductive reasoning and inductive learning to evolve requirements specifications. In *IEE Proceedings - Software*, volume 150, pages 25–38, 2003.

[36] A. S. d'Avila Garcez and G. Zaverucha. The connectionist inductive learning and logic programming system. *Applied Intelligence*, 11(1):59–77, 1999.

[37] Artur S. d'Avila Garcez, Krysia Broda, and Dov M. Gabbay. Symbolic knowledge extraction from trained neural networks: A sound approach. *Artificial Intelligence*, 125(1-2):155–207, 2001.

[38] Marc Denecker and Antonis C. Kakas. Abduction in logic programming. In *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part I*, pages 402–436, London, UK, 2002. Springer-Verlag.

[39] J. Deshmukh, E. Emerson, and S. Sankaranarayanan. Symbolic deadlock analysis in concurrent libraries and their clients. In *ASE*, pages 480–491, 2009.

[40] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Inf.*, 1:115–138, 1971.

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

[42] E. A. Emerson and E. M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 169–181, London, UK, 1980. Springer-Verlag.

[43] Edward A. Feigenbaum. Some challenges and grand challenges for computational intelligence. *Journal of ACM*, 50(1):32–40, 2003.

[44] M. Fisher, D. Gabbay, and L. Vila, editors. *Handbook of temporal reasoning in artificial intelligence*. Elsevier, 2005.

[45] Dov M. Gabbay, Ian Hodkinson, and Mark Reynolds. *Temporal logic (vol. 1): mathematical foundations and computational aspects*. Oxford University Press, Inc., New York, NY, USA, 1994.

[46] Antony Galton. Temporal logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Stanford University, 2003.

[47] Dimitra Giannakopoulou, Corina S. Păsăreanu, and Howard Barringer. Component verification with automatically generated assumptions. *Automated Software Engineering*, 12:297–320, July 2005.

[48] Stijn Goedertier, David Martens, Jan Vanthienen, and Bart Baesens. Robust process discovery with artificial negative events. *J. Mach. Learn. Res.*, 10:1305–1340, 2009.

[49] A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. In *TACAS '02: 8th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pages 357–370. Springer, 2002.

[50] J. Y. Halpern, N. Harper, P. Kolaitis, M. Y. Vardi, and Vianu. On the unusual effectiveness of logic in computer science. *Bulletin of Symbolic Logic*, 7(2):213–236, 2001.

[51] S. Haykin. *Neural Networks: A Compreensive Foundation*. Prentice Hall, 2nd edition, 1999.

[52] M. Hilario. An overview of strategies for neurosymbolic integration. In *Proc. Workshop on Connectionist-Symbolic Integration: from Unified to Hybrid Approaches, IJCAI 95*, 1995.

[53] P. Hitzler, S. Hölldobler, and A. K. Seda. Logic programs and connectionist networks. *J. Applied Logic*, 2(3):245–272, 2004.

[54] S. Hölldobler and Y. Kalinke. Towards a new massively parallel computational model for logic programming. In *ECAI-94: Workshop on Combining Symbolic and Connectionist Processing*, volume 2, pages 68–77, Amsterdan, 1994.

[55] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[56] Michael R. A. Huth and Mark D. Ryan. *Logic in Computer Science: modelling and reasoning about systems*. Cambridge University Press, Cambridge, UK, 2000.

[57] Finn V. Jensen. *Bayesian Networks and Decision Graphs*. Information Science and Statistics. Springer, July 2002.

[58] Maurice Karnaugh. The map method for synthesis of combinational logic circuits. *Transactions of American Institute of Electrical Engineers*, 72:593–599, 1953.

[59] Nikola K. Kasabov. *Foundations of Neural Networks, Fuzzy Systems, and Knowledge Engineering*. MIT Press, Cambridge, MA, USA, 1996.

[60] Nikola K. Kasabov and Robert Kozma. Self-organization and adaptation in intelligent systems. *JACIII*, 2(6):177, 1998.

[61] Christos Kloukinas and Sergio Yovine. A model-based approach for multiple QoS in scheduling: from models to implementation. *Automated Software Engineering*, 18:5–38, March 2011.

