SpringerBriefs in Computer Science

Paulo Shakarian · Chitta Baral · Gerardo I. Simari ·
Bowen Xi · Lahari Pokala

Neuro Symbolic Reasoning and Learning

Springer

SpringerBriefs in Computer Science

SpringerBriefs present concise summaries of cutting-edge research and practical applications across a wide spectrum of fields. Featuring compact volumes of 50 to 125 pages, the series covers a range of content from professional to academic.

Typical topics might include:

- A timely report of state-of-the art analytical techniques
- A bridge between new research results, as published in journal articles, and a contextual literature review
- A snapshot of a hot or emerging topic
- An in-depth case study or clinical example
- A presentation of core concepts that students must understand in order to make independent contributions

Briefs allow authors to present their ideas and readers to absorb them with minimal time investment. Briefs will be published as part of Springer's eBook collection, with millions of users worldwide. In addition, Briefs will be available for individual print and electronic purchase. Briefs are characterized by fast, global electronic dissemination, standard publishing contracts, easy-to-use manuscript preparation and formatting guidelines, and expedited production schedules. We aim for publication 8–12 weeks after acceptance. Both solicited and unsolicited manuscripts are considered for publication in this series.

**Indexing: This series is indexed in Scopus, Ei-Compendex, and zbMATH **

Paulo Shakarian • Chitta Baral • Gerardo I. Simari •
Bowen Xi • Lahari Pokala

Neuro Symbolic Reasoning and Learning

Springer

Paulo Shakarian
Arizona State University
Tempe, AZ, USA

Chitta Baral
Arizona State University
Tempe, AZ, USA

Gerardo I. Simari
Department of Computer Science and Engineering, Universidad Nacional del Sur (UNS)
Institute for Computer Science and Engineering (UNS-CONICET)
Bahía Blanca, Argentina

Bowen Xi
Arizona State University
Tempe, AZ, USA

Lahari Pokala
Arizona State University
Tempe, AZ, USA

ISSN 2191-5768 ISSN 2191-5776 (electronic)
SpringerBriefs in Computer Science
ISBN 978-3-031-39178-1 ISBN 978-3-031-39179-8 (eBook)
https://doi.org/10.1007/978-3-031-39179-8

Foreword

Manipulating symbols is an essential part of intelligence. Symbolic reasoning allows us achieve tasks such as planning an event, programming a computer, solving algebra problems, investigating a crime, and explaining the reasoning behind AI solutions to humans. In the early days of artificial intelligence, symbolic manipulation ranging from logic programming to Bayesian reasoning represented the bulk of AI research. However, repeated cycles of unrealistic expectations led to several "AI winters" where the symbolic-based systems of the 1980s and 1990s proved to be brittle and require unrealistic amounts of customization.

Perhaps the most glaring shortcoming of the early work on symbolic AI was the inability to learn. Recent years, however, have seen a trifecta of factors: vast bodies of data now available on the Internet and via IoT devices, extraordinarily high-performance CPU/GPU compute clusters, and the ability to adapt and leverage these two factors to make decades of neural networks research practically realizable. The ability of deep learning to obtain seeming arbitrarily improved performance by adding more layers and data—an essentially brute-force approach—has led large firms to make massive hardware and personnel investments to push the technology to its limit. As a result, we have deep learning models with billions of parameters showcasing an increasing array of new capabilities. However, these systems are brittle in many ways. Combined with the hype generated by commercial entities, neural systems' abilities may be being oversold without adequate discussion of the risks and shortcomings.

However, despite the dichotomy between "connectionist" deep learning and symbolic AI, the two subfields have not always been worlds apart. Research linking optimization and logic is more than three decades old, and the use of neural networks and programming languages has been studied on and off for decades. It is now the right time to take these studies of "neurosymbolic" methods that combine neural and symbolic reasoning to the next level. This book examines several of the key advances in a single volume. Building on top of annotated logic, fuzzy logic, and differentiable logics, this book explores a variety of techniques for reasoning and learning that use these techniques in new and exciting ways. It explores applications

to agent-based systems and language as well as the key issue of symbol grounding—the all-important interface between neural and symbolic layers.

This book is timely and written with the practitioner in mind. Several of the frameworks described in this volume have roots in companies such as IBM and Sony. More recently, we have also seen an interest in the defense community with firms like Lockheed Martin and government organizations such as DARPA making investments toward neuro symbolic systems. Even consumer-oriented companies like Meta have shown recent interest with the Chief AI Scientist, a notable connectionist, stating that symbolic manipulation is necessary for human-like AI. But it is no accident that industry recognizes this importance, as neuro symbolic AI represents the most promising hope for enabling a more modular, robust, safe, and explainable AI while still leveraging the important learning advances offered by deep learning. Paulo Shakarian and his colleagues have provided a concise text to introduce the reader to some of the key advances in this important field of AI.

V.S. Subrahmanian
Walter P. Murphy Professor of Computer Science,
Northwestern University,
Chicago, IL, USA
March 2023

Acknowledgements

The authors would like to thank the Fulton Schools of Engineering at Arizona State University for their support in writing this book. Some of the authors are supported by U.S. Army Small Business Technology Transfer Program Office, the Army Research Office under Contract No. W911NF-22-P-0066, CONICET under grant PIP 11220210100577CO, and Agencia Nacional de Promoción Científica y Tecnológica, Argentina under grant PICT-2018-0475 (PRH-2014-0007).

The authors would also like to thank Al Mehdi Saadat Chowdhury for his valuable feedback in reviewing several of the chapters, Alvaro Velasquez for his valuable discussions and presentation at our 2023 AAAI tutorial of the same name, V.S. Subrahmanian for writing the forward, as well as the team at Springer, in particular Susan Lagerstrom-Fife, for their support in editing and publishing this volume.

Contents

1 New Ideas in Neuro Symbolic Reasoning and Learning 1
References 2

2 Brief Introduction to Propositional Logic and Predicate Calculus ... 3
2.1 Introduction 3
2.2 Propositional Logic 3
2.2.1 Syntax and Semantics 4
2.2.2 Consistency and Entailment 7
2.2.3 Entailment and the Fixpoint Operator 8
2.3 Predicate Calculus 9
2.4 Conclusion 13
References 14

3 Fuzzy and Annotated Logic for Neuro Symbolic Artificial Intelligence 15
3.1 Introduction 15
3.2 Generalized Annotated Programs 15
3.2.1 Syntax of GAPs 17
3.2.2 Semantics of GAPs 18
3.2.3 Satisfaction 18
3.2.4 Theoretical Results for Annotated Logic on a Lower Semi-Lattice 19
3.3 Fuzzy Logic 20
3.3.1 Properties of Fuzzy Operators Relevant to NSR 21
3.3.2 Fuzzy Negation and Considerations for Unit Interval Annotations 22
3.3.3 T-Norms 22
3.3.4 T-Conorms 24
3.3.5 Fuzzy Implication 24
3.3.6 Aggregators 26
3.3.7 Product Real Logic 28
3.3.8 Conorms to Approximate the GAPs Fixpoint Operator .. 29

3.4 Chapter Conclusion 30
References 30

4 LTN: Logic Tensor Networks 33
4.1 Introduction and Underlying Language 33
4.2 From Real Logic to Logic Tensor Networks 34
4.2.1 Representing Knowledge 35
4.3 LTN Tasks 37
4.3.1 Satisfiability and Learning 37
4.3.2 Querying 37
4.3.3 Reasoning 38
4.4 Use Cases 38
4.4.1 Basic Use Cases: Classification and Clustering 38
4.4.2 Other Use Cases 40
4.5 Discussion 40
4.6 Chapter Conclusions 41
References 41

5 Neuro Symbolic Reasoning with Ontological Networks 43
5.1 Introduction and Underlying Language 43
5.2 Recursive Reasoning Networks 45
5.2.1 RRNs: Intuitive Overview 45
5.2.2 RRNs: A Closer Look 46
5.3 Use Cases 48
5.4 Related Approaches 49
5.5 Chapter Conclusions 50
References 50

6 LNN: Logical Neural Networks 53
6.1 Introduction 53
6.2 Logic and Inference in LNNs 54
6.3 Training LNNs 57
6.4 Discussion 59
6.5 Chapter Conclusion 60
References 60

7 NeurASP 63
7.1 Introduction 63
7.2 ASP: Answer Set Programming 63
7.3 NeurASP 66
7.3.1 Semantics 67
7.3.2 Inference in NeurASP 68
7.3.3 Learning in NeurASP 71
7.4 Discussion 72
References 73

8 Neuro Symbolic Learning with Differentiable Inductive Logic Programming 75
8.1 Introduction 75
8.2 ILP Framework 75
8.2.1 Problem Formulation 76
8.2.2 Solving ILP Problems 76
8.3 A Neural Framework for ILP 78
8.3.1 Loss-Based ILP 78
8.3.2 Architecture 79
8.3.3 Complexity 81
8.3.4 Training Considerations and Empirical Results 82
8.4 Extensions to *δILP* 83
8.5 Conclusion 85
References 86

9 Understanding SATNet: Constraint Learning and Symbol Grounding 89
9.1 Introduction 89
9.2 SATNet to Learn Instances of MAXSAT 90
9.2.1 Problem Relaxation 90
9.2.2 SATNet Forward Pass 91
9.2.3 Learning the Relaxed Constraint Matrix 92
9.2.4 Experimental Findings 92
9.3 Symbol Grounding 93
9.3.1 Rebuttal on SATNet's Performance on Visual Sudoku ... 94
9.4 Discussion 95
9.5 Chapter Conclusion 96
References 96

10 Neuro Symbolic AI for Sequential Decision Making 99
10.1 Introduction 99
10.2 Deep Symbolic Policy Learning 99
10.2.1 Deep Symbolic Regression 100
10.2.2 Deep Symbolic Policy Learning 102
10.3 Verifying Neural-Based Models 103
10.3.1 STLNet 104
10.4 Discussion 106
10.5 Chapter Conclusion 107
References 107

11 Neuro Symbolic Applications 109
11.1 Introduction 109
11.2 Neuro Symbolic Reasoning in Visual Question Answering 109
11.3 Neuro Symbolic Reasoning involving Natural Language Processing 112
11.3.1 Concept Learning 112

11.3.2 Reasoning Over Text 112
11.3.3 Using ASP in Reasoning Over Text 113
11.3.4 Solving Logic Grid Puzzles Described in Text 114
11.4 Neuro-Symbolic Reinforcement Learning 116
11.5 Chapter Conclusion 117
References 117

Chapter 1
New Ideas in Neuro Symbolic Reasoning and Learning

The significant advances in machine learning due to neural network research over the past decade have inspired interest in "neuro symbolic" reasoning and learning. The key idea is that while neural systems have shown great results for learning models from data, there are certain shortcomings such as explainability, lack of modularity, difficulty in adhering to constraints or making assurances, multi-step reasoning, and other such issues [4, 5]. While ideas from symbolic disciplines of artificial intelligence such as formal logic provide the ability to address such concerns, they must be properly combined with neural methods to take advantage of the recent advances in machine learning. While neuro symbolic reasoning was formally introduced in 2005,[1] ideas concerning the integration of logic with neural architectures are actually decades old. However, the aforementioned gaps in machine learning, along with advances in neuro symbolic reasoning and learning over the past 5 years have led to more researchers exploring the topic at a much faster pace—this is sometimes referred to as the "third wave" of AI [2]. Industry and government organizations are both looking closely at this area in an effort to further advance artificial intelligence. For example, IBM has made major investments in Logical Neural Network (LNN) research and DARPA has recently announced the "Assured Neuro Symbolic Learning and Reasoning" (ANSR) program.[2]

In this book, we review several key lines of research from the past 5 years that are exemplary of advances in neuro symbolic reasoning and learning. We note that many compelling neuro symbolic frameworks are not covered in depth (e.g., [1, 3]) as this is a short volume and our intent is to communicate research indicative of the direction of the field. That said, we tried to cover significant breadth to provide good insights, covering basic principles of logic in Chaps. 2–3, exploring four of the most impactful approaches to neuro symbolic reasoning in Chaps. 4–7, examining

[1] See the IJCAI 2005 workshop dedicated to the topic https://people.cs.ksu.edu/~hitzler/nesy/NeSy05/.

[2] https://www.darpa.mil/program/assured-neuro-symbolic-learning-and-reasoning.

P. Shakarian et al., *Neuro Symbolic Reasoning and Learning*, SpringerBriefs in Computer Science, https://doi.org/10.1007/978-3-031-39179-8_1

different approaches to neuro symbolic learning in Chaps. 8–10, and concluding with a discussion on applications in Chap. 11.

Online Resources for this Book

- **ASU Neuro Symbolic AI Resource Page**
  https://neurosymbolic.asu.edu
  Includes chapter supplements, slides, and other related materials.
- **Neuro Symbolic Channel on YouTube**
  https://www.youtube.com/@neurosymbolic
  Includes in-depth video lectures on the frameworks covered in this book.

References

1. Dai, W.Z., Xu, Q., Yu, Y., Zhou, Z.H.: Bridging Machine Learning and Logical Reasoning by Abductive Learning. Curran Associates, Red Hook (2019)
2. d'Avila Garcez, A., Lamb, L.C.: Neurosymbolic AI: the 3rd wave. CoRR abs/2012.05876 (2020). https://arxiv.org/abs/2012.05876
3. Duan, X., Wang, X., Zhao, P., Shen, G., Zhu, W.: Deeplogic: joint learning of neural perception and logical reasoning. IEEE Trans. Pattern Anal. Mach. Intell. 1–14 (2022). https://doi.org/10.1109/TPAMI.2022.3191093
4. Marcus, G.: Deep learning: a critical appraisal. CoRR abs/1801.00631 (2018). http://arxiv.org/abs/1801.00631
5. Shakarian, P., Koyyalamudi, A., Ngu, N., Mareedu, L.: An independent evaluation of ChatGPT on mathematical word problems (MWP). In: AAAI Spring Symposium (2023)

Chapter 2
Brief Introduction to Propositional Logic and Predicate Calculus

2.1 Introduction

In this chapter, we introduce propositional logic and first order predicate calculus, adapted to the way we make use of these languages in the rest of this book[1]—given that we wish to provide a short treatment of this material, we prioritize intuition over rigor. For a thorough discussion of these topics, we refer the reader to [5, 8]. In Sect. 2.2, we review propositional logic covering syntax, semantics, as well as consistency, entailment, and the relationship to lattice theory and fixpoint operators.

In Sect. 2.3, we extend this simple logic to the first order case; we introduce the notion of predicates, ground vs. non-ground atoms, and quantifiers. We note that these concepts underlie major frameworks such as Logic Tensor Networks (LTN) [3], Logical Neural Networks (LNN) [10], Differentiable ILP [4], NeurASP [13], ontological neuro symbolic approaches [6], and neuro symbolic approaches using annotated logic [1, 7, 11].

2.2 Propositional Logic

A simple subset of computational logic is known as "propositional logic." The basic construct in propositional logic is a "propositional atom" or simply "atom." An atom is an aspect of the world observed by the system thought to be true. Simply said, a proposition is a statement, e.g., "alice is a nurse", "bob is a policeperson," etc. The relationship between statements is a central aspect propositional logic. Atomic propositions are the simplest statements since they refer to a single relation. It is possible to determine how the constituents of a compound proposition

[1] For instance, we deal only with finite representations.

P. Shakarian et al., *Neuro Symbolic Reasoning and Learning*, SpringerBriefs in Computer Science, https://doi.org/10.1007/978-3-031-39179-8_2

impact its truth value by using propositional logic. A compound proposition, also referred to as a formula, can be created by combining several propositions using logical connectives—the three fundamental connectives are conjunction ("and"), disjunction ("or"), and negation ("not").

What follows is a formal treatment of propositional logic, and we use the convention in the field of computational logic when introducing these ideas. The first concept we introduce is the *syntax* of the logic; here we define the constructs and how they relate to one another. The second concept is the *semantic structure*, which refers to how the underlying meaning is conveyed. The third concept is the *satisfiability relationship*, which can be thought of as how statements relate to models of the world. These fundamental building blocks of logic are important and often lead to a major difference in expressive power and computational complexity. For example, a casual inspection of the syntax (how the logic looks) of annotated logic [7] and PSL [2] may lead a reader to believe they are the same. However, they are vastly different in terms of both syntax and semantics.

2.2.1 Syntax and Semantics

Example 2.1 (Atomic Propositions) The following is a set of ground atoms:

$$A = \{\mathsf{alice\_nurse}, \mathsf{bob\_police}\}$$

Where alice_nurse can be interpreted as "alice is a nurse" and bob_police can be interpreted as "bob is a policeperson."
Based on A, the set of worlds $W = \{w_1, w_2, w_3, w_4\}$ is the following:

$$w_1 : \{\}$$
$$w_2 : \{\mathsf{alice\_nurse}\}$$
$$w_3 : \{\mathsf{bob\_police}\}$$
$$w_4 : \{\mathsf{alice\_nurse}, \mathsf{bob\_police}\}$$

Starting with syntax, as per [8] we assume the existence of a universe of ground atomic propositions ("atoms" or "ground atoms", where "ground" refers to a lack of variables) $A = \{a_1, \ldots, a_n\}$. In propositional logic, the main semantic structure is the assignment of a truth value (true or false) to each atom, which represents an interpretation that can be thought of as a "world". Formally, a world (usually denoted w) is simply a subset of A, and the set of all possible worlds 2^A is denoted with W. Intuitively, an atomic proposition (or "atom" for short) represents some aspect of reality in a certain domain of reference. A world w tells us that all atoms in w are true, while the rest are not.

Formulas Formulas are a key extension to syntax in that they allow us to combine atoms together with connectives. For propositional logic, we define a formula f over A using three such connectors: conjunction ("and"), disjunction ("or"), and negation ("not"). The symbols for these connectors are $\wedge$, $\vee$, and $\neg$. Formulas are defined *inductively* which allows us to show how formulas build on each other (this comprises a grammar). For propositional logic, the following is a definition of formulas using the three connectors.

- Each single $a \in A$ is a formula.
- If f' is a formula, $\neg f'$ is also a formula.
- If f', f'' are formulas, $f' \wedge f''$ is also a formula.
- If f', f'' are formulas, $f' \vee f''$ is also a formula.

Below we show formulas following our running example.

Example 2.2 (Formulas) Using set A from *Example 2.1*, consider formulas

$$f_1 : \mathsf{alice\_nurse} \wedge \mathsf{bob\_police}$$

and

$$f_2 : \neg\mathsf{alice\_nurse}.$$

Formula f_1 can be read as "Alice is a nurse and Bob is a policeperson", while f_2 can be read as "Alice is not a nurse".

Satisfiability As mentioned earlier, *satisfiability* refers to how the syntax (how the logic looks) relates to the semantics (the underlying meaning). Simply put, given a world w and formula f, we may ask the question "is f true in world w?"—satisfiability allows us to answer this question. To deal with this issue, we define satisfiability, denoted with the symbol $\models$. If a formula holds true for a particular assignment of truth values to its components, it is said to be satisfiable. We say w satisfies f (written $w \models f$) if f is true in world w. Likewise, we will use $w \not\models f$ to denote that w does not satisfy f.

In order to see precisely how satisfiability specifies the relationship between syntax and semantics, we define this relationship between a world w and formula f inductively as follows:

- If $f = a$, where a is an atom, then $w \models f$ iff $a \in w$.
- If $f = \neg f'$, then $w \models f$ iff $w \not\models f'$.
- If $f = f' \wedge f''$, then $w \models f$ iff $w \models f'$ and $w \models f''$.
- If $f = f' \vee f''$, then $w \models f$ iff $w \models f'$ or $w \models f''$.

Example 2.3 (Satisfiability) Consider sets A and W from *Example 2.1*, and formulas f_1, f_2 from *Example 2.2*. Recall w_4 is {alice_nurse, bob_police}.

We can say that world w_4 satisfies formula f_1 only if w_4 satisfies alice_nurse and w_4 also satisfies bob_police. As both alice_nurse and bob_police are in w_4, this condition holds, and we write $w_4 \models f_1$. However, f_2 is only satisfied by worlds that do *not* contain alice_nurse, so world w_4 does not satisfy formula f_2 since w_4 contains this atom.

We will now cover two other constructs that typically appear in languages that are based on propositional logic. The first is implication, and it is simple to translate into an equivalent propositional logic formula.

Implication Another connective between two formulas, called *implication* or *material implication*, is represented by a single arrow $\rightarrow$ and encodes the "implies" or "if/then" relationship between two formulas. Given formulas f_1, f_2, we say world $w \models f_1 \rightarrow f_2$ iff w satisfies f_2 whenever w satisfies f_1; formula f_1 is typically referred to as the antecedent, while f_2 is the consequent. Note that the semantics is such that the formula is satisfied also when the antecedent is false.[2] This is clearly equivalent to the following formula that uses the previously defined connectives:

$$f_1 \rightarrow f_2 \equiv \neg f_1 \vee f_2$$

Note that connective "$\equiv$" ("equivalence") is simply an abbreviation for double implication, so $f_1 \equiv f_2$ simply means $f_1 \rightarrow f_2$ and $f_2 \rightarrow f_1$. Another equivalence, called the contrapositive, is the following:

$$f_1 \rightarrow f_2 \equiv \neg f_2 \rightarrow \neg f_2$$

Rules We now introduce the concept of a "logical rule", which is not a construct included in propositional logic but is part of many languages that are closely related to it. A rule consists of two parts: a "body", which intuitively is interpreted as a condition, and a "head", which occurs as a result of the body being true. Rules are written as follows:

$$b \leftarrow a_1 \wedge a_2 \wedge \ldots \wedge a_n.$$

Note that the arrow points to the left—this is so the construct is not confused with material implication that in general has a different behavior in terms of its role with respect to entailment given that the contrapositive does not hold for rules.

[2] An intuitive way to interpret the formula $f_1 \rightarrow f_2$ is "I promise that whenever f_1 is true, f_2 will also be true; so, in situations in which f_1 does not hold, the promise is not broken.

In the rule above, we have that if all of the atoms $a_1, \ldots, a_n$ (the "body") are true, then atom b (the "head") must also be true, and the satisfiability relationship with a world is defined accordingly. These rules can be learned from historical data (e.g., a child touching a hot stove and getting burned), extracted from historical data (e.g., every email with a link to evilstuff.ru is malicious), or can come from an expert (e.g., a boxer with long arms throws a jab). Additionally, we shall also include a special atom TRUE where for all worlds w, $w \models \mathsf{TRUE}$. This allows us to have rules like $b \leftarrow \mathsf{TRUE}$, meaning that b is always true—this is called a *fact*.

So, let's suppose we have an agent that collects information in the form of formulas (sometimes called sentences). Such a collection of formulas is often called a "program" or a "knowledge base" depending on the specific literature. Simply put, a program Π is a set of formulas that are known to describe reality in a certain domain. A world w is such that $w \models \Pi$ if and only if for all $f \in \Pi$ we have $w \models f$.

Example 2.4 (Implication) Consider sets A and W from *Example 2.1*, formulas f_1: alice_nurse, f_2: bob_police, and world w : {alice_nurse, bob_police} from W. We can say that $f_1 \leftarrow f_2$ (alice_nurse $\leftarrow$ bob_police) is satisfied as by w since w satisfies both f_2 and f_1.

We now move on to the concepts of consistency and entailment, which are built from the basic concept of satisfiability introduced above.

2.2.2 Consistency and Entailment

We now discuss two problems that are of particular importance in the study of computational logic: consistency and entailment. The former allows us to understand if a set of logical statements is contradictory, while the latter refers to understanding when a logical statement can be included as a consequence obtained from other sentences in a knowledge base.

Consistency If it is conceivable for two or more propositions to hold true simultaneously, they are said to be logically consistent. On the other hand, they are inconsistent if there is no way for them to all be true simultaneously . Formally, a program Π is consistent if and only if there is a world w that satisfies all elements of Π. A simple example of an inconsistent program is $\{a, \neg a\}$, as clearly a cannot be both true and false at the same time. Determining if a program is consistent is generally an NP-hard problem as it can be viewed as equivalent to satisfiability.

Example 2.5 (Consistency) Consider the setting from *Example 2.2*. Formulas f_1 and f_2 taken together are inconsistent, since there is no world that can satisfy both alice_nurse and ¬alice_nurse.

Consistency is particularly important in the area of neuro symbolic AI, as the training data used to learn the parameters associated with a knowledge base may be contradictory. For example, logical neural networks [10] use a measure of consistency as a term in the loss function, and recent work [11] has explored how to ensure consistency in a learned model.

Entailment Given a program Π (which encodes what we know about the domain) and some formula f, we want to determine if f can be concluded from the information in Π. We define this problem formally as follows. Let M_Π and M_f denote the sets of all worlds that satisfy Π and f, respectively. We say that Π entails f (written $\Pi \models f$) if and only if $M_\Pi \subseteq M_f$. The set of sentences that are entailed by Π is also commonly referred to the set of its *consequences*.

This problem is also very difficult computationally, with complexity coNP-hard (coNP is the complexity class complementary to NP) in the general case as the complement of satisfiability can be encoded in this case. However, as is also the case with consistency, this problem may become easier for certain subsets of a given logical language.

2.2.3 Entailment and the Fixpoint Operator

With these preliminaries in mind, we can now present a method for addressing entailment problems in a simple logical language that is comprised of rules and does not include negation. We use this simple subset to illustrate a concept from lattice theory, namely the fixpoint operator.

Returning to the problem of entailment, let us consider the problem in our restricted logic: we want to see if a particular atom a is entailed by Π (which consists of a set of rules as described above). To solve this, there is always the option of taking a brute-force approach, which involves listing out all possible worlds and checking if they satisfy Π. Then, of the worlds that satisfy Π, we could see which ones contain a. If all of them satisfy f, then Π entails f. As we can easily see that the number of possible worlds is intractable, this is not efficient.

So, let's introduce what is called a *fixpoint operator*. For a given program Π, we define the function $\Gamma_\Pi : 2^W \to 2^W$. This function is defined as follows:

$$\Gamma_\Pi(W) = W \cup \bigcup_{r \in \Pi} \{head(r) \mid body(r) \subseteq W\}, \tag{2.1}$$

where $head(r)$ is the atom in the head of rule r and $body(r)$ is the set of atoms in the body of r. Next, we define multiple applications of Γ_Π.

$$\Gamma_\Pi^{(n)}(W) = \begin{cases} \Gamma_\Pi(W) & \text{if } n = 1 \\ T_\Pi(T_\Pi^{(n-1)}(W)) & \text{otherwise} \end{cases} \tag{2.2}$$

Example 2.6 (Fixpoint operator) Considering the language from *Example 2.1*:

$$\Pi = \{\mathsf{alice\_nurse}, \mathsf{alice\_nurse} \rightarrow \mathsf{bob\_police}\}$$
$$\Gamma_\Pi^{(1)}(\Pi) = \Gamma_\Pi(\Pi) = \{\mathsf{alice\_nurse}, \mathsf{bob\_police}\}$$
$$\Gamma_\Pi^{(2)}(\Pi) = \Gamma_\Pi(\Gamma_\Pi(\Pi)) = \{\mathsf{alice\_nurse}, \mathsf{bob\_police}\}$$
$$\Gamma_\Pi^{(3)}(\Pi) = \Gamma_\Pi(\Gamma_\Pi(\Gamma_\Pi(\Pi))) = \{\mathsf{alice\_nurse}, \mathsf{bob\_police}\}$$

Suppose that there exists, as we saw in the example above, a natural number x such that $\Gamma_\Pi^{(x-1)}(W) = \Gamma_\Pi^{(x)}(W)$. If such a number exists, then we say that Γ_Π has a (least) *fixed point*. The good news is that we can prove that Γ_Π as defined here *does* have a fixed point using Lattice Theory [12].

Now, suppose we calculate the least fixed point of $\Gamma_\Pi(\emptyset)$ (denoted $lfp(\Gamma_\Pi(\emptyset))$)—this gives us a set of atoms. It can be proved that any world that satisfies Π is a superset of $lfp(\Gamma_\Pi(\emptyset))$. Hence, this set is called the *minimal model* of Π. So, if a (the atom we are checking for entailment) is not in the minimal model of Π, then we know that Π does not entail a, as the minimal model is a world that is not in the set $M(a)$ yet satisfies Π. If a is in the minimal model, then we know that all worlds satisfying Π must contain a, and hence we can be assured of entailment in this case.

2.3 Predicate Calculus

In the previous sections, we learned how to represent atomic propositions using propositional logic. However, the expressive potential of that language is limited, as we can only describe facts that are either true or false and do not generalize, so complex sentences or natural language statements cannot be easily represented with propositional logic alone.

Consider the graph in Fig. 2.1, which shows relationships among different people, and those people have different attributes. We can certainly imagine cases where there are multiple attributes associated with the people, and multiple relationship

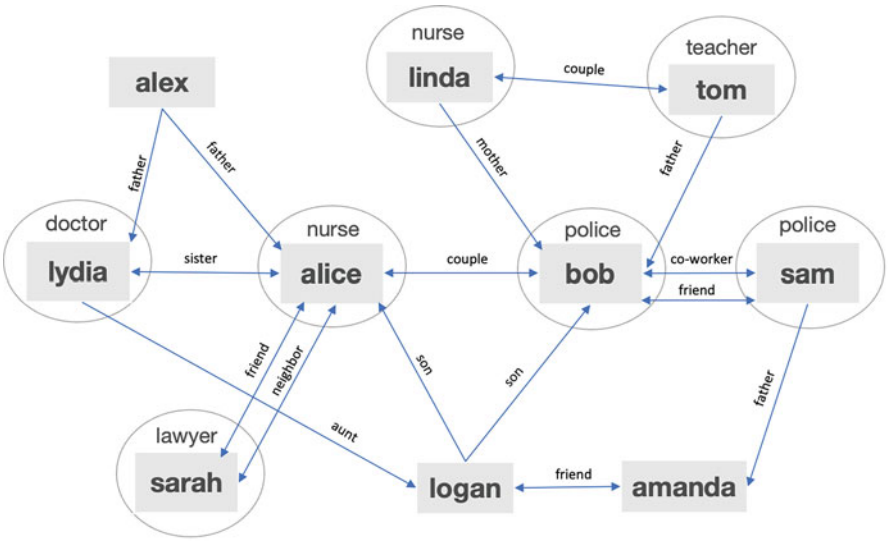


Fig. 2.1 Example multi-modal graph representing relationships among different people

types among them (e.g., Alice and Sarah are both neighbors and friends, so there are two different edges between them).

To capture relationships shown in structures such as Fig. 2.1, we require a language capable of representing sets of individual objects, defining relationships among these objects, enabling the construction of statements that hold true for some or all objects satisfying particular conditions, and encompassing all the expressions of propositional logic. One way of doing this is via "Predicate Calculus," which is also known as First-order Logic (often abbreviated "FOL"). This form of representing knowledge is an extension of propositional logic, as we still have basic statements (e.g., Alice is a nurse), but formulas in Predicate Calculus may contain more information than propositions in propositional logic. Predicate Calculus is thus more adequate for articulating natural language statements briefly compared to propositional logic.

We now discuss the basic elements of Predicate Calculus without function symbols, which is the subset we are interested in for the purposes of this book.

Constant Terms In Fig. 2.1, the entities that we wish to reason about such as alice, bob, etc., are called *constant terms*. Note that these are not propositional atoms, as there is no concept of alice being true but rather we use these entities to build up the idea of atoms. Note that it is a standard convention to represent constant symbols in lowercase (which is why we use that here, despite the symbols representing proper names of people).

Predicates In Fig. 2.1, we see that there are both binary relations (e.g., co-worker) and attributes (e.g., nurse)—the latter can also be seen as unary relations.

While depicted in a slightly different manner, both of these are *predicates* in the predicate calculus and can be used to express relationships among the constants. The combination of predicates and constants give rise to atomic propositions. So, for example, nurse(alice) is an atomic proposition that is either true or false and couple(alice, bob) is also an atomic proposition. The *arity* of a predicate specifies how many such constants are involved. In a graphical representation such as Fig. 2.1, we only have predicates of arity 1 and 2; however, in general, this is not a constraint. For example, we may have a predicate that specifies two people and a day of the week, such as hasMeeting(bob, sam, tuesday), which would perhaps mean that Bob and Sam have a meeting on Tuesday. It is also possible to have predicates with an arity of zero, which act as atomic propositions.

Atomic propositions, formed with predicates and constants in predicate calculus, can be connected to each other with the same connectors as used in propositional logic, leading to formulas. Semantics, in turn, is still based on worlds, as we can consider worlds as subsets of all possible ground atoms determined by a finite set of constants. In fact, we can view Fig. 2.1 as a world—every grounded atom depicted in that figure can be considered true in the world that it specifies. We can think of such a graph structure as representing data collected under perfect conditions.

Variables Looking back at our examples in the propositional setting (i.e., starting with Example 2.1) the difference between alice_nurse in propositional logic and nurse(alice) in predicate calculus may not be evident, but the concept that makes predicate calculus powerful is the idea of variable terms.[3] Such terms allow us to talk about relationships more generally. For example, with constant alice, the atom nurse(alice) is an atom stating that the specific person "Alice" is a nurse. But, we can now replace alice with a variable such as X, which is a symbol that does not refer to any specific entity, so we can express an arbitrary entity that is a nurse using nurse(X). Note that the construct nurse(X) is not possible in propositional logic, but it is still atomic, so we shall call such atoms that contain variables "non-ground", while atoms that only contain constants are called "ground". [4] Also, as with ground atoms, the non-ground atoms can be connected by the usual connectives to create non-ground formulas. This may lead the reader to wonder how truth or falsehood is assigned if a non-ground atom is not specific to a given constant. This is where another key concept in predicate calculus comes into play: quantification.

