Rule Induction and Approximate Symbolic Guidance in Neuro-Symbolic Learning

Lucile Dierckx

Thesis submitted in partial fulfilment of the requirements for the Degree of Doctor in Applied Sciences

January 2026

ICTEAM
Louvain School of Engineering
Université catholique de Louvain
Louvain-la-Neuve
Belgium

Thesis Committee:

Pr. Siegfried **Nijssen** (Advisor)	UCLouvain/ICTEAM, Belgium
Pr. Pierre **Schaus** (Advisor)	UCLouvain/ICTEAM, Belgium
Pr. Peter **Van Roy**	UCLouvain/ICTEAM, Belgium
Pr. Hélène **Verhaeghe**	UCLouvain/ICTEAM, Belgium
Pr. Benoît **Frenay**	UNamur/NaDI, Belgium
Pr. Eleonora **Giunchiglia**	Imperial College London, UK
Pr. Antonio **Vergari**	University of Edinburgh, UK

Rule Induction and Approximate Symbolic Guidance in Neuro-Symbolic Learning

by Lucile Dierckx

ICTEAM/INGI
UCLouvain
Place Sainte-Barbe, 2
1348 Louvain-la-Neuve
Belgium

This work was supported by Service Public de Wallonie Recherche under grant n°2010235 – ARIAC by DIGITALWALLONIA4.AI.

Abstract

While machine learning techniques, particularly deep learning, have achieved remarkable success across diverse domains, they often lack interpretability, robustness, and the ability to incorporate prior knowledge. On the other hand, symbolic reasoning methods provide a structured and verifiable approach to problem-solving, but struggle with noisy data and the reliance on expert-crafted knowledge bases. Neuro-symbolic approaches seek to combine the strengths of neural and symbolic methods to overcome these limitations.

This thesis explores two complementary axes of neuro-symbolic integration. First, we study how symbolic knowledge can be learned within differentiable neural architectures. To this end, we propose `RL-Net`, a neural framework for learning interpretable ordered rule lists that supports multi-class and multi-label classification while remaining fully differentiable. Second, we investigate how neural models can be trained under the guidance of symbolic knowledge. Here, the challenge we focus on lies in the high computational cost of evaluating complex logical rules. We explore approximation-based methods for incorporating such constraints into training, analyse their impact on learning, and identify limitations of existing approaches. Building on this, we introduce LUBAC, a framework that makes use of both lower- and upper-bounds of complex logical evaluations, compiled into arithmetic circuits. We also provide theoretical guarantees on the quality of gradient approximations obtained through these methods.

Acknowledgments

After giving myself a year to see how things would unfold and ultimately staying a bit more than four, along with many thoughts of "in the worst case scenario, I can always quit", I am now completing my PhD thesis. This work would not have been possible without the help, support, and encouragement of many people, to whom I am deeply grateful.

I am particularly grateful to my advisor, Siegfried, for his kindness, patience, and constant involvement, as well as for sharing his knowledge and experience. Throughout these years, I never felt stressed or pressured; instead, he consistently conveyed positivity and confidence, even during my less productive periods. I also thank him for his availability for discussions on any topic and for his attentive listening. His style of supervision was exactly what I needed to fully enjoy my experience as a PhD student. I also extend my sincere thanks to the members of my thesis jury for taking the time to read and evaluate my work.

During these years, I had the opportunity to take part in several collaborations. I would like to thank Rosana for our work on RL-Net, the many discussion, laughs, and occasionnal tears we shared. I thank Alexandre for having me into the Schlandals Corp.© and for our discussions. I also want to thank all the TRAIL members with whom I had the chance to work during the summer workshops and on the grands défis.

Beyond the work itself, these four years also allowed me to meet many great people, whom I thank for the moments we shared and hope to keep in touch with. I think in particular of the first-floor team (Achille, Alice, Augustin, Benoît, Emma, ...) and the Stevin team (Amaury, Benoît, Damien, Donatien, Édouard, François, Juliette, Théo, Wei), as well as other researchers I met during my time in INGI (François, Louis, Mathias, ...). A special thought goes to Vanessa for her constant kindness, energy and ability to answer any question; sharing the joys of teaching load assignments with her was always a pleasure. Throught my funding with TRAIL I also had the opportunity to meet and collaborate with many great people; among them, I especially thank Laura for first being the perfect roommate and soon becoming a good friend and a mutual emotional support.

Last but not least, I thank my family, former flatmates, friends, and cats for the joy, love, and encouragement they provided throughout these years.

Contents

Introduction 1

The rise of deep learning has significantly influenced many areas of computer science and artificial intelligence (AI), enabling the achievement of state-of-the-art performance in a wide range of tasks such as image segmentation and classification, text generation, unsupervised learning, and temporal data prediction [LBH15; Zha+23]. However, despite its capabilities, deep learning has several limitations. For instance, it generally requires large amounts of labelled data for training, lacks interpretability, robustness, can produce predictions that are inconsistent with domain knowledge, and performs poorly on tasks that demand complex reasoning [VT19; Von+21].