[62] T. Kohonen, M. R. Schroeder, and T. S. Huang, editors. *Self-Organizing Maps*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 3rd edition, 2001.

[63] R Kowalski and M Sergot. A logic-based calculus of events. *New Gen. Comput.*, 4:67–95, January 1986.

[64] L. C. Lamb, R. V. Borges, and A. S. d'Avila Garcez. A connectionist cognitive model for temporal synchronization and learning. In *Proc. of 22nd AAAI Conf. on Artificial Intelligence*, pages 827–832, 2007.

[65] Leslie Lamport. "sometime" is sometimes "not never": on the temporal logic of programs. In *POPL '80: Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 174–185, New York, NY, USA, 1980. ACM Press.

[66] T. Lin, B.G. Horne, P. Tino, and C. L. Giles. Learning long-term dependencies in narx recurrent neural networks. *IEEE Transactions on Neural Networks*, 7(6):1329–1338, 1996.

[67] John W. Lloyd. *Foundations of logic programming*. Springer-Verlag New York, Inc., 1987.

[68] John W. Lloyd. *Logic for learning: learning comprehensible theories from structured data*. Springer, Berlin, 2003.

[69] Yogesh S. Mahajan, Zhaohui Fu, and Sharad Malik. Zchaff2004: An efficient sat solver. In Holger H. Hoos and David G. Mitchell, editors, *SAT (Selected Papers*, volume 3542 of *Lecture Notes in Computer Science*, pages 360–375. Springer, 2004.

[70] John McCarthy. Programs with common sense. In *Proceedings of the Teddington Conference on the Mechanization of Thought Processes*, pages 75–91, London, 1959. Her Majesty's Stationary Office.

[71] Kenneth L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, School of Computer Science, Carnegie Mellon, Pittsburgh, PA, US, 1992.

[72] R. S. Michalski. Learning strategies and automated knowledge acquisition: an overview. *Computational models of learning*, pages 1–19, 1987.

[73] M. L. Minsky and S. A. Papert. *Perceptron*. MIT Press, Cambridge, MA, 1969.

[74] T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[75] Stephen Moyle and Stephen Muggleton. Learning programs in the event calculus. In *Proceedings of the 7th International Workshop on Inductive Logic Programming*, pages 205–212, London, UK, 1997. Springer-Verlag.

[76] S. Muggleton. Inverse Entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286, 1995.

[77] Stephen Muggleton and Luc de Raedt. Inductive logic programming: Theory and methods. *J. Logic Programming*, 19/20:629–679, 1994.

[78] Wonhong Nam and Rajeev Alur. Learning-based symbolic assume-guarantee reasoning with automatic decomposition. In *In ATVA*, pages 170–185, 2006.

[79] Wonhong Nam, P. Madhusudan, and Rajeev Alur. Automatic symbolic compositional verification by learning assumptions. *Form. Methods Syst. Des.*, 32:207–234, June 2008.

[80] Allen Newell and Herbert A. Simon. Computer science as empirical inquiry: symbols and search. *Commun. ACM*, 19(3):113–126, 1976.

[81] B. Nuseibeh. Mobile privacy requirements on demand. In *Product-Focused Software Process Improvement, 11th International Conference, PROFES 2010*, page 1, 2010.

[82] M. Page. Connectionist modelling in psychology: A localist manifesto. *Behavioural and Brain Sciences*, 23:443–512, 2000.

[83] D. Peled, M. Y. Vardi, and M. Yannakakis. Black box checking. *J. of Automata Languages and Combinatorics*, 7(2):225–246, 2001.

[84] S. Pinker. *The stuff of thought: Language as a window into human nature*. Viking, 2007.

[85] A. Pnueli. The temporal logic of programs. In *FOCS '77: Proc. 18th IEEE Symp.on Foundations of Computer Science*, pages 46–67. IEEE Computer Society, 1977.

[86] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[87] J. R. Quinlan and R. M. Cameron-Jones. Induction of logic programs: Foil and related systems. *New Generation Computing, Special issue on Inductive Logic Programming*, 13:287–312, 1995.