Quantification The truth of a non-ground formula such as nurse(X) can be interpreted in various ways through what is known as quantification. The two most common ways of quantification are *existential* and *universal*.

[3] Variable terms and constant terms together are referred to as "terms."

[4] Note that in neuro symbolic AI, the term "grounding" may be used in another context, commonly known as "symbol grounding" which refers to how predicates and constants themselves are defined. This is especially prevalent in logic tensor networks [3]. However, in this chapter, we shall refer to "grounding" in the way traditionally used in computational logic.

Existential Quantification With existential quantification, denoted with the symbol $\exists$, we are stating that "there exists an X" such that the formula is true. Thus, the existentially quantified formula

$$\exists \mathsf{X}.\mathsf{nurse}(\mathsf{X})$$

can be read as "there exists some X such that $\mathsf{nurse(X)}$ is true." If we consider the world depicted by the graph in Fig. 2.1, we see that this is true as $\mathsf{nurse(alice)}$ is in the world. However, it is important to note that we can have existentially quantified formulas that also include constants. For instance,

$$\exists \mathsf{X}.\mathsf{nurse}(\mathsf{X}) \wedge \mathsf{sister}(\mathsf{lydia}, \mathsf{X})$$

states that "there exists X such that X is a nurse and lydia and X are sisters." Again, as $\mathsf{nurse}(\mathsf{alice}) \wedge \mathsf{sister}(\mathsf{lydia}, \mathsf{alice})$ is true in the world depicted in Fig. 2.1, then such a world would satisfy that existentially quantified formula.

Universal Quantification While existential quantification states that there is at least one X, universal quantification, denoted $\forall$, means that the statement is true for *all* possible instantiations of X. So, in our example,

$$\forall \mathsf{X}.\mathsf{nurse}(\mathsf{X})$$

would not be satisfied by the world in Fig. 2.1, as $\mathsf{nurse(lydia)}$ is not in that world. Alternatively, the rule

$$\forall \mathsf{X}, \mathsf{Y}\big(\mathsf{police}(\mathsf{X}) \leftarrow \mathsf{coworker}(\mathsf{X}, \mathsf{Y})\big)$$

would be satisfied by the world in Fig. 2.1 as every pair of variables where $\mathsf{coworker}$ is true, the first argument is a policeperson.

Grounding As mentioned earlier, ground atomic propositions consist of a predicate and constant(s). In other words, formulas that use (quantifiers and) variable symbols can be grounded, which refers to transforming non-ground formulas into ground ones. So, if we have n constants, checking if $\forall \mathsf{X}.\mathsf{nurse}(\mathsf{X})$ is true means we have to check n propositional formulas, instead of just one. This complexity issue is compounded when we have predicates with a greater arity or variables spanning multiple predicates. For instance,

$$\forall \mathsf{X}, \mathsf{Y}\big(\mathsf{police}(\mathsf{X}) \leftarrow \mathsf{coworker}(\mathsf{X}, \mathsf{Y})\big)$$

requires us to check all pairs of variables. Of course, there are algorithmic and practical techniques to avoid some brute-force checks. For example, in PyReason [1], the software uses "type checking" by which only certain constants can be associated with certain predicates.

Graphical Representation Even with the improvements described above, the problem of grounding is often a major source of computational cost. As a result, we typically see the arity of predicates limited in many practical applications. For example, in [6] and [4], the arity of predicates is no greater than two in both cases. However, limiting the arity of predicates to two still allows for very expressive languages, as in many applications knowledge is expressed in a graphical format.

Perhaps one of the most common ways to practically represent knowledge in a multi-attribute formalism is as a knowledge graph. In such a structure, only binary and unary predicates are permitted, and unary predicates are often represented as binary predicates where both arguments are the same. The Resource Description Framework (RDF) [9] is perhaps the most common framework for representing data in such a manner. With RDF, a graph is represented as a set of triples of the form

$$\langle subject, predicate, object \rangle.$$

Here, the subject and object are the left and right arguments of the binary predicate (specified as the middle item of the triple). These tuples indicate either a subject-object relationship or an individual's membership in a class, where subject refers to an individual, predicate to a particular membership relation, and object to a class. We can thus write

$$sister(alice, lydia) \text{ as } \langle alice, sister, lydia \rangle, \text{ and}$$
$$police(bob) \text{ as } \langle bob, memberof, police \rangle.$$

Furthermore, we can represent negated facts as triples with a different relation symbol; for instance,

$$\neg sister(alice, lydia) \text{ as } \langle alice, not\_sister, lydia \rangle, \text{ and}$$
$$\neg police(bob) \text{ as } \neg\langle bob, not\_memberof, police \rangle.$$

Since negation isn't native to the language, adequately establishing the relationship between symbols such as *sister* and *not_sister* falls within the responsibility of the knowledge engineer.

2.4 Conclusion

In this chapter, we briefly reviewed some of the main concepts from propositional logic and predicate calculus in order to provide the reader with the basic concepts from computational logic to understand more advanced concepts in neuro symbolic artificial intelligence, such as fuzzy, real-valued, and annotated logic, as well as frameworks that provide neural equivalences to logic-based concepts, which we will cover in the following chapters.

References

1. Aditya, D., Mukherji, K., Balasubramanian, S., Chaudhary, A., Shakarian, P.: PyReason: software for open world temporal logic. In: AAAI Spring Symposium (2023)
2. Bach, S.H., Broecheler, M., Huang, B., Getoor, L.: Hinge-loss markov random fields and probabilistic soft logic. J. Mach. Learn. Res. **18**, 1–67 (2017)
3. Badreddine, S., d'Avila Garcez, A., Serafini, L., Spranger, M.: Logic tensor networks. Artif. Intell. **303**, 103649 (2022). https://doi.org/10.1016/j.artint.2021.103649. https://www.sciencedirect.com/science/article/pii/S0004370221002009
4. Evans, R., Grefenstette, E.: Learning explanatory rules from noisy data. J. Artif. Int. Res. **61**(1), 1–64 (2018)
5. Genesereth, M., Chaudhri, V.K.: Introduction to logic programming. Synth. Lect. Artif. Intell. Mach. Learn. **14**(1), 1–219 (2020)
6. Hohenecker, P., Lukasiewicz, T.: Ontology reasoning with deep neural networks. J. Artif. Intell. Res. **68**, 503–540 (2020)
7. Kifer, M., Subrahmanian, V.: Theory of generalized annotated logic programming and its applications. J. Log. Program. **12**(3&4), 335–367 (1992)
8. Lloyd, J.W.: Foundations of Logic Programming. Springer-Verlag, New York (1987)
9. Miller, E.: An introduction to the Resource Description Framework. D-lib Magazine (1998)
10. Riegel, R., Gray, A., Luus, F., Khan, N., Makondo, N., Akhalwaya, I.Y., Qian, H., Fagin, R., Barahona, F., Sharma, U., Ikbal, S., Karanam, H., Neelam, S., Likhyani, A., Srivastava, S.: Logical neural networks (2020). https://org.doi/10.48550/ARXIV.2006.13155. https://arxiv.org/abs/2006.13155
11. Shakarian, P., Simari, G.: Extensions to generalized annotated logic and an equivalent neural architecture. In: IEEE TransAI. IEEE (2022)
12. Tarski, A.: A lattice-theoretical fix point theorem and its applications. Pac. J. Math. **5**, 285–309 (1955)
13. Yang, Z., Ishay, A., Lee, J.: NeurASP: embracing neural networks into answer set programming. In: Bessiere, C. (ed.) Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20, pp. 1755–1762. International Joint Conferences on Artificial Intelligence Organization (2020). Main track

Chapter 3
Fuzzy and Annotated Logic for Neuro Symbolic Artificial Intelligence

3.1 Introduction

Most of the research reviewed in this book uses some type of real-valued or "fuzzy" formalism as the underlying logic for a given neuro symbolic framework. For example, in [2, 18] the authors use "real logic" while work on differentiable inductive logic programming [3, 16] is based on a subset of that language. The framework of Logical Neural Networks [10, 11] uses a related but different variant they refer to as "weighted real-valued logic." All of these logics rely on a collection of well-studied "fuzzy operators" [19]. Further, it turns out that the logic used in the aforementioned frameworks can be viewed as a special case of generalized annotated logic [6] whose application to neuro symbolic reasoning was studied in [1, 12]. In this chapter, we review generalized annotated logic and fuzzy operators, which are used in some of the major frameworks described in this text (e.g., LNN in Chap. 6, LTN in Chap. 4, and δILP in Chap. 8).

3.2 Generalized Annotated Programs

Neuro symbolic reasoning frameworks are data-driven, and as a result deal with noisy, incomplete, and potentially inconsistent data. Additionally, these frameworks are powered by gradient descent. As a direct result, associating logical propositions with a real value in the interval [0, 1] can make sense, as it can be associated with a "confidence" or "degree of truth" and in some cases a probability. The decision to associate logical propositions with a scalar in that interval became popular in research on logic programming (e.g., Van Emden Logic [17]). This has also been the design choice for several recent neuro symbolic frameworks such as LTN [2] and δILP [3]. However, by its very nature, it is making a *closed world* assumption, as an atom is associated with a value of 0 for false and 1 for true. In alternatives that

P. Shakarian et al., *Neuro Symbolic Reasoning and Learning*, SpringerBriefs in Computer Science, https://doi.org/10.1007/978-3-031-39179-8_3

have been seen in various forms of logic [5, 9, 14, 15], atoms are instead associated with a subset of the interval [0, 1]. This allows for *open world* reasoning as we have included in the interval a notion of uncertainty about the confidence itself. So, for example, while "true" is associated with [1, 1] and "false" is associated with [0, 0], we have various levels of uncertainty, with [0, 1] representing total uncertainty. In applications where the absence of data refers to "unknown" as opposed to "false", this type of reasoning becomes more applicable. The introduction of Generalized Annotated Programs ("GAPs") by [6] provided a unified framework that generalized both of these results by instead associating atomic propositions with elements of a lattice structure. The results of [6] and others (including the more recent [1, 12]) hold in this very general case, yet provide utility into the aforementioned special cases.

In this section, we recapitulate the definition of GAPs from [6], but with a focus on certain syntactic elements as well as the use of lower-lattice semantics—this is more relevant for neuro symbolic reasoning. As mentioned earlier, GAPs allow atomic propositions to be associated with values from a lattice structure, which generalizes other real-valued logic paradigms used in various neuro symbolic approaches; additionally, annotations based on a lattice structure can support describing an atomic proposition not only as true but as false or uncertain (i.e., no knowledge), allowing for the modeling of open-world novelty.

Use of Lower Semi Lattice In [6], the authors assumed the existence of an upper semi-lattice $\mathcal{T}$ (not necessarily complete) with ordering $\sqsubseteq$. In the extensions of [1, 12, 14, 15], it was proposed to instead use a *lower* semi-lattice structure, where we have a single element $\bot$ and multiple top elements $\top_0, \ldots \top_i \ldots \top_{max}$. Here, we also use the notation $height(\mathcal{T})$ to denote the maximum number of elements in the lattice in a path between $\bot$ and a top element (including $\bot$ and the top element).[1] In this chapter, we shall review some formal results from [12] in which certain results from [6] are also shown to hold (under certain conditions) on a lower semi-lattice.

The employment of a lower semi-lattice structure enables two desirable features. First, atoms can be annotated with subsets of the real unit interval as done in several frameworksk [10, 13, 15]. Second, it allows for reasoning about such intervals whereby the amount of uncertainty (i.e., for interval $[l, u]$ the quantity $u - l$) decreases monotonically as an operator proceeds up the lattice structure.

Therefore, we define the bottom element $\bot = [0, 1]$ and a set of top elements $\{[x, x] \mid [x, x] \subseteq [0, 1]\}$ (see note[2]). Specifically, we set $\top_0 = [0, 0]$ and $\top_{max} = [1, 1]$. An example of such a semi-lattice structure is shown in Fig. 3.1.

[1] In general, we shall assume that the lattice has a finite and discrete carrier set.

[2] N.B.: When using a semi-lattice of bounds, the notation "$\sqsubseteq$" loses its "subset intuition", as $[0, 1] \sqsubseteq [1, 1]$ in this case, for example.

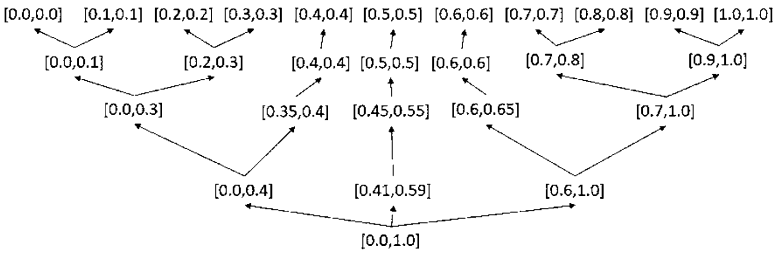


Fig. 3.1 Example of a lower semi-lattice structure where the elements are intervals in [0, 1]

3.2.1 Syntax of GAPs

We assume the existence of a set AVar of variable symbols ranging over $\mathcal{T}$ and a set $\mathcal{F}$ of function symbols, each of which has an associated arity. We start by defining the concept of annotation.

Definition 3.1 (Annotation) (i) Any member of $\mathcal{T} \cup$ AVar is an annotation. (ii) If f is an n-ary function symbol over $\mathcal{T}$ and $t_1, \ldots, t_n$ are annotations, then $f(t_1, \ldots, t_n)$ is an annotation.

Example 3.2 (Annotation) The following are all annotations: 0.2, 1, 0.05, and X (here X is assumed to be a variable in AVar). If $+$ and $*$ are binary function symbols in $\mathcal{T}$, then the following is an example of an annotation:

$$\frac{0.2}{(X * 0.05) + 1} \tag{3.1}$$

One specific function we define is "$\neg$", which is used in the semantics of [6]: for a given $[l, u]$, $\neg([l, u]) = [1 - u, 1 - l]$. Note that we also use the symbol $\neg$ in our first-order language (following the formalism of [6]).

We define a separate logical language whose constants are members of set C and whose predicate symbols are specified by set $\mathcal{P}$. We also assume the existence of a set $\mathcal{V}$ of variable symbols ranging over the constants, that no function symbols are present, and that terms and atoms are defined in the usual way (cf. [8]). We shall assume that C, $\mathcal{P}$, and $\mathcal{V}$ are discrete and finite. In general, we shall use capital letters for variable symbols and lowercase letters for constants. Similar to previous work [3, 4], we assume that all elements of $\mathcal{P}$ have an arity of either 1 or 2—we use $\mathcal{P}_{una}$ to denote the set of unary predicates and $\mathcal{P}_{rel}$ the set of binary predicates. We shall also denote a subset of $\mathcal{P}$ to include "target predicates" written $\mathcal{P}_{tgt}$ that can consist of either binary or unary predicates ($\mathcal{P}_{tgt\_rel}$, $\mathcal{P}_{tgt\_una}$) provided that they are not reserved words. We shall use the symbol $\mathcal{L}$ to denote the set of all ground

literals and $\mathcal{A}$ for the set of all ground atoms. We now define the syntactical structure of GAPs that will be used in this work.

Definition 3.3 (Annotated atoms, negations, literals) If $\mathbf{a}$ is an atom and $\boldsymbol{\mu}$ an annotation, then $\mathbf{a} : \boldsymbol{\mu}$ is an *annotated atom* and $\neg \mathbf{a} : \boldsymbol{\mu}$ is a *negated annotated atom*. Collectively, such formulas are called *annotated literals*.

Definition 3.4 (GAP Rule) If $\ell_0 : \mu_0, \ell_1 : \mu_1, \ldots, \ell_m : \mu_m$ are annotated literals (such that for all $i, j \in 1, m, \ell_i \not\equiv \ell_j$), then:

$$r \equiv \ell_0 : \mu_0 \leftarrow \ell_1 : \mu_1 \wedge \ldots \wedge \ell_m : \mu_m$$

is called a *GAP rule*. We will use the notation $head(r)$ and $body(r)$ to denote ℓ_0 and $\{\ell_1, \ldots, \ell_m\}$, respectively. When $m = 0$ ($body(r) = \emptyset$), the above GAP rule is called a *fact*. A GAP rule is *ground* if there are no occurrences of variables from either AVar or $\mathcal{V}$ in it. For ground rule r and ground literal ℓ, $bodyAnno(\ell, r) = \mu$ *such that* $\ell : \mu$ *appears in the body* of r. A generalized annotated program Π is a finite set of GAP rules.

3.2.2 Semantics of GAPs

The formal semantics of GAPs are defined as follows. Note that we extend the notion of an interpretation to allow for a mapping of literals to annotations (as opposed to atoms). However, we add a requirement on the annotation between each atom and negation that ensures equivalence to the semantic structure of [6]. The intuition behind this extension is that we can more easily detect inconsistencies using the fixpoint operator, as we can just compare the annotations of each pair of literals (the atom and its negation).

Definition 3.5 (Interpretation) An interpretation I is any mapping from the set of all grounds literals to $\mathcal{T}$ such that for literals $a, \neg a$, we have $I(a) = \neg(I(\neg a))$. The set $\mathcal{I}$ of all interpretations can be partially ordered via the ordering: $I_1 \preceq I_2$ iff for all ground literals l, $I_1(\ell) \sqsubseteq I_2(\ell)$. $\mathcal{I}$ forms a complete lattice under the $\preceq$ ordering.

3.2.3 Satisfaction

We now present the concept of satisfaction of literals and rules by interpretations:

Definition 3.6 (Satisfaction) An interpretation I *satisfies* a ground literal $\ell : \mu$, denoted $I \models \ell : \mu$, iff $\mu \sqsubseteq I(\ell)$. I satisfies the ground GAP rule

$$\ell_0 : \mu_0 \leftarrow \ell_1 : \mu_1 \wedge \ldots \wedge \ell_m : \mu_m$$

(denoted $I \models \ell_0 : \mu_0 \leftarrow \ell_1 : \mu_1 \ \wedge \ \ldots \ \wedge \ \ell_m : \mu_m$) iff either

1. I satisfies $\ell_0 : \mu_0$ or
2. There exists an $1 \leq i \leq m$ such that I does not satisfy $\ell_i : \mu_i$.

I satisfies a non-ground literal or rule iff I satisfies all ground instances of it.

We say that an interpretation I is a *model* of program Π if it satisfies all rules in Π. Likewise, program Π is *consistent* if there exists some I that is a model of Π. We say Π *entails* $\ell : \mu$, denoted $\Pi \models_{ent} \ell : \mu$, iff for every interpretation I s.t. $I \models \Pi$, we have that $I \models \ell : \mu$. As shown in [6], we can associate a fixpoint operator with any GAP Π that maps interpretations to interpretations.

Definition 3.7 Suppose Π is any GAP and I is an interpretation. The mapping Γ_Π that maps interpretations to interpretations is defined as

$$\Gamma_\Pi(I)(\ell_0) = \textbf{sup}(annoSet_{\Pi,I}(\ell_0)),$$

where $annoSet_{\Pi,I}(\ell_0) = \{I(\ell_0)\} \cup \{\mu_0 \mid \ell_0 : \mu_0 \leftarrow \ell_1 : \mu_1 \wedge \ldots \wedge \ell_m : \mu_m$ *is a ground instance of a rule in* Π, *and for all* $1 \leq i \leq m$, *we have* $I \models \ell_i : \mu_i\}$.

The key result of [6] (Theorem 2 in that paper, discussed in the next section) tells us that $lfp(\Gamma_\Pi)$ precisely captures the ground atomic logical consequences of Π. We show this is also true (under the condition that Π is consistent) even if the annotations are based on a lower lattice (see Theorem 3.10). In [6], the authors also define the *iteration* of Γ_Π as follows:

- $\Gamma_\Pi \uparrow 0$ is the interpretation that assigns $\bot$ to all ground literals.
- $\Gamma_\Pi \uparrow (i+1) = \Gamma_\Pi(\Gamma_\Pi \uparrow i)$.

For each ground $\ell \in \mathcal{L}$, the set $\Pi(\ell)$ is the subset of ground rules (to include facts) in Π where ℓ is in the head. We will use the notation m_ℓ to denote the number of rules in $\Pi(\ell)$. For a given ground rule, we will use the symbol $r_{\ell,i}$ to denote that it is the i-th rule with atom ℓ in the head.

3.2.4 Theoretical Results for Annotated Logic on a Lower Semi-Lattice

In [12], the authors revisit some of the results of [6] that depend upon an upper semi-lattice structure and obtain comparable results that apply to a lower semi-lattice.[3] Further, note that in UGAPs we shall refer to interpretations defined only

[3] Note that we shall refer to results specific to the upper semi-lattice as "UGAPs," and those for a lower semi-lattice as "LGAPs".

as mappings of atoms to annotations (i.e., not including negated atoms). This makes it relatively straightforward to have consistency; consider the following proposition shown in [12].

Proposition 3.8 *Any UGAP* Π *consisting of UGAP rules where all bodies and heads are atoms is consistent.*

To provide a specific example of an LGAP that is not consistent, consider the following two rules (here a is a ground literal):

$$a : [0, 0] \leftarrow$$

$$a : [1, 1] \leftarrow$$

Take note of the key result of [6] below.

Theorem 3.9 *[Theorem 2 in [6]] For UGAP* Π, Γ_Π *is monotonic and has a least fixpoint* $lfp(\Gamma_\Pi)$. *Moreover, for this case* Π *entails* $a : \mu$ *iff* $\mu \sqsubseteq lfp(\Gamma_\Pi)(a)$.

However, under the condition that Π is consistent, [12] shows a similar result.

Theorem 3.10 *If LGAP* Π *is consistent, then:*

1. Γ_Π *is monotonic,*
2. Γ_Π *has a least fixpoint* $lfp(\Gamma_\Pi)$, *and*
3. Π *entails* $a : \mu$ *iff* $\mu \leq lfp(\Gamma_\Pi)(a)$.

In [12] it is also shown that for both LGAPs and UGAPs, there is a bound on the number of applications of Γ that are required until convergence.

Theorem 3.11 *If (LGAP or UGAP)* Π *is consistent, then* $lfp(\Gamma_\Pi) \equiv \Gamma_\Pi \uparrow x$, *where* $x = height(\mathcal{T}) * |\mathcal{L}|$.

In the next section, we present some background concepts related to fuzzy logic.

3.3 Fuzzy Logic

Generalized annotated logic uses arbitrary function symbols to specify how annotations are combined; we now focus on how those functions can be defined. Specifically, we explore the various fuzzy operators [18, 19] used in neuro symbolic frameworks, including: fuzzy negation (Sect. 3.3.2); triangular norms (T-norms), which are associated with conjunctions (Sect. 3.3.3); T-conorms, which are associated with disjunction (Sect. 3.3.4); fuzzy implications (Sect. 3.3.5); and aggregators, which are associated with first-order logic quantifiers. However, first we discuss some properties that are useful in understanding fuzzy operators and their associated derivatives.

3.3.1 Properties of Fuzzy Operators Relevant to NSR

The paradigms we describe in this book that employ fuzzy operators—LNN, LTN, and δILP —leverage the operators within the neural structure where they either are used as activation functions or produce the argument of an activation function. As such, their derivatives are used during the backpropagation process, and this can impact the learning process. There also is a natural trade-off between the function of these operators (in terms of resembling certain logical formalisms) vs. the utility in the backpropagation process. Understanding such trade-offs is crucial to the success of an NSR system.

Functions as Classical Operators on Classical Input If we restrict the inputs to a fuzzy operator to either 0 or 1, it is sensible to expect the output to resemble that of the associated truth table. For example, this is a property of the product T-norm, as (in the case of two inputs) it returns 1 only for the input 1, 1 and 0 for the other three sets of binary inputs, capturing the truth table in a precise manner. However, the product T-norm is also clearly an example of vanishing gradient at such edge cases. In [2], the authors address the vanishing gradient in the product T-norm by proposing an adjustment to the inputs as follows (input x is replaced with $\pi_0(x)$ and ϵ is a very small number).

$$\pi_0(x) = (1 - \epsilon)x + \epsilon \tag{3.2}$$

However, this quite clearly leads to the classical functioning as being relaxed. LTN's, which use intervals instead of scalars, propose a different solution to dealing with vanishing gradients while maintaining classical functionality. In that work, they propose a threshold α that is greater than 0.5 and less than or equal to 1. For a given bound $[L, U]$, consider the semantics presented in Table 3.1.

Vanishing Derivative It is well known that, when the gradient vanishes (goes to zero), the learning process halts. When using fuzzy operators, it is thus important to understand if vanishing gradients occur, and if so at what portions of the domain (e.g., in an edge case). This topic is explored in depth in the literature on LTN's [2, 18].

Exploding Derivative Exploding gradients (i.e., very large gradients) impact the stability of the learning process. These have been primarily associated with operators used for aggregation (e.g., fuzzy versions of quantifiers) [2].

Table 3.1 Semantics of intervals used in LNN

True	False	Unknown	Inconsistent
$[L, U] \subseteq [\alpha, 1]$	$[L, U] \subseteq [0, 1 - \alpha]$	$L \in [0, 1 - \alpha]$ and $U \in [\alpha, 1]$	$U < L$

Single Passing Single passing refers to derivatives where the derivative is non-zero for only one argument. This is considered inefficient as it provides very little learning signal, yet the complete forward pass must still be computed.

Parameterization Several frameworks such as LNN and recent work on learning GAPs [12] take the step of parameterizing the operator itself, while LTN and *δILP* do not. With the LNN framework, weights are associated with the inputs to the operator, and a bias term is added. These parameters are learned through the training process and steps are taken with an activation function wrapped around the operator to ensure the weights are learned in a manner that ensures classical functionality (as per α, see Table 3.1). These parameters allow LNN's to obtain a high degree of model fit to the training data. However, parameters associated with an operator may impact interpretability (e.g., the authors of [10] point out for instance that the bias, in particular, can lead to interpretability/explainability issues). The parameterization of [12] differs in that the parameters are essentially used to learn the structure of the rule, gating certain predicates from impacting the outcome of the operator. This is discussed more in-depth in Chap. 8.

3.3.2 Fuzzy Negation and Considerations for Unit Interval Annotations

A function for fuzzy negation, $N : [0, 1] \rightarrow [0, 1]$ is generally expected to exhibit classical behavior on classical inputs in addition to adhering to the property that for an $x \leq x'$, $N(x) \geq N(x')$. A negation is *strong* if $N(N(x)) = x$ and strict if it is continuous and strictly decreasing. The term *strong negation* is also used to refer to the specific fuzzy negation where $N(x) = 1 - x$ is the most common form of negation used in the NSR literature. In both LNN and LTN, as well as the variant of *δILP* that includes negation [7], negation is not parameterized.

3.3.3 T-Norms

Triangular norms, or t-norms, are the fuzzy operator used for conjunction. T-norms are defined as functions that are commutative, associative, and enjoy monotonicity (i.e., when all but one argument is fixed, the function monotonically increases as that argument increases) and neutrality (arguments set to 1 are ignored).

Common t-norms used in NSR

Goedel t-norm:

$$T_G(\{x_1, \ldots, x_n\}) = \min_i(x_i) \tag{3.3}$$

Product (Goguen) t-norm:

$$T_{prod}(\{x_1, \ldots, x_n\}) = \prod_i x_i \tag{3.4}$$

Łukasiewicz t-norm:

$$T_{LK}(\{x_1, \ldots, x_n\}) = \max\left(1 + \sum_i (x_i - 1), 0\right) \tag{3.5}$$

LNN variant of the Łukasiewicz t-norm:

$$T_{LNN}(\{x_1, \ldots, x_n\}) = \max\left(0, \min\left(1, \beta + \sum_i w_i(x_i - 1)\right)\right) \tag{3.6}$$

Note that t-norms (Eqs. (3.3)–(3.5)) are commonly found in the literature (Goedel, Goguen/Product, and Łukasiewicz t-norms) and also summarized in the recent work of [18] as they are used in the LTN framework [2]. The last t-norm (Eq. (3.6)) is the weighted t-norm of the LNN framework [10], which is commonly used within an activation function (e.g., $relu$ or the "tailored activation function" presented in that work) and $\beta, w_1, \ldots, w_n$ are parameters. We note that there is a parameterized version of the Godel t-norm presented in [10] as well (not shown here). Note that all of these t-norms can be used as functions of the head atom in generalized annotated programs [6].

The Godel t-norm is single-passing while the product t-norm suffers from vanishing gradient on the edge cases (as described in Sect. 3.3.1—we note that the vanishing gradient problem can be addressed with a technique described in that section, though sacrificing the classical function). The Łukasiewicz t-norm also suffers from the vanishing gradient problem. Empirical studies for both LTN and δILP have shown that product t-norms generally provide the best performance. However, the literature on LNN's shows a clear preference for a parameterized Łukasiewicz t-norm, which has also been noted to outperform parameterized versions of product and Godel t-norms in empirical evaluations [11] (using the LNN framework).

3.3.4 T-Conorms

Triangular conorms, or t-conorms, are the fuzzy operator used for disjunction and are defined as functions that are commutative, associative, and enjoy monotonicity (i.e., when all but one argument is fixed, the function monotonically increases as that argument increases) and neutrality (arguments set to 0 are ignored).

We show t-conorms (Eqs. (3.7)–(3.10)) that are analogous to the t-norms of the last section. When t-norms and t-conorms are paired, some important differences arise. For example. only the Godel t-norm/conorm pair allows for distribution over the pair of operators and also maintains the identity $T_G(x, x) = S_G(x, x) = x$.

Common t-conorms used in NSR

Goedel (max) t-conorm:

$$S_G(\{x_1, \ldots, x_n\}) = \max_i(x_i) \tag{3.7}$$

Product (Probabilistic Sum) t-conorm:

$$S_{prod}(\{x_1, \ldots, x_n\}) = 1 - \prod_i (1 - x_i) \tag{3.8}$$

Łukasiewicz t-conorm:

$$S_{LK}(\{x_1, \ldots, x_n\}) = \min\left(\sum_i x_i, 1\right) \tag{3.9}$$

LNN variant of the Łukasiewicz t-conorm:

$$S_{LNN}(\{x_1, \ldots, x_n\}) = \max\left(0, \min\left(1, 1 - \beta + \sum_i (w_i \cdot x_i)\right)\right) \tag{3.10}$$

3.3.5 Fuzzy Implication

The main requirement for a fuzzy implication is to adhere to classical functionality; namely, for a fuzzy implication $I : [0, 1]^2 \rightarrow [0, 1]$, we expect that $I(0, 0) =$

$I(0,1) = I(1,1) = 1$ and $I(1,0) = 0$. There are two categories of fuzzy implication, defined as follows.

Definition 3.12 (Strong (S-) Implication) A strong fuzzy implication is defined using a t-conorm and a fuzzy negation where $x \rightarrow x' \equiv \neg x \vee x'$. Likewise, for values $x, x' \in [0, 1]$, we have $I(x, x') = S(N(x), x')$.

Definition 3.13 (Residual (R-) Implication) A residual fuzzy implication is defined using a t-norm as follows: $I(x, x') = \sup\{x'' \in [0, 1] \textit{ such that } T(x, x'') \leq x'\}$.

We note that the Łukasiewicz operators are the only ones that provide the identity $x \rightarrow x' \equiv (\neg x \vee x')$. As a result, S- and R- implications formed with Łukasiewicz t-norm/conorm and strong negation are the same. These are shown in Eqs. (3.11) and (3.12).

Łukasiewicz fuzzy implications
(Both Strong and Residual are equivalent for Łukasiewicz fuzzy implication)

Łukasiewicz fuzzy implication:

$$I_{LK}(x, x') = \min(1 - x + x', 1) \tag{3.11}$$

LNN variant the Łukasiewicz implication:

$$I_{LNN}(x, x') = \max(0, \min(1, 1 - \beta + w(1 - x) + w'x')) \tag{3.12}$$

When Goedel and product t-norms/conorms are used, the S-implications differ from the R-implications. We show S-implications in Eqs. (3.13) and (3.14), and R-implications in Eqs. (3.15) and (3.16).

Common fuzzy S-implications used in NSR
Kleene-Dienes S-implication (uses Goedel t-conorm) :

$$I_{KD}(x, x') = \max(1 - x, x') \tag{3.13}$$

Reichenbach S-implication (uses Product t-conorm) :

$$I_R(x, x') = 1 - x + x \cdot x' \tag{3.14}$$

Common fuzzy R-implications used in NSR
Goedel R-implication (uses Goedel t-norm):

$$I_G(x, x') = \begin{cases} 1 & \text{if } x \leq x' \\ x' & \text{otherwise} \end{cases} \tag{3.15}$$

Goguen R-implication (uses product t-norm):

$$I_P(x, x') = \begin{cases} 1 & \text{if } x \leq x' \\ x'/x & \text{otherwise} \end{cases} \tag{3.16}$$

Both S- and R- implications created with Goedel t-norm/conorms are single-passing, which is not surprising as both Goedel t-norm and conorm suffer from this as well. All R-implications, including Łukasiewicz, suffer from the vanishing gradient problem, though the Gougen R-implication only has this issue at the edge cases (and this is due to the same issue with the product t-norm described earlier). Meanwhile, the Kleiner-Dienes S-implication also suffers from vanishing gradients, in addition to exploding gradients in certain cases.

Empirically, [18] has shown that Goedel and Goguen R-Implications perform poorly, while the Reichenback S-implication performs well. In [18], a sigmoidal variant of the Reichenbach S-implication that, once tuned (based on hyperparameters), can outperform the Reichenbach S-implication. In [18], the Łukasiewicz variant also performed well.