In contrast, symbolic reasoning has a long history of addressing problems involving structured reasoning, such as rule discovery, planning, and knowledge representation. It relies on the manipulation of discrete symbols and formal rules, making it well-suited to structured reasoning, verifiability, and capable of providing human-readable explanations for its conclusions. Nevertheless, symbolic systems also face important challenges. They typically require large expert-made knowledge bases, and are very sensitive to noise or incomplete information [VT19].

Parallels have been drawn between these two paradigms and the cognitive process of the human brain, suggesting that both paradigms should work together to be able to tackle more complex tasks. According to dual-process theories of reasoning, human cognition is governed by two systems. System 1, which is fast, intuitive, and automatic, and System 2, which is slower, more deliberate, and logical [Kah11]. Deep learning resembles System 1 in its efficiency and ability to handle low-level perceptual tasks, while symbolic reasoning is more similar to System 2, providing structured, rule-based reasoning that supports deliberative decision-making.

The interactions of these two paradigms have become a growing area of research in recent years, with the goal of developing AI systems that combine the flexibility and learning capacity of deep learning with the rigour and structure of symbolic reasoning [Giu+25]. This emerging field, known as neuro-symbolic artificial intelligence (NeSy AI), seeks to overcome the limitations of each paradigm by leveraging their complementary strengths [VT19; GL23].

Neuro-symbolic AI covers a broad range of topics and objectives,

all centred around the various forms of interactions between symbolic reasoning and deep learning. These interactions can take many forms. For instance, symbolic knowledge can be used to guide a gradient-based learning process, such as ensuring that predictions about road events in images align with the plausible actions of each type of actor [Giu+23]. Alternatively, neural networks can support symbolic reasoning to make a choice, for example, serving as a learned heuristic policy in a Monte-Carlo tree search algorithm [Sil+16]. Neuro-symbolic systems can also be designed to perform reasoning tasks directly over high-level perceptual inputs such as images or text, for example, answering questions about scenes composed of objects with various shapes, sizes, and colours [Joh+17; Mao+19].

1.1 Research Goals

This thesis investigates how the combination of symbolic reasoning and differentiable learning can address tasks that are challenging to solve using either paradigm on its own. Given the diversity of neuro-symbolic methods, this section clarifies the specific scope of the thesis and the main research axes pursued in our contributions.

From Interaction to Integration. Neuro-symbolic systems cover a spectrum of interactions between symbolic and neural components, varying in the degree of integration between the two. At one end are loosely coupled approaches, where the symbolic and neural parts merely *interact*. Examples include the use of symbolic simulations to generate training data for neural models or the embedding of symbolic inputs (e.g., text) into vector spaces for further processing by neural networks. These methods do not involve joint training or direct integration of symbolic and neural components.

At the other end are tightly *integrated* approaches, where symbolic knowledge is embedded into the neural architecture or vice versa, forming a unified framework that supports end-to-end training and inference.

This thesis focuses exclusively on this class of tightly integrated approaches, where symbolic and neural components cooperate within a single end-to-end learning system.

Symbolic Knowledge as Logic. The symbolic component used in neuro-symbolic frameworks can take many forms, such as physical laws, logical rules, or the manipulation of discrete symbols like words. However, many approaches represent symbolic knowledge in the form of logical constraints.

These logical constraints may be encoded in various formalisms, such as first-order logic or propositional logic, which are both commonly used.

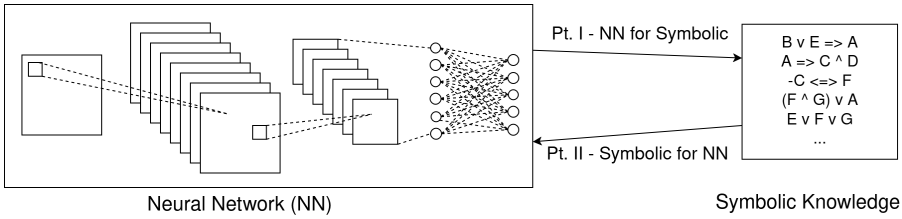


Figure 1.1: The two directions of neuro-symbolic integration explored in this thesis. The first part focuses on learning symbolic knowledge with the help of deep learning tools. The second part addresses the use of symbolic constraints to guide the training of neural models.

In this thesis, we focus on propositional logic, the simplest form of logic, which, despite its limited expressiveness, remains widely used. Indeed, it already allows for modelling a wide range of knowledge and is easily compatible with differentiable frameworks. Even in approaches based on more expressive logic, such as first-order, a grounding into propositional logic is often performed to enable integration with gradient-based learning [Fie+15; Vla+16].

Two Axes of Neuro-Symbolic Integration. This thesis explores two complementary aspects of *neuro-symbolic integration*, depicted in Figure 1.1.

1. **Learning symbolic knowledge within a differentiable architecture:** We investigate how symbolic structures, such as interpretable logical rules, can be learned directly from data within neural architectures while preserving the interpretability and formal structure of symbolic representations.

2. **Training neural models under symbolic constraints:** We study how domain-specific logical knowledge can be embedded into the training process of a neural model, encouraging logical consistency and incorporating domain knowledge into the training. A particular challenge addressed in this direction is the high computational cost of evaluating the satisfaction degree of complex logical constraints. Accordingly, we explore the use of methods that allow for working with complex symbolic knowledge in a differentiable training setting.