[88] J. Ross Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[89] Oliver Ray. Nonmonotonic abductive inductive learning. *Journal of Applied Logic*, 2008.

[90] F. Rosemblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65:386–408, 1958.

[91] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. *Parallel distributed processing: explorations in the microstructure of cognition, vol. 1: foundations*, pages 318–362, 1986.

[92] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.

[93] R. Setiono, B. Baesens, and C. Mues. Recursive neural network rule extraction for data with mixed attributes. *IEEE Transactions on Neural Networks*, 19(2):299–307, 2008.

[94] R. Setiono, W.K. Leow, and J.M. Zurada. Extraction of rules from artificial neural networks for nonlinear regression. *IEEE Transactions on Neural Networks*, 13(3):564–577, 2002.

[95] Hava T. Siegelmann, Bill G. Horne, and C. Lee Giles. Computational capabilities of recurrent narx neural networks. Technical report, U. Maryland College Park, College Park, MD, USA, 1995.

[96] Hava T. Siegelmann and Eduardo D. Sontag. On the computational power of neural nets. In *COLT '92: Proceedings of the fifth annual workshop on Computational learning theory*, pages 440–449, New York, NY, USA, 1992. ACM Press.

[97] Joseph Sifakis. A unified approach for studying the properties of transition systems. *Theor. Comput. Sci.*, 18:227–258, 1982.

[98] Kendal Simon and Creen Malcolm. *An Introduction to Knowledge Engineering*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[99] P. Smolensky. On the proper treatment of connectionism. *Behavioral and Brain Sciences*, 11(1):1–23, 1988.

[100] P. Smolensky. Tensor product variable binding and the representation of symbolic structures in connectionist systems. *Artificial Intelligence*, 46(1-2):159–216, 1990.

[101] Ian Sommerville. *Software Engineering (7th Edition)*. Pearson Addison Wesley, 2004.

[102] Richard S. Sutton. *Reinforcement Learning*. Kluwer Academic Publishers, Norwell, MA, USA, 1992.

[103] Geoffrey G. Towell and Jude W. Shavlik. Knowledge-based artificial neural networks. *Artificial Intelligence*, 70(1-2):119–165, 1994.

[104] Stavros Tripakis and Karine Altisen. On-the-fly controller synthesis for discrete and dense-time systems. In *In FM'99, volume 1708 of LNCS*, pages 233–252. Springer Verlag, 1999.

[105] A. M. Turing. Computing machinery and intelligence. *Mind*, 59:433–460, 1950.

[106] L. G. Valiant. Three problems in computer science. *Journal of ACM*, 50(1):96–99, 2003.

[107] L. G. Valiant. Knowledge infusion: In pursuit of robustness in artificial intelligence. In *FSTTCS*, pages 415–422, 2008.

[108] Alexander Waibel, Toshiyuki Hanazawa, Geofrey Hinton, Kiyohiro Shikano, and Kevin J. Lang. Phoneme recognition using time-delay neural networks. *Readings in speech recognition*, pages 393–404, 1990.

[109] P.J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(5):1550–1560, 1990.

[110] S. Wermter and R. Sun. *Hybrid Neural Systems*. Springer, 2000.

[111] Ronald J. Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Comput.*, 1:270–280, June 1989.

[112] Jon Williamson. Probability logic. In Dov M. Gabbay, Ralph H. Johnson, Hans Jürgen Ohlbach, and John Woods, editors, *Handbook of the Logic of Argument and Inference - The Turn Towards the Practical*, volume 1 of *Studies in Logic and Practical Reasoning*, pages 397 – 424. Elsevier, 2002.

[113] B. Yang, R. E. Bryant, D. R. O'Hallaron, A. Biere, O. Coudert, G. Janssen, R. K. Ranjan, and F. Somenzi. A performance study of bdd-based model checking. In *FMCAD '98: Intl. Conf. Formal Methods in Computer-Aided Design*, pages 255–289, London, UK, 1998. Springer-Verlag.

[114] Lotfali Askar Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965.

[115] D. Zhang and J. P. Tsai. *Machine Learning Applications In Software Engineering*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 2005.