We note that for frameworks such as δILP, its variants (e.g., [7, 16]) and for the NSR applications of GAPs [12] that implication is treated somewhat differently. In these studies, the implication of a rule is handled in the semantics of the language in a feed-forward manner. This is mainly due to the inductive nature of those frameworks. Hence, fuzzy implications are not explored in those papers, unlike in LTN and LNN where the logic program is known ahead of time.

3.3.6 Aggregators

In another work, t-norms and t-conorms are presented as functions with two arguments, while in this chapter we presented them as functions over sets of real values in the interval [0, 1]. In [10], the authors also take this approach while in [2] the authors refer to the versions of the operators over sets as "aggregators" as they view them as primarily handling existential and universal quantification over variables. For example, in classical logic we can view existential and universal

quantification as follows:

$$\exists X.pred(X) \equiv \bigvee_{c \in C} pred(c) \tag{3.17}$$

$$\forall X.pred(X) \equiv \bigwedge_{c \in C} pred(c) \tag{3.18}$$

Hence, the t-norms and t-conorms introduced so far can be used for this type of aggregation. From [2], we present smoothed aggregators A_{pM} and A_{pME} that can be associated with existential and universal quantification, respectively, and are defined in Eqs. (3.19) and (3.20). Additionally, a log-product aggregator was introduced in [18] that performed well empirically.

Smoothed Aggregators

Smoothed Existential Aggregator [2]:

$$A_{pM}(\{x_1, \ldots, x_n\}) = \left(\frac{1}{n}\sum_i x_i^p\right)^{1/p} \tag{3.19}$$

Smoothed Universal Aggregator [2]:

$$A_{pME}(\{x_1, \ldots, x_n\}) = 1 - \left(\frac{1}{n}\sum_i (1 - x_i)^p\right)^{1/p} \tag{3.20}$$

Log-Product (Universal) Aggregator [18]:

$$A_{\log T_P}(\{x_1, \ldots, x_n\}) = \sum_i \log(x_i) \tag{3.21}$$

These aggregators are designed to avoid single-passing and vanishing gradient issues. We also note that A_{pME} can be viewed as measuring the distance of each atom from 1, and when $p = 2$ this becomes equivalent to one minus the root mean squared error (hence called A_{RMSE} in [18]). For A_{pME}, an increase in the hyperparameter p will give a higher power to false (while such a change will give a higher power to true for A_{pM}). We note that A_{pME} is referred to as A_{GME} in [18] (for "General Mean Error").

Empirically, for universal quantification, log-product and A_{pME} ($p = 1.5$ and $p = 2$ a.k.a. A_{RMSE}) were shown to perform well in [18], outperforming Godel and Łukasiewicz t-norms (over multiple inputs) by significant margins. The authors

believe that vanishing gradients played a large role in that outcome. Existential quantification via A_{pME} ($p = 1.5$) provides the best performance, though accuracy results were much closer. The authors noted that the Godel t-conorm (max) had one of the poorest performances as an existential quantifier.

3.3.7 Product Real Logic

Based on empirical results, the authors of [2] suggest a "product real logic" consisting of strong negation (N), product t-norm (T_P), probabilistic sum t-conorm (S_P), Reichenbach S-implication (I_R), and the smoothed aggregators (A_{pM}, A_{pME}). However, they note the following edge cases where these operators have vanishing gradients. That said, the authors provide an extension to circumvent this problem (we illustrated this technique in Sect. 3.3.2). Here, we show the full technique that consists of using the following two functions to adjust the input. These are shown in expressions (3.22) and (3.23) below, where ϵ is an arbitrarily small number.

$$\pi_0(x) = (1 - \epsilon)x + \epsilon \tag{3.22}$$

$$\pi_1(x) = (1 - \epsilon)x \tag{3.23}$$

We describe the vanishing gradient problems with the specific operators in real product logic, and show how π_0, π_1 are used to modify the operator input in Table 3.2 below. They refer to this technique as *stable product semantics*.
We note that the authors of [2] point out that T_P ceases to be a t-norm when inputs are modified in this manner as it loses identity; specifically, we have:

$$T_P(\pi_0(x), \pi_0(1)) = (1 - \epsilon)a + \epsilon \neq x$$

However, empirical results reported in [2] suggest that stable product semantics improves performance.

Table 3.2 Table of real product logic operators

Operator	Vanishing gradient case	Adjustment
$N(x)$	n/a	$N(x)$
$T_P(\{x_1, \ldots, x_i, \ldots, x_n\})$	$x_i = 0$	$T_P(\{\pi_0(x_1), \ldots, \pi_0(x_n)\})$
$S_P(\{x_1, \ldots, x_i, \ldots, x_n\})$	$x_i = 1$	$S_P(\{\pi_1(x_1), \ldots, \pi_1(x_n)\})$
$I_R(x, x')$	$x = 0, x' = 1$	$I_R(\pi_0(x), \pi_1(x'))$
$A_{pM}(\{x_1, \ldots, x_i, \ldots, x_n\})$	$x_i = 0$	$A_{pM}(\{\pi_0(x_1), \ldots, \pi_0(x_n)\})$
$A_{pME}(\{x_1, \ldots, x_i, \ldots, x_n\})$	$x_i = 1$	$A_{pME}(\{\pi_1(x_1), \ldots, \pi_1(x_n)\})$

3.3.8 Conorms to Approximate the GAPs Fixpoint Operator

We note that the fixpoint operator of GAPs, if applied to scalars, will use max in place of sup. Likewise, in a framework like that of [12], where literals are assigned annotations by the interpretation (as described in this chapter), if we only modify the lower bound of an interval, we can also implement the fixpoint operator with the max operator.

In some ways, taking the maximum annotation can be thought of as a disjunction over the bodies of the rules. It is interesting to note that in δILP [3] the authors used max to aggregate the results of pairs of rules, but used probabilistic sum to aggregate the weighted activations of all pairs. This raises the question: can GAPs be correctly implemented in a neural architecture? While the results of [3] suggest this may be the case, the poor results obtained with a Goedel t-conorm aggregator in [2, 18] can give us pause. Here we propose an approximate fixpoint semantics for GAPs that can replace max with a conorm. We present approximate operator $\hat{\Gamma}_\Pi$ below.

Definition 3.14 Suppose Π is any GAP and I is an interpretation. The mapping $\hat{\Gamma}_\Pi$ that maps interpretations to interpretations is defined as

$$\hat{\Gamma}_\Pi(I)(\ell_0) = \mathbf{agg}(annoSet_{\Pi,I}(\ell_0)),$$

where:

- **agg** is a deterministic function that maps a subset of $\mathcal{T}$ to an element of $\mathcal{T}$ such that for any $\mathcal{T}' \subseteq \mathcal{T}$, $\mathbf{sup}(\mathcal{T}') \sqsubseteq \mathbf{agg}(\mathcal{T}')$.
- $annoSet$ is defined as per Definition 3.7.

As we did with Γ, we define multiple applications of $\hat{\Gamma}$ as well:

- $\hat{\Gamma}_\Pi \uparrow 0 = \Gamma_\Pi \uparrow 0$
- $\hat{\Gamma}_\Pi \uparrow (i+1) = \hat{\Gamma}_\Pi(\hat{\Gamma}_\Pi \uparrow i)$.

Finally, we can show a simple result that ensures that such an operator will indeed be sound.

Proposition 3.15 *If* $lfp(\hat{\Gamma}_\Pi)(\ell) = \mu$, *then* $\Pi \models \ell : \mu$.

Proof Note that $\hat{\Gamma}$ is also guaranteed to have a least fixed point by a mirror of the proof for the existence of such a fixed point for Γ. For any ℓ, by Definitions 3.7 and 3.14, we have $lfp(\hat{\Gamma}_\Pi)(\ell) \sqsubseteq lfp(\Gamma_\Pi)(\ell)$. The statement follows from the results of [6] and [12]. □

This result suggests that an operator such as S_P could be used in place of max, and still ensure soundness.

3.4 Chapter Conclusion

In this chapter, we reviewed generalized annotated logic [6, 12] that captures many of the logics used in NSR. Then, we studied various fuzzy operators used in NSR frameworks, and described their properties, shortcomings, and associated empirical results. We also discussed "real product logic" that consists of a collection of operators that were shown to work well for LTN and suggested an approximate fixpoint operator for generalized annotated programs that may be useful in an NSR context. In the coming chapters, we shall see how this knowledge is applied in various NSR frameworks.

References

1. Aditya, D., Mukherji, K., Balasubramanian, S., Chaudhary, A., Shakarian, P.: PyReason: Software for open world temporal logic. In: AAAI Spring Symposium (2023)
2. Badreddine, S., d'Avila Garcez, A., Serafini, L., Spranger, M.: Logic tensor networks. Artif. Intell. **303**, 103649 (2022)
3. Evans, R., Grefenstette, E.: Learning explanatory rules from noisy data. J. Artif. Int. Res. **61**(1), 1–64 (2018)
4. Hohenecker, P., Thomas, L.: Ontology reasoning with deep neural networks. J. Artif. Intell. Res. **68**, 503–540 (2020)
5. Khuller, S., Martinez, M.V., Nau, D.S., Sliva, A., Simari, G.I., Subrahmanian, V.S.: Computing most probable worlds of action probabilistic logic programs: scalable estimation for $10^{30,000}$ worlds. Ann. Math. Artif. Intell. **51**(2–4), 295–331 (2007)
6. Kifer, M., Subrahmanian, V.: Theory of generalized annotated logic programming and its applications. J. Log. Program. **12**(3&4), 335–367 (1992)
7. Krishnan, G.P., Maier, F., Ramyaa, R.: Learning rules with stratified negation in differentiable ILP. In: Advances in Programming Languages and Neurosymbolic Systems Workshop (2021)
8. Lloyd, J.W.: Foundations of Logic Programming. Springer-Verlag, New York (1987)
9. Nilsson, N.J.: Probabilistic logic. Artif. Intell. **28**(1), 71–87 (1986)
10. Riegel, R., Gray, A., Luus, F., Khan, N., Makondo, N., Akhalwaya, I.Y., Qian, H., Fagin, R., Barahona, F., Sharma, U., Ikbal, S., Karanam, H., Neelam, S., Likhyani, A., Srivastava, S.: Logical neural networks (2020). https://org.doi/10.48550/ARXIV.2006.13155. https://arxiv.org/abs/2006.13155
11. Sen, P., Carvalho, B.W.S.R.D., Riegel, R., Gray, A.: Neuro-symbolic inductive logic programming with logical neural networks. Proc. AAAI Conf. Artif. Intell. **36**(8), 8212–8219 (2022)
12. Shakarian, P., Simari, G.: Extensions to generalized annotated logic and an equivalent neural architecture. In: IEEE TransAI. IEEE (2022)
13. Shakarian, P., Parker, A., Simari, G.I., Subrahmanian, V.S.: Annotated probabilistic temporal logic. ACM Trans. Comput. Logic **12**(2) (2011)
14. Shakarian, P., Simari, G.I., Callahan, D.: Reasoning about complex networks: a logic programming approach. Theory Pract. Log. Program. **13**(4–5-Online-Supplement) (2013)
15. Shakarian, P., Simari, G.I., Schroeder, R.: MANCaLog: a logic for multi-attribute network cascades. In: Gini, M.L., Shehory, O., Ito, T., Jonker, C.M. (eds.) International conference on Autonomous Agents and Multi-Agent Systems, AAMAS, pp. 1175–1176. IFAAMAS (2013)
16. Shindo, H., Nishino, M., Yamamoto, A.: Differentiable inductive logic programming for structured examples. In: Thirty-Fifth AAAI Conference on Artificial Intelligence, pp. 5034–5041. AAAI Press (2021)

17. Van Emden, M.: Quantitative deduction and its fixpoint theory. J. Logic Program. **3**(1), 37–53 (1986)
18. van Krieken, E., Acar, E., van Harmelen, F.: Analyzing differentiable fuzzy logic operators. Artif. Intell. **302**, 103602 (2022)
19. Zadeh, L.A.: Fuzzy logic. Computer **21**(4), 83–93 (1988)

Chapter 4
LTN: Logic Tensor Networks

4.1 Introduction and Underlying Language

Logic Tensor Networks (from now on referred to as "LTNs", for short) is a recently-developed [2] NSR framework based on fuzzy differentiable logic that supports different tasks based on manipulating data and knowledge. It has been shown to effectively tackle many different problems that are central to the construction of effective intelligent systems, such as classical machine learning tasks like regression, data clustering, (multi-label) classification, embedding learning, other, less basic tasks like relational learning, and semi-supervised learning, as well as more complex ones like query answering under uncertainty. The underlying language adopted by LTNs is *Real Logic*, an infinitely-valued fuzzy logical language that is a special case of the annotated logic discussed in Chap. 3; as such, we provide a brief recap here for completeness and refer the interested reader to Chap. 3 for more details.

Real Logic assumes that the domains of interest are interpreted by tensors in the Real field, which are n-dimensional algebraic objects that include scalars (0 dimensions), vectors (1 dimension), matrices (2 dimensions), as well as higher-dimensional structures that generalize these well-known basic cases. To emphasize the close relationship to these structures, the term *grounding* is used instead of *interpretation*, which is the more commonly used term for this concept in logic.[1] Groundings are denoted with the symbol "$\mathcal{G}$", and can be intuitively understood simply as mappings from terms to tensors of real numbers,[2] and formulas to real numbers on the unit interval. The language includes the following typical elements:

[1] The creators of LTN use the word "grounding" as a reference to "symbol grounding", a concept covered further in Chap. 9.

[2] From now on, we use "tensor" to abbreviate "tensor in the Real field".

P. Shakarian et al., *Neuro Symbolic Reasoning and Learning*, SpringerBriefs in Computer Science, https://doi.org/10.1007/978-3-031-39179-8_4

- Constants denoting individuals, associated with a space of tensors of any rank. Individuals can be either *pre-defined* (i.e., a data point) or *learnable* (i.e., an embedding). Intuitively, each dimension corresponds to a feature in the space associated with the individual.
- Variables denoting sequences of individuals. Intuitively, such sequences represent the possible values that the variable can take, and they mainly contain more than one instance of the same value.
- Functions, which may correspond to any pre-defined or learnable mathematical function. Typical examples of functions are distance measures and regressors.
- Predicates, which in Real Logic are represented as mathematical functions that map an n-ary domain of individuals to a value in [0, 1] that is interpreted as a truth degree. Examples of such constructs are similarity measures and classifiers.

The language also provides the usual connectives and quantifiers. The former is modeled using fuzzy semantics: conjunction with t-norms, disjunction with t-conorms, implication with fuzzy implication, and negation with fuzzy negation. As discussed in Chap. 3, there is a wide range of such functions to choose from; though they could in principle all be used in LTNs, they are not all equivalent in terms of the computational properties they afford to the algorithms that depend on their differentiability. In terms of quantifiers, the usual existential and universal are provided, along with two less typical ones. They are all defined using operators called *aggregators*, which are symmetric and continuous functions that take n values and yield a value in [0, 1] (again, interpreted as a truth degree). Though such operators are meant to generalize classical logic quantification, the specific definition used may imply that certain properties (e.g., commutativity of quantifiers of the same type) are not guaranteed. Apart from universal and existential quantification, the language considers also *diagonal* and *guarded* quantification. The former assumes that all variables in the argument are grounded onto sequences with the same number of elements, and it quantifies by selecting tuples such that the i-th tuple contains the i-th instance of the argument (hence the name "diagonal"). The latter quantifies over a variable that satisfies a condition called the *mask*.

4.2 From Real Logic to Logic Tensor Networks

The discussion of the underlying language in Sect. 4.1 already contains the intuition behind the symbolic-subsymbolic connection that the LTN model establishes. In summary, as usual in machine learning setups, objects are represented by points embedded in a feature space, but this is generalized in LTNs by making functions and predicates learnable as well. The power then lies in that logical combinations of these basic elements can be used during the learning process in order to guide it—thus, weights are optimized both when learning basic classifiers and regressors, but also in such a way that additional constraints provided via formulas are satisfied as much as possible for all points in the feature space. There are three basic tasks that can then be defined in LTNs:

1. *Learning*, which refers to the task of generalizing from specific observations (i.e., data). This is also sometimes referred to as *inductive inference*.
2. *Reasoning*, the task of deriving what follows from a set of facts This is also typically referred to as *deductive inference*.
3. *Query Answering*, or the evaluation of the truth of a given logical expression (called a *query*). As a generalization of such Boolean queries, the task can also refer to the search for a set of objects that satisfy a given query.

Before discussing in more detail how each of these tasks is carried out in LTNs, we will first provide an overview of the different kinds of knowledge that can be represented in the model.

4.2.1 Representing Knowledge

The main connection between the domain and symbols lies in the grounding. Real Logic knowledge bases (KBs) are therefore defined by formulas in the language along with a grounding. Given this setup, there are three main forms of knowledge that can be represented; we discuss each of them in turn.

Knowledge Represented via Symbol Groundings The first type of knowledge in this category is related to the need for specifying domain groundings involves setting *boundaries* for the logical expressions. Typical examples of this are ranges for attributes such as weight (e.g., between 1 and 300 lbs.) or height (e.g., between 5 and 100 in.), or encodings of an attribute, such as color represented by 8-bit RGB values (thus, points are integer triples in $[0, 255]^3$).

Another kind of knowledge via groundings is the explicit assignment of values to symbols—so, if for instance an individual a has a set of known features in the color space described above, then we can directly define $\mathcal{G}(a) = (0, 33, 71)$. This kind of assignment is typically used for elements in training sets, where attribute values of objects are known. Other, less basic, examples include encoding of functions and predicates (e.g., fixing the use of Hamming distance as a function or a specific pre-trained classifier as a predicate). Finally, the generalization of explicit assignment is the parametric definition of groundings, where the precise values are not known but are assumed to be learnable. Typical examples of this are classifiers that take images as input and are trained to recognize specific objects (say, cats), or regressors that take as input a set of attributes of a property and predict its current market value.

Knowledge Represented via Formulas Factual propositions specifying knowledge regarding specific objects can be simply represented by logical propositions. In its simplest forms, atomic formulas may specify that an object is known to belong to a specific class; for instance, if picture pic_1 is known to contain a cat, we may have an atomic formula $cat(pic_1)$. This can be generalized easily by adding connectives:

- f_1: $\neg cat(pic_2)$ states that picture pic_2 does not contain a cat,
- f_2: $cat(pic_1) \wedge pet(pic_1)$ states that pic_1 contains both a cat and a pet, and
- f_3: $lion(pic_3) \rightarrow \neg pet(pic_3)$ says that if pic_3 contains a lion then it does not contain a pet.

(We assume that such pictures contain a single animal each.). Note that such formulas allow to incorporate of semi-supervision naturally by specifying disjunctions, such as $dog(pic_1) \vee wolf(pic_1)$. Formulas with connectives are also ideal for relational learning [8], whose goal it is to specify relationships among multiple objects. For instance, the formula

$$\neg cat(pic_4) \rightarrow \neg cat(pic_5)$$

encodes the knowledge that it is concluded that one of the pictures in the dataset (pic_4) does not contains a cat, then another (pic_5) doesn't either.

This type of knowledge can be further generalized by adding variables and quantifiers so that statements can be made about groups of elements not fixed a priori, which allows specifying general constraints that do not depend on a specific dataset. Following our pets and animals domain, we can have the following formulas:

$$\forall X \; cat(X) \rightarrow pet(X)$$

$$\forall X \; lion(X) \rightarrow \neg pet(X)$$

respectively specifying the constraints that cats are pets and that lions are not. Recall that the underlying semantics is fuzzy, so these by nature soft constraints.

Knowledge Represented via Fuzzy Semantics It is evident from the discussions above that formulas essentially combine groundings in order to approximate and generalize how truth values are manipulated in first order logic. As such, the selection of different functions for each connective and quantifier will give rise to different interpretations of the concept of satisfaction of formulas.

An interesting case is that of quantifiers, which are implemented via aggregation functions that may be parameterized so that different values of parameters cause the quantification to be done in a different manner. For instance, consider the A_{pME} function taking $a_1, \ldots, a_n \in [0, 1]$ and $p \geq 1$:

$$A_{pME}(a_1, \ldots, a_n) = 1 - \left(\frac{1}{n} \sum_{i=1}^{n} (1 - a_i)^p \right)^{\frac{1}{p}}$$

This aggregator can be seen as a "smooth minimum": the higher the value of p, the more importance true values will be given by the function. For universal quantification, this translates to a range between "for some" and "for all", generalizing

the universal ($\forall$) quantifier in classical logic. An important thing to note is that the process of defining groundings in LTNs allows different choices of operators to be made for different formulas, so such choices represent an important part of knowledge representation in formalism.

4.3 LTN Tasks

We now discuss in further detail the three main tasks associated with LTNs. We begin with satisfiability, which is at the heart of learning, then move on to querying, and finally cover reasoning. In the rest of this section, we will use the notation $\mathcal{T} = \langle \mathcal{K}, \mathcal{G}(\cdot \mid \theta), \Theta \rangle$ to denote Real Logic *theories*, comprised of a set of closed first order logic formulas $\mathcal{K}$ and a parametric grounding $\mathcal{G}(\cdot|\theta)$ for all symbols (θ is the set of parameter values and Θ the set of their possible values, also referred to as the *hypothesis space*). We refer to theories with fixed parameter values as *grounded* and denote them with $\langle \mathcal{K}, \mathcal{G}_\theta \rangle$.

4.3.1 Satisfiability and Learning

The task of learning in LTNs is inherently tied to the concept of *satisfiability*, which can be defined in this context as an optimization problem. In order to do so, we make use of a formula aggregating operator of the form $SatAgg : [0, 1]^* \rightarrow [0, 1]$. Now, learning a theory $\mathcal{T} = \langle \mathcal{K}, \mathcal{G}(\cdot \mid \theta), \Theta \rangle$ is simply the search for values in $\theta^* \in \Theta$ that maximize this expression:

$$\theta^* = \arg\max_{\theta \in \Theta} SatAgg_{\phi \in \mathcal{K}} \mathcal{G}_\theta(\phi),$$

where $SatAgg_{\phi \in \mathcal{K}} \mathcal{G}_\theta(\phi)$ simply refers to the satisfiability of the theory with respect to the aggregation operator $SatAgg$. This formulation allows learning all components, from embeddings to functions and predicates, taking into account the formulas in the theory. As described in [2], this learning process allows for the application of regularization in order to impose preferences over space Θ as usual.

4.3.2 Querying

When provided with a grounded theory, query answering is the process of checking if a certain condition holds, which is expressed via a formula called a query. There are two basic kinds of queries:

- *Truth queries*: Asks for the truth value of a formula in the language, which can be closed (i.e., with no variables, which yields a scalar) or not (i.e., it has n free variables, which yields a tensor of order n).
- *Value queries*: These are analogous to truth queries, except that they are evaluated over terms rather than formulas. The answer to such queries is the tensor that corresponds to the grounding of the term. As before, if the query has free variables then it yields a tensor of tensors as its answer.

Each of these has a *generalized* counterpart, which applies to unseen sets of objects in the domain of application.

4.3.3 Reasoning

The third and final task involves verifying if a formula is logically entailed in the theory. As in classical logic, a formula ϕ is a fuzzy consequence of a set of formulas $\mathcal{K}$ if and only if every model of $\mathcal{K}$ is also a model of ϕ. In Real Logic, the truth of a formula is decided by fixing a value $0.5 < q \leq 1$ and checking whether the truth value of the formula lies in $[q, 1]$. Unfortunately, given that there are in general infinitely many possible groundings, deciding entailment in this manner is not feasible.

There are at least two ways in which this can be addressed. The first is called *querying after learning* and is akin to brave reasoning; it involves considering only grounded theories that maximally satisfy the input theory—one way of coming up with a set of such grounded theories is to run optimizations several different times. The answer is then computed based on the obtained theories. The other option is called *proof by refutation*, which involves searching for a counterexample and, if no such counterexamples are found, then it is assumed to hold. Though the general formulation of this procedure cannot be directly applied due to null derivatives, a soft version of the constraint is described in [2].

4.4 Use Cases

We now discuss several use cases that show how LTNs can be applied to practical problems in data management and decision support systems. We begin by discussing the main details of how LTNs can address clustering problems, and then briefly discuss other use cases.

4.4.1 Basic Use Cases: Classification and Clustering

One of the simplest use cases of LTNs is one-class classification, in which the task is to decide if an input belongs not to a class of interest. In order to tackle such a

problem with LTNs, we can simply have a predicate $Class(X)$, which in this case is of arity 1 since it is simply stating that X belongs to *the* class, and two formulas:

$$(\forall X : Pos(X))\ Class(X) \tag{4.1}$$

$$(\forall X : Neg(X))\ \neg Class(X) \tag{4.2}$$

where $Pos(X)$ and $Neg(X)$ denote the set of positive and negative training examples, respectively. This can be naturally extended to more complex classification problems, such as incorporating multiple classes and/or labels, extensions that classical machine learning algorithms cannot handle easily.

Another use case that can be simply encoded in LTNs is that of clustering, which is the problem that seeks to assign one or more classes to data points, typically seeking to group together in the same class those data points that share certain characteristics. In this simple use case, we will assume that the data correspond to points on the Real plane. We then have, analogous to the classification problem, a predicate $Class(X, C)$ that encodes that data point X belongs to class C (thus, we need arity 2 here), and the following formulas encoding the basic axioms that we wish the solutions to obey:

$$\forall X\ \exists C\ Class(X, C) \tag{4.3}$$

$$\forall C\ \exists X\ Class(X, C) \tag{4.4}$$

$$\forall C\ X\ Y : dist(|X, Y) < 0.2\ (Class(X, C) \leftrightarrow Class(Y, C)) \tag{4.5}$$

$$\forall C\ X\ Y : dist(|X, Y) > 1\ \neg(Class(X, C) \wedge Class(Y, C)) \tag{4.6}$$

where "$dist(|X, Y)$" is a function that calculates the Euclidean distance between two points. Formulas (4.3) and (4.4) simply state that each point must belong to a class, and each class must contain at least one point, respectively. Formulas (4.5) and (4.6) encode the essential conditions of clustering: points that are close to each other should belong to the same class, while those that are farther apart should not, respectively.

Now, training an LTN on a dataset will simply involve maximizing an aggregator of these formulas, as discussed above, while querying can be done as desired; for instance:

- Atomic queries such as $Class(X, c_1)$ will give us truth values for all data points indicating their membership to class c_1. Another example of an atomic query, without variables, is $Class(x_8, c_2)$, which will yield the truth value associated with data point X_8 belonging to class c_2.
- More complex queries such as $Class(X, c_1) \wedge Class(X, c_2)$ allow us to calculate the data points that belong to both c_1 and c_2. Clearly, much more complex queries can also be designed.

These examples correspond to *truth* queries; as explained above, *value* queries can also be issued over terms. In this case, we might wish to calculate the value of $dist(x_3, x_7)$.

4.4.2 Other Use Cases

As discussed in [2], LTNs can also be applied to a wide variety of other problems, such as semi-supervised pattern recognition (such as in images), regression, and statistical relational learning. The latter is perhaps one of the most interesting and powerful use cases, given that it offers an alternative approach to problems that have been tackled with other models like Markov Random Fields [10], Probabilistic Soft Logic [1], and Markov Logic Networks [9]. Typical examples within this broad class of use cases include problems that involve taking into account relations between objects, such as link prediction, entity resolution, and community detection in social networks.

Other use cases for LTNs have been discussed in [5, 6], where Randomly Weighted Feature Networks (RWFNs) are introduced as a model inspired by the insect brain. In their work, the authors evaluated both LTNs and RWFNs on Semantic Image Interpretation and Visual Relationship Detection tasks, showing that the latter outperform LTNs in certain aspects such as model size.

4.5 Discussion

Logic Tensor Networks fit quite nicely within the "*Neuro_{Symbolic}* category in Kautz's taxonomy [4, 7], which contains architectures that transform symbolic rules into templates that give rise to structures within the neural component. The added value of this setup is apparent in comparison to the standard way in which machine learning typically operates, which corresponds to the first level of this taxonomy ("*Symbolic Neuro symbolic*"). In that case, only objects/individuals are tensorized, and there is no symbolic intervention until the output is produced and translated back into symbols. On the other hand, in the LTN model, all logical components are tensorized: objects, predicates, functions, and formulas.

Another way of looking at LTNs is as "*Informed learning with prior knowledge*", which is one of the compositional patterns proposed in [3] as part of their effort to derive a unifying view of neuro-symbolic approaches via the identification of a small number of elementary building blocks. In this case, LTNS are an example of a pattern in which domain knowledge is used to guide the training process via a so-called *semantic loss function*, which refers to cases in which at least part of the loss function is formulated in relation to the degree of satisfaction of a body of symbolic knowledge.

4.6 Chapter Conclusions

In this chapter, we have described the basics of the Logic Tensor Networks formalism, beginning with the underlying representation language of Real Logic, a real-valued fuzzy logic that is a special case of the general representation language discussed in Chap. 3. The symbolic-subsymbolic connection in this formalism is centered on the concept of grounding, which essentially assigns terms, and formulas to tensors in the Real field—tensors that are part of the training data then correspond to ground truth, while those that are unknown are learnable. The learning process, as usual in this type of setting, is directly dependent on having a differentiable function to optimize, and thus special care must be taken that the different kinds of fuzzy operators (connectives and aggregators) ensure this property.

LTNs have been shown to be applicable to a wide range of problems, and as such have great potential for use in more complex applications in intelligent systems. The fact that connection between inferences and the logical formulas that support them is explicit and also makes formalism more amenable to the development of mechanisms for deriving explanations.

References

1. Bach, S.H., Broecheler, M., Huang, B., Getoor, L.: Hinge-loss markov random fields and probabilistic soft logic. J. Mach. Learn. Res. **18**, 1–67 (2017)
2. Badreddine, S., d'Avila Garcez, A., Serafini, L., Spranger, M.: Logic tensor networks. Artif. Intell. **303**, 103649 (2022)
3. van Bekkum, M., de Boer, M., van Harmelen, F., Meyer-Vitali, A., Teije, A.T.: Modular design patterns for hybrid learning and reasoning systems. Appl. Intell. **51**(9), 6528–6546 (2021)
4. d'Avila Garcez, A., Lamb, L.C.: Neurosymbolic AI: the 3rd wave. CoRR abs/2012.05876 (2020). https://arxiv.org/abs/2012.05876
5. Hong, J., Pavlic, T.P.: An insect-inspired randomly, weighted neural network with random fourier features for neuro-symbolic relational learning (2021). https://doi.org/10.48550/ARXIV.2109.06663. https://arxiv.org/abs/2109.06663
6. Hong, J., Pavlic, T.P.: Representing prior knowledge using randomly, weighted feature networks for visual relationship detection (2021). https://doi.org/10.48550/ARXIV.2111.10686. https://arxiv.org/abs/2111.10686
7. Kautz, H.A.: The third AI summer: AAAI Robert S. Engelmore memorial lecture. AI Magazine **43**(1), 105–125 (2022)
8. Koller, D., Friedman, N., Džeroski, S., Sutton, C., McCallum, A., Pfeffer, A., Abbeel, P., Wong, M.F., Meek, C., Neville, J., et al.: Introduction to Statistical Relational Learning. MIT Press, Cambridge (2007)
9. Richardson, M., Domingos, P.: Markov logic networks. Mach. Learn. **62**(1), 107–136 (2006)
10. Sherman, S.: Markov random fields and Gibbs random fields. Isr. J. Math. **14**(1), 92–103 (1973)

Chapter 5
Neuro Symbolic Reasoning with Ontological Networks

5.1 Introduction and Underlying Language

Recursive Reasoning Networks, from now on referred to as RRNs for short, were recently presented in [13] as a neuro-symbolic formalism with the capability to carry out ontological reasoning via machine learning-based mechanisms. The authors' motivation is focused on the need for logic-based inference for realizing the broader goals of AI, in opposition to the way that "reasoning" is typically conceived of in machine learning approaches, which is centered on carrying out specific tasks of varying degrees of difficulty.

Ontological reasoning [2, 3, 8] is a task that is of particular interest in many approaches to developing intelligent systems; this is because it lies at the intersection of data management (which in computer science typically falls within the realm of Databases [1]) and automated reasoning (which has been addressed from many different perspectives, such as logic programming [14, 15], non-monotonic reasoning [20], and structured argumentation [5, 11, 17, 22], just to name a few). A salient property of ontology-based systems is that their knowledge bases are bipartite: on the one hand they have a set of *facts*, often called the *extensional* KB, the database, or the ABox (for *assertional* in description logics, while on the other they boast a set of *rules*, typically referred to as the *intensional* KB, the program, or the TBox (for *terminological*) in description logics. This separation of knowledge representation interests means that an agent that has incorporated a set of rules into its knowledge does not necessarily have to keep its set of facts fixed; consider our knowledge as humans of family relationships:

- A *sibling* of person X is a person who shares parents with X.
- An *uncle* (resp., *aunt*) of person X is a male (resp., female) person who is a sibling of one of X's parents.

Once we learn about how these (and many other) relationships are defined, we can reuse that knowledge to reason not only about our own families, but that of

P. Shakarian et al., *Neuro Symbolic Reasoning and Learning*, SpringerBriefs in Computer Science, https://doi.org/10.1007/978-3-031-39179-8_5

our friends, politicians, and historical figures alike. Another very powerful aspect of ontology languages in general is that of *value invention*, which refers to the capability of a rule to contain an existential quantifier in the head; a typical example in the family relationships domain is the following:

$$\forall X \; person(X) \rightarrow \exists Y \; person(Y), mother(Y, X),$$

which simply states that for every person there exists another person that is the mother of the first. This is general knowledge that can always be leveraged; though the mother of every person in the database can never be known (in a finite set of facts), we can reason with placeholder values, leading to more powerful inferences.