This thesis thus provides new insights in two directions of neuro-symbolic integration: learning symbolic models using deep learning tools, and training gradient-based models under symbolic constraints to enforce consistency with domain knowledge. These two directions will be explored in the two distinct parts of the thesis.

Technical Foundations. Achieving the integration of symbolic knowledge into neural networks in practice requires making symbolic knowledge differentiable. In the first part of the thesis, this is done through the use of specific neural architectures that incorporate rule-like behaviour, using dedicated *activation functions*, *layer types*, and *neural network constraints*.

In the second part, symbolic knowledge is compiled into a form that supports efficient gradient-based optimisation while allowing for the evaluation of the satisfaction of the symbolic constraints. Specifically, we rely on *weighted model counting (WMC)* to evaluate the degree of constraint satisfaction, and we compile the WMC of the logical constraints into *arithmetic circuits*, which support tractable and differentiable reasoning over symbolic knowledge.

These technical foundations, neural rule architectures, and symbolic circuit compilation are central tools on which our contributions are built. They will be introduced in more detail in the first and second parts of the thesis, respectively.

Handling Multi-Valued Settings. A recurring theme across both parts of the thesis is the treatment of *multi-valued* settings. These include multi-class and multi-label classification tasks, as well as logical models using multi-valued distributions. While many existing neuro-symbolic methods are designed for binary variables or binary classification, real-world tasks often require reasoning over multi-valued domains.

Accordingly, we extend or improve existing binary-focused techniques to effectively handle multi-valued predictions and distributions throughout the thesis.

1.2 Contributions

The main contributions of this thesis are as follows:

- **RL-Net, a differentiable rule-learning framework.** We introduce `RL-Net`, a framework for learning ordered rule lists within a differentiable neural architecture. By leveraging the structure of ordered rules lists, `RL-Net` supports both multi-class and multi-label classification tasks while maintaining full interpretability. This rule learning approach can thus be integrated in a wider differentiable framework, allowing it to be trained end-to-end with other differentiable modules.
- **A first approximation-based approach for parameter learning under complex constraints.** We propose a first method based on using approximations for incorporating complex symbolic constraints into a knowledge-guided learning process. This approach enables the

study of the impact on the learning process and the generalisation ability of the model of using approximations methods that provide various guarantees.

- **Identification of an important limitation in existing lower-bound methods for learning with complex constraints.** We show how the use of a lower-bound on the weighted model count for a formula can lead to the computation of gradients that significantly deviate from their exact value. However, in a learning setting, it is essential for the approximated gradient to serve as a reliable proxy for the gradient of the exact constraint. This insight highlights an important limitation of existing methods that rely solely on lower-bounds to handle complex symbolic constraints in knowledge-guided learning.

- **Upper-bound compilation into arithmetic circuits.** We present a method for compiling the upper-bound of the weighted model count of a formula into an arithmetic circuit.

- **LUBAC, a framework for learning with upper- and lower-bounds of complex constraints.** We introduce LUBAC, a framework for guiding the training of a neural network under symbolic constraints that are too complex to be fully compiled. LUBAC makes use of both lower- and upper-bound circuits of the true weighted model count.

- **Theoretical guarantees on gradient approximation.** We derive formal proofs establishing error bounds on the gradients obtained via the two proposed partial compilation techniques. These results offer theoretical guarantees on the quality of the gradient approximation when learning under complex symbolic constraints.

1.3 Publications

The main contributions of this thesis have been published in the following papers:

- L. Dierckx, R. Veroneze, and S. Nijssen. "Rl-net: Interpretable rule learning with neural networks". In: *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer. 2023, pp. 95–107. This paper presents `RL-Net`, a neural network architecture that learns a set of ordered rule lists for multi-class and multi-label classification tasks. The rules are learned in a differentiable manner, allowing the combination of the model with other neural components.

- L. Dierckx, A. Dubray, and S. Nijssen. "Parameter Learning Using Approximate Model Counting". In: *International Conference on Neural-Symbolic Learning and Reasoning*. Springer. 2024, pp. 80–88. This work focuses on neural parameter learning under complex symbolic constraints. More particularly, it introduces a first method based on partially compiled circuits and approximation techniques to handle learning with such hard constraints. This approach enables the study of how constraint approximations impact both the learning process and generalisation when full evaluation of the constraints is computationally infeasible. The paper also provides a theoretical bound on the gradient error resulting from the approximate constraints evaluation.

- L. Dierckx, A. Dubray, and S. Nijssen. "Learning from Logical Constraints with Lower- and Upper-Bound Arithmetic Circuits". In: *International Joint Conference on Artificial Intelligence*. 2025. This paper introduces LUBAC, a framework also designed for neural parameter learning under complex symbolic constraints. Its main contribution is the use of both a lower- and an upper-bound of the weighted model count of the logical constraints, enabling the computation of gradients that are guaranteed to lie within a bounded error of the true gradient.

In addition to the publications directly related to the main topics of this thesis, other works have been published during the course of it. Although they are not directly related to neuro-symbolic learning and will not be detailed in this manuscript, they are listed here for completeness:

- L. Dierckx, M. Beauvois, and S. Nijssen. "Detection and multi-label classification of bats". In: *International Symposium on Intelligent Data Analysis*. Springer. 2022, pp. 53–65.. This paper, extending the master's thesis co-authored with Mélanie Beauvois, presents a pipeline for detecting and classifying bat calls in audio recordings. It addresses multi-class and multi-label classification of individual calls. Given that existing open datasets' annotations only support binary detection or recording-level classification, the paper describes a method for the creation of an artificial dataset that permits fine-grained call-level classification.

- L. G. Jiménez, L. Dierckx, M. Amodei, H. R. Khosroshahi, N. Chidambaran, A.-T. P. Ho, and A. Franzin. "Computational Evaluation of the Combination of Semi-Supervised and Active Learning for Histopathology Image Segmentation with Missing Annotations". In: *Proceedings of the IEEE/CVF International Conference on Computer*

Vision. 2023, pp. 2552–2563.. This paper, presenting work that was accomplished for TRAIL's Grands Défis, explores semi-supervised and active learning approaches for gland segmentation in colorectal cancer tissue. It specifically considers scenarios where a portion of the dataset lacks annotations, a common occurrence in real-world digital pathology applications.

1.4 Outline

The thesis is organized as follows. Chapter 2 provides an overview of the main research directions in the field of neuro-symbolic AI and reviews proposed taxonomies for a wide range of existing neuro-symbolic contributions.

The thesis is then divided into two parts. Part I explores *Symbolic Integration in Neural Networks*. It focuses on the integration of symbolic knowledge into neural network architectures. It contains Chapter 3, which introduces `RL-Net`, a framework for learning ordered rule lists within a differentiable neural architecture. The use of ordered rules enables interpretability and supports both multi-class and multi-label classification tasks.

Part II is about *Learning from Complex Symbolic Constraints*. It explores how symbolic constraints can guide the training process of gradient-based models. Chapter 4 introduces the foundational concepts necessary for understanding knowledge-guided learning with symbolic constraints, and serves as a basis for the following chapters.

Chapter 5 presents a first approach for handling complex constraints through approximation techniques. It introduces bounds on the gradients computed using these approximations and analyses their impact on both the learning process and the model's generalisation ability. While this method allows the study of the impact of using approximations, it remains limited in its practical applicability.

To address this limitation, Chapter 6 introduces LUBAC, a framework for training under complex symbolic constraints by combining both lower- and upper-bounds of the weighted model count. The chapter also details a method for compiling upper bounds into arithmetic circuits and provides theoretical guarantees on the approximation error of the resulting gradients.

Finally, Chapter 7 concludes the thesis and outlines potential directions for future work.

Neuro-Symbolic Artificial Intelligence

2

The field of Neuro-Symbolic Artificial Intelligence (NeSy AI) is a rapidly evolving area of research that, as introduced in the previous chapter, aims to combine the strengths of neural models and symbolic reasoning. This hybrid approach is motivated by the recognition that, while deep learning has achieved state-of-the-art performance in a wide range of tasks, and particularly those involving high-dimensional data [LBH15; Zha+23], it also presents important limitations. These include a lack of interpretability, robustness issues, and a difficult incorporation of prior knowledge, all of which can hinder its development in critical domains [GAS16; MD19; VT19; Von+21].

In contrast, symbolic reasoning methods, though sensitive to noise and often requiring expert knowledge, perform well in areas where neural methods fall short [VT19]. These include robustness to out-of-distribution data, interpretability, and the ability to perform explicit reasoning [VT19]. As such, neuro-symbolic AI seeks to develop systems that can integrate the learning and generalisation capabilities of neural networks with the structure, interpretability, and reasoning capabilities of symbolic approaches.

This chapter provides an overview of the wide range of research directions falling into the neuro-symbolic AI domain. It discusses the motivations for integrating neural and symbolic methods, the challenges that arise from this integration, and the diversity of existing approaches.

In the literature, the *symbolic* component of neuro-symbolic AI is interpreted in two main ways: broadly, to include any form of explicit symbol manipulation (e.g., natural language), or, more narrowly, as a logic-based reasoning [Sar+22]. As the contributions of this thesis align with the latter interpretation, this chapter will primarily focus on logic-based neuro-symbolic systems, while still presenting a more general categorisation to provide a complete view of the field.

The chapter is structured as follows: Section 2.1 introduces the characteristics of symbolic reasoning and deep learning, detailing their respective strengths and the motivation for their integration. It also outlines the main challenges involved in combining the two paradigms. Section 2.2 then reviews existing categorisations of neuro-symbolic approaches, illustrating

the variety of integration strategies. Section 2.3 focuses on logic-based neuro-symbolic systems, discussing the main types of logic used and how these can be imposed as hard or soft constraints. Finally, Section 2.4 concludes the chapter by summarising the main concepts and situating the contributions of this thesis within the presented neuro-symbolic landscape.

2.1 Neural and Symbolic Methods Characteristics and Combination

Symbolic and neural methods represent two distinct paradigms in artificial intelligence, each having its own strengths and limitations. Understanding their complementary nature is important for motivating their integration in neuro-symbolic AI.