Language As briefly described above, ontological knowledge bases can be seen as a way to formalize information in terms of individuals, classes, and roles. Individuals are the objects of interest (constants in logical languages), while classes and roles in ontologies commonly refer to unary and binary relations; some languages also incorporate the possibility to represent and reason with n-ary relations with $n > 2$. It is common in this kind of setting to make certain simplifies assumptions, such as that of a fixed vocabulary, which means that the sets of constants and relations are fixed. Though this setup may seem weak in terms of expressive power, it is interesting to note that knowledge graphs [12] are special cases of this kind of knowledge base, where nodes correspond to individuals, labeled directed edges correspond to binary relations, and vertex labels correspond to unary relations.

For the purposes of this chapter, we consider the language adopted in [13] that is a subset of the well-known Datalog language [1]:

- *Predicates* are either unary or binary and encode relations as described above.
- No *functional symbols* present in the language.
- Rules of the forms (in the following, $\beta_1, \ldots \beta_n$ and α are *atoms*):
  - $\beta_1 \wedge \ldots \wedge \beta_n \rightarrow \alpha$: Such rules encode knowledge of the form "if β_1, β_2, ..., and β_n all hold, then α must also hold.
  - $\beta_1 \wedge \ldots \wedge \beta_n \rightarrow \perp$: This type of rule, called a *negative constraint*, simply encodes that $\beta_1, \ldots \beta_n$ cannot all hold at once (such a situation leads to a contradiction).
- *Facts*: Rules without bodies. As usual, *literals* are either facts α or negated facts, denoted with $\neg\alpha$.
- *Databases*: Finite sets of facts.

The semantics of this language is straightforward and defined as usual via Herbrand interpretations [15]. In particular, we will follow the notation used in [13] and denote logical entailment with the symbol "$\shortmid\sim$". The problem that we are interested in solving is then:

Problem statement: Given a database D, a set of rules (also referred to as the *program*) Σ, and literal ℓ, decide whether or not $D \cup \Sigma \mathrel{|\!\sim} \ell$.

We assume in the rest of this chapter that D is of variable size, which we denote with k, and that program Σ is fixed.

5.2 Recursive Reasoning Networks

The approach taken to solve the problem stated above is to derive a neural network $N[\Sigma, k]$ with binary output such that given an *arbitrary* D of size at most k and an arbitrary literal ℓ, we have that $N[\Sigma, k] = 1$ if and only if $D \cup \Sigma \mathrel{|\!\sim} \ell$. *Recursive Reasoning Networks* are a class of neural networks designed specifically for this task. In the following, we first provide an intuitive discussion of the approach and then go on to present further details.

5.2.1 RRNs: Intuitive Overview

Intuitively, the goal is to replace formal reasoning mechanisms with neural computations carried out in an RRN, which is simply a type of deep neural network. The RRN is trained relative to a particular ontology and, as its counterpart, is independent of specific facts. The vocabulary (which, as mentioned above, is typically assumed to be fixed) determines the structure of the layers in the neural network. However, it is important to note that, unlike other paradigms, the rules themselves are not provided to the training process[1] and must instead be learned from the training data.

The process of learning such an RRN is the same as that for classical recurrent neural networks:

(i) Start by randomly generating initial embeddings for all the individuals that appear in the training data; and
(ii) iterate over the training data and update embeddings of individuals involved; this considers also the possible inferences that are fired based on the data in question.

Even though this process may seem inefficient given that it scans the training data multiple times, it should be noted that this is essential for learning *reasoning chains*. It is interesting to note that if only rules are available, facts can be generated from such a set of formulas to be used instead.

[1] Paradigms that *do* require the logic program a-priori include LTN (Chap. 4), (Chap. 6), and NeurASP (Chap. 7).

Once the trained model is obtained, its application on a set of given facts is carried out in two stages: first, vector representations (embeddings) are generated for the individuals in the data, and then predictions for queries are computed solely based on such representations. Towards this end, the model leverages multi-layer perceptrons (MLPs) for both relations (between two individuals) and class memberships (of single individuals); such a step intuitively provides access to the knowledge that was encoded during the training process, allowing to make sense of complex combinations by considering unary and binary relations simultaneously.

5.2.2 RRNs: A Closer Look

Embedding of individuals is done as usual in a D-dimensional Real space $[0, 1]^d$, where d is a hyperparameter. Finding the best value for this hyperparameter will be based on the ontology's expressiveness, vocabulary size, and a number of individuals. The embedding of the i-th individual is denoted with $\mathbf{e}_i$. It is interesting to note that, in the language adopted here, databases can be seen as sets of triples:

- Atoms representing unary relations of the form $C(i)$ correspond to triples of the form $\langle i, \texttt{MemberOf}, C\rangle$, where `MemberOf` is a special symbol.
- Binary relations—atoms of the form $R(i, j)$—are encoded as $\langle i, R, j\rangle$.

Also, negated facts can be encoded with a special symbol for each relation that corresponds to its negation; for instance: $\neg C(i)$ as $\langle i, \texttt{not\_MemberOf}, C\rangle$, where `not_MemberOf` is a special predicate that will be interpreted as "¬`not_MemberOf`". The same can be done for each binary relation symbol in the vocabulary.

For each knowledge base *KB* which is associated with a set of individuals denoted *individuals*(*KB*), we have an *indicator function* of the form:

$$\mathbb{1}_{KB} : \mathit{individuals}(KB) \rightarrow \{-1, 0, 1\}^{|\mathit{classes}(KB)|}$$

Using this function, we can build vectors that summarize all the information about an individual's class memberships provided explicitly in the data:

$$[\mathbb{1}_{KB}(i)] = \begin{cases} 1, & \text{if } \langle i, \texttt{MemberOf}, C_\ell\rangle \\ -1, & \text{if } \langle i, \texttt{not\_MemberOf}, C_\ell\rangle \\ 0, & \text{otherwise} \end{cases}$$

The trained model over program Σ is then of the following form:

$$\mathit{RRN}_\Sigma(D, T) = \mathbb{P}\{T \text{ is true} \mid \langle \Sigma, D\rangle\}$$

where T is an arbitrary query of the form $\langle i, R, j \rangle$. This probability distribution represents the degree of belief that the model has regarding the *truth* of a query, which allows for training using methods like cross-entropy error. The term "truth" here is emphasized because the model is not necessarily tied to deciding $D \cup \Sigma \mid\sim \ell$ because a query might be true even if this relation does not hold (i.e., for whatever reason, the ontology fails to infer it).

RRNs are simply a special kind of *gated network*, where gates control how candidate updates are applied or not. In this model, two kinds of update gates are needed: for binary relations and for classes (unary relations). Since binary relations are in general not symmetric, four update layers are needed (subject, object, positive, and negative). The following is a one-sided update for relation P; we only show the function for updating the *subject's* embedding—those for the subject and the negated relation are analogous:

$$\mathbf{g}^{\triangleleft P}(s, o) = \sigma\left(\mathbf{V}_1^{\triangleleft P}\mathbf{e}_s + \mathbf{V}_2^{\triangleleft P}\mathbf{e}_o\right)$$

Calculation of candidate update steps is done as follows:

$$\hat{\mathbf{e}}_s^{(1)} = ReLU\left(\mathbf{W}_1^{\triangleleft P}\mathbf{e}_s + \mathbf{W}_2^{\triangleleft P}\mathbf{e}_o + \mathbf{e}_s\mathbf{e}_o^T\mathbf{w}^{\triangleleft P}\right)$$

In the above equations, $\mathbf{V}_1^{\triangleleft P}$, $\mathbf{V}_2^{\triangleleft P}$, $\mathbf{W}_1^{\triangleleft P}$, $\mathbf{W}_2^{\triangleleft P}$, and $\mathbf{w}^{\triangleleft P}$ correspond to model parameters. Also, σ is the logistic function,[2] and $ReLU(x)$ is the rectified linear unit function.[3] The remaining elements are defined as follows:

$$\hat{\mathbf{e}}_s^{(2)} = \mathbf{e}_s + \hat{\mathbf{e}}_s^{(1)} \circ \mathbf{g}^{\triangleleft P}(s, o)$$

$$\mathbf{e}_s = Update^{\triangleleft P}(s, o) = \frac{\hat{\mathbf{e}}_s^{(2)}}{||\hat{\mathbf{e}}_s^{(2)}||_2}$$

The corresponding functions for *unary* relations (which we recall are much simpler since they are inherently one-sided and we don't need to distinguish between positive and negative) are as follows:

$$\mathbf{g}(i) = \sigma\left(\mathbf{V} \cdot [\mathbf{e}_i : \mathbb{1}_{KB}(i)]\right)$$

$$\hat{\mathbf{e}}_i^{(1)} = ReLU\left(\mathbf{W} \cdot [\mathbf{e}_i : \mathbb{1}_{KB}(i)]\right)$$

As before, in the above equations $\mathbf{V}$ and $\mathbf{W}$ correspond to model parameters.

[2] $\sigma(x) = \frac{1}{(1+\exp(-x))}$.

[3] $ReLU(x) = \max\{0, x\}$.

$$\hat{\mathbf{e}}_i^{(2)} = \mathbf{e}_i + \hat{\mathbf{e}}_i^{(1)} \circ \mathbf{g}(i)$$

$$\mathbf{e}_i = Update(i) = \frac{\hat{\mathbf{e}}_i^{(2)}}{||\hat{\mathbf{e}}_i^{(2)}||_2}$$

Finally, we have a *simultaneous update operation* defined as follows:

$$\langle \mathbf{e}_s, \mathbf{e}_o \rangle = Update(\langle s, P, o \rangle) = \left\langle Update^{\triangleleft P}(s, o), Update^{P \triangleright}(s, o) \right\rangle.$$

These update functions are used in the two central algorithms: one for generating individual embeddings, and the other to train the RRN based on such values and minimize a loss function based on facts and inferences with respect to the input ontology.

The only remaining aspect to discuss is that of query answering, which can be framed as a prediction problem. Toward this end, RRNs incorporate a single multi-layer perceptron (MLP) for all classes, which is denoted with $MLP^{(classes)}$ and, given a *single* embedding, returns the probabilities associated with the corresponding individual's membership in *each class* in the vocabulary—note that the probabilities of negated atoms in this case, can be simply obtained by taking one minus the probability of the value corresponding to the non-negated atom. For relations, we have one MLP for each relation R, which is denoted with $MLP^{(R)}$. Such model receives a *pair* of embeddings and returns the probability that the corresponding ordered pair of individuals is in relation R. For the negation of the relation, we simply compute $1 - MLP^{(R)}(\mathbf{e}_1, \mathbf{e}_2)$.

5.3 Use Cases

In [13], the authors discuss a variety of use cases for RRNs, and carry out a range of empirical evaluations on both synthetic and real-world datasets. For the former, they propose taking problems based on family trees (the domain we discussed in the introduction) and geographic relationships between countries [7]. The family relations setting is interesting because it contains a non-trivial number of relation types—consider the existence of not only the most typical relations like "parent of", "sibling of", and "grandparent of", but also more complex ones like "great aunt of" or "male first cousin once removed". In the case of real-world data, the authors explore the use of DBpedia [6] (101 class types, 518 relation types), Claros [18] (33 class types, 77 relation types), and UMLS [16] (127 class types, 53 relation types). These datasets provide an adequate range of attributes not only in terms of vocabulary size but also a number of individuals, positive vs. negative class balance, and relations specified vs. inferable.

Performance measured in terms of accuracy and F-1 score was in general very good; additional experiments also showed that the formalism is tolerant to noise in the form of missing information and inconsistency, and the number of layers incorporated in order to perform updates corresponds closely to the number of inference steps necessary to compute each relation.

There are many other use cases associated with approximate inference in ontological settings, many of which have been addressed by closely-related approaches such as statistical relational learning. In the following section, we briefly discuss some of the most important works in this vein.

5.4 Related Approaches

Perhaps one of the most well-known approaches to relational learning is Markov Logic [21] (also known as Markov Logic Networks, or MLNs), a formalism proposed within the last two decades but which rests on the foundation of Markov Random Fields (or Markov Networks), a classical model for reasoning under uncertainty developed much earlier in Physics to represent joint probability distributions capable of modeling the particle interactions. MLNs are essentially first-order logic templates that, when fully grounded, yield MRFs over which queries can be answered.

In the same spirit as MLNs, Probabilistic Soft Logic (PSL) [4] is a declarative language proposed for the specification of probabilistic graphical models, where logical atoms are assigned soft truth values, dependencies are encoded via weighted first-order rules, and support is included for similarity functions and aggregation. The latter is central to many applications, such as link prediction and opinion modeling in social networks. PSL programs are shown in [4] to ground out to a special kind of MRF called Hinge-loss MRF, in which the solution space is log-concave and thus allows the corresponding optimization problem to be solved in a computationally tractable manner. Recently, Neural Probabilistic Soft Logic (NeuPSL) [19] was proposed as an NSR framework that leverages neural networks for low-level perception, integrating their outputs into a set of symbolic potential functions that are created by a PSL program. Such functions together with the neural networks define so-called *Deep Hinge-Loss Markov Random Fields*, which support scalable convex joint inference.

Finally, *deep deductive reasoners* [9] have also been recently proposed with the goal of developing effective neuro symbolic tools. The authors are interested in their application to ontological settings via the languages of RDF and description logics, as well as first-order logic. following this same motivation, in [10] the authors address the problem of developing an NSR approach to entailment in knowledge graphs in the more general setting where the graph over which entailment queries are issued may not be the same as the ones over which the training was performed.

5.5 Chapter Conclusions

In this chapter, we discussed neuro symbolic frameworks centered on ontological reasoning, a task that is closely related to reasoning in first-order logic but that has traditionally been developed with a focus on balancing expressive power and scalability. Our presentation focused mainly on the formalism of Recursive Reasoning Frameworks, an elegant and effective approach to learning neural encodings of rules in Datalog that works well in a variety of settings. RRNs correspond to level 4 of the Kautz taxonomy, where symbolic knowledge is compiled into a neural model, and as such it is not clear how interpretability and explainability can be achieved—further investigation towards this end is needed. Another avenue for future work involves expanding expressive power since Datalog is quite weak as an ontology language.

Other related approaches that we discussed here, such as NeuPSL and deep deductive reasoners, also have similar limitations in terms of understanding the connection between logical and neural structures, which is essential to the goal of interpretability and explainability.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases, vol. 8. Addison-Wesley, Reading (1995)
2. Antoniou, G., Van Harmelen, F.: A Semantic Web Primer. MIT Press, Cambridge (2004)
3. Baader, F., Horrocks, I., Lutz, C., Sattler, U.: Introduction to Description Logic. Cambridge University Press, Cambridge (2017)
4. Bach, S.H., Broecheler, M., Huang, B. and Getoor, L.: Hinge-loss Markov random fields and probabilistic soft logic. J. Mach. Learn. Res. **18**(1), 3846–3912 (2017)
5. Besnard, P., Hunter, A.: Constructing argument graphs with deductive arguments: a tutorial. Argument Comput. **5**(1), 5–30 (2014)
6. Bizer, C., Lehmann, J., Kobilarov, G., Auer, S., Becker, C., Cyganiak, R., Hellmann, S.: DBpedia – a crystallization point for the web of data. web Semant. **7**(3), 154–165 (2009)
7. Bouchard, G., Singh, S., Trouillon, T.: On approximate reasoning capabilities of low-rank vector spaces. In: 2015 AAAI Spring Symposium Series (2015)
8. Calì, A., Gottlob, G., Lukasiewicz, T.: A general Datalog-based framework for tractable query answering over ontologies. In: Proceedings of the Twenty-Eighth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, pp. 77–86 (2009)
9. Ebrahimi, M., Eberhart, A., Bianchi, F., Hitzler, P.: Towards bridging the neuro-symbolic gap: deep deductive reasoners. Appl. Intell. **51**(9), 6326–6348 (2021)
10. Ebrahimi, M., Sarker, M.K., Bianchi, F., Xie, N., Eberhart, A., Doran, D., Kim, H., Hitzler, P.: Neuro-symbolic deductive reasoning for cross-knowledge graph entailment. In: AAAI Spring Symposium: Combining Machine Learning with Knowledge Engineering (2021)
11. García, A.J., Simari, G.R.: Defeasible logic programming: DeLP-servers, contextual queries, and explanations for answers. Argument Comput. **5**(1), 63–88 (2014)
12. Hogan, A., Blomqvist, E., Cochez, M., d'Amato, C., Melo, G.d., Gutierrez, C., Kirrane, S., Gayo, J.E.L., Navigli, R., Neumaier, S., et al.: Knowledge graphs. ACM Comput. Surv. **54**(4), 1–37 (2021)

13. Hohenecker, P., Lukasiewicz, T.: Ontology reasoning with deep neural networks. J. Artif. Intell. Res. **68**, 503–540 (2020)
14. Lifschitz, V.: Answer Set Programming. Springer, Heidelberg (2019)
15. Lloyd, J.W.: Foundations of Logic Programming, 2nd edn. Springer (1987). https://doi.org/10.1007/978-3-642-83189-8
16. McCray, A.T.: An upper-level ontology for the biomedical domain. Comp. Funct. Genomics **4**(1), 80–84 (2003)
17. Modgil, S., Prakken, H.: The ASPIC+ framework for structured argumentation: a tutorial. Argument Comput. **5**(1), 31–62 (2014)
18. Nenov, Y., Piro, R., Motik, B., Horrocks, I., Wu, Z., Banerjee, J.: RDFOx: a highly-scalable RDF store. In: International Semantic Web Conference, pp. 3–20. Springer, Berlin (2015)
19. Pryor, C., Dickens, C., Augustine, E., Albalak, A., Wang, W., Getoor, L.: NeuPSL: neural probabilistic soft logic (2022). arXiv preprint. arXiv:2205.14268
20. Reiter, R.: Nonmonotonic reasoning. In: Exploring Artificial Intelligence, pp. 439–481. Elsevier, Amsterdam (1988)
21. Richardson, M., Domingos, P.: Markov logic networks. Mach. Learn. **62**(1), 107–136 (2006)
22. Toni, F.: A tutorial on assumption-based argumentation. Argument Comput. **5**(1), 89–117 (2014)

Chapter 6
LNN: Logical Neural Networks

6.1 Introduction

Logic Neural Networks (LNN's) [14] is a neuro symbolic framework created by IBM Research, leveraging the substantial expertise in both symbolic AI and optimization within that organization. Like LTN's (Chap. 4), NeurASP (Chap. 7), and other NSR frameworks [13, 17], there is an assumed existence of a logic program *a-priori*.[1] Like several of the other neuro symbolic paradigms, it uses a fuzzy logic that can be viewed as a subset of generalized annotated logic [1, 7, 16]. However, unlike the majority of other neuro symbolic frameworks,[2] where atoms are typically associated with scalar values (e.g., LTN's [2]), LNN's associate atoms with subsets of the unit interval [0, 1]. This intuition, combined with a parameterized version of fuzzy operators, leads to an explainable output that is based on a threshold parameter α. This leads to deterministic output in a manner similar to how voltage levels are treated as logical signals in physical digital circuits. With threshold α, any interval associated with a ground atom that is a subset of $[\alpha, 1]$ is treated as "true", while an interval in the subset of $[0, 1-\alpha]$ is treated as false. The twin intuitions of atoms-associated-with-intervals and parameterized fuzzy operators provide several desirable characteristics.

- *Open world reasoning.* In other frameworks, such as LTN, atoms are generally assumed to have an initial value of 0, which can be interpreted as "false." Hence, the inference process (i.e., the forward pass) in such frameworks is conducting a process that generally seeks to increase the truth of atoms. LNN's, on the other hand, initialize atoms as "uncertain", which is associated with the interval

[1] Note that this assumption is relaxed in an inductive logic programming extension for LNN's [15] which we cover in Chap. 8.

[2] Perhaps with the exception of [16] that directly works with annotated logic where annotations are a subset of the unit interval.

P. Shakarian et al., *Neuro Symbolic Reasoning and Learning*, SpringerBriefs in Computer Science, https://doi.org/10.1007/978-3-031-39179-8_6

[0, 1]. Here, the inference process will seek to tighten the bound, so rather than increasing truth, it will increase certainty, and the interval will iteratively tighten in a region around true, false, or uncertain based on the threshold.
- *Omnidirectional inference.* LNN's conduct a forward pass through what is referred to as an "upward-downward" algorithm, which is mathematically equivalent to a fixpoint operator (see Chap. 2 for a description of fixpoint operators and Chap. 3 describing a fixpoint operator for annotated logic). In doing so, LNN's support both multi-step inference as well as giving optionality in terms of which logical elements (e.g., atoms, formulae) are considered input or output during training.
- *Explainability.* Atom-to-neuron correspondence afforded by LNN's combined with the thresholding of fuzzy values provides a level of explainability not present in embedding-based approaches (e.g., see Chap. 5). Further, as the underlying logic is a subset of annotated logic, the forward pass (a deduction process) is deterministic and guaranteed to terminate in a finite number of steps.

The remainder of this chapter is organized as follows. First, we review the underlying logic and inference used in the framework (Sect. 6.2). This is followed by a discussion of the training process (Sect. 6.3). We then discuss several characteristics with respect to LNN's (Sect. 6.4) prior to concluding the chapter in Sect. 6.5.

6.2 Logic and Inference in LNNs

LNN's employ a first order logic in which logical propositions are associated with a subset of the [0, 1] interval (as opposed to scalars). This is referred to simply as a "weighted real-valued logic" in [14]. As mentioned in the introduction, a hyperparameter, $\alpha \in (0.5, 1]$ defines acceptable ranges for truth and falsehood: values in the range of $[\alpha, 1]$ are considered true and $[0, 1 - \alpha]$ considered as false (see the y-axis of Fig. 6.3 later in this chapter). This leads to the use of fuzzy operators. Interestingly, unlike LTN [2, 17] and δILP [3], LNN research has shown a clear preference for Lukasiewicz operators over product operators [14, 15], though a key difference is that the LNN work studies parameterized operators while the LTN and δILP works study non-parameterized variants. Further, as pointed out in [15], in the LNN framework the idea of returning classical output for classical input differs due to the alpha parameter. Figure 6.1 shows a comparison between truth-tables for "classical functionality" requirements for a traditional fuzzy disjunction (i.e., as described in Chap. 3) and the requirements for a real-valued disjunction used in the LNN paradigm. The key intuition is that as a range of real values such as $[\alpha, 1]$ (resp., $[0, 1 - \alpha]$) is considered "true" (resp., "false") then the results of an operator that combines such inputs should also fall into the same range. As an example, it is easy to see how one of the most popular t-norms, the product t-norm, fails in this regard. Suppose we have two atoms associated each associated with a real-valued activation that is less than 1, but equal to α, to the minimum threshold for truth.

a	b	a ∨ b
0	0	0
0	1	1
1	0	1
1	1	1

Classical functionality for fuzzy disjunction

a	b	a ∨ b
[0,1-α]	[0,1-α]	[0,1-α]
[0,α)	[α,1]	[α,1]
[α,1]	[0,α)	[α,1]
[α,1]	[α,1]	[α,1]

Classical functionality for LNN disjunction

Fig. 6.1 Comparison between the ideas of classical functionality for a fuzzy disjunction (t-conorm) and a real-valued disjunction used in LNN

The resulting product t-norm is then associated with a value of α^2 that clearly falls outside of the range of truth.

In order to ensure classical function with respect to the α parameter is adhered to, as well as to fit the data, the operators are parameterized. Later in this chapter (Sect. 6.3) we shall discuss approaches to constraining weight values to ensure this type of classical functionality. One method to provide this control is by leveraging an activation function in conjunction with the parameterized operator. This extends the idea of a fuzzy operator described in Chap. 3. As such is the case, in this chapter, we illustrate the functions with a generic activation function, denoted f, shown in Eqs. (6.2)–(6.4). We note that these reduce to the variants shown in Chap. 3 when $f = \mathsf{relu1}$, which essentially performs clipping.[3] We note that LNN's use *strong negation*, unchanged from Chap. 3 where $N(x) = 1 - x$ (i.e., not parameterized).

Parameterized Lukasiewicz-style Fuzzy Operators used in LNN's with a generic activation function (f)
Strong Negation (used in LNN without parameters):

$$N(x) = 1 - x \tag{6.1}$$

LNN variant of the Lukasiewicz t-norm w. activation function:

$$T_{LNN}(\{x_1, \ldots, x_n\}) = f\left(\beta + \sum_i w_i(x_i - 1)\right) \tag{6.2}$$

LNN variant of the Lukasiewicz t-conorm w. activation function:

(continued)

[3] $\mathsf{relu1}(x) = \max(0, \min(1, x))$. This was originally introduced in [10] and discussed with respect to LNN's in [15].

$$S_{LNN}(\{x_1, \ldots, x_n\}) = f\left(1 - \beta + \sum_i (w_i \cdot x_i)\right) \tag{6.3}$$

LNN variant the Lukasiewicz implication w. activation function:

$$I_{LNN}(x_1, x_2) = f(1 - \beta + w_1(1 - x_1) + w_2 x_2) \tag{6.4}$$

We note that the logic program—the set of logical sentences of interest—is known *a-priori* in the LNN framework. As such, a syntax tree can be produced that decomposes these sentences into logical components and ultimately atomic propositions. As the framework has a direct correspondence between neurons and atoms, this syntax tree directly serves as the neural architecture used during the backpropagation process. We show an example of a syntax tree in Fig. 6.2. In that example, we have a logic program Π_{grass} defined as follows:

$$\begin{aligned}\Pi_{grass} = \{\ &\mathsf{hasRained}() \vee \mathsf{sprinklerUsed}() \rightarrow \mathsf{grassWet},\\ &\neg\mathsf{hasRained}() \wedge \mathsf{thunderHeard}() \rightarrow \neg\mathsf{grassWet}\ \}\end{aligned}$$

We see in Fig. 6.2 how these sentences are decomposed.

In inference, this syntax tree is also used in an "upward-downward" algorithm that iteratively tightens the bounds via proof rules that are created through the use of the bounds associated with nodes in the syntax tree. The upward pass uses the truth bounds starting at the atoms to compute the truth of formulas while the downward pass uses the truth bounds on the formulas to tighten the bounds on subformulas and atoms. The use of strong negation aids in this process. For a given formula ϕ with associated truth bounds $[L_\phi, U_\phi]$, we have the following relationships:

$$L_{\neg\phi} \geq \neg U_\phi = 1 - U_\phi \tag{6.5}$$

$$U_{\neg\phi} \leq \neg L_\phi = 1 - U_\phi \tag{6.6}$$

$$L_\phi \geq \neg U_\phi = 1 - U_{\neg\phi} \tag{6.7}$$

$$U_\phi \leq \neg L_\phi = 1 - L_{\neg\phi} \tag{6.8}$$

Note that any proof rule can be represented as a rule in annotated logic as per Chap. 3; also, note that the full syntax of [7] directly supports all syntactic elements of the underlying logic of LNN's.

The upward-downward algorithm is shown to converge in a finite number of iterations in [14]; however tighter, polynomial bounds on the number of iterations are likely due to the results of [16] and the fact that annotated logic captures the logic

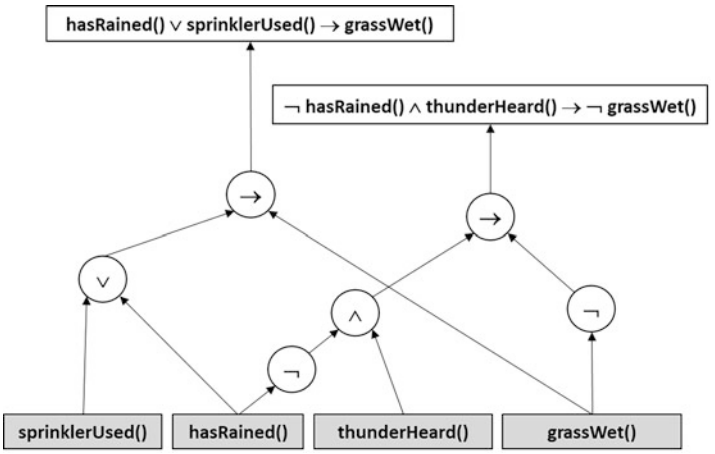


Fig. 6.2 Syntax tree for a logic program that also serves as a neural architecture for learning

of LNN's. The upward-downward algorithm functions as both a deduction process and as the forward pass in learning.

6.3 Training LNNs

As mentioned in Sect. 6.2, the syntax tree (e.g., Fig. 6.2) also serves as the neural architecture for learning the weights associated with the operators (the w_i and β parameters in Eqs. (6.2)–(6.4)). A key challenge that the designers of the LNN architecture sought to address is to maintain interpretability while giving flexibility in the parameters to fit the data. A key idea is to restrict the values of the parameters in order to ensure the classical behavior of the operators. Again, here "classical behavior" must consider the α threshold (this differs from classical behavior in standard fuzzy logic—see Fig. 6.1 for an example comparison). One way to think of this in a generic sense is as a constrained optimization problem that restricts the values of the parameters. However, framing learning as an optimization problem leads to several technical difficulties:

- Parameter updates would require expensive constraint satisfaction operations.
- The operation would require slack parameters, limiting explainability and possibly leading to over-fitting.
- The β parameter when learned for each neuron is difficult to interpret from an explainability perspective.

Tailored Activation Function One method to avoid learning by a constrained optimization problem is to use a specially-engineered activation function, referred to as the "tailored activation function" in [14]. Below we show such an activation

function for use with disjunction (S_{LNN} as shown in Eq. (6.3), but assuming two arguments).

$$f(x) = \begin{cases} x \cdot (1-\alpha)/x_F & \text{if } 0 \leq x \leq x_F \\ (x - x_F) \cdot (2\alpha - 1)/(x_T - x_F) + 1 - \alpha & \text{if } x_F < x \leq x_T \\ (x - x_T) \cdot (1-\alpha)/(x_{max} - x_T) + \alpha & \text{if } x_T < x \leq x_{max} \end{cases} \tag{6.9}$$

$$x_F = \sum_i w_i(1-\alpha) \qquad x_T = w_{max} \cdot \alpha \qquad x_{max} = \sum_i w_i$$

This activation function avoids the need for constraints as it ensures that classical inputs lead to classical outputs. It also assumes that $\beta = 1$ for all operators resolves a lingering explainability issue associated with bias parameters. We show an example of the tailored activation function applied to a sample input of two real-valued atoms connected by a real-valued operator in Fig. 6.3, where two different settings of α are shown.

Loss Function and Logical Consistency LNN's do not make the assumption that the input logic program is consistent, and prior to training, it is really not relevant as the consistency of the program is inherently tied to the parameters, as the parameters directly affect how the operators function. While the original work on LNN's hypothesizes the ability to enforce consistency constraints during the training process (e.g., disallowing assignments of weights that cause inconsistency), at the time of this writing such a technique has not been implemented and empirically evaluated. While we note that for certain application requirements (e.g., safety, legal requirements, etc.) hard constraints may be necessary, for other user cases, one may only need to ensure a certain degree of consistency. This consistency is measured and incorporated into the loss function, which is shown below (Expression (6.10)).

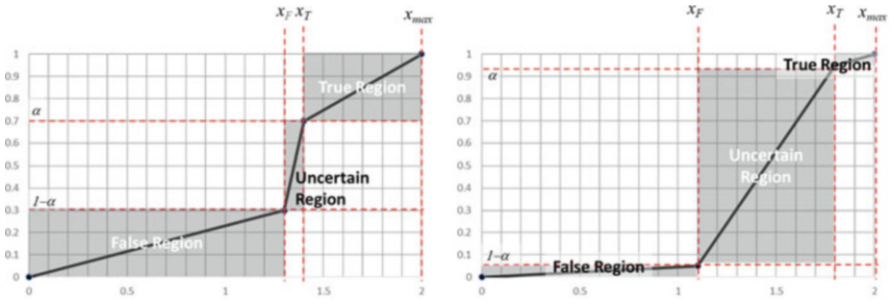


Fig. 6.3 Example activation functions used in LNN's

$$\min_{B,W} E(B, W) + \sum_{k \in N} \max(0, L_{B,W,k} - U_{B,W,k}) \tag{6.10}$$

In this loss function, B and W are vectors for biases and weights, respectively. The function E is a loss function for the problem (e.g., MSE, cross-entropy, etc.). Here, N is the set of neurons. Note that each neuron is associated with an atom or formula, hence an $[L, U]$ bound. When $L > U$, the neuron is associated with an inconsistent logical construct. Hence, the term $\sum_{k \in N} \max(0, L_{B,W,k} - U_{B,W,k})$ provides a total of the amount of such inconsistency (as it will equal 0 for each consistent neuron).

6.4 Discussion

In this section, we discuss several aspects of LNN's and consider them in the overall context of the neuro symbolic advances covered in this volume.

Perceptual Integration and Symbol Grounding As LNN's have a direct correspondence between neurons and atoms, unlike the vector representations used in LTN's (Chap. 4), there is an implicit assumption that symbols are properly defined prior to reasoning at the LNN layers. Therefore, if one adds LNN layers on top of a perceptual model (e.g., a convolutional neural network) and performs end-to-end training, it may become unclear what the underlying symbols represent. This is an instance of the *symbol grounding problem* [4], the problem of properly defining symbols. Some would argue that LTN's have some of this capability "built in" by virtue of their vector representation, which is parameterized, something that to-date is absent from the LNN literature.[4] This is striking when one considers presentations of evaluations in various other neuro symbolic frameworks that incorporate perception experiments such as LTN [5], NeurASP [19], δILP [3], SATNet [18], and NeuPSL [13]. To date, there have been recorded LNN experiments for problems such as knowledge graph reasoning [11], natural language processing [6], and reinforcement learning tasks [9]. However, as of the time of this writing, there are little or no published experimental results where LNN's are used in conjunction with a perceptual system, even in cases where a pre-trained network is used for perception. The integration of LNN's with perceptual systems and the associated symbol grounding problem is an important area for future LNN research.