Neural Methods. Deep learning methods, particularly neural networks, have shown remarkable success in learning patterns and representations from large datasets, without requiring preliminary feature engineering. They are especially well-suited for handling high-dimensional data and are robust to noise, commonly found in real-world datasets. This makes them highly effective in a wide range of applications such as detection and classification [LBH15], image segmentation [RFB15] and generation [Goo+20], object detection [Red+16], and natural language processing [Ach+23].

However, neural methods also come with notable drawbacks. Beyond achieving good task performance, many applications require models that are *interpretable* [Rud19]. A popular definition of interpretability is provided by Miller [Mil19], who describes it as "the degree to which a human can understand the cause of a decision". Neural networks, with their large number of parameters and non-linear operations, generally lack this property, making them less suitable for use on sensitive applications. In addition, they typically require large amounts of labelled data for training, are vulnerable to adversarial perturbations, and often fail to generalise to out-of-distribution data [VT19; Von+21]. Furthermore, incorporating prior domain knowledge into such models remains challenging [Giu+25]. These limitations are especially problematic in safety-critical domains, where robustness, trust, and interpretability are essential [Rud19].

Symbolic Methods. Symbolic reasoning, on the other hand, is based on the manipulation of discrete symbols and formal rules, allowing for structured reasoning and globally explainable decision-making [Gri09]. It allows for the precise encoding of domain knowledge and can generalise well on limited data [Man+21a]. Symbolic methods are abstract, which facil-

itates out-of-distribution generalisation [GS19]. Additionally, their symbolic representation can provide clear justification for their outputs [GS19].

Nonetheless, symbolic approaches also face limitations [VT19]. They are ill-suited for processing raw high-dimensional data (e.g., images) and are sensitive to noise or incomplete information, which can lead to incorrect conclusions. Moreover, the construction of effective symbolic systems often requires extensive domain expertise, making them less accessible for many applications. Reasoning with complex symbolic representations can also be computationally expensive, which limits scalability.

Neuro-Symbolic Combination. Neuro-symbolic AI seeks to combine the strengths of both paradigms, leveraging the generalisation power of neural models and the structured reasoning capabilities of symbolic methods. Such hybrid systems aim to learn from high-dimensional data with fewer labelled samples, while improving robustness, interpretability, and reasoning ability [Sar+22].

This integration has been described as the future of AI in Kautz's lecture on the "third AI summer" [Kau22], delivered at AAAI 2020 when he received the Robert S. Engelmore award. Kautz characterises the first summer as dominated by logic-based approaches, the second summer by expert-based systems, and the third by deep learning. Looking ahead, he argues that the future of AI lies in systems that can not only learn from data but also reason about it in a structured and interpretable manner.

A central challenge in neuro-symbolic integration is the representation mismatch between the two paradigms. Symbolic reasoning, in its stricter logical sense, typically operates on discrete symbols and Boolean logic, relying on enumeration and search to find an optimal or exact solution. In contrast, neural models process and output continuous values, and use gradient-based optimisation [Sar+22]. Addressing this discrete–continuous incompatibility requires relaxing logical constraints to permit differentiable, continuous representations. Two main strategies have been considered to address this representation issue [Mar+24]. The first one is to use probabilistic logic, which associates uncertainty with the logical facts, allowing for a probabilistic interpretation of the logical constraints. The second one is to introduce a degree of truth to the logical facts, allowing for a continuous representation of the symbolic knowledge using fuzzy logic. Both approaches are discussed in more detail in Section 2.3.

Another fundamental requirement in this neuro-symbolic integration is the need for differentiability within the symbolic component (i.e., through the logical operators) in order to support end-to-end gradient-based learning. This can be achieved in various ways, for example, by embedding logical rules within neural architectures [QWL21], or by

compiling symbolic constraints into differentiable representations such as arithmetic circuits [Dar03]. These two strategies are explored in the first and second parts of this thesis, respectively.

The nature of imposed symbolic constraints also plays a role in the neuro-symbolic task. Hard constraints require complete satisfaction of the knowledge, while soft constraints allow the model to balance between constraint satisfaction and data-driven learning. This distinction is further explored in Section 2.3.

Finally, scalability and tractability are additional central aspects in neuro-symbolic AI. Symbolic inference can become computationally intractable when scaling to large datasets or complex constraints [Val79]. To address this, recent works explore the use of approximation methods, partial knowledge compilation, or neural approximations [MMD21; Hua+21; Kri+23; MDD24; Ver+24; Dan+24]. These aspects are central to the second part of this thesis, with Chapter 5 exploring approximation techniques and Chapter 6 working with partial compilation.

2.2 Categorisations of Neuro-Symbolic Approaches

A significant body of work has focused on combining logical background knowledge with neural models. Numerous surveys have been published to identify the main trends and challenges in the field [BH05; HH07; GLG09; Bes+21; HS22; Sar+22; GL23]. As the field continues to evolve rapidly, it still lacks of a unified terminology and conceptual framework, which has motivated many literature reviews to propose structured taxonomies [BH05; VT19; Bek+21; Von+21; HS22; Kau22; Fel+24]. Among these, none has achieved consensus among researchers, yet the one presented by Kautz in its AAAI 2020 lecture [Kau22] is often referred to. We describe it here, as it attempts to capture the wide range of approaches that fall under the umbrella of neuro-symbolic AI and remains a central reference point in discussions of the field's scope. This section hence presents Kautz's broad categorisation, followed by brief summaries of other taxonomy works.

Kautz identifies six classes of neuro-symbolic systems, based on how the symbolic and neural components interact:

1. **Symbolic Neuro Symbolic.** Symbolic data is first vectorised and processed by a neural model, which in turn produces symbolic outputs. This is the common methodology in natural language processing, where neural networks handle text input and generate symbolic outputs such as translations or summaries.

2. **Symbolic[Neuro].** A neural model is embedded within a symbolic system to replace or enhance specific components, such as heuristics

or search models. AlphaGo [Sil+16], which uses a neural network as a heuristic to guide the symbolic game tree search, exemplifies this category.

3. **Neuro|Symbolic.** Neural models convert non-symbolic data (e.g., images) into symbolic representations, which are then processed by a symbolic reasoning system. Both components are trained separately, with a unidirectional data flow.

4. **Neuro: Symbolic → Neuro.** The neural model is trained classically but follows a training that is somehow based on symbolic knowledge. This is, for instance, the case when training on synthetic data generated by a symbolic system, as, for instance, additive manufacturing processes, where physical simulations are used to generate the training data.

5. **Neuro_{Symbolic}.** Neural networks are explicitly designed to learn symbolic operations or emulate symbolic behaviour. This includes cases where neural networks are trained to perform logical operations on tensors, such as Logic Tensor Networks [DSG17], or where neural architectures are designed to learn ordered rule lists classification, as in our work on `RL-Net` [DVN23b] described in Chapter 3.

6. **Neuro[Symbolic].** Symbolic reasoning is tightly integrated in the architecture of a neural model, enabling joint, end-to-end training. This includes DeepProbLog [Man+18], where neural outputs parametrise a probabilistic logic program. Our works in the second part of the thesis [DDN24b; DDN25b] also fit into this category, as it is based on the same basic pipeline as DeepProbLog.

Beyond Kautz's categorisation, several other surveys offer complementary perspectives.

Bader et al. [BH05] present a unifying view of the neuro-symbolic cycle, where neural and symbolic components interact and mutually benefit from each other. Their framework classifies approaches along three dimensions: interrelation, language, and usage, which cover a total of eight different aspects. This classification is analysed, along with Kautz's categorisation, in the survey of Sarker et al. [Sar+22], which also offers a historical overview of the terminology and a breakdown of the distribution of research across subcategories. Other surveys, such as [VT19; Bek+21], make use of design patterns as building blocks to characterise the variety of approaches in the field.

From a neural-based perspective, Von Rueden et al. [Von+21] introduce a taxonomy of informed machine learning methods, which broadly covers

methods that integrate prior knowledge into learning. It distinguishes the knowledge sources, knowledge representations, and integration strategies, and identifies mechanisms such as constraints, regularisers, and circuits as common integration methods. Also using a neural point of view, Feldstein et al. [Fel+24] propose a map of neuro-symbolic frameworks focused on augmenting neural models with symbolic components. Their taxonomy classifies systems based on architecture, supervision type, and the form of logical integration.

This overview highlights the diversity of approaches and interpretations in neuro-symbolic AI. In the rest of the thesis, we focus on cases where the "symbolic" part of "neuro-symbolic" refers to the use of formal logic instead of the more general view of symbol manipulation (such as handling of text). Given the wide scope of the field and the more focused nature of our contributions, we do not review individual methods in depth here. Instead, each chapter of the thesis includes a dedicated related work section that discusses prior work most relevant to that chapter's contributions. Nevertheless, the surveys and categorisations introduced in this section offer a broad entry point into the field and serve as valuable references for readers wishing to explore further.

2.3 Logic as Neuro-Symbolic Constraint

Logic provides a formal language for representing and reasoning about knowledge. In the context of neuro-symbolic AI, logical representations can encode prior knowledge, enforce desirable properties, or shape learning objectives.

In this section, we present the main types of logic commonly used in neuro-symbolic systems, highlighting their respective expressiveness, suitability for integration with neural components, and typical roles in learning and inference. We then discuss how these logical representations can be imposed as hard or soft constraints on neural models, enabling the integration of symbolic knowledge into the learning process.

2.3.1 Types of Logic

Different logic systems offer varying levels of expressivity and reasoning capabilities. In neuro-symbolic AI, the most commonly used forms include propositional logic, first-order logic, probabilistic logic, and fuzzy logic.

Propositional Logic Propositional logic [BL99] is the simplest form of logic, where atomic statements, represented as propositional variables (e.g., proposition P, representing the fact that "it is sunny today"), are assigned

binary truth values: true or false. Propositional formulas are constructed using logical connectives $\wedge$ (and), $\vee$ (or), $\neg$ (not), $\rightarrow$ (implication), and $\leftrightarrow$ (biconditional). Given the truth values of the atomic propositions, the truth of the overall formula can be derived following the rules of operator precedence and the truth tables of the logical connectives.

> **Example 2.3.1.** (Propositional Logic)
> The formula $P \wedge Q \rightarrow R$ expresses that if both P and Q are true, then R must also be true.