Refinement and Consistency As discussed earlier, LNN's do not ensure consistency of the parameterized logic program resulting from the training process, only that the resulting parameters have been regularized with respect to consistency. In many real-world applications, such as robotics and aerospace, hard constraints may be desirable to ensure safety or to provide a specification when used as a component of a larger system. The designers of LNN's do hypothesize a potential solution to

[4] At the time of this writing in late 2022.

this issue, which is similar to the idea of logical refinement and has appeared in other contexts. In one such case, in a framework called STLNet, logical refinement is used to update the output of a symbol-producing black-box neural model to ensure its output adheres to a temporal logic specification [12]. Similar techniques for LNN's can potentially allow for the enforcement of hard consistency requirements. One step toward this direction from the reinforcement learning community is the work of [8]. Here, the authors introduce the idea of LNN Shielding to enforce a stronger consistency constraint in what amounts to a refinement process to avoid certain actions that are inconsistent with logical specifications. In the process, called "LNN Shielding," they are only using the real valued logic introduced in [14]. They also build on this idea with "LNN Guiding" where alternative actions are selected based on an LNN model.

6.5 Chapter Conclusion

In this chapter, we reviewed LNN's, a framework where the logic program is known *a-priori* like several of the other paradigms discussed in this book (e.g., LTN, NeurASP). However, a key difference is that with LNN's, the fuzzy operators themselves are parameterized, as opposed to parameters occurring in lower layers. This, combined with the use of confidence intervals (as opposed to scalars) gives rise to several desirable properties such as open world reasoning, bidirectional inference, and a degree of explainability. However, like many of the frameworks in this volume, there are still many open questions concerning LNN's, such as the ability to enforce hard consistency constraints and interaction with perceptual systems.

References

1. Aditya, D., Mukherji, K., Balasubramanian, S., Chaudhary, A., Shakarian, P.: PyReason: software for open world temporal logic. In: AAAI Spring Symposium (2023)
2. Badreddine, S., d'Avila Garcez, A., Serafini, L., Spranger, M.: Logic tensor networks. Artif. Intell. **303**, 103649 (2022)
3. Evans, R., Grefenstette, E.: Learning explanatory rules from noisy data. J. Artif. Int. Res. **61**(1), 1–64 (2018)
4. Harnad, S.: The symbol grounding problem. Physica D **42**(1–3), 335–346 (1990)
5. Hong, J., Pavlic, T.: Representing prior knowledge using randomly, weighted feature networks for visual relationship detection. In: Combining Learning and Reasoning: Programming Languages, Formalisms, and Representations (2022)
6. Jiang, H., Gurajada, S., Lu, Q., Neelam, S., Popa, L., Sen, P., Li, Y., Gray, A.: LNN-EL: a neuro-symbolic approach to short-text entity linking. In: Proceedings of ACL, pp. 775–787. Association for Computational Linguistics, Online (2021)
7. Kifer, M., Subrahmanian, V.: Theory of generalized annotated logic programming and its applications. J. Log. Program. **12**(3&4), 335–367 (1992)
8. Kimura, D., Chaudhury, S., Wachi, A., Kohita, R., Munawar, A., Tatsubori, M., Gray, A.: Reinforcement learning with external knowledge by using logical neural networks. CoRR abs/2103.02363 (2021). https://arxiv.org/abs/2103.02363

9. Kimura, D., Ono, M., Chaudhury, S., Kohita, R., Wachi, A., Agravante, D.J., Tatsubori, M., Munawar, A., Gray, A.: Neuro-symbolic reinforcement learning with first-order logic. In: Proceedings of EMNLP, pp. 3505–3511. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic (2021)
10. Krizhevsky, A.: Convolutional deep belief networks on CIFAR-10 (2010). https://api.semanticscholar.org/CorpusID:15295567
11. Luus, F.P.S., Sen, P., Kapanipathi, P., Riegel, R., Makondo, N., Lebese, T., Gray, A.G.: Logic embeddings for complex query answering. CoRR abs/2103.00418 (2021). https://arxiv.org/abs/2103.00418
12. Ma, M., Gao, J., Feng, L., Stankovic, J.A.: STLnet: signal temporal logic enforced multivariate recurrent neural networks. In: 34th Conference on Neural Information Processing Systems (NeurIPS 2020)
13. Pryor, C., Dickens, C., Augustine, E., Albalak, A., Wang, W., Getoor, L.: NeuPSL: neural probabilistic soft logic (2022). arXiv preprint. arXiv:2205.14268
14. Riegel, R., Gray, A., Luus, F., Khan, N., Makondo, N., Akhalwaya, I.Y., Qian, H., Fagin, R., Barahona, F., Sharma, U., Ikbal, S., Karanam, H., Neelam, S., Likhyani, A., Srivastava, S.: Logical neural network, arXiv, eprint 2006.13155 (2020)
15. Sen, P., Carvalho, B.W.S.R.d., Riegel, R., Gray, A.: Neuro-symbolic inductive logic programming with logical neural networks. In: Proceedings of AAAI, pp. 8212–8219 (2022)
16. Shakarian, P., Simari, G.: Extensions to generalized annotated logic and an equivalent neural architecture. In: IEEE TransAI. IEEE (2022)
17. van Krieken, E., Acar, E., van Harmelen, F.: Analyzing differentiable fuzzy logic operators. Artif. Intell. **302**, 103602 (2022)
18. Wang, P., Donti, P.L., Wilder, B., Kolter, J.Z.: SATNet: bridging deep learning and logical reasoning using a differentiable satisfiability solver. In: Chaudhuri, K., Salakhutdinov, R. (eds.) Proceedings of ICML, pp. 6545–6554. PMLR (2019)
19. Yang, Z., Ishay, A., Lee, J.: NeurASP: embracing neural networks into answer set programming. In: Bessiere, C. (ed.) Proceedings of IJCAI, pp. 1755–1762. International Joint Conferences on Artificial Intelligence Organization (2020)

Chapter 7
NeurASP

7.1 Introduction

One motivation behind developing neuro-symbolic reasoning methods and architectures is to incorporate explicit knowledge about a domain while learning a model. This knowledge can be incorporated into the model in multiple ways: (a) using distillation [9, 10], (b) encoding the knowledge as part of the loss function [15], and (c) having an explicit symbolic representation, reasoning and filtering layer using a differentiable logic [12–14]. In this chapter, we explore the third approach using a differentiable extension of the declarative problem solving language ASP (Answer Set Programming) that is inspired by Logic Programming. In this chapter, we describe the basic concepts and syntax of ASP (Sect. 7.2), the associated differentiable extension known as "NeurASP" (Sect. 7.3), including semantics, inference, and learning procedures.

7.2 ASP: Answer Set Programming

An ASP program is a collection of rules of the form:

$$a_o \leftarrow a_1, \ldots, a_m, \mathsf{not}\ a_{m+1}, \ldots \mathsf{not}\ a_n.$$

where a_is are atoms in the sense of predicate logic. Intuitively, the meaning of the above rule is that if $a_1, \ldots, a_m$ are true and $a_{m+1}, \ldots, a_n$ can be safely assumed to be false, then a_0 must be true. If the right-hand side of a rule is empty (i.e., $m = n = 0$) then we refer to it as a fact. The semantics of ASP programs that do not have not in their rules are defined by their unique least model based on subset order.

P. Shakarian et al., *Neuro Symbolic Reasoning and Learning*, SpringerBriefs in Computer Science, https://doi.org/10.1007/978-3-031-39179-8_7

The following two ASP rules express the ancestor relation *anc* in terms of the parent relation *par*.

$$anc(X, Y) \leftarrow par(X, Y). \qquad anc(X, Y) \leftarrow par(X, Z), anc(Z, Y).$$

Now consider the ASP program Π_1 consisting of the following facts and the above two rules:

$$par(a, b).par(b, c).par(c, d).par(e, f).$$

Using the above program, we will not only conclude the given facts:

$$par(a, b).par(b, c).par(c, d).par(e, f).$$

but, we will also conclude the following facts, as they are present in the least model of the program:

$$anc(a, b).anc(b, c).anc(c, d).anc(a, c).anc(b, d).anc(a, d).anc(e, f).$$

The above rules about ancestors in terms of parents are an example of transitive closure. It should be noted that, in general, transitive closure can not be expressed using first-order predicate logic. See Example 54 of [2] for more on this.

A key feature of ASP is its use of the default negation operator not . Its use in the body of rules makes it easy to express non-monotonic reasoning aspects such as normative and default statements. For example, ASP rules can be used to express statements such as "Normally crows are black. However, albino crows, which are crows, are white." in the following way:

$$black(X) \leftarrow crow(X), \text{not } ab(X). \qquad ab(X) \leftarrow albino\_crow(X).$$
$$crow(X) \leftarrow albino\_crow(X). \qquad white(X) \leftarrow albino\_crow(X).$$

So if we add the facts *crow*(*banjo*) and *albino_crow*(*willow*) to the above rules to form the ASP program Π_2, then from that program we will be able to conclude *black*(*banjo*) and *white*(*willow*). Moreover, even though we will be able to conclude *crow*(*willow*) from Π_2, we will not be able to conclude *black*(*willow*).

When the dependency graph of an ASP program has a loop with not in it then the program may no longer have the least model and may have multiple minimal models or none at all. For example, consider the program Π_3 given by the following rules:

$$p \leftarrow a. \qquad p \leftarrow b.$$
$$a \leftarrow \text{not } b. \qquad b \leftarrow \text{not } a.$$

The above program has two minimal models: $\{p, a\}$, and $\{p, b\}$. Based on the minimal models, one can infer $a \vee b$ from this program. However, minimal models do not always capture the intuitive meaning of an ASP program. For example, consider the program Π_4 consisting of the following single rule:

$$a \leftarrow \mathsf{not}\ b.$$

This program has two minimal models $\{a\}$, and $\{b\}$. However, the second one is not meaningful as there is no rule that makes b to be true. Thus, to define the semantics of ASP programs a more discriminative notion of *stable models* [8] has been proposed.

Given a ground ASP program Π a set of ground atoms S is a stable model of Π iff S is the unique least model of the program Π^S obtained from Π by (a) removing all rules from Π that have a $\mathsf{not}\ f$ in its body where $f \in S$, and (b) removing all not literals from the body of the rest of the rules. Stable models of non-ground ASP programs are defined as the stable models of their ground version. For example, consider the program Π_5 consisting of the following two rules:

$$q \leftarrow \mathsf{not}\ p.$$
$$p \leftarrow \quad p.$$

Consider the set $\{q\}$. $\Pi_5^{\{q\}}$ consists of the following two rules:

$$q \leftarrow.$$
$$p \leftarrow p.$$

and its least model is $\{q\}$. Thus $\{q\}$ is a stable model of Π_5. Note that $\{p\}$ is not a stable model of Π_5 as $\Pi_5^{\{p\}}$ consists of the following rule:

$$p \leftarrow p.$$

and its least model is $\{\}$, which is different from $\{p\}$. Note that $\{p\}$ is a minimal model of Π_5 and $\{p\}$ is also a supported model of p supported by the rule $p \leftarrow p$. Regardless, it is not a stable model.

A minor syntactic extension of ASP allows rules where the head of the rule can be empty. In that case, models are not allowed to satisfy the body of any such rule.

In systems implementing ASP, such as Clingo [7], several syntactic enhancements are made so as to make the programs more concise to write. For example, if we have a multi-valued fluent f then atoms can be of the form $f = v$. Another syntactic feature is choice rules. Using choice rules, Π_3 can be written as:

$$p \leftarrow a.$$
$$p \leftarrow b.$$
$$1\{a; b\}1.$$

The last rule above can also be written as:

$$\{a; b\} = 1.$$

In general, a choice rule of the following form intuitively means that in each stable model at least u and at most v elements of the set $\{a_1; a_2; \ldots; a_n\}$ must be true.

$$u\{a_1; a_2; \ldots; a_n\}v.$$

Suppose we have an image d and we would like to express that in each stable model, d takes a unique value from 0 to 9. This can be written as the rule:

$$\{val(d) = 0; \ldots; val(d) = 9\} = 1.$$

There have been several proposals to incorporate probabilities into the ASP framework such as P-log [3], Prob-log [5] and LPMLN [11]. The formulation of NeurASP [14] that combines neural networks with ASP uses a probability framework close to the one in LPMLN.

7.3 NeurASP

A NeurASP [14] program combines a set of neural networks with an ASP program. Each neural network is assigned a name, and it takes an input and assigns values (with assigned probabilities) to one or more of its (output) attributes. For example, consider a neural network for MNIST. This neural network will take an image as an input and assign probabilities to the single output attribute *digit* with respect to the values 0 ... 9. In the NeurASP framework, this neural network denoted by M_{mnist_nn} will be specified by the following neural atom:

$$nn(mnist\_nn, mnist\_input, 1, \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}).$$

where, *mnist_nn* is the name of the neural network, *mnist_input* refers to an input, 1 denotes the number of attributes, and $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ refers to the value the attribute can take. The neural network M_{mnist_nn} will then define a probability function denoted by $M_{mnist\_nn}(mnist\_input)[1, v]$ such that $\sum_{v \in \{0,\ldots,9\}} M_{mnist\_nn}(mnist\_input)[1, v] = 1$.

Now consider a neural network M_{sudoku_nn} that takes an image of the initial state of a Sudoku game and assigns probabilities to the 81 output attributes, each expressing what is the value in one of the 81 squares, with respect to the values *empty*, and 1 ... 9. In the NeurASP framework this neural network M_{sudoku_nn} will be specified by the following neural atom:

$$nn(sudoku\_nn, sudoku\_input, 81, \{empty, 1, 2, 3, 4, 5, 6, 7, 8, 9\}).$$

where, *sudoku_nn* is the name of the neural network, *sudoku_input* refers to an input, 81 denotes the number of attributes, and $\{empty, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ refers to the value the attributes can take. The neural network M_{sudoku_nn} will then define a probability function denoted by $M_{sudoku\_nn}(sudoku\_input)[i, v]$ such that for values $1 \leq i \leq 81$, $\sum_{v \in \{empty,1,\ldots,9\}} M_{sudoku\_nn}(sudoku\_input)[i, v] = 1$.

In general, a neural network M_{name} is specified by a neural atom of the form[1]

$$nn(name, input, no\_attr, value\_set)$$

where *name* is the name of the neural network, *input* is the input to the neural network, *no_attr* refers to the number of attributes that the neural network is trying to assign values, and *value_set* is a set of values that each of the attributes can take. The neural network M_{name} then defines a probability function denoted by $M_{name}(input)[no\_attr, v]$ such that for $1 \leq i \leq no\_attr$,

$$\sum_{v \in value\_set} M_{name}(input)[i, v] = 1.$$

Here, for simplicity, the assumption has been made that each of the (output) attributes of a particular neural network take a value from a common set *value_set*.

7.3.1 Semantics

Given a set of neural atoms denoted together as Π_{nn}, by $ASP(\Pi_{nn})$ we denote the set of choice rules obtained by converting each neural atom

$$nn(name, input, no\_attr, \{v_1, \ldots, v_n\})$$

in Π_{nn} to the following set of choice rules:

[1] Our representation is slightly different from the representation in the original paper [14]. There the representation will be $nn(name(no\_attr, input), value\_set)$, with the value_set given in square brackets.

$$\{name_i(input) = v_1; \ldots, name_i(input) = v_n\} = 1, \text{for } i \in \{1, \ldots, no\_attr\}.$$

We use σ^{nn} to denote the set of atoms $name_i(input) = v_j$ obtained in the above conversion. A NeurASP program Π is a pair consisting of a set of neural atoms denoted together as Π_{nn} and an ASP program Π_{asp} with the stipulation that atoms of the form $name_i(input) = v_j$ obtained from the neural atoms in Π_{nn} do not appear in the head of the rules of Π_{asp}, thus layering the NeurASP program to two layers, where the ASP layer is computed on top of the neural layer. The stable models of a NeurASP program $\Pi = \langle \Pi_{nn}, \Pi_{asp} \rangle$ are the stable models of $ASP(\Pi_{nn}) \cup \Pi_{asp}$.

Given a NeurASP program Π, the probability of each atom of the form $name_i(input) = v_j$ denoted by $P_\Pi(name_i(input) = v_j)$ is defined as:

$$P_\Pi(name_i(input) = v_j) = M_{name}(input)[i, v_j]$$

The (unnormalized) probability[2] of a stable model S of Π, denoted by $P_{u\_\Pi}(S)$ is the product of the probabilities of all atoms in $S \cap \sigma^{nn}$ divided by the number of stable models that have the same atoms. That is,

$$P_{u\_\Pi}(S) = \frac{\Pi_{c=v \in S \cap \sigma^{nn}} P_\Pi(c = v)}{\text{Number of stable models of } \Pi \text{ that have the same } S \cap \sigma^{nn}}$$

The division by the number of stable models is necessary because the same set of atoms $S \cap \sigma^{nn}$ may lead to multiple stable models. In that case, we distribute the probability equally across those stable models. Next, we need to account for the possibility that for some combination of atoms, there may not be any stable models. Accounting for that, we define the (normalized) probability of a stable model S of Π, denoted by $P_\Pi(S)$ as follows:

$$P_\Pi(S) = \frac{P_{u\_\Pi}(S)}{\sum_{I \text{ is a stable model of } \Pi} P_{u\_\Pi}(I)}$$

The probability of an observation formula O, denoted by $P_\Pi(O)$ is given as:

$$P_\Pi(O) = \sum_{S \models O} P_\Pi(S)$$

7.3.2 Inference in NeurASP

Recall that one of our motivations behind developing neuro-symbolic reasoning methods and architectures was to incorporate explicit knowledge about a domain

[2] For efficiency purposes, in implementations, this normalization often is not done.

while learning a model. In that context let us consider models that learn to recognize handwritten digits. While there is less of an issue of explicit knowledge when there is only one digit to recognize, knowledge could play an important role when there are multiple digits. For example, consider the two digits representing a number between 1 to 16.

Reading a Number Between 1 to 16 To read a number between 1 to 16 that is made of two numerals, we can use a single neural network M_{num}, but have two ground neural atoms, such as the following.

$$nn(num, l, 1, \{blank, 0, \ldots, 9\}),$$

$$nn(num, r, 1, \{blank, 0, \ldots, 9\}).$$

One need not list the ground neural atoms, One can use an ASP program that has a unique stable model which gives us the set of neural atoms. For example, instead of the above two ground neural atoms, we can use a predicate *img*, and have the facts: $img(l)$. and $img(r)$.; and the following rule:

$$nn(num, X, 1, \{blank, 0, \ldots, 9\}) \leftarrow img(X).$$

Knowing that the two numerals that the model is reading represent a number between 1 to 16, we can express that knowledge as follows: (a) the image l is either a blank or a one and (b) the image r is (i) between 1 to 9, if the image l is a blank, and (ii) between 0 to 6, if the image l is a one. This is expressed as the ASP program in Fig. 7.1. Here, even if the neural network originally had assigned a higher probability to $num_1(r) = 8$, the ASP program will eliminate that possibility in the answer sets where $num_1(l) = 1$.

Solving the Kubok-16 Puzzle Figure 7.2 illustrates a Kubok-16 puzzle instance with X,Y co-ordinates added to it. The X, Y co-ordinates are not part of the puzzle but will help illustrate our solution. The goal of the Kubok-16 puzzle is to fill in the empty squares with numbers from 1 to 16 so that no number is repeated among the 16 squares and so that the sum of the numbers in each row and column matches with the row and column sum given in the top of each column and left of each row.

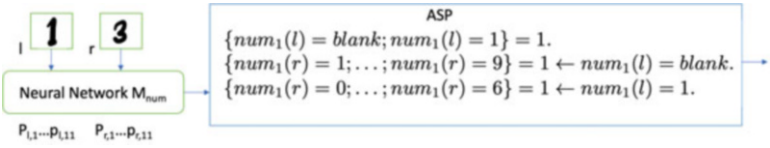


Fig. 7.1 A NeurASP program illustration showing two numeral inputs, a neural network whose output feeds into an ASP program

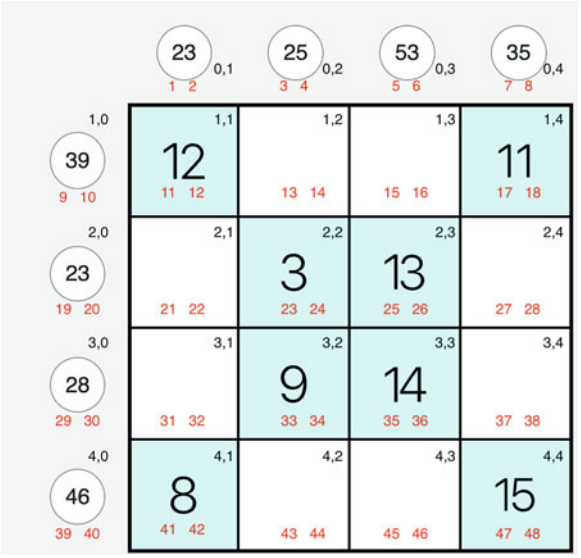


Fig. 7.2 A Kubok-16 puzzle instance with X,Y co-ordinates added to it. The 48 output attributes are also numbered from 1 to 48

We now present a NeurASP program that can solve a Kubok-16 puzzle instance given as an image. First a neural network M_{kubok} will take an image of the initial state of a Kubok-16 puzzle and assigns probabilities to 11 possible values to each of the 48 output attributes. The 48 attributes correspond to two sub-images for each of the 16 squares and the 8 row and column sums. For the 16 squares, the first sub-image is either a blank, or a one, and the second sub-image is (i) either a blank, or a number between 1 to 9 if the first sub-image is a blank, and (ii) between 0 to 6, if the first sub-image is a one. For the 8 rows and columns, the first sub-image is between 1 to 5, and the second sub-image is (i) a number between 0 to 8 if the first sub-image is a five, and (ii) between 0 to 9 otherwise. In the NeurASP framework this neural network M_{kubok} can be specified by the following neural atom:

$$nn(kubok, img, 48, \{blank, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}).$$

where, *kubok* is the name of the neural network, *img* refers to an input, 48 denotes the number of attributes, and $\{blank, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ refers to the 11 values the attributes can take. The neural network M_{kubok} will then define a probability function denoted by $M_{kubok}(img)[i, v]$ such that for $1 \leq i \leq 48$, $\sum_{v \in \{blank,0,\ldots,9\}} M_{kubok}(img)[i, v] = 1$.

The above neural atom would be translated to the following ASP rules:

$\{kubok_i(img) = blank; kubok_i(img) = 0; \ldots kubok_i(img) = 9\} = 1$, where $1 \leq i \leq 48$, which we can alternatively write as:

$\{kubok(X, img, blank); kubok(X, img, 0); \ldots kubok(X, img, 9)\} = 1 \leftarrow 1 <= X, X <= 48.$

Since there is only one input *img*, we can simplify the above as follows:

$$\{kubok(X, blank); kubok(X, 0); \ldots kubok(X, 9)\} = 1 \leftarrow 1 <= X, X <= 48.$$

We will now have an ASP program that will do the subsequent reasoning and filtering. In the ASP code below, we add some explanations as comments.

```
% Blank is viewed as 0.
kubok(X,0) :- kubok(X,blank).
% This rule computes the number in the various X,Y co-ordinates
% based on the numeral corresponding to each of the 48
% attributes of the image.
val(X,Y,Z) :- kubok(M,U), kubok(N,V), Z = 10*U + V,
              N = 10*X+2*Y, M = N-1, U!= blank, V !=blank.

num4(1..4).  num16(1..16).
% This choice rule states that each square in the 4 X 4 grid
% should have a number between 1 to 16.
{val(X,Y,Z): Z <=16, num16(Z)} = 1 :- X <=4, Y <=4, num4(X),
     num4(Y).
% This choice rule states that  for any number between 1 to 16
% only one square in the  4 X 4 grid will have that number.
{val(X,Y,Z): X <=4, Y <=4, num4(X), num4(Y)} = 1 :- Z <=16,
     num16(Z).
% The following two rules compute the sum of the numbers in
% each row and column
result(X,0, S) :- S = #sum{ Z : val(X,Y,Z) , num4(Y) }, num4(X).
result(0,Y, S) :- S = #sum{ Z : val(X,Y,Z) , num4(X) }, num4(Y).
% The following two rules filter out the possible assignments
% where the computed row or the colunm sum does not match
% with what is given.
:- result(0,Y,Z), val(0,Y,ZZ), Z != ZZ.
:- result(X,0,Z), val(X,0,ZZ), Z != ZZ.
```

7.3.3 *Learning in NeurASP*

In NeurASP learning can be framed as learning the parameters of the neural network part so as to maximize the probability of the outputs for the corresponding inputs in the training set. In the context of ASP, the outputs can be thought of as observations. Following the formulation in [14], let θ be the parameters of the neural network part, and let us denote the NeurASP program by $\Pi(\theta)$. Thus learning parameters to maximize the probability of the outputs can be formulated as follows: Given a set $\mathcal{O}$ of observations, such that $P_{\Pi(\theta)}(O) > 0$ for each $O \in \mathcal{O}$, we need to find the θ that maximizes $\Pi_{O \in \mathcal{O}} P_{\Pi(\theta)}(O)$, which is equivalent to maximizing their log, which is, $\sum_{O \in \mathcal{O}} \log(P_{\Pi(\theta)}(O))$. Backpropagation using gradient ascent can then be used for this purpose. The gradient of $\sum_{O \in \mathcal{O}} \log(P_{\Pi(\theta)}(O))$ with respect to θ is given by:

$$\frac{\partial \sum_{O \in \mathcal{O}} \log(P_{\Pi(\theta)}(O))}{\partial \theta} = \sum_{O \in \mathcal{O}} \frac{\partial \log(P_{\Pi(\theta)}(O))}{\partial p} \times \frac{\partial p}{\partial \theta}$$

where p denotes probability of atoms $c = v$ in the neural network, and $\frac{\partial p}{\partial \theta}$ is computed with respect to the neural network in the standard back propagation way. The challenge then is to compute:

$$\frac{\partial \log(P_{\Pi(\theta)}(O))}{\partial p},$$

whose computation is based on the following proposition from [14].

Proposition ([14]) *Let $\Pi(\theta)$ be a NeurASP program and let O be an observation such that $P_{\Pi(\theta)}(O) > 0$. Let p denote the probability of an atom $c = v$ in σ^{nn}, i.e., p denotes $P_{\Pi(\theta)}(c = v)$, and I denote the stable models of $P_{\Pi(\theta)}$. We have that:*

$$\frac{\partial \log(P_{\Pi(\theta)}(O))}{\partial p} = \frac{\sum_{I:I\models O, I\models c=v} \frac{P_{\Pi(\theta)}(I)}{P_{\Pi(\theta)}(c=v)} - \sum_{I,v':I\models O, I\models c=v', v\neq v'} \frac{P_{\Pi(\theta)}(I)}{P_{\Pi(\theta)}(c=v')}}{\sum_{I:I\models O} P_{\Pi(\theta)}(I)}$$

In [14], Clingo [7] is used to compute the stable models of $\Pi(\theta)$, and a PyTorch implementation of the neural network is used to compute the probability of each atom $c = v$ in σ^{nn}.

7.4 Discussion

The NeurASP framework allows disentangling aspects where a neural network excels, and aspects that a logical reasoning and problem solving system provide desirable characteristics. In the specific examples that we discussed in this chapter perception is disentangled from reasoning and problem solving. Such disentangling has also been explored in several other neuro-symbolic frameworks [4, 16]. There have been some past works (such as in [1]) where filtering modules were used to correct perception errors done by machine learning modules; but without the end-to-end mechanism for end-to-end training or fine tuning. In NeurASP, the ASP part is differentiable leading to an end-to-end system, the ASP can not only filter our perception errors but the knowledge in the ASP part can be used to fine tune the parameters in the neural network part. Consider the example in Fig. 7.1. After a neural network is good at recognizing numerals, say a convolutional neural network trained on numerals is plugged into the NeurASP program fine tuning it would lead to a neural network model that is better at recognizing numerals in that particular context. In general, NeurASP facilitates semantics-based regularization [6], as neural modules may not perfectly learn from training data the underlying constraints of a task domain, but when explicitly coded in the ASP, the NeurASP program guarantees that the output satisfies the constraints. In [14] an illustration of this is given with respect to finding the shortest paths. In [14] it is also shown how NeurASP can be used for context relational classification where the ASP module

helps in distinguishing a car from a toy car using the additional knowledge encoded in the ASP part. This is also useful in concept learning.

The ASP component of NeurASP automatically inherits the various niceties and distinctions of ASP. This includes its expressiveness to represent problem solving domains, such as puzzle solving and planning and elaboration tolerance. In [14] Sudoku and several of its elaborations are represented using NeurASP, and it is shown that the elaborations require small changes in the ASP part. Similar to the example in Fig. 7.1, one can imagine elaborations such as (i) the two numbers being both odd or both even; (ii) if one is odd then the other being even and vice versa; (iii) the second number being always smaller or equal to the first number; and so on. Expressing these constraints is straightforward in ASP. In using a NeurASP framework, one can start with a CNN trained on just the numerals and then use the ASP module with the constraints.

A key drawback of NeurASP is that for computing the gradient over the ASP part one needs to compute all the stable models, and this can be time intensive. This contrasts with frameworks such as LNN (Chap. 6) and LTN (Chap. 4) where the logic itself is differentiable. Thus alternate approaches, such as the use of approximate inference, and the use of tractable alternatives to ASP, such as probabilistic soft logic need to be explored. Another drawback of NeurASP is its need to precisely define the interface between the neural network module and the ASP module. An intriguing prospect is to replace the ASP part with a natural language understanding module and explore ways that the interfacing between the two modules is learned rather than explicitly given by a human.

Recently a few extensions and alternatives of NeurASP have been proposed. In [12] a probabilistic circuit is allowed in between the neural module and the ASP module. This allows the expression of conditional probabilities and thus provides a more general framework. In [13] an abstract alternative to NeurASP is proposed where the neural and symbolic modules are considered as black boxes and the authors discuss the necessary properties of these black boxes that allow them to be composed into a single integrated end-to-end system.

References

1. Aditya, S., Yang, Y., Baral, C., Aloimonos, Y., Fermüller, C.: Image understanding using vision and reasoning through scene description graph. Comput. Vis. Image Underst. **173**, 33–45 (2018)
2. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, Cambridge (2003)
3. Baral, C., Gelfond, M., Rushton, N.: Probabilistic reasoning with answer sets. Theory Pract. Logic Program. **9**(1), 57–144 (2009)
4. Chen, W., Ma, X., Wang, X., Cohen, W.W.: Program of thoughts prompting: disentangling computation from reasoning for numerical reasoning tasks (2022). arXiv preprint. arXiv:2211.12588
5. De Raedt, L., Kimmig, A., Toivonen, H.: Problog: a probabilistic prolog and its application in link discovery. In: Proceedings of the 20th International Joint Conference on Artificial Intelligence, pp. 2468–2473 (2007)

6. Diligenti, M., Gori, M., Sacca, C.: Semantic-based regularization for learning and inference. Artif. Intell. **244**, 143–165 (2017)
7. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T. Thiele, S.: A user's guide to gringo, clasp, clingo, and iclingo. Technical report, 2008. Available at http://wp.doc.ic.ac.uk/arusso/wp-content/uploads/sites/47/2015/01/clingo_guide.pdf
8. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, USA, August 15–19, 1988, 2 Volumes, pp. 1070–1080. MIT Press (1988)
9. Hu, Z., Ma, X., Liu, Z., Hovy, E., Xing, E.: Harnessing deep neural networks with logic rules. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp. 2410–2420 (2016)
10. Hu, Z., Yang, Z., Salakhutdinov, R., Xing, E.: Deep neural networks with massive learned knowledge. In: Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, pp. 1670–1679 (2016)
11. Lee, J., Yang, Z.: LPMLN, weak constraints, and P-log. In: Thirty-First AAAI Conference on Artificial Intelligence (2017)
12. Skryagin, A., Stammer, W., Ochs, D., Dhami, D.S. Kersting, K.: Neural-probabilistic answer set programming. In: Proceedings of the 19th International Conference on Principles of Knowledge Representation and Reasoning, pp. 463–473 (2022). https://doi.org/10.24963/kr.2022/48
13. Tsamoura, E., Hospedales, T., Michael, L.: Neural-symbolic integration: a compositional perspective. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 35, pp. 5051–5060 (2021)
14. Yang, Z., Ishay, A., Lee, J.: NeurASP: embracing neural networks into answer set programming. In: 29th International Joint Conference on Artificial Intelligence (IJCAI 2020) (2020)
15. Yang, Z., Lee, J., Park, C.: Injecting logical constraints into neural networks via straight-through estimators. In: International Conference on Machine Learning, pp. 25096–25122. PMLR (2022)
16. Yi, K., Wu, J., Gan, C., Torralba, A., Kohli, P., Tenenbaum, J.: Neural-symbolic VQA: disentangling reasoning from vision and language understanding. In: Advances in Neural Information Processing Systems, vol. 31 (2018)

Chapter 8
Neuro Symbolic Learning with Differentiable Inductive Logic Programming

8.1 Introduction

In Chaps. 4 and 6, we explored frameworks such as [2] and [10] that assumed the a-priori existence of a set of logical sentences or rules, and used this information to construct a neural architecture that would use those rules to guide the training process, in which certain aspects of the logic are parameterized to better fit the training data.