Propositional logic is widely used in neuro-symbolic AI to encode structured domain knowledge [Xu+18; Ahm+22b; Giu+24]. In this thesis, all symbolic aspects are expressed using propositional logic as well, both for the rule lists learned in Part I of the thesis and the different knowledge structures used in Part II and described in Section 4.1 of the thesis.

First-Order Logic First-order logic extends propositional logic by enabling reasoning over objects and their relations. It introduces quantifiers $\forall$ (for all) and $\exists$ (there exists), as well as variables (e.g., x,y), predicates (e.g., $P(x)$), constants (specific objects in the domain), and functions (mapping from objects to objects). This expressivity allows for more compact and general knowledge representation, especially when dealing with structured domains.

> **Example 2.3.2.** (First-Order Logic)
> The formula $\forall x \exists y (P(x) \wedge Q(y))$ states that for every object x, there exists an object y such that both P on x and Q on y hold.

First-order logic is commonly used in neuro-symbolic systems to capture relational knowledge [Man+18; MMD21; Man+21a; Bad+22]. However, its integration with neural networks often requires grounding, a process in which variables and quantifiers are substituted with concrete objects from a finite domain, resulting in a propositional formula, which is required for in many neuro-symbolic integrations [Fie+12; Fie+15; Vla+16]. This grounding of first order logic into a propositional form for integration in a learning setting is further discussed in Section 4.2.4.

In a neuro-symbolic learning setting, propositional and first-order logic cannot be directly combined with neural models because they operate on discrete truth values, whereas neural networks rely on continuous computations and gradient-based optimisation. To address this incompatibility, most approaches introduce a form of relaxation. The binary constraints of logic are replaced or approximated with continuous semantics, such as fuzzy logic operators or probabilistic interpretations. In this way, logical

rules can be mapped into differentiable functions that are compatible with neural architectures. This relaxation step makes it possible to propagate gradients through symbolic constraints, thereby enabling the joint optimisation of neural parameters and logical knowledge. The next two paragraphs introduce the main aspects of fuzzy and probabilistic logic, which are the most commonly used methods to bind the discrete values of logic to the continuous outputs of neural models.

Probabilistic Logic One possible solution to this domain discrepancy is to use probabilistic logic [Nil86]. The latter offers a principled way to incorporate uncertainty by assigning probabilities to logical statements, thus allowing propositions to be true with some likelihood. Probabilistic logic can be used to extend both propositional and first-order logic to represent knowledge with uncertainty [DKT07].

> **Example 2.3.3.** (Probabilistic Logic)
> Knowing that P is true with a probability of 0.8, Q is true with a probability of 0.9, and R is true with a probability of 0.2, we can compute that the implication $P \wedge Q \longrightarrow R$ has a probability of $0.8 \times 0.9 \times (1 - 0.2) = 0.576$ to be violated.

Many works in neuro-symbolic AI use probabilistic logic to represent knowledge and to enable reasoning and inference with uncertainty. In that case, the neural model is often used to learn the probabilities of the propositional variables [Man+18; Man+21a; MMD21; Xu+18]. The logical formula is then commonly used to assess the model's consistency with the knowledge encoded in the logical formula. It can be used to guide the model towards a desired behaviour, which is what we explore in Part II of the thesis, or to ensure that the model's outputs are at all times consistent with the knowledge encoded in the logical formula.

Fuzzy Logic Another approach to reconciling the continuous outputs of neural networks with the discrete representations of Boolean logic is the use of fuzzy logic [Háj01]. Fuzzy logic extends classical Boolean logic to a many-valued framework, allowing truth values to range continuously within the interval $[0, 1]$. Rather than being simply true or false, a statement in fuzzy logic can be partially true, capturing the vagueness of many real-world concepts. For instance, the proposition "it is early" may hold to a degree of 0.9 at 6 AM and of 0.6 at 10 AM, depending on how "early" is defined.

This degree of truth is defined through a membership function that maps elements to their degree of truth with regard to a defined concept

(i.e., mapping hours to a degree of earliness). Importantly, such degrees are not probabilities. A membership degree expresses how much a statement holds (vagueness), while a probability indicates how likely a statement is to hold (uncertainty). In fuzzy logic, logical connectives such as conjunction ($\wedge$), disjunction ($\vee$), and implication ($\rightarrow$) are redefined using smooth operations over these truth degrees. For instance, the conjunction of two fuzzy statements is replaced by an operation called the t-norm, the disjunction with the t-conorm operation, and the negation and implication with additional generalised fuzzy operations. Different fuzzy logics can be defined by choosing different t-norms and t-conorms, leading to various interpretations of the logical connectives. Popular fuzzy logics include Łukasiewicz logic, Gödel logic, and product logic, each with its own semantics for the logical operators. However, not all of these are equally suited to gradient-based learning. For instance, [VAV22] shows that some popular fuzzy operators are ill-suited for use in a differentiable learning task.

Example 2.3.4. (Fuzzy Logic)
Suppose we have a fuzzy logical formula $P \wedge Q \rightarrow R$, with fuzzy truth values: $P = 0.8$, $Q = 0.9$, and $R = 0.2$.