In this chapter, we essentially remove the assumption that we know the logic program ahead of time and instead learn it from data. However, we note that the current work in this area generally still assumes some structural format of the program (similar to how we assume the knowledge of a neural structure). In the area of logic programming, inductive logic programming (ILP) [4] has emerged as a paradigm for this task. However, as originally formulated, ILP was generally not designed to handle noisy data or real-valued/annotated logic. The work of [5] combined ideas from ILP with gradient-based learning in "differentiable ILP" (δILP). Here, we first introduce some basic ILP concepts (Sect. 8.2) using the setup of [5] as a guide. Then, we summarize δILP, which extends ILP for real-valued logic and noisy data via gradient-based training (Sect. 8.3). We also take a close look at complexity and training issues in that section. Finally, in Sect. 8.4 we look at extensions to δILP and alternative approaches.

8.2 ILP Framework

In this section, we describe a formulation for a classical ILP problem, following the formalism of [5], but pointing out some key differences with other ILP frameworks. In the following Sect. 8.3 we shall extend this to the case where it can handle fuzzy logic and conduct ILP with respect to a loss function.

P. Shakarian et al., *Neuro Symbolic Reasoning and Learning*, SpringerBriefs in Computer Science, https://doi.org/10.1007/978-3-031-39179-8_8

8.2.1 Problem Formulation

The set of predicates in the logical language is classified as *extensional* and *intensional*. An extensional predicate only appears in ground atoms (i.e., the database) and can only appear in rule bodies, while an intensional predicate does not appear in the database and can only be inferred by rules, so each intensional predicate must appear in the head of at least one rule. Additionally, intensional predicates can also appear in rule bodies. In [5], only two types of intensional predicates are permitted: *target predicates*, which are the predicates we wish to draw inferences about, and *invented predicates*,[1] which are produced as part of the induction process.

We now introduce the concept of an ILP problem and the definition of a solution to such a problem.

Definition 8.1 (ILP Problem) Given a sets of background atoms $\mathcal{B}$, positive instances (ground atoms) $\mathcal{P}$, and negative instances (ground atoms) $\mathcal{N}$, an ILP problem is of the form $(\mathcal{B}, \mathcal{P}, \mathcal{N})$. We note that the predicates used to form atoms in $\mathcal{P}$, $\mathcal{N}$ are *target predicates*. A *solution* to an ILP problem is a logic program Π such that the following conditions hold:

- For all $a \in \mathcal{P}$, $\mathcal{B} \cup \Pi \models a$
- For all $a \in \mathcal{N}$, $\mathcal{B} \cup \Pi \not\models a$

We note that in a solution to an ILP problem *predicate invention* can be used in which a new intensional predicate is created and appears in the constructs of Π to create a solution. As mentioned earlier, in [5] all intensional predicates are either target predicates or invented predicates.

8.2.2 Solving ILP Problems

General Approaches to ILP There are two common approaches to solving ILP problems, regardless of any neuro symbolic characteristics. The first is a *bottom-up* approach such as Progol [9] in which specific (e.g., ground) rules are learned from the data and further generalized. Conversely, *top-down* approaches (used in [5, 11, 12, 15]) generate candidate rules that are tested and pruned. Typically, a template is used to generate candidates in such an approach.

ILP as SAT-Solving The work of [5] was inspired by an approach presented by [3] where the authors show how an ILP problem can be reformulated as an instance of satisfiability of Boolean formulas. Simply put, the approach is to generate all potential candidate rules to be considered (this is done with templates, and we shall

[1] Invented predicates are often referred to as auxiliary predicates.

discuss the specifics of how templates can lead to rules later on). Let us suppose that Π' is the program consisting of all candidate (non-ground) rules. For each rule $r_i \in \Pi'$, we create a new ground atom f_i and add that ground atom to the conjunction in the body of r_i. So, the rules are of the following form:

$$r_i \equiv p(X) \leftarrow q_1(X) \wedge \ldots \wedge q_m(X),$$

and modified to become:

$$r_i \equiv p(X) \leftarrow q_1(X) \wedge \ldots \wedge q_m(X) \wedge f_i.$$

Let F be the set of all such f_i atoms. We shall refer to the modified set of candidates (with the additional f_i atoms) as Π''. Now we can re-frame the problem of Definition 8.1 as satisfiability by finding set $F' \subseteq F$ such that

$$\forall a \in \mathcal{P} : \mathcal{B} \cup F' \cup \Pi'' \models a$$
$$\forall a \in \mathcal{N} : \mathcal{B} \cup F' \cup \Pi'' \not\models a$$

So, the transformation to SAT is complete—we simply find a subset F' of atoms in F that are considered true that meet the above criteria. From that, it is easy to show that taking all rules containing an atom from F' is guaranteed to yield a solution to the ILP problem.

Templates With this setup, the next question is: "where do the candidates in set Π' come from?". The key idea that is not only used in [5] but subsequent work ([15] and [11]) is the concept of template, which can be thought of as a generalization of a non-ground rule that specifies rule structure that can be directly associated with a set of such rules. In this way, a designer can create several templates based on a given problem and use them to generate the candidate rules. In [5] a *rule template*, denoted τ_i, is a tuple (v_i, int_i) where v_i is a natural number specifying the number of existentially quantified variables allowed in a rule and int_i is a number in $\{0, 1\}$ and set to 1 if intensional predicates are allowed in the body and 0 otherwise.[2] With this notion of a rule template in mind, the authors of [5] create a *program template* that adds further specifications as to what the learned program Π will look like. We present this definition below.

Definition 8.2 (Program Template [5]) Given a set of auxiliary (invented) predicates P_a, a map of elements of P_a to naturals specifying the arity of each predicate in that set $arity_a$, a map *rules* from each intensional predicate to a set of rule templates, and the natural number $\mathcal{T}$ specifying the number of inference steps, a *program template* is of the form $(P_a, arity_a, rules, \mathcal{T})$.

[2] Note that in our complexity analysis (Sect. 8.3.3) we generalize this concept further where int_i is the *number* of intensional predicates permitted in the body.

8.3 A Neural Framework for ILP

At its core, the key contribution of [5] is the development of an ILP framework that can handle noisy data. This extends the formalism introduced in the previous section to allow for the handling of real-valued logic. This intuition is comparable to other frameworks such as LNN [10] and LTN [2]. In all three papers, differentiable fuzzy operators are used instead of classical ones, which have the advantage of being amenable to gradient descent. As with [2, 10], the logic of [5] is also a special case of generalized annotated programs [1, 7, 12] (see Chap. 3 for further discussion).

8.3.1 Loss-Based ILP

In [5], the authors assign a label to each ground atom formed with a target predicate. Recall that for an instance of an ILP problem, the set of such atoms is comprised of $\mathcal{P} \cup \mathcal{N}$. Ground atoms in $\mathcal{P}$ are associated with the value 1, and the rest are associated with the value 0. The set Λ is the set of all atom-label pairs in a given ILP instance. The notation (a, λ) will be used for an arbitrary atom-label pair (element of Λ).

The proposed differentiable model of [5] can be viewed as returning a conditional probability associated with each atom $a \in \mathcal{P} \cup \mathcal{N}$. In [5], that is presented as follows.[3]

$$p(\lambda \mid a, W, \Pi, \mathcal{B}) \tag{8.1}$$

However, an alternate formulation would be to consider an annotated logic program [7] Π_W resulting from the gradient-based training process in which rules are associated with weights resulting from the training process and can inherently consider fuzzy input. In this case, an equivalent formulation of Expression (8.1) is shown as follows:

$$\Gamma_{\Pi_W \cup \mathcal{B}} \uparrow \mathcal{T}(a), \tag{8.2}$$

where Γ is the fixpoint operator of [7] and $\mathcal{T}$ is the number of permitted inference steps. Going forward, to maintain consistency with [5], we shall use the conditional probability notation of Expression (8.1). With that in mind, we can now express the cross-entropy loss as follows.

$$-\mathbb{E}_{(a,\lambda)\sim\Lambda}\big[\lambda \cdot \log(p(\lambda \mid a, W, \Pi, \mathcal{B})) + (1-\lambda) \cdot \log(1 - p(\lambda \mid a, W, \Pi, \mathcal{B}))\big].$$

[3] Note that we have omitted the conditional from the language in this chapter, and instead treat it as a defining characteristic of the environment.

In the next section, we shall describe how these values are calculated as well as the architecture of the model.

8.3.2 *Architecture*

In this section, we describe the architecture of the differentiable model for conducting the inference.

Pre-processing The approach of [5], as well as related papers, carries out pre-processing where logical structures are converted into vector representations. As with a wide variety of NSR papers [2, 10], including follow-on work to [5] such as [8, 11, 12, 15], the background information (the logical "facts" for each training instance or the current environment that the model is being applied to) are converted into vector representations in a manner described in the previous section (i.e., a vector associated with the "label" for each atom in the set Λ). The initial vector resulting from this conversion is denoted with $\mathbf{a}_0$.

Another pre-processing step entails the creation of candidate rules from the templates—this is also replicated in later δILP work [8, 11, 15]. In [5], this step is particularly expensive as it results in a combinatorial number of candidate rules generated. Later, we discuss alternative approaches that use specialized templates and candidate generation techniques (namely [11, 15]) that reduce some of this computational overhead. This also results in the neural structures that produce the $\mathbf{c}_t$ vectors (see figure in the supplement), which become quite large (we discuss the bounds on the size of these vectors in Sect. 8.3.3).

The pre-processing steps are conversions from logic into vector representations or neural architecture—they are not themselves differentiable, though they occur outside of the portion of the architecture where backpropagation occurs, so this does not cause any difficulties. Next, we examine the neural portion of the architecture.

Neural Architecture Positions in vectors in the neural architecture of [5] all correspond to real values associated with ground atoms, and there are three such groups of vectors denoted $\mathbf{a}, \mathbf{b}, \mathbf{c}$. However, as there are multiple inference steps supported, each of these is repeated based on the number of inference steps. Further, $\mathbf{a}_0$ corresponds to the initial input. These vectors are illustrated in Fig. 8.1.

Vector $\mathbf{a}_t$ contains real values for each ground atom resulting from t steps of inference (in the case of $\mathbf{a}_0$, it contains the initial input). These values are connected to the positions in vector $\mathbf{c}_t$ in that it has a position for every pair of grounded rules from the candidates. This is because [5] restricts each intensional predicate to only two templates, and each pair has one candidate rule from each template. The position receives the maximum scalar value for the atom formed by the intensional predicate of the pair. Note that this is consistent with the semantics of generalized annotated programs [7]. We also note that the scalar value produced by conjunction

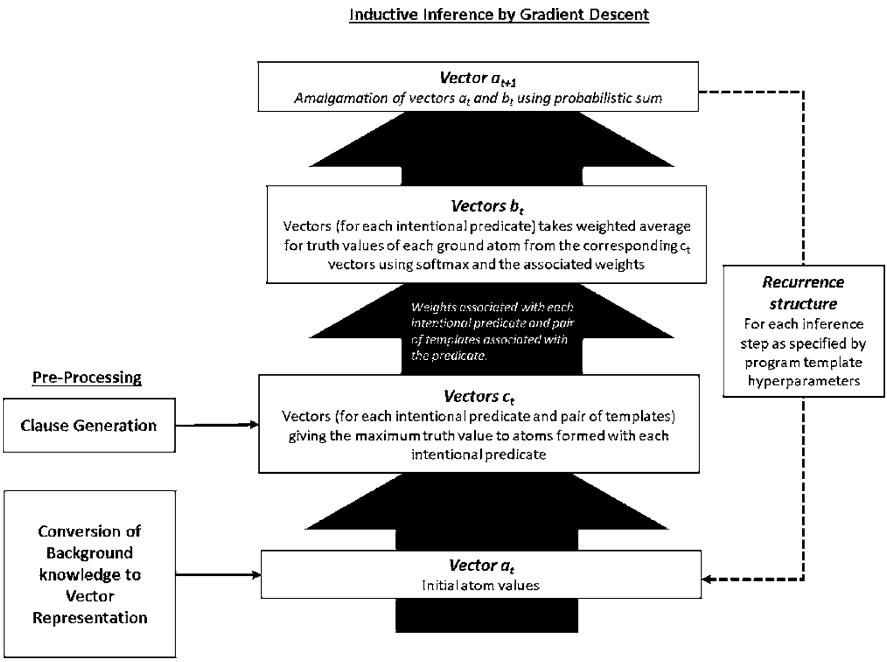


Fig. 8.1 δILP architecture of [5]

is created using a product t-norm, though experiments were performed with Goedel and Lukasiewicz t-norms as well.

The parameters of the model lie in the connections from the $\mathbf{c}_t$ to $\mathbf{b}_t$ vectors. The initial intuition of the authors of [5] was to associate weights with each rule and allow for an arbitrary number of rules. However, this provided poor results, so instead, they associated weights with pairs of rules, and the intuition is that the weights should be associated with pairs. As a result, parameters are associated with rule pairs (hence correspond to individual positions in $\mathbf{c}_t$. Hence, for a given intensional predicate p, the portion of the $\mathbf{b}_t$ vector associated with that predicate is given as follows:

$$\mathbf{b}_t^p = \sum_{j,k} \mathbf{c}_t^{p,j,k} \cdot \frac{e^{\mathbf{W}_p[j,k]}}{\sum_{j',k'} e^{\mathbf{W}_p[j',k']}} \tag{8.3}$$

Finally, the result of the inference step vector $\mathbf{a}_{t+1}$, is produced by taking the probabilistic sum of $\mathbf{a}_t$ and $\mathbf{b}_t$. The authors noted that other possibilities were also considered here, such as max, but the best performance was obtained by using a probabilistic sum.

8.3.3 Complexity

In [5], the authors present an analysis of space (memory) complexity equal to the sum of the following two expressions:

$$2 \cdot n \cdot \mathcal{T} \cdot \sum_{i=1}^{|P_{in}|} \prod_{j}^{num_\tau} |cl(\tau_i^j)| \tag{8.4}$$

$$3 \cdot n \cdot |C| \cdot \sum_{i=1}^{|P_{in}|} \prod_{j}^{num_\tau} |cl(\tau_i^j)| \tag{8.5}$$

These correspond with memory requirements in terms of the number of floats for primary vectors (Eq. (8.4)) and intermediate vectors (Eq. (8.5)) used in the neural architecture. Again, here $\mathcal{T}$ is the number of inference steps permitted and n is defined as the number of ground atoms, which is bounded by the following:

$$n = |P| \cdot |C|^2 + 1 \tag{8.6}$$

Set P_{in} is comprised of intensional predicates, which are predicates that only appear in rule heads (which include invented predicates and any target predicates for which one wishes to make inferences). The entire set of predicates (P) includes P_{in} and P_{ex}, which are the extensional predicates (which only appear in rule bodies and never in the head). The number num_τ is the number of rules learned per intensional predicate, which in [5] is always set to 2.

The set $cl(\tau_i^j)$ contains the clauses (rules) generated by template τ_i^j, and the size of this set is determined by several factors. First is the number of intensional predicates allowed in the body of each clause (*int*), and second is the number of variables that can be existentially quantified in the body (v). For *int*, v, we note that they are different for each intensional predicate, so we shall index these values in that case and use the subscript *max* to refer to the max of such values. Third is the arity of the intensional predicates (which in [5] can be one of {0, 1, 2} for each predicate, the maximum value of this we shall denote as $arity_i$ for the i template and this is bounded by $arity_{max}$, which will be the maximum arity for any predicate), and fourth is the number of atoms per body (which is 2 in [5]; they argue that this is without loss of generality as more atoms per body can be permitted by creating more invented predicates; however, it is noteworthy that doing so also increases computational complexity). We shall denote this last item with $body_{max}$.

Now we shall create a bound on $|cl(\tau_i^j)|$. We note that the total number of variables for a given template is $arity_i + int_i$. This means that for a given predicate in the body, there are at most $(arity_i + int_i)^{arity_{max}}$. The number of possible predicates for one of the atoms in the body is $|P_{ext}| + int_i$. We note that the number of variable combinations for a template and arity of $arity_i$ is $(arity_i + v)!/v! \leq (arity_i + v)^{arity_i}$. Hence, the number of possible variable arrangements and predicates for each atom is

bounded by $(|P_{ext}| + int_i)(arity_i + v)^{arity_{max}}$. The sum over all intensional predicates from Expressions (8.4) and (8.5) is bounded by the following:

$$|P_{in}| \cdot \left(|P|(arity_{max} + v)^{arity_{max}}\right)^{body_{max} \cdot num_\tau} \tag{8.7}$$

We note that based on expected use cases as expressed in [5], we have $num_\tau = 2$, $body_{max} = 2$, $arity_{max} = 2$, $v_{max} = 1$, and thus get the following.

$$6561 \cdot |P_{in}| \cdot |P|^4 \tag{8.8}$$

While this is an upper bound on space complexity, this is mainly due to smaller-arity-sized predicates as well as pruning (e.g., two non-ground atoms create the same clause if their order is reversed). However, it is noteworthy that such exact pruning only reduces space complexity, but has minimal effect on time complexity (as such clauses are generated). We also note that the above result, especially taken into consideration with Expressions (8.4)–(8.6) is in line with a quintic run-time (in terms of predicates). We can re-write overall space complexity as being bounded by the following:

$$6561 \cdot (|P| \cdot |C|^2 + 1) \cdot (2 \cdot \mathcal{T} + 3 \cdot |C|) \cdot |P_{in}| \cdot |P|^4 \tag{8.9}$$

$$\approx 6561 \cdot (2 \cdot \mathcal{T} + 3 \cdot |C|) \cdot |P_{in}| \cdot |C|^2 \cdot |P|^5 \tag{8.10}$$

$$\leq K \cdot |P_{in}| \cdot |C|^3 \cdot |P|^5 \tag{8.11}$$

In line (8.11) (where K is a large constant), it could also be the case that $\mathcal{T} > |C|$, and theoretically we may want this to be true (for correct reasoning); however, we note that in the current work on δILP, scalability precludes the full chaining of inference rules (and so $\mathcal{T}$ is typically set to a small natural number, e.g. $\mathcal{T} = 3$).

8.3.4 Training Considerations and Empirical Results

In this section, we highlight some of the key points from the experiments of [5], where we focus on specific details of the implementation and what their implementation demonstrated.

Training Setup The authors of [5] modify mini-batch stochastic gradient descent to better suit first order logic. In their paradigm, training data consists of many ILP problem instances in the form of $(\mathcal{B}, \mathcal{P}, \mathcal{N})$ triples. However, sampling from this data consists of the following procedure, which is considered one "step" of training.

1. Sample one $(\mathcal{B}, \mathcal{P}, \mathcal{N})$ triple
2. From the triple, sample a mini-batch of $\mathcal{P} \cup \mathcal{N}$

A sampling of $\mathcal{P} \cup \mathcal{N}$ has been shown to help escape local minima. The authors conduct 6000 such steps in their experiments. They use a validation set consisting of a new set of background facts, measuring MSE, which is the primary metric they report in their results. Some key aspects of training include the following.

- Weights are initialized from a normal distribution on the unit interval (mean of zero, standard deviation between 0 and 2).
- RMS Prop was used as the optimizer with a learning rate of 0.5, though the authors reported that Adam also provided good results and RMS prop performed well with much lower learning rates as well (e.g., 0.01).
- Cross-entropy loss was used as the loss function throughout the experiments.

Local Minima Issues The authors note that their approach is prone to get stuck in local minima, which they attribute to the initialization of weights. As a result, it only finds a solution a certain proportion of the time. However, they note that they can model select based on training data after multiple runs, citing that 100 training runs worked well in practice. They noted that the top-scoring model generalized well to unseen data, and attribute this to the fact that the system learns non-ground rules where variables are universally quantified. They hypothesize that future work could address the issue by conducting a search over the initial weights.

Experimental Tasks The authors performed a variety of experiments in [5] that demonstrate the performance of the approach. A few items can be noted:

- First, the authors show that the solution provides very good performance on simple ILP problems in which they examined different candidates for t-norms. By looking at the percentage of runs with less than 10^{-4} MSE, they found that the product t-norm performed significantly better (analogous results were found for LTN's [2, 16]).
- The authors performed tests where portions of the training data were purposely mislabeled and studied the effects of varying the amount of mislabeled data on MSE. In general, performance was robust to such perturbations, and the authors noted robustness up to 30% mislabeled data.
- Further tests were performed where they connect *δILP* to a pre-trained CNN for digit classification to train *δILP* on tasks dealing with digits (e.g., determining if two handwritten digits have a "less-than" relationship). They showed the *δILP* outperforms a multi-layer perceptron baseline.

8.4 Extensions to *δILP*

In this section, we examine several extensions and alternatives to *δILP* for learning logic programs using a neuro symbolic approach.

***δILP* with Negation [8]** One key shortcoming of [5] is the lack of support for negation. The subsequent work of [8] adds stratified, safe negation to *δILP*. The key intuition is to induce a *stratified* logic program Π that is partitioned into $\Pi_1, \ldots, \Pi_k$ where for any atom b in the body of a rule in Π_i, the rules defining b must appear in a partition Π_j where $j \leq i$. For any negation $\neg b$ in the body of a rule Π_i, the rules defining $\neg b$ must appear in a partition Π_j where $j < i$. With such a partition, we can compute $lfp(\Gamma_{\bigcup_{j \leq i} \Pi_j})$ in an iterative fashion as it is guaranteed to be the same as $lfp(\Gamma_{\Pi_i \cup lfp(\Gamma_{\bigcup_{j \leq i-1} \Pi_j})})$. Intuitively, the outcome of the fixpoint operator for the lower partitions will conclude all required negations for the upper ones. The authors use the standard fuzzy definition for negation (see Chap. 3) and design a clause generation process around the stratified program paradigm. In this work, the authors also use a modified loss function, differing from [5], which can be described as follows:

$$0.5 \cdot H(\mathcal{P}) + 0.5 \cdot H(\mathcal{N}) + \gamma \cdot avgH(W).$$

Here, W is the set of all weights, H is cross-entropy, $avgH$ is average entropy (based on the probability distribution shown in Eq. (8.3)), and γ is a hyperparameter for scaling. The authors demonstrated the effectiveness of the approach on several small ILP tasks where negation can impact results.

***δILP* with Function Symbols and Improving Template Generation [15]** In [15], the authors extend *δILP* to allow for function symbols. However, it is noteworthy that this addition complicates the grounding process, as arbitrary function symbols over the set of constants can lead to an infinite number of ground atoms. To address the problem, the authors use beam search to identify candidates that contribute to accuracy and then use them to enumerate ground facts, taking into consideration the function symbols. In this manner, they achieve an efficient generation of candidates and select a useful and relevant subset of ground facts. Experiments only demonstrated the case where rules consisted of a single atom in the body (as opposed to two atoms as shown in [5]); however, this does not appear to be a prohibition of the approach, though it is unclear why the authors did not demonstrate this capability empirically.

ILP for Logical Neural Networks [11] In [11], the authors present a variant of ILP for logical neural networks [10]. One of the key points of departure is that here the logic associates atoms with subsets of the unit interval as opposed to scalars (as done in [5]). The framework permits the entire language of logical neural networks but still relies on a template. Here, a template resembles a syntax tree (of the type described in [10]), which is parameterized with "gates" that allow for the selection of predicates within the template, thereby avoiding the expensive combinatorial candidate generation of [5]. As a result, the implementation shown in that paper scales to larger problems than [5] while also supporting negation.[4]

[4] However, it is noted that consistency is not guaranteed and, as in [10], is addressed by adding a term to the loss function aiming to reduce the amount of inconsistency.

The authors demonstrate experiments on graph-based data with 14,505 constants and 237 predicates.

ILP for Generalized Annotated Logic [12] In [12], the authors combine ideas from generalized annotated programs [7, 13, 14] with concepts from binarized neural networks [6] to also generalize the idea of template further by parameterizing operators in a manner that gates certain inputs (as weights are binarized). The key idea is to make use of a rule with a parameterized annotation function, as follows:

$$r \equiv a : \left[f^{(i)}_{a,\theta^i_a}(X^{(i)}_a), 1 \right] \leftarrow \bigwedge_{\ell_j \in \mathcal{L}'} \ell_j : \left[x_j, x'_j \right]. \tag{8.12}$$

In Expression (8.12), rule r has literal a in the head and is assigned a lower bound on the lattice element based on function $f^{(i)}_a$, which uses learned parameters θ^i_a. Note that $X^{(i)}_a$ is the vector of the lower bound of the annotations of each literal in the body (x_j is the j-th component of $X^{(i)}_a$) and that the x'_j is unused by the annotation function for this particular rule, as the upper bounds are set based on rules applied to negations.

In [12], the annotation function $f^{(i)}_{a,\theta^i_a}$ has the capability to use parameters to "turn off" a given literally in the body, which differs from the other work described in this chapter. This is done via binarized weights [6] that are restricted to values in the set $\{-1, 1\}$. The work of [6] has led to the successful use of a "gradient descent"-style approach to discrete optimization for model training. This avoids the well-known problem of vanishing gradients by substituting the partial derivative for a "pseudo gradient" of the activation function. In [12], the annotation function translates directly to an activation function in the neural architecture. The parameters turn off individual body literals. For example, let $\theta^i_{a,j}$ be the j-th component of θ^i_a; if this value is 1, that means that literal ℓ_j should be considered in the body of the rule –likewise, if it is -1, then it should not. In this way, training can be thought of as a method to erase unneeded body atoms. The following function accomplishes this requirement:

$$f^{(i)}_{a,\theta^i_a}\left(X^{(i)}_a\right) = Sign\left(ReLU\left(1 + \sum_j 0.5\big(1 + \theta^i_{a,j}\big)(x_j - 1)\right)\right) \tag{8.13}$$

8.5 Conclusion

In this chapter, we discussed δILP [5], as well as some of its extensions. This work is significant for two reasons. First, from the perspective of ILP, it allows for the handling of noisy data. Second, from a neuro symbolic perspective, it provides a neural infrastructure to learn logic programs, which in the future can possibly be

integrated with perceptual-level neural layers for end-to-end System 1/System 2 reasoning. However, to date, this has not been demonstrated with a differentiable ILP approach. Further, *δILP* in general does not scale. Even the most promising method for scalability based on empirical results [11] only scales to problems of the size with an order of magnitude 10^5 number of inputs, which is significantly smaller than today's connectionist methods (e.g., GNN methods scale to graphs several orders of magnitude larger). There is also the issue of templates, which based on the current line of work assume some sort of underlying knowledge of the problem. That said, there has been significant progress in *δILP*: ideas like refinement [15], gating [11], stratification [8], and binarization [12] all hold promise to address various challenges. The future of *δILP* will most likely leverage a combination of these ideas to achieve scale and end-to-end training integrated with perceptual-level architectures.

References

1. Aditya, D., Mukherji, K., Balasubramanian, S., Chaudhary, A., Shakarian, P.: PyReason: software for open world temporal logic. In: AAAI Spring Symposium (2023)
2. Badreddine, S., d'Avila Garcez, A., Serafini, L., Spranger, M.: Logic tensor networks. Artif. Intell. **303**, 103649 (2022)
3. Chikara, N., Koshimura, M., Fujita, H., Hasegawa, R.: Inductive logic programming using a maxsat solver. In: 25th International Conference on Inductive Logic Programming (ILP 2015) (2015)
4. Cropper, A., Dumančić, S., Evans, R., Muggleton, S.H.: Inductive logic programming at 30. Mach. Learn. **111**(1), 147–172 (2022)
5. Evans, R., Grefenstette, E.: Learning explanatory rules from noisy data. J. Artif. Int. Res. **61**(1), 1–64 (2018)
6. Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., Bengio, Y.: Binarized neural networks. In: Lee, D.D., Sugiyama, M., von Luxburg, U., Guyon, I., Garnett, R. (eds.) Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems, pp. 4107–4115 (2016)
7. Kifer, M., Subrahmanian, V.: Theory of generalized annotated logic programming and its applications. J. Log. Program. **12**(3&4), 335–367 (1992)
8. Krishnan, G.P., Maier, F., Ramyaa, R.: Learning rules with stratified negation in differentiable ILP. In: Advances in Programming Languages and Neurosymbolic Systems Workshop (2021)
9. Muggleton, S.: Inverse entailment and progol. N. Gener. Comput. (Special issue on Inductive Logic Programming) **13**(3–4), 245–286 (1995)
10. Riegel, R., Gray, A., Luus, F., Khan, N., Makondo, N., Akhalwaya, I.Y., Qian, H., Fagin, R., Barahona, F., Sharma, U., Ikbal, S., Karanam, H., Neelam, S., Likhyani, A., Srivastava, S.: Logical neural networks (2020). https://doi.org/10.48550/ARXIV.2006.13155. https://arxiv.org/abs/2006.13155
11. Sen, P., Carvalho, B.W.S.R.d., Riegel, R., Gray, A.: Neuro-symbolic inductive logic programming with logical neural networks. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 36(8), pp. 8212–8219 (2022)
12. Shakarian, P., Simari, G.: Extensions to generalized annotated logic and an equivalent neural architecture. In: IEEE TransAI. IEEE (2022)
13. Shakarian, P., Simari, G.I., Callahan, D.: Reasoning about complex networks: a logic programming approach. Theory Pract. Log. Program. **13**(4–5(Online-Supplement)) (2013)

14. Shakarian, P., Simari, G.I., Schroeder, R.: MANCaLog: a logic for multi-attribute network cascades. In: Gini, M.L., Shehory, O., Ito, T., Jonker, C.M. (eds.) International Conference on Autonomous Agents and Multi-Agent Systems AAMAS, pp. 1175–1176. IFAAMAS (2013)
15. Shindo, H., Nishino, M., Yamamoto, A.: Differentiable inductive logic programming for structured examples. In: Thirty-Fifth AAAI Conference on Artificial Intelligence, pp. 5034–5041. AAAI Press (2021)
16. van Krieken, E., Acar, E., van Harmelen, F.: Analyzing differentiable fuzzy logic operators. Artif. Intell. **302**, 103602 (2022)

Chapter 9
Understanding SATNet: Constraint Learning and Symbol Grounding

9.1 Introduction

SATNet [21] provides a different viewpoint on neuro symbolic reasoning from many other frameworks in that it has the capability to learn an instance of a combinatorial problem. The key intuition is to consider a semidefinite relaxation of the maximum satisfiability (MAXSAT) problem [9]. In MAXSAT, one is given a set of clauses and must find a set of truth assignments to Boolean variables that maximize the number of satisfied clauses (constraints). During training, a matrix representing a relaxation of the clauses is learned, but it is done via backpropagation. Hence, the forward pass and ultimate result of the training process consists of an instance of MAXSAT with a relaxation of a set of constraints that are learned from data. This can capture a variety of interesting problems that typically are not amenable to neural approaches. The canonical problem studied empirically with respect to SATNet is Sudoku [15], a combinatorial puzzle game that can be viewed as an instance of MAXSAT. SATNet demonstrated the ability to learn to successfully play the game from training samples alone without *a priori* knowledge of the rules. Traditional neural approaches (e.g., dense multi-layer perceptron) attempting to accomplish this task fail catastrophically due to the learning of superficial patterns as opposed to the underlying combinatorial problem. SATNet was also demonstrated on instances of Sodku where the input was visually represented using MNIST digits. Initial experiments showing successful end-to-end training [21] by connecting SATNet with an untrained convolutional neural network [11] showed promise. However, this approach was later shown to perform poorly if inadvertent supervision via "label leakage" was accounted for [3]. This is due to the fact that the original work did not properly account for *symbol grounding* [10], which refers to how symbols used in reasoning are defined and *transduction* [2] the problem of translating neural inputs into symbols. However, later work using unsupervised methods for the perceptual layers seems to address this problem [19], at least for the case of Sudoku.

P. Shakarian et al., *Neuro Symbolic Reasoning and Learning*, SpringerBriefs in Computer Science, https://doi.org/10.1007/978-3-031-39179-8_9

The remainder of this chapter is outlined as follows. In Sect. 9.2 we review the basic framework of SATNet and how it can learn instances of MAXSAT from data. In Sect. 9.3 we discuss symbol grounding and the work around the use of MAXSAT in connection to perceptual layers for end-to-end training. This is followed by a discussion of what SATNet provides as well as potential research directions (Sect. 9.4) before concluding the chapter (Sect. 9.5).

9.2 SATNet to Learn Instances of MAXSAT

In this section, we review the MAXSAT problem (Sect. 9.2), describe the forward pass of SATNet, which approximates the solution to MAXSAT (Sect. 9.2.2), describe how SATNet learns the relaxed constraint matrix (Sect. 9.2.3), and review some of the experimental findings for SATNet (Sect. 9.2.4).

MAXSAT The maximum satisfiability (MAXSAT) problem is a well-known combinatorial problem based on the following input and output.

MAXSAT (Problem LO5 in [9])
INPUT: Set A of n atomic propositions, set C of m disjunctions over the set of A. Positive integer $k \leq m$.
OUTPUT: "Yes" if there is a world w that is a subset of A such that w satisfies at least k disjunctions in C.
Even in the case where each clause only has two literals, MAXSAT as framed above is NP-Complete [9], with the case where $k = m$ is solvable in polynomial time. Clearly, the optimization variant (where we find a world such that it satisfies the maximum number of disjunctions in S) is NP-hard. We can re-write the problem as the following optimization function (as per [21]). Here, $\tilde{s}_{ij}$ is 1 if atom a_i is in clause c_j, -1 if $\neg a_i$ is in clause c_j and 0 if it is absent from c_j. Meanwhile, $\tilde{v}_i$ is 1 if atom a_i is in the world that is the solution, and -1 otherwise.

$$\max_{\tilde{v} \in \{-1,1\}^n} \sum_j \bigvee_i \mathbf{1}(\tilde{s}_{ij} \tilde{v}_i > 0) \tag{9.1}$$

9.2.1 Problem Relaxation

Following [20], SATNet leverages a semidefinite program (SDP) relaxation of SATNet. Here each $\tilde{v}_i$ variable is related to a k-sized vector v_i based on randomized rounding-forming matrix V. Here, S is the relaxation of the constraint matrix formed by the collection of $\tilde{s}_{ij}$. Note that it is also required that $\|v_i\| = 1$. This gives us the following optimization problem.

$$\min_{V \in \mathbb{R}^{k \times (n+1)}} \langle S^T S, V^T V \rangle \tag{9.2}$$

Note that the objective function in (Expression (9.2)) can be solved optimally by coordinate descent, which is how SATNet accomplishes its forward pass as we shall describe in the next section.

9.2.2 SATNet Forward Pass

Now we turn our attention to the forward pass of SATNet which amounts to solving the MAXSAT problem given a relaxed constraint matrix S as described in Sect. 9.2.1. Later, in Sect. 9.2.3 we shall review how backpropagation is used to produce S on each iteration of training. Figure 9.1 illustrates the process of the forward pass.

Inputs and Outputs We note that the primary goal of SATNet is not trying to solve an instance of SATNet, but rather learning an instance of it from data. Specifically, we look to learn the relaxed constraint matrix S. Therefore, in a given training sample, we know there is a world w that maximizes the number of constraints satisfied, and the set of literals[1] associated with w is sub-divided into input and output sets, denoted with $\mathcal{I}$, $\mathcal{O}$. To follow the convention of [21], we shall treat both sets as sets of indices—i.e., if $i \in \mathcal{I}$ then $\tilde{(v)}_i$ is an input value and v_i is the associated relaxation.

As the first step to the forward pass, the inputs (each $\tilde{(v)}_i$) are relaxed into k-sized vectors. The value k is set to $\sqrt{2n}+1$, which is the minimum size that ensures coordinate descent will provide an optimal solution to the relaxed objective function (Expression (9.2)). To support this, a random unit vector associated with each input is generated (v_i^{rand}), along with vector $v_\top$. Using the discrete-to-continuous relaxation employed for SDP relaxation of MAXSAT, the relaxed input is created as follows.

$$v_i = -\cos(\pi \tilde{v}_i) v_\top + \sin(\pi \tilde{v}_i)(I_k - v_\top v_\top^T) v_i^{rand} \tag{9.3}$$

Here, I_k refers to the identity matrix.

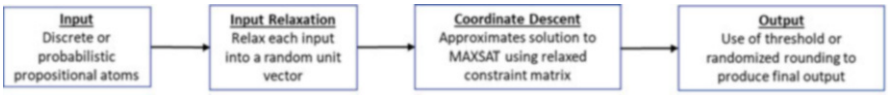


Fig. 9.1 Overview of the forward pass of SATNet

[1] The set of literals would consist of $\{a$ such that $a \in w\} \cup \{\neg a$ such that $a \notin w\}$.

Coordinate Descent to Compute Outputs We note that normally, coordinate descent would be used to identify all literals associated with the instance of MAXSAT. However, in the context of SATNet's forward pass, we wish to only use the MAXSAT approximation to identify the outputs (elements of O). However, the use of coordinate descent makes this the most expensive part of the forward pass operation, running in time $O(nmk)$ each iteration (the authors note that empirically much fewer iterations are required to converge). This process results in relaxed vectors, each of size k. These vectors need to be subsequently converted into discrete output values, completing the approximation of MAXSAT. While randomized rounding can provide an optimal approximation ratio, a probabilistic output is used for training (likely to facilitate more fine-grain gradient adjustments). Derived from results on randomized rounding, the probability is given as follows.

$$P(\tilde{v}_i) - \cos^{-1}(-v_i^T v_\top)/\pi$$

For testing, we note that randomized rounding is performed multiple times, with the best result used as the solution—this practice is known to work well for MAXSAT approximation.

9.2.3 Learning the Relaxed Constraint Matrix

We now discuss the backpropagation process used to learn the relaxed constraint matrix. At a high level, this involves propagating the gradient from the outputs with respect to the loss function (denoted ℓ) by computing the gradient with respect to the relaxation of the outputs ($\frac{\partial \ell}{\partial V_O}$), then pushing the gradients through the semidefinite program routine, resulting in matrix U, which in turn allows for the computation of the gradient with respect to the relaxed inputs ($\frac{\partial \ell}{\partial V_I}$) and with respect to the relaxed constraint matrix ($\frac{\partial \ell}{\partial S}$) and with respect to the original inputs ($\frac{\partial \ell}{\partial \tilde{V}_I}$). We provide further technical details in the supplement.

9.2.4 Experimental Findings

The canonical problem studied with respect to SATNet is the Sudoku puzzle, where a square grid is provided that is partially filled-in with digits. The puzzle is solved by filling in the remaining digits such that each row and column sum to the same value. An example is shown in Fig. 9.2.

This problem can be captured in an instance of MAXSAT which includes inputs and outputs. Inputs, set I, consist of the digits already filled-in, while the outputs,

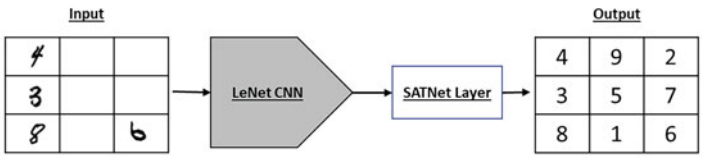


Fig. 9.2 Overview of SATNet integrated with a CNN perceptual layers

set O are the remaining digits. This also makes for a natural set of data to use in a machine learning context [15].

Using a CNN trained on 9×9 Sudoku games from [15], the best performance obtained in testing is 15.1% accuracy despite achieving 91.4% accuracy on training. Even worse, performance drops to 0.01% in training and 0% in testing when the game is permuted. This is likely due to the CNN improperly leveraging local structures in the data. SATNet, by contrast, achieves 98.3% accuracy in testing even with permutations.[2]

SATNet obtains these results by learning 600 constraints and using an MSE loss function. Note that unlike Sudoku solving performed by NeurASP (Chap. 7), SATNet achieves its results with no *a-priori* knowledge of the game. However, since only a relaxation of the constraint matrix is learned, it is not inspectable. Further, the number of constraints set by SATNet is about two orders of magnitude larger than the number of rules required for the game. It is also unclear as to what is a reasonable number of constraints to set for a given run of SATNet in order to both ensure convergence and avoid overfitting.

9.3 Symbol Grounding

In [21] the authors presented another compelling result, an evaluation of the performance of SATNet on a "visual" Sudoku problem. Here, the input would consist of MNIST digits as opposed to printed ones and an untrained version of the LeNet CNN [11] would be used for perceptual layers, along with a cross-entropy loss function (as opposed to MSE used in other experiments). Note that a different learning rate was required for the CNN and SATNet layers. In Fig. 9.2 we show a diagram representing this problem. It is noteworthy here, that as opposed to other integrations of perception and reasoning the perceptual-reasoning system of [21] was trained end-to-end.

The results reported for 9×9 visual Soduku in [21] were quite promising. It achieved 63.2% accuracy on test data while pure CNN methods failed catastrophically (under 1% accuracy). Further, as the CNN of [11] provides a 99.2% test

[2] The results in this paragraph are as reported in [21].

accuracy, the authors of [21] argued that a theoretical best accuracy for testing would be 74.7%. While these results seemed promising, several issues were then raised in later work.

9.3.1 Rebuttal on SATNet's Performance on Visual Sudoku

However, despite the promising results for the visual case reported in [21], a paper published in the following year [3] highlighted several significant limitations to the approach. Specifically, three items were noted. First, when [3] reproduced the visual Sudoku results of [21], they found that the results were highly dependent on weight initialization and the number of constraints. They found that the weight initialization would cause failure (under 1% accuracy) in about 80% of cases and that arbitrarily increasing the number of constraints would also cause the system to fail. They noted that a successful run of SATNet on visual Sudoku (using the approach of [21]) was highly dependent on the perceptual layer rapidly learning how to classify the digits.

However, the third issue raised by [3] was perhaps the most important. They argue that through what they termed as "label leakage" the LeNet-SATNet combination was indirectly supervising the learning of the digits in the perceptual layer. This is due to the fact that the ground truth contained printed versions of the MNIST digits in the training data. They argue that the handwritten digits should have been masked in the output to remove this issue. The concepts of label leakage and masking are illustrated in Fig. 9.3. When the combined LeNet-SATNet architecture was evaluated by [3] with masking implemented, the system failed catastrophically in both training and testing.

The Importance of Symbol Grounding The authors of [3] argue that SATNet's failure in visual Sudoku with masking was due to its inability to handle *symbol grounding*. Symbol grounding is a concept from the cognitive sciences that are concerned with the definition of symbols used in reasoning [10]. It is related to a complementary problem of *transduction* which involves the translation of neural

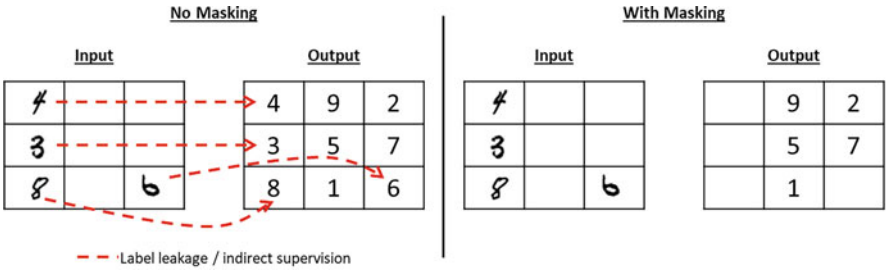


Fig. 9.3 Illustration of the Visual Sudoku problem with and without masking, highlighting the label leakage problem

signals into symbols [2]. The argument is that the LeNet-SATNet architecture did not address symbol grounding and that simply connecting two architectures was likely inadequate to successfully both define symbols and learn a model to reason over them in an end-to-end fashion.

Addressing Symbol Grounding in SATNet The research gap identified by [3] led to further investigation into SATNet's performance. There was interest to see if a combined perceptual-reasoning framework could be trained. A promising approach emerged in [19]. The idea here is to use unsupervised learning, self-grounded training, and an additional proofreading layer to train the model without any supervision. Further description can be found in the online supplement.

9.4 Discussion

In this section, we briefly discuss some of the key issues surrounding SATNet research as well as the current and future directions relating to these issues.

Generalized Symbol Grounding While the work of [19] illustrated that SATNet can be used with a perceptual system in a situation with no supervision, the work is still limited by the preprocessing imposed by the unsupervised algorithm and the associated assumptions that come with it (e.g., that the number of clusters is known *a-priori*). The authors of [19] also point out that their method is susceptible to overfitting which they hypothesize can be overcome with additional regularization. However, perhaps the largest shortcoming is that it is not clear that the method can generalize to situations other than combinatorial problems that consist of hand-written digits—e.g., image keypoint matching, as studied in [16].[3] Creating a general method to address symbol grounding remains an open question, though some recent work shows promise. Specifically, abduction-induction learning [4, 5], appreciation [8], and DeepLoigc [6].

Explainability While several neuro symbolic frameworks covered in this book give rise to explainability [1, 7, 18], the constraints learned by SATNet are not directly understood as the framework actually learns a relaxation of the constraint matrix. However, SATNet is typical of recent efforts to learn instances of combinatorial problems using gradient descent—for example, [14, 16, 17] learn constraints but do not provide interpretable results. Meanwhile, the non-neural approach of Meng et al. [12] to this problem does provide interpretable constraints. Post-processing of the constraint matrix, regularization, and binarization all offer potential solutions, but to date, there is no empirical evidence pointing to one of these methods working.

[3] For clarity, note that the keypoint matching problem in [16] does not use symbol grounding; instead, it uses known key points provided in the dataset [13]. However, learning the key points and the associated combinatorial matching problem together in an end-to-end fashion requires proper symbol grounding.

Scalability A key limitation of SATNet and several follow-on papers looking to learn combinatorial constraints [16, 17] is the requirement to solve a combinatorial problem on each forward pass of the algorithm, thereby greatly increasing runtime. For SATNet, this scalability issue is further compounded by sensitivity to hyper-parameters which can lead to non-convergence [3]. The recent work of Nandwani et al. [14] by generating negative examples from combinatorial examples (e.g., where at least one constraint is violated) and learning hyperplanes that separate these samples, thereby avoiding the need to solve the combinatorial problem. Techniques to scale constraint learning, likely through avoiding repeated instances of solving the combinatorial problem, will be needed in order for practical applications of this line of research to be admitted.

9.5 Chapter Conclusion

In this chapter, we explored the SATNet framework, which provides a neuro symbolic approach to learning combinatorial problems by producing a relaxation of a constraint matrix from the training process. This research has led to the study of many important questions, not only in the area of constraint learning but also in the area of symbol grounding as SATNet has been shown to interface with perceptual layers to learn visual versions of combinatorial problems.

References

1. Badreddine, S., d'Avila Garcez, A., Serafini, L., Spranger, M.: Logic tensor networks. Artif. Intell. **303**, 103649 (2022)
2. Barsalou, L.W.: Perceptual symbol systems. Behav. Brain Sci. **22**(4), 577–660 (1999)
3. Chang, O., Flokas, L., Lipson, H., Spranger, M.: Assessing SATNet's ability to solve the symbol grounding problem. In: Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., Lin, H. (eds.) Advances in Neural Information Processing Systems, vol. 33, pp. 1428–1439. Curran Associates, Inc., Red Hook (2020)
4. Dai, W.Z., Muggleton, S.: Abductive knowledge induction from raw data. In: Zhou, Z.H. (ed.) Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21, pp. 1845–1851. International Joint Conferences on Artificial Intelligence Organization (2021)
5. Dai, W.Z., Xu, Q., Yu, Y., Zhou, Z.H.: Bridging Machine Learning and Logical Reasoning by Abductive Learning. Curran Associates Inc., Red Hook (2019)
6. Duan, X., Wang, X., Zhao, P., Shen, G., Zhu, W.: Deeplogic: joint learning of neural perception and logical reasoning. IEEE Trans. Pattern Anal. Mach. Intell. **45**(4), 1–14 (2022)
7. Evans, R., Grefenstette, E.: Learning explanatory rules from noisy data. J. Artif. Int. Res. **61**(1), 1–64 (2018)
8. Evans, R., Bošnjak, M., Buesing, L., Ellis, K., Pfau, D., Kohli, P., Sergot, M.: Making sense of raw input. Artif. Intell. **299**, 103521 (2021)
9. Garey, M.R., Johnson, D.S.: Computers and Intractability; A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York (1979)
10. Harnad, S.: The symbol grounding problem. Physica D **42**(1–3), 335–346 (1990)

11. Lecun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proc. IEEE **86**(11), 2278–2324 (1998)
12. Meng, T., Chang, K.W.: An integer linear programming framework for mining constraints from data. In: ICML (2021)
13. Min, J., Lee, J., Ponce, J., Cho, M.: SPair-71k: A large-scale benchmark for semantic correspondence. ArXiv, abs/1908.10543 (2019). https://api.semanticscholar.org/CorpusID:201653520
14. Nandwani, Y., Mausam, M., Singla, P.: A solver-free framework for scalable learning in neural ILP architectures (2022). https://arxiv.org/abs/2210.09082
15. Park, K.: Can convolutional neural networks crack Sudoku puzzles? (2018). https://github.com/Kyubyong/sudoku
16. Paulus, A., Rolínek, M., Musil, V., Amos, B., Martius, G.: CombOptNet: fit the right NP-hard problem by learning integer programming constraints. In: Proceedings of the 38th International Conference on Machine Learning, Proceedings of Machine Learning Research, vol. 139, pp. 8443–8453. PMLR (2021)
17. Pogančić, M.V., Paulus, A., Musil, V., Martius, G., Rolinek, M.: Differentiation of blackbox combinatorial solvers. In: International Conference on Learning Representations (2020)
18. Riegel, R., Gray, A., Luus, F., Khan, N., Makondo, N., Akhalwaya, I.Y., Qian, H., Fagin, R., Barahona, F., Sharma, U., Ikbal, S., Karanam, H., Neelam, S., Likhyani, A., Srivastava, S.: Logical neural networks (2020). https://doi.org/10.48550/ARXIV.2006.13155. https://arxiv.org/abs/2006.13155
19. Topan, S., Rolnick, D., Si, X.: Techniques for symbol grounding with SATNet. Adv. Neural Inf. Proces. Syst. **34**, 20733–20744 (2021)
20. Wang, P.W., Kolter, J.Z.: Low-rank semidefinite programming for the MAX2SAT problem. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 33, no. 01, pp. 1641–1649 (2018)
21. Wang, P., Donti, P.L., Wilder, B., Kolter, J.Z.: SATNet: bridging deep learning and logical reasoning using a differentiable satisfiability solver. In: Chaudhuri, K., Salakhutdinov, R. (eds.) Proceedings of the 36th International Conference on Machine Learning, ICML, vol. 97, pp. 6545–6554. PMLR (2019)

Chapter 10
Neuro Symbolic AI for Sequential Decision Making

10.1 Introduction

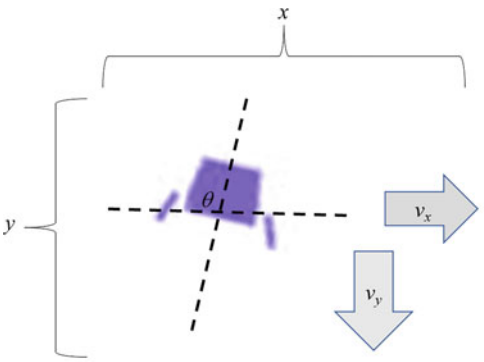
Since the seminal work of [11], there has been great interest in the integration of deep neural networks with sequential decision making, particularly in the context of deep reinforcement learning. However, it has been noted [9] that such approaches have difficulties in generalizing to perturbed situations, explainability, and verification. In this chapter, we examine symbolic methods that intend to overcome these limitations. First, we look at deep symbolic policies [7] in Sect. 10.2 which combines ideas from symbolic regression with deep reinforcement learning [12] to create an equation-based policy to dictate the movement of an agent. Then we look at how to enforce constraints specified using temporal logic in Sect. 10.3 where we look at the use of logic to enforce constraints on a neural network [5, 8] This is followed by a discussion of this active area of research (Sect. 10.4) before concluding the chapter in Sect. 10.5.

10.2 Deep Symbolic Policy Learning

In this section, we review deep symbolic policy (DSP) learning [7]. The idea originates from a desire that deep reinforcement learning (RL) of agent policies provides a black box for understanding how an agent moves through a given environment. Further, the models produced by deep RL are often large. This contrasts sharply with control theory, whereby simple mathematical equations can govern the behavior of a physical device. Deep symbolic policy learning attempts to provide models styled after control theory that can still be learned from data.

The intuition is that the state space is represented by a vector and a policy provides multi-dimensional actions over a continuous space specified by mathematical equations. In Fig. 10.1 we show an example of the Lunar Lander problem from

P. Shakarian et al., *Neuro Symbolic Reasoning and Learning*, SpringerBriefs in Computer Science, https://doi.org/10.1007/978-3-031-39179-8_10



State	Definition
s_1	x
s_2	y
s_3	v_x
s_4	v_y
s_5	θ
s_6	$\frac{d\theta}{dt}$
s_7	First leg contact
s_8	Second leg contact

Action	Definition
a_1	Main engine
a_2	Engine left or right

Fig. 10.1 Example state and actions used in DSP for the OpenAI Gym Lunar Lander

OpenAI Gym [1] with the associated policy learned by DSP [7]. In the figure, the state space is comprised of a vector with eight components that are associated with the lander's location, velocity in two dimensions, angular velocity, and if either of the legs has contact with the ground. The actions are determined by a two-dimensional vector that specified the main engine and lateral movement (negative for left, positive for right). In this example, DSP set a policy where each dimension of the action space is specified with a mathematical equation over the components of the state vector. We note that, for this example, DSP reportedly had near state-of-the-art performance in terms of reward maximization while using the shown equations, a much smaller and more explainable model than any of the standard deep RL approaches.

Note that when there is one action dimension, the problem is reduced to symbolic regression, which is the problem of fitting an arbitrary mathematical formula to data [4, 12]. In Sect. 10.2.1 we review the work of deep symbolic regression [12] which lays the groundwork for deep symbolic policy learning. In Sect. 10.2.2 we then discuss how to move from symbolic regression to DSP learning which involves handling multiple action dimensions as well as leveraging an environment-based reward function.

10.2.1 Deep Symbolic Regression

The task of symbolic regression consists of finding a mathematical equation that fits the top of a set of training data. Formally, given historical data (X, y) where each $X_i \in \mathbb{R}^d$ and $y_i \in \mathbb{R}$, find a function $f : \mathbb{R}^d \rightarrow \mathbb{R}$ that fits the dataset,

usually optimize normalized root mean squared error (NRMSE) below where σ is the standard deviation of the ground truth.

$$NRMSE = \frac{1}{\sigma}\sqrt{\frac{1}{d}\sum_{i}\left(y_i - f(X_i)\right)^2} \tag{10.1}$$

Previous approaches to symbolic regression often relied on genetic algorithms. However, as the focus of this book is neuro symbolic reasoning, we shall focus on deep learning based approaches to this problem, specifically deep symbolic regression (DSR) [12] as it lays the foundation for deep symbolic policy learning. In the supplement, we also will briefly describe a very recent transformer-based approach to this problem introduced by Meta AI at NeurIPS [4].

In this work, the authors optimize NRMSE (Expression (10.1)) which relies on the function that results from each iteration of the learning process. As a result, reinforcement learning is used in this problem. In the remainder of this section, we shall review the steps involved which include the following.

1. A distribution of expressions is generated using a recurrent neural network (RNN) model
2. The reward associated with each expression is evaluated based on NRMSE
3. A "risk-seeking" gradient is computed
4. RNN parameters are updated based on the gradient

Generating Candidate Expressions with an RNN The primary neural component to the approach of DSR is an RNN that produces the expressions. The key data structure used for the mathematical expressions is an "expression tree" which is a syntax tree (see Chap. 6) consisting of mathematical syntax.

Each recurrent unit of the RNN produces a distribution of symbols for the current position. The symbols to select from include the mathematical operators, trigonometric functions, logarithm, power, a constant symbol, and variables representing each of the n inputs in an X_i vector. The input for each recurrent unit consists of a concatenation of the parent and sibling for the output of that unit. As each recurrent unit produces a distribution of symbols, the end results is a distribution of N candidate expressions. A refinement process (outside of the RNN) is used to optimize the constants (i.e., find the constant values that optimize NRMSE). Additional refinement also occurs at this step to ensure expressions are of a certain length, that the children of a mathematical operator cannot all be constants, a mathematical operator cannot have its inverse as a child, and to disallow nested trigonometric functions.

Evaluation of Expressions Each of the N expressions sampled from the RNN is evaluated in terms of a reward for expression τ which is defined based on NRMSE as follows.

$$R(\tau) = 1/(1 + NRMSE) \tag{10.2}$$

From this, a reward threshold for the top $1 - \epsilon$ quantile expressions (denoted R_ϵ) is calculated. The set T is the subset of equations that meet or exceed this threshold.

Compute "Risk Seeking" Gradient to Update RNN Parameters In most reinforcement learning, the policy objective is defined in terms of expectation. However, with DSR, the learning objective is defined based on the top $1 - \epsilon$ quantile.

$$J_{risk}(\theta; \epsilon) = \mathbb{E}_{\tau \sim p(\tau|\theta)}[R(\tau) \mid R(\tau) \geq R_\epsilon(\theta)] \tag{10.3}$$

To find the gradient of J_{risk}, DSR leverages the concept of "conditional value-at-risk" (CVaR) in which use a risk-averse policy gradient [14], except modified (as DSR seeks the top $1 - \epsilon$ quantile) to be a "risk-seeking" policy gradient. Adapting the proof techniques from this work, DSR uses the following policy gradient.

$$\nabla_\theta J_{risk}(\theta; \epsilon) = \mathbb{E}_{\tau \sim p(\tau|\theta)}[(R(\tau) - R_\epsilon(\theta)) \cdot \nabla_\theta \log p(\tau|\theta) | R(\tau) \geq R_\epsilon(\theta)] \tag{10.4}$$

In practice, this gradient is approximated using Monte Carlo sampling.

Putting It All Together While this section was designed to introduce DSR as a component of DSP learning, it is worth noting that DSR was thoroughly evaluated against numerous academic and commercial solutions for symbolic regression across multiple benchmarks and was generally shown to provide state-of-the-art performance. In the inset, we also describe a recently-introduced alternative to symbolic regression based on the transformer architecture.

10.2.2 Deep Symbolic Policy Learning

With the preliminaries of DSR in mind, we now return our attention to the problem of deep symbolic policy (DSP) learning described earlier (e.g., see Fig. 10.1). There are two key aspects by which DSP differs from DSR. First, the objective function differs. DSP looks to optimize a reward function specific to the environment of the agent as opposed to standard symbolic regression criteria. Second, DSP learning entails multiple action dimensions. For example, in the lunar landing example of Fig. 10.1 there are two different dimensions in which the lander can move

Avoiding Objective Function Mismatch As DSP deals with the actions of an agent, the ability of the agent to reach a goal has primacy over a symbolic policy that consists of equations that fit historical data. In fact, in [7], agents often experienced catastrophic failure when relying on DSR-style objective functions, in some cases experiencing an order-of-magnitude reduction in performance. Instead, for DSP-learning, the following reward function is used.

$$R(\tau) = \frac{1}{N_{ep}} \sum_{i=1}^{N_{ep}} \sum_{t=1}^{lth(i)} r_t(i) \tag{10.5}$$

Where N_{ep} is the number of episodes, $lth(i)$ is the length of episode i, and $r_t(i)$ is the instantaneous reward for episode i at time t. This reward is then integrated into a risk-seeking objective function and its associated gradient: J_{risk} and $\nabla_\theta J_{risk}$, defined earlier in Eqs. (10.3) and (10.4) respectively. Here two, the gradient is approximated using Monte Carlo sampling.

Adding Multiple Action Dimensions As the goal of DSP learning is to create symbolic control policies for an agent, the addition of multiple actions leads to a potentially combinatorial explosion in the search space. However, to deal with this problem, DSP learning leverages an "anchor model" which is a black-box model that learns a policy that considers all action dimensions. DSP learning then proceeds sequentially. At a given iteration i, it uses the previously learned symbolic policies for action dimensions 1 through $i-1$ and uses the black-box anchor model for action dimensions $i+1$ and greater. We provide an illustrative example in the supplement.

10.3 Verifying Neural-Based Models

The work of DSP suggests that a symbolic policy for agent actions can be learned from data. When we think about this in the broader context of neuro symbolic reasoning, for example considering differentiable ILP (Chap. 8) or constraint learning (Chap. 9) we can envision future approaches of being able to train an agent with symbolic policies not only limited to mathematical expressions but described in a first order logic. With that in mind, it would also make sense to envision agents that after training can be guaranteed to meet certain safety specifications. Additionally, verification of the results of sequential models has relevance outside of agent models and may have applicability to pure neural approaches. For example, responses produced by ChatGPT for math problems are highly sensitive to factors such as the number of additions and subtractions [13].

In this section, we give an overview of a method by which neural models can yield results that meet a logical specification, which is in some ways an attempt to mimic practices using temporal logic from the software verification community [6]. Specifically, in Sect. 10.3.1 we examine STLNet [8] that allows for the training of RNN or transformer-based models that produce a sequential output that adheres to a signal temporal logic (STL) constraint. Additionally, in the supplement, we provide a discussion of an application of logical neural networks (LNN)[1] to create

[1] See Chap. 6 for an overview of LNNs.

deep reinforcement learning agents guaranteed to adhere to constraints through techniques referred to as "shielding" and "guiding" [5].

10.3.1 STLNet

The concept of STLNet [8] is a black box (e.g. neural) model that one may desire to ensure certain constraints are met around the output of a model that provides a sequential output. The solution is designed to function in a manner agnostic to the underlying black box model. The specifications are created in Signal Temporal Logic which is reviewed in the supplement.

Problem Setup In the problem, we assume m signals $X = \{x^1, \ldots, x^m\}$ that span n time units. We use the notation $x^k_{[t,t']}$ to denote the subsequence of the kth sequence from time t to t'. For some time point i, we will refer to $x^k_{[1,i]}$ as the *prefix* of signal k and $x^k_{[i+1,n]}$ as the suffix. A given parameterized (e.g., neural) model Ψ_θ, once trained, accepts a m prefixes and predicts m suffixes.

$$\Psi_\theta\left((x^1_{[1,i]}, \ldots, x^m_{[1,i]})\right) = \left(\hat{x}^1_{[i+1,n]}, \ldots, \hat{x}^m_{[i+1m,n]}\right) \tag{10.6}$$

To denote the concatenation of an input signal with the predicted output, we use the notation $\hat{x}^k = x^k_{[1,i]}\hat{x}^k_{[i+1,n]}$. Likewise, x^k is used to denote the true sequence and we assume this is drawn from a distribution ($x \leftarrow \mathbf{x}$). Given a series of STL properties, $\varphi_1, \ldots, \varphi_v$, we wish to solve the following problem.

$$\theta = \mathbf{arg\,min}_\theta \mathbb{E}_{x \leftarrow \mathbf{x}}[\mathcal{L}(\hat{x}, x)] \tag{10.7}$$

$$\textit{such that } \hat{x} \models \varphi_1 \wedge \ldots \wedge \varphi_v$$

Student-Teacher Training Paradigm Next, we describe a potential solution to solving the problem specified in Eq. (10.7). Following the process described in Fig. 10.2, a black-box model first finds a set of parameters theta such that the model can predict sequence $\hat{x}$ that minimizes the expected error with the ground truth (x). In [8] this first model is referred to as the "student network."

We note that this initial $\hat{x}$ in no way is guaranteed to adhere to the constraints specified by formulas $\varphi_1, \ldots, \varphi_v$. This is where the "trace generation" step comes into play. The idea of trace generation is to take the result of a model (such as the student network) and modify it in a way to satisfy the constraints. The trace generator examines each temporal component of $\hat{x}$ and ensures it adheres to the constraints. If it does not do so, it then conducts a minimal modification. This process involves converting the conjunction of constraints $\varphi_1 \wedge \ldots \wedge \varphi_v$ to disjunctive normal form (DNF), providing a disjunction of conjunctions of atoms over time. From here, the modification process is easy—simply iterate over all conjunctions

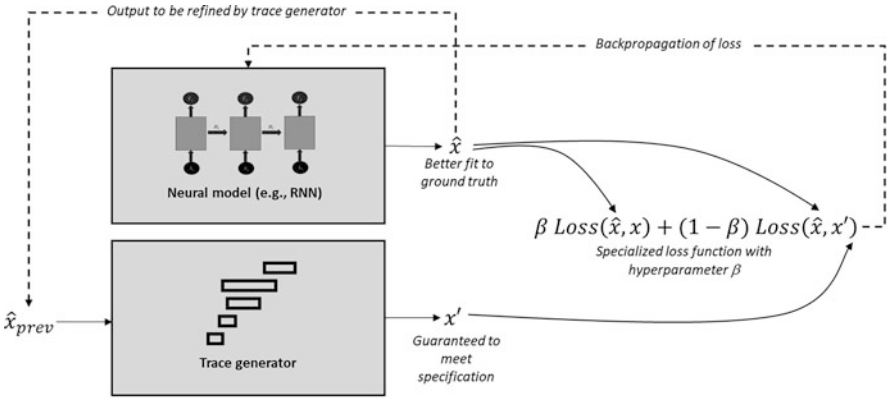


Fig. 10.2 Overview of STLNet

in the DNF formula and make modifications to $\hat{x}$ based on a minimal difference (based on a distance measure). The trace generator returns x' as the result, which is guaranteed to adhere to the constraints. In [8] this is referred to as the "teacher network." Note that this is not a neural network itself, but rather a procedure for modifying the output of the student network.[2]

Now, the algorithm proceeds to retrain with the student network but uses a different loss function, specified as follows:

$$\beta \mathcal{L}(\hat{x}, x) + (1 - \beta)\mathcal{L}(\hat{x}, x') \tag{10.8}$$

Here the loss function consists of two terms: a loss between the result of the model and ground truth and a loss between the model result and the result of the teacher network (which is guaranteed to adhere to constraints). The hyperparameter β adjusts the balance between these two objectives. The intuition is the resulting parameters for the student network trained with this loss function will satisfy more of the constraints. Now the new $\hat{x}$ produced by the student network can be modified by the trace generator where it will make fewer modifications. This process repeats iteratively until convergence (see Fig. 10.2). Note that the end result is a trained model that may not necessarily produce a result that adheres to constraints, but can be modified via the trace generator to do so. Depending on the application, the result $\hat{x}$ or x' can be used, the former prioritizing model fit and the latter prioritizing adherence to constraints.

Empirical results on simulated data presented in [8] show that the "student-teacher" paradigm can provide models that produce a relatively low error and adhere

[2] Note that this point is not explicitly stated in the original paper, but the authors clarify this point in a response to reviewers, see https://proceedings.neurips.cc/paper/2020/file/a7da6ba0505a41b98bd85907244c4c30-Review.html for a copy of the review.

to all constraints (for the result of the teacher network) or very little constraint violation (for the final result of the student network). This was shown for both transformer and RNN architecture. Some limited evaluation of real-world data was also considered. However, the scalability of the approach with respect to the constraints was not explored, which could potentially lead to a combinatorial number of disjunctive clauses. This could potentially hinder practical application (see discussion in Sect. 10.4.

10.4 Discussion

In this section, we reflect on how the research in this chapter can lead to real-world agent-based systems built from a data-driven approach yet provide explainable and verifiable action policies. This is because there are true potential applications of these techniques. In the U.S., DARPA has invested significant funding in programs such as "Assured Autonomy" and "Assured Neuro Symbolic Reasoning" with such goals in mind.