Using the Łukasiewicz interpretation of the logical operators, we can first compute the conjunction $P \wedge Q = \max(0, P+Q-1) = \max(0, 0.8+0.9-1) = 0.7$. Then, we can compute the implication as $P \wedge Q \rightarrow R = \min(1, 1 - (P \wedge Q) + R) = \min(1, 1 - 0.7 + 0.2) = 0.5$.

Hence, the fuzzy logical formula is satisfied to a degree of 0.5.

Fuzzy logic has been widely used in neuro-symbolic systems, often to implement soft logical constraints, like a loss term, that can guide the neural learning. For instance, Logic Tensor Networks [DSG17; Bad+22] handle the discrete/continuous gap between logic and neural models by performing fuzzy logical inference over continuous neural embeddings of logical atoms.

2.3.2 Soft and Hard Constraints

The integration of symbolic knowledge with neural models can be approached in different ways, depending on the type of knowledge and the desired degree of compliance between the model and the symbolic structure. The two major paradigms of integration emerge in this context. First, the use of *soft constraints*, which act as a guiding signal during training. Second, the use of *hard constraints*, which enforce strict compliance with the knowledge at all times of the training and inference process.

In the soft constraint paradigm, the symbolic knowledge is incorporated as an inductive bias. Violations of the constraints are tolerated but penalised, allowing for flexibility in learning. A common approach is to incorporate the constraints into the loss function as a regularisation term [Xu+18; Ahm+22a]. A well-known example of this approach is the semantic loss [Xu+18], which penalises predictions that are inconsistent with a given logical formula. Another common way to enforce soft constraints is illustrated with DeepProbLog [Man+18], where the neural module outputs probability distributions over atoms, and inference is performed over a probabilistic logic program. The system is trained end-to-end, with gradients backpropagated through the symbolic reasoning component. Although the model is encouraged to comply with the symbolic knowledge, strict satisfaction is not enforced. This soft constraining of the neural model output is what we explore in Part II of the thesis, following a similar setting to the one defined for DeepProbLog. While previous examples are based on probabilistic logic, soft constraints are also employed in settings based on fuzzy logic. For instance, in Logic Tensor Networks [DSG17; Bad+22], predicates, functions, and constants are modelled by neural components, and logical formulas are evaluated using fuzzy semantics. Here again, the model learns to approximate consistency with the logical knowledge without guaranteeing full compliance.

In contrast, hard constraints impose strict adherence to symbolic knowledge, requiring the model to fully comply with the constraints during both training and inference. One approach is to augment the neural model with a symbolic reasoning component that enforces compliance, adding extra layer(s) to the neural component [GL20; GL21; Ahm+22b; Sto+24; Giu+24]. For example, the Semantic Probabilistic Layer [Ahm+22b] performs structured output prediction by adding an additional layer that ensures the model's output is consistent with the symbolic knowledge. This layer combines information from a probabilistic circuit and a constraint circuit, effectively enforcing hard constraints on the model's predictions. Similarly, in the Coherent by Construction Network [Giu+24], extending [GL20; GL21], propositional logic constraints are enforced through a dedicated requirement layer and requirement loss that guarantees that all outputs satisfy the given logical knowledge. Another form of hard constraint enforcement lies in designing the architecture of the model itself to satisfy specific symbolic properties by construction. This is what we do in Part I of the thesis and what is done in `DR-Net` [QWL21], which serves as a basis for our contribution. In these works, the network is constrained to learn decision rules directly. The model is designed such that all learned functions always represent rule-based classifiers, thereby guaranteeing interpretability and compliance with the rule structure.

2.4 Conclusion

This chapter introduced the dual paradigms of symbolic reasoning and deep learning, highlighting their respective strengths and limitations, and motivating their integration within the field of neuro-symbolic AI. It then discussed important challenges associated with this integration, namely the representational mismatch between symbolic and sub-symbolic components, the requirement for differentiability, and the issues of scalability and tractability in logical reasoning.

Two important considerations when using logic as the symbolic component were subsequently examined: the different types of logic commonly used in neuro-symbolic systems, and the distinction between soft and hard constraints in integrating symbolic knowledge into learning models.

With this background, we can now position the contributions of this thesis within the broader presented neuro-symbolic landscape. Regarding the type of logic used, both parts of the thesis operate in the propositional logic setting. The first part employs binary propositional logic, while the second part extends this to a probabilistic relaxation to train a neural model end-to-end.

The first part of the thesis fits the category of a *Neuro_{Symbolic}* system, as it focuses on learning symbolic knowledge in the form of interpretable rules within a neural network architecture. This part enforces hard constraints, as the neural model is explicitly constrained to represent a rule classifier, and as the constraint is enforced at all times by the network's architecture.

The second part aligns with a *Neuro[Symbolic]* approach, in which the parameters of a neural model are directly integrated into a symbolic reasoner, used to guide the model towards a desired behaviour. The symbolic knowledge is incorporated as soft constraints, guiding the model towards a desired logical consistency but without requiring full compliance. In addition, the second part addresses the scalability and tractability challenges of symbolic inference by proposing to work with approximate reasoning mechanisms and partial compilation of the logical knowledge, enabling integration with neural learning while maintaining computational feasibility.

Part I

Symbolic Integration in Neural Networks

Interpretable Rule Learning with Neural Networks