Such programs have led to active research in cyber physical systems (CPS). While this chapter focused on a select subset of neuro symbolic approaches for creating autonomous, verifiable cyber physical systems, this is part of a broader area of research. Applications include defense, aerospace, autonomous driving, mining, and other areas where safety, assurances, and modular design (enabled by strict safety guarantees on components) are prized. See [2] for a recent survey.

However, it is worth noting that the constraints discussed in this chapter were either based on a variant of annotated logic or temporal logic. However, many target applications deal with agent behavior through space, it may be desirable to formally represent constraints with spatial requirements, such as Spatio-Temporal Perception Logic [3]. However, even with a language that allows describing spatio-temporal constraints, we will likely still encounter grounding issues, as space is often treated in a continuous manner, which can lead to an infinite set of ground atoms. Selecting an appropriate set of ground atoms based on spatial information will be an important aspect of any framework that considers spatial constraints.

Spatial constraints will also become important in transitioning DSP learning to real-world robotic systems as we would want symbolic policies to avoid unsafe actions. Further, a refinement process for DSP learning, similar to the "teacher network" in STLNet may allow for adjustment of learned deep symbolic policies to conform with constraints, though to date this has not been studied. However, it should be noted that DSP/DSR already leverages refinement processes to restrict the mathematical equations that result from the learning process (e.g. constant optimization, avoiding nested trigonometric functions, etc.). Perhaps extending such refinement processes could be an avenue toward real-world usage.

Finally, the expression of the constraints in a manner that allows for easy refinement is another area of concern. Note that the refinement processes in DSP/DSR are fairly straightforward based on well-known desirable characteristics

of formulas used in control and do not consider logical constraints from an external source. The ideas of guarding and shielding [5] have not been widely studied in experimental settings and do not support temporal logic constraints. The work of STLNet showed promise in learning a model that adheres to constraints, but it relies on converting constraints to disjunctive normal form (DNF). We note that the conversion from conjunctive normal form (CNF) to DNF can lead to a combinatorial explosion [10]. This implies that the approach put forth by STLNet could be intractable in certain cases. Understanding the complexity involved in generating a DNF version of the constraints and placing a bound on the size is an important issue to be addressed in order to determine if the method presented in [8] is viable as a refinement process for ensuring constraints are satisfied.

10.5 Chapter Conclusion

In this chapter, we discussed neuro symbolic approaches that allow for symbolic agent policies and the enforcement of constraints. Specifically, we looked at DSP learning which allows for a symbolic, mathematical control-theory style model for agent actions as well as STLNet which allows for the learning of a sequential neural model that adheres to constraints. While this is an important area that affects various types of autonomous systems, there are still many problems that must be addressed in order for these frameworks to make real world impact.

References

1. Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., Zaremba, W.: Openai gym. arXiv preprint arXiv:1606.01540 (2016)
2. Corso, A., Moss, R., Koren, M., Lee, R., Kochenderfer, M.: A survey of algorithms for black-box safety validation of cyber-physical systems. J. Artif. Int. Res. **72**, 377–428 (2022)
3. Hekmatnejad, M., Hoxha, B., Deshmukh, J.V., Yang, Y., Fainekos, G.: Formalizing and evaluating requirements of perception systems for automated vehicles using spatio-temporal perception logic (2022). https://doi.org/10.48550/ARXIV.2206.14372. https://arxiv.org/abs/2206.14372
4. Kamienny, P.A., d'Ascoli, S., Lample, G., Charton, F.: End-to-end symbolic regression with transformers. In: Oh, A.H., Agarwal, A., Belgrave, D., Cho, K. (eds.) Advances in Neural Information Processing Systems (2022)
5. Kimura, D., Chaudhury, S., Wachi, A., Kohita, R., Munawar, A., Tatsubori, M., Gray, A.: Reinforcement learning with external knowledge by using logical neural networks. CoRR abs/2103.02363 (2021). https://arxiv.org/abs/2103.02363
6. Lamport, L.: "Sometime" is sometimes "not never". In: Proceedings of the Seventh ACM Symposium on Principles of Programming Languages, ACM SIGACT-SIGPLAN (1980)
7. Landajuela, M., Petersen, B.K., Kim, S., Santiago, C.P., Glatt, R., Mundhenk, N., Pettit, J.F., Faissol, D.: Discovering symbolic policies with deep reinforcement learning. In: International Conference on Machine Learning, pp. 5979–5989. PMLR (2021)

8. Ma, M., Gao, J., Feng, L., Stankovic, J.A.: Stlnet: signal temporal logic enforced multivariate recurrent neural networks. In: 34th Conference on Neural Information Processing Systems (NeurIPS 2020)
9. Marcus, G.: Deep learning: a critical appraisal. CoRR abs/1801.00631 (2018). http://arxiv.org/abs/1801.00631
10. Miltersen, P.B., Radhakrishnan, J., Wegener, I.: On converting CNF to DNF. Theor. Comput. Sci. **347**(1), 325–335 (2005)
11. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., Hassabis, D.: Human-level control through deep reinforcement learning. Nature **518**(7540), 529–533 (2015)
12. Petersen, B.K., Larma, M.L., Mundhenk, T.N., Santiago, C.P., Kim, S.K., Kim, J.T.: Deep symbolic regression: recovering mathematical expressions from data via risk-seeking policy gradients. arXiv preprint. arXiv:1912.04871 (2019)
13. Shakarian, P., Koyyalamudi, A., Ngu, N., Mareedu, L.: An independent evaluation of ChatGPT on mathematical word problems (MWP). In: AAAI Spring Symposium (2023)
14. Tamar, A., Glassner, Y., Mannor, S.: Optimizing the cvar via sampling (2014). https://doi.org/10.48550/ARXIV.1404.3862. https://arxiv.org/abs/1404.3862

Chapter 11
Neuro Symbolic Applications

11.1 Introduction

This will be the concluding chapter in the book. So far, we have explored what neuro-symbolic reasoning is about and discussed various frameworks for neuro symbolic reasoning, such as logical neural networks (Chap. 6), logic tensor networks (Chap. 4), and NeurASP (Chap. 7). We have also discussed various approaches to developing intelligent systems with the help of neuro symbolic reasoning. All these concepts are used in making systems intelligent with sufficient training and reasoning abilities. Neuro symbolic reasoning has applications in various areas such as image understanding and visual question answering [3, 16, 29, 42, 44]; natural language processing, question answering, and knowledge-based reasoning [9, 12, 13, 23, 30, 39, 43, 45]; spatio-temporal reasoning [26]; robotics [18, 22]; reasoning about actions and agents, including using reinforcement learning [4, 40]; image and text generation [1, 24, 31]; and real world applications in Healthcare [15, 25], Smart Cities [37], and Business Management and Finance [14, 19]. A few surveys and tutorials on this have also been written [10, 38, 41].

In the following, we will first illustrate in some detail the neuro-symbolic reasoning application to visual question answering (VQA) that involves understanding both images and text (Sect. 11.2. Then we turn our attention to neuro symbolic applications to natural language processing (NLP) in Sect. 11.3.

11.2 Neuro Symbolic Reasoning in Visual Question Answering

One type of application of neuro-symbolic reasoning is disentangling sub-tasks where neural approaches shine from sub-tasks where symbolic approaches seem to do better in a task. For example, in [3, 44] the authors propose to disentangle logical

P. Shakarian et al., *Neuro Symbolic Reasoning and Learning*, SpringerBriefs in Computer Science, https://doi.org/10.1007/978-3-031-39179-8_11

reasoning from the vision in visual question answering (VQA) tasks, and in [13] the authors propose disentangling numerical computation from linguistic reasoning in numerical reasoning tasks.

Visual question answering involves understanding images, understanding questions about the images asked in natural language, linking them, and reasoning with them to come up with an answer, often in natural language. VQA, and question answering in general, are ways to evaluate how good a system is at understanding images, texts, and multi-modal objects or documents. Over the years, several VQA datasets have been proposed, and one particular VQA dataset that stands out is CLEVR [21], as it helps in testing the abilities and limitations of models with respect to a range of visual reasoning abilities. The images in CLEVR are synthetic ones and are of limited types, but that makes it a good fit for focusing on visual reasoning.

Consider the CLEVR image in Fig. 11.1 and the question: *How many matte objects are behind the small rubber object and on the left side of the purple ball?* Neuro-symbolic approaches to answer this question include the approaches explored in [16, 44], and such approaches involve the following steps:

1. Use of a neural module that converts the image to a scene representation.
2. Use of a neural module that converts the question to an execution program.
3. A program executor that takes the structural scene representation, and the program obtained from the question, to generate an answer.

For the CLEVR image in Fig. 11.1, a scene representation as obtained in [44], is a table of the form given in Fig. 11.2.

In [44], the neural module that converts the image to a structural scene representation uses Mask R-CNN for segmentation and predicts the size, shape, material, and color attributes. A ResNet-34 module then takes segments for each object and the original image as input and determines the 3D coordinates. In [16], YOLOv3 is used to predict the object attributes and bounding-box.

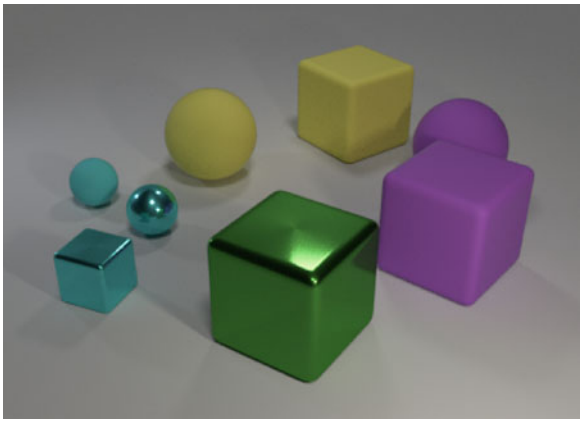


Fig. 11.1 A CLEVR image from the CLEVR data set. It has two matte objects behind the small rubber object and on the left side of the purple ball

ID	Size	Shape	Material	Color	x	y	z
1	small	cube	metal	cyan	-1.5	-0.4	0.35
2	small	sphere	rubber	cyan	-1.3	0.3	0.35
3	small	sphere	metal	cyan	-0.9	-0.1	0.35
4	large	sphere	rubber	yellow	-0.4	0.5	0.7
5	large	cube	metal	green	0.1	-0.9	0.7
6	large	cube	rubber	yellow	0.7	0.8	0.7
7	large	cube	rubber	purple	0.9	-0.42	0.7
8	large	sphere	rubber	purple	1.1	0.45	0.7

Fig. 11.2 Structural scene representation of the CLEVR image in Fig. 11.1

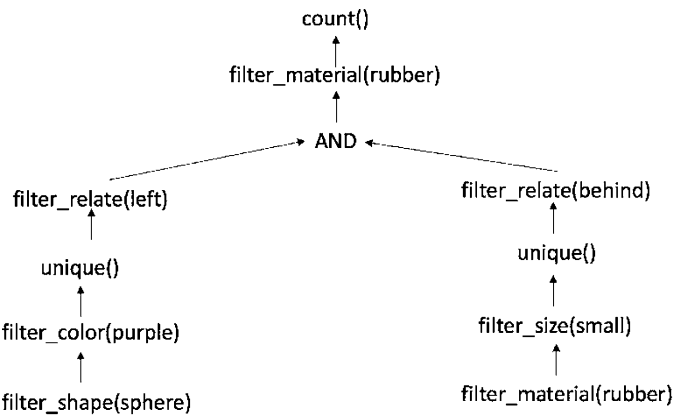


Fig. 11.3 CLEVR functional program of the question: *How many matte objects are behind the small rubber object and on the left side of the purple ball?*

For the question *How many matte objects are behind the small rubber object and on the left side of the purple ball?*, the CLEVR functional program is given in Fig. 11.3.

In [44], the neural module that converts the question to a program is an attention-based seq2seq model with a bidirectional LSTM used for both encoder and decoder components. The decoder generates an output vector fed to an attention layer to obtain a context vector. Both the output vector and the context vector are fed to a fully connected layer with softmax to make the next prediction. In [16], instead of questions in natural language, programs are directly given.

A symbolic procedural program executor is proposed in [44] that takes the structural scene representation, and the program obtained from the question, to generate an answer. In [16], instead of a symbolic procedural program executor, the CLEVR program, and the structural scene representation of the image are converted to answer set programming format, additional answer set rules are added and an answer set solver is used to answer the question.

11.3 Neuro Symbolic Reasoning involving Natural Language Processing

Excluding the last decade or so, natural language processing (NLP) has been primarily a symbolic task. More recently, neural approaches have been very successful in various NLP tasks; regardless, for many tasks involving natural language processing, a neuro-symbolic approach has many advantages. In the CLEVR-centric VQA discussed in the previous section, questions in natural language are converted to programs using a neural approach; however, the programs that are generated are executed as part of a symbolic module.

11.3.1 Concept Learning

In [32], concepts are learned by processing (a) an input pair $\langle x, y \rangle$, where x is an image and y is a natural language sentence about the image, describing a particular concept using the image; (b) together with a supplementary sentence. For example, for the image x in Fig. 11.1, a corresponding y may be the sentence "The cube object to the right of the green metal cube is purple." In addition, a supplementary sentence could be "green, purple, yellow are colors". During the concept learning phase, y is translated to a program using a neural approach, and the supplementary sentence is translated to tuples of the form $(green, isa, color)$. A symbolic program executor is used to execute the program with respect to the image x to locate the object in x referred to by y. Subsequently, an embedding prediction module takes the located object o, the concept c pointed out in y, and the relationship between c and the other concepts to output an embedding for c.

The learning process is evaluated using question answering, where questions are expressed in natural language. For example, to test if the various CLEVR concepts (such as color, shape, size, and materials) have been learned, the evaluation question answer may be of the following kind:

Q: How many spherical objects have the same color as the metal cyan cube?
A: 2.

During the evaluation phase, the question in natural language is neurally translated to a program and the symbolic program executor is used to execute the program.

11.3.2 Reasoning Over Text

We now consider some purely natural language tasks. In [17], a neuro symbolic approach, referred to as neural module networks (NMNs) [5] are used for answering compositional questions that require multiple reasoning steps in an interpretable

manner. NMNs are composed of multiple learnable modules. The question answering approach in [17] has many similarities with the VQA approach in [44]. In particular, as in [17], a question in [17] is translated by a neural module to a program that is used by a symbolic program executor. Each function in the program is implemented by a neural module, while in [44] the functions focus on or identify parts of the image, in [17] the functions focus on or identify spans of the text. The following are some of the functions proposed and used in [17].

- filter(Q, P) → P′: Given a question Q and a span P in the passage find the relevant (with respect to Q) sub-span P′ in P.
- relocate(Q, P) → P′: Given a question Q and a span P in the passage, find P′ (in P) corresponding to the argument asked in question Q.
- find-num(P) → N: Find the number(s) N associated with the input paragraph span P.
- find-date(P) → D: Find the date(s) D associated with the input paragraph span P.
- compare-num-less-than(P1, P2) → {P1, P2}: Determines which of the spans P1 or P2 is associated with a smaller number.
- find-max-num(P) → P′: Find P′ (in P) that is associated with the largest number.

Consider the question Q from [17] given as "Who kicked the longest field goal in the second quarter?", asked with respect to paragraph P. To answer this question, the question is translated to the following program: relocate(Q,find-max-num(filter(Q, P))). When this program is executed, the first filter(Q, P) is executed to find a span P1 of P relevant to the question Q. Next when find-max-num(P1) is executed a span P2 of P1 is identified that is associated with the largest number. Then, relocate(Q, P2) is executed to find who kicked the field goal, which earlier had been identified as the longest.

11.3.3 Using ASP in Reasoning Over Text

In the previous subsection, not only the translation of a question to a program was neurally implemented, but each of the functions was also neurally implemented. However, when processing natural language text, one comes across aspects spelled out in natural language that can be more easily implemented in a declarative manner by using declarative language. Thus, similar to the use of function modules in the previous subsection, one can declaratively define various natural language concepts and use them in question answering in a neuro symbolic architecture where a neural module is used to extract facts from the question and the text, and NLI (natural language inference) and concept definitions in a declarative language are used to answer questions. Such an approach is used by us in [36] to answer a multiple-choice question about the life cycle of a frog.

Consider the following question asked with respect to a paragraph P_f about the life cycle of a frog:

What best indicates that a frog has reached the adult stage?

(A) When it has lungs.
(B) When its tail has been absorbed by the body.

Consider the word "indicates" in the question. Intuitively, while an adult frog satisfies both (a) has lungs, and (b) its tail has been absorbed by the body, only the latter "indicates" that the frog has reached the adult stage, as a froglet also satisfies the property of having lungs. Thus, a property indicates a particular life stage if that property is unique to that life stage. This can be expressed in the declarative language of Answer Set Programming by the following rule:

```
indicator(O, S, P) :- organism(O), stage(S),
                      property(P), has(O,S,P),
                      #count{ has(O,S',P) : stage(S')} = 1.
```

Some of the other terms that are defined using declarative rules in [36] are "middle", and "between" used in the questions "*What is the middle stage in a frog's life?*", and "*What is a stage that comes between tadpole and adult in the life cycle of a frog?*".

11.3.4 Solving Logic Grid Puzzles Described in Text

Logic grid puzzles are often asked in standardized analytical tests to evaluate the reasoning ability of test takers. One of the most well known is the Zebra puzzle, which is about uniquely associating five houses with five colors, five inhabitants of different nationalities, and five different pets, with the preference for five different beverages and five different cigarette brands so that a set of given clues are satisfied. The clues of the Zebra puzzle are:

- The Englishman lives in the red house.
- The Spaniard owns the dog.
- Coffee is drunk in the green house.
- The Ukrainian drinks tea.
- The green house is immediate to the right of the ivory house.
- The Old Gold smoker owns snails.
- Kools are smoked in the yellow house.
- Milk is drunk in the middle house.
- The Norwegian lives in the first house.
- The man who smokes Chesterfields lives in the house next to the man with the fox.
- Kools are smoked in the house next to the house where the horse is kept.
- The Lucky Strike smoker drinks orange juice.
- The Japanese smoke Parliaments.
- The Norwegian lives next to the blue house.

This puzzle appeared in Life magazine in 1962, had 15 such clues, and asked the questions:

Who drinks water?
Who owns the zebra?

A neuro symbolic approach to solving such puzzles is to have a neural module that translates the clues to a formal representation, and a reasoning module that solves the puzzle using the outputted formal representation. In early work [27], classical NLP techniques were used for translation, and a theorem prover was used as the reasoning module. Alloy, a formal language for modeling software designs, was used in [33]. In other work [6, 35], a non-neural machine learning module was used for translation and Answer Set Programming (ASP) for reasoning. Later work [20] proposed the use of a DistilBERT-based classifier for translation and Prolog for reasoning.

In the approach in [6, 35], a general representation was developed for the reasoning module for (m, n) puzzles where m is the number of categories and n is the number of elements in each category (for the zebra puzzle, $m = 6$ and $n = 5$). The 6 categories are houses, colors, nationality, pets, beverages, and cigarette brands. This information is expressed in ASP in the following way:

```
cindex(1..6).
eindex(1..5).

etype(1, houses).
etype(2, colors).
etype(3, nationality).
etype(4, pets).
etype(5, beverages).
etype(6, cigarets).
```

The elements of each of the categories are also clearly spelled out. We list the elements of the category house and colors below.

```
element(1, 1). element(2, yellow).
element(1, 2). element(2, blue).
element(1, 3). element(2, red).
element(1, 4). element(2, ivory).
element(1, 5). element(2, green).
```

We represent the puzzle solution using the predicate *tuple*. Suppose house 1 has the color yellow, and the person living there is of nationality Norwegian, has a pet fox, drinks water, and smokes Kools. Let this be expressed by tuple 1. This and a second tuple of the solution will be represented by using the predicate *tuple* as follows:

```
tuple(1, 1, 1).              tuple(2, 1, 2).
tuple(1, 2, yellow).         tuple(2, 2, blue).
tuple(1, 3, norwegian).      tuple(2, 3, ukrainian).
tuple(1, 4, fox).            tuple(2, 4, horse).
tuple(1, 5, water).          tuple(2, 5, tea).
tuple(1, 6, cools).          tuple(2, 6, chesterfield).
```

The rest of the tuples that represent the solution are:

```
tuple(3, 1, 3).       tuple(4, 1, 4).        tuple(5, 1,5).
tuple(3, 2, red).     tuple(4, 2, ivory).    tuple(5, 2,green).
tuple(3, 3, english). tuple(4, 3, spaniard). tuple(5, 3,japaneese).
tuple(3, 4, snails).  tuple(4, 4, dog).      tuple(5, 4,zebra).
```

```
tuple(3, 5, milk).    tuple(4, 5, oj).       tuple(5, 5,coffee).
tuple(3 ,6, old-g).   tuple(4, 6, lucky).    tuple(5, 6,parliament).
```

Such a solution is obtained by enumerating the *tuple* predicate and representing the clues as constraints that filter out enumerations that do not satisfy the clues. The *tuple* predicate is enumerated by the following ASP rules:

```
1 {tuple(G, C, E) : element(C,E) } 1 :- cindex(C), eindex(G).
   :- tuple(G1,C,E), tuple(G2, C, E), G1 != G2.
```

Below are examples of ASP translations of some of the clues.

```
   :- tuple(I,2,red), tuple(J,3,english), I != J.
   :- not tuple(1,3,norwegian).
   :- tuple(I,3,norwegian),tuple(J,2,blue), not next_to(I,J).
next_to(I,J) :- I = J+1.
next_to(I,J) :- J = I+1.
```

One challenge here is that, while a neural ML system can be taught to translate clues to the first three rules above, the rules defining next_to would need to be given or learned using a separate Inductive Logic Programming (ILP) like module.

11.4 Neuro-Symbolic Reinforcement Learning

Reinforcement learning (RL) is often formulated as learning what actions to take in a state in an environment so as to maximize the expected cumulative reward. The dynamics of the environment are modeled as a Markov decision process (MDP), with a set of states S, a set of actions A, a probabilistic transition function P that defines the probability of transitioning from one state to another due to an action, and a reward function associated with each transition. The goal of the reinforcement learning algorithm is then to learn a policy of what action to take in which state so as to maximize the expected cumulative reward. Deep RL is often used when the dimension of a state becomes large, such as when the state is expressed as a screenshot of a game or images. In the case of a game, a function of the game score is often used as the reward function.

However, in many RL scenarios, there are often additional conditions such as safety that are not always baked into the reward function. An example of a safety condition is that an autonomous car should never jump a red light, while a typical reward based criteria may be to reach a destination quickly [2]. Safety conditions and other temporal goals [7, 8] that an agent may want to satisfy are difficult to capture in a state based reward function [28] as capturing the history aspect by a single state would usually mean blowing up the number of states. One way to address this is to generalize the RL framework to allow policies to be from sequence of states to actions, rather than just from states to actions, and rewards to be also similarly defined. To do this in a succinct manner a formal representation of sets of sequences of actions can be done using symbolic methods such as temporal logics. Such a generalization is done in [11] and the notion of reward machines is introduced.

Another way to address this, which also requires the use of symbolic reasoning, is by using a shield [2] that is pre-synthesized symbolically based on safety properties normally specified using temporal logic. That paper introduces the notion of safe RL as "learning an optimal policy while satisfying a temporal logic safety specification". It presents two approaches. In one approach, the shield suggests which actions are safe, and in another approach the shield monitors and makes corrections if a safety specification would be violated. In case of the former, the paper gives an algorithm to synthesize the reactive system component of shields that are correct in that they enforce the safety conditions, and are minimally interfering. For the latter, the paper presents the joint operation of a learner and a shield. In essence the shield monitors the actions of the agent and when it finds an action to be unsafe then it substitutes it with a safe action. A more general approach is proposed in [4] which allows "learning over the continuous state and action spaces, and the monitor and the shield are continually updated as learning progresses".

In hierarchical RL, while the lower level may use Deep RL, symbolic rule learning is at times used at higher levels to aid in interpretability. In [34] Inductive Answer Set Learning is used to learn meta policies in the Animal AI environment. It is able to learn interpretable rules such as: "If a ramp is available then climb it. If the agent is on a platform and there is lava near the goal, then observe the arena dynamics."

11.5 Chapter Conclusion

In this chapter, we went over some of the noteworthy current research on neuro symbolic applications. We focused on VQA and NLP use cases. In both cases, we looked at how neuro symbolic techniques bridge the gap between machine learning perception and symbolic reasoning. We then touched upon the necessity of neuro symbolic reasoning in reinforcement learning. As the field progresses, we expect a tighter coupling of such approaches to emerge.

References

1. Aggarwal, G., Parikh, D.: Neuro-symbolic generative art: a preliminary study. Preprint (2020). arXiv:2007.02171
2. Alshiekh, M., Bloem, R., Ehlers, R., Könighofer, B., Niekum, S., Topcu, U.: Safe reinforcement learning via shielding. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 32 (2018)
3. Amizadeh, S., Palangi, H., Polozov, A., Huang, Y., Koishida, K.: Neuro-symbolic visual reasoning: Disentangling visual from reasoning. In: International Conference on Machine Learning, pp. 279–290. PMLR (2020)
4. Anderson, G., Verma, A., Dillig, I., Chaudhuri, S.: Neurosymbolic reinforcement learning with formally verified exploration. Adv. Neural Inf. Process. Syst. **33**, 6172–6183 (2020)

5. Andreas, J., Rohrbach, M., Darrell, T., Klein, D.: Neural module networks. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 39–48. IEEE Computer Society (2016)
6. Baral, C., Dzifcak, J.: Solving puzzles described in English by automated translation to answer set programming and learning how to do that translation. In: 2011 AAAI Fall Symposium Series (2011)
7. Baral, C., Eiter, T., Bjäreland, M., Nakamura, M.: Maintenance goals of agents in a dynamic environment: formulation and policy construction. Artif. Intell. **172**(12–13), 1429–1469 (2008)
8. Baral, C., Zhao, J.: Goal specification, non-determinism and quantifying over policies. In: Proceedings of the 21st national conference on Artificial intelligence, vol. 1, pp. 231–237 (2006)
9. Bosselut, A., Le Bras, R., Choi, Y.: Dynamic neuro-symbolic knowledge graph construction for zero-shot commonsense question answering. In: AAAI, pp. 4923–4931 (2021)
10. Bouneffouf, D., Aggarwal, C.C.: Survey on applications of neurosymbolic artificial intelligence. Preprint (2022). arXiv:2209.12618
11. Camacho, A., Icarte, R.T., Klassen, T.Q., McIlraith, S.A.: Ltl and beyond: formal languages for reward function specification in reinforcement learning. In: IJCAI (2019)
12. Cambria, E., Liu, Q., Decherchi, S., Xing, F., Kwok, K.: Senticnet 7: a commonsense-based neurosymbolic ai framework for explainable sentiment analysis. In: Proceedings of LREC 2022 (2022)
13. Chen, W., Ma, X., Wang, X., Cohen, W.W.: Program of thoughts prompting: disentangling computation from reasoning for numerical reasoning tasks. Preprint (2022). arXiv:2211.12588
14. Corchado, J.M., Borrajo, M.L., Pellicer, M.A., Yáñez, J.C.: Neuro-symbolic system for business internal control. In: Industrial Conference on Data Mining, pp. 1–10. Springer (2004)
15. Drancé, M.: Neuro-symbolic xai: application to drug repurposing for rare diseases. In: International Conference on Database Systems for Advanced Applications, pp. 539–543. Springer (2022)
16. Eiter, T., Higuera, N., Oetsch, J., Pritz, M.: A neuro-symbolic ASP pipeline for visual question answering. Preprint (2022). arXiv:2205.07548
17. Gupta, N., Lin, K., Roth, D., Singh, S., Gardner, M.: Neural module networks for reasoning over text. In: International Conference on Learning Representations (2019)
18. Hanson, D., Imran, A., Vellanki, A., Kanagaraj, S.: A neuro-symbolic humanlike arm controller for Sophia the robot. Preprint (2020). arXiv:2010.13983
19. Hatzilygeroudis, I., Prentzas, J.: Fuzzy and neuro-symbolic approaches to assessment of bank loan applicants. In: Artificial Intelligence Applications and Innovations, pp. 82–91. Springer (2011)
20. Jabrayilzade, E., Tekir, S.: LGPSolver – solving logic grid puzzles automatically. In: Findings of the Association for Computational Linguistics: EMNLP 2020, pp. 1118–1123 (2020)
21. Johnson, J., Hariharan, B., van der Maaten, L., Fei-Fei, L., Zitnick, C.L., Girshick, R.B.: Clevr: a diagnostic dataset for compositional language and elementary visual reasoning. In: CVPR (2017)
22. Kalithasan, N., Singh, H., Bindal, V., Tuli, A., Agrawal, V., Jain, R., Singla, P., Paul, R.: Learning neuro-symbolic programs for language guided robot manipulation. Preprint (2022). arXiv:2211.06652
23. Kapanipathi, P., Abdelaziz, I., Ravishankar, S., Roukos, S., Gray, A., Astudillo, R., Chang, M., Cornelio, C., Dana, S., Fokoue, A., et al.: Question answering over knowledge bases by leveraging semantic parsing and neuro-symbolic reasoning. Preprint (2020). arXiv:2012.01707
24. Karth, I., Aytemiz, B., Mawhorter, R., Smith, A.M.: Neurosymbolic map generation with VQ-VAE and WFC. In: The 16th International Conference on the Foundations of Digital Games (FDG) 2021, pp. 1–6 (2021)
25. Lavin, A.: Neuro-symbolic neurodegenerative disease modeling as probabilistic programmed deep kernels. In: International Workshop on Health Intelligence, pp. 49–64. Springer (2021)
26. Lee, J.H., Sioutis, M., Ahrens, K., Alirezaie, M., Kerzel, M., Wermter, S.: Neuro-symbolic spatio-temporal reasoning. Preprint (2022). arXiv:2211.15566

27. Lev, I., MacCartney, B., Manning, C.D., Levy, R.: Solving logic puzzles: from robust processing to precise semantics. In: Proceedings of the 2nd Workshop on Text Meaning and Interpretation, pp. 9–16 (2004)
28. Liao, H.C.: A survey of reinforcement learning with temporal logic rewards. TUM (2020). https://mediatum.ub.tum.de/doc/1579215/1579215.pdf
29. Mao, J., Gan, C., Kohli, P., Tenenbaum, J.B., Wu, J.: The neuro-symbolic concept learner: Interpreting scenes, words, and sentences from natural supervision. In: International Conference on Learning Representations. International Conference on Learning Representations, ICLR (2019)
30. Marques, N.C., Bader, S., Rocio, V., Hölldobler, S.: Neuro-symbolic word tagging. In: International Review on Computers and Software (2021)
31. Martin, L.J.: Neurosymbolic automated story generation. Ph.D. Thesis, Georgia Institute of Technology, 2021
32. Mei, L., Mao, J., Wang, Z., Gan, C., Tenenbaum, J.: Falcon: fast visual concept learning by integrating images, linguistic descriptions, and conceptual relations. In: International Conference on Learning Representations (2022)
33. Milicevic, A., Near, J.P., Singh, R.: Puzzler: An automated logic puzzle solver (2012). http://people.csail.mit.edu/jnear/puzzler/writeup.html
34. Mitchener, L., Tuckey, D., Crosby, M., Russo, A.: Detect, understand, act: a neuro-symbolic hierarchical reinforcement learning framework. Mach. Learn. **111**(4), 1523–1549 (2022)
35. Mitra, A., Baral, C.: Learning to automatically solve logic grid puzzles. In: Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, pp. 1023–1033 (2015)
36. Mitra, A., Clark, P., Tafjord, O., Baral, C.: Declarative question answering over knowledge bases containing natural language text with answer set programming. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 33, pp. 3003–3010 (2019)
37. Morel, G.: Neuro-symbolic AI for the smart city. In: Journal of Physics: Conference Series, vol. 2042. IOP Publishing (2021)
38. Palangi, H., Bosselut, A., Dasigi, P.: Neuro-symbolic methods for language and vision. In: AAAI 2022 Tutorial (2022)
39. Park, K.W., Bu, S.J., Cho, S.B.: Evolutionary optimization of neuro-symbolic integration for phishing URL detection. In: International Conference on Hybrid Artificial Intelligence Systems, pp. 88–100. Springer (2021)
40. Shakya, A., Rus, V., Venugopal, D.: Student strategy prediction using a neuro-symbolic approach. In: International Educational Data Mining Society (2021)
41. Wang, W., Yang, Y.: Towards data-and knowledge-driven artificial intelligence: A survey on neuro-symbolic computing. Preprint (2022). arXiv:2210.15889
42. Yang, Z., Ishay, A., Lee, J.: NeurASP: embracing neural networks into answer set programming. In: 29th International Joint Conference on Artificial Intelligence (IJCAI 2020) (2020)
43. Yang, S., Zhang, R., Erfani, S., Lau, J.H.: An interpretable neuro-symbolic reasoning framework for task-oriented dialogue generation. Preprint (2022). arXiv:2203.05843
44. Yi, K., Wu, J., Gan, C., Torralba, A., Kohli, P., Tenenbaum, J.: Neural-symbolic VQA: disentangling reasoning from vision and language understanding. Adv. Neural Inf. Process. Syst. **31** (2018)
45. Zheng, K., Zhou, K., Gu, J., Fan, Y., Wang, J., Li, Z., He, X., Wang, X.E.: Jarvis: a neuro-symbolic commonsense reasoning framework for conversational embodied agents. Preprint (2022). arXiv:2208.13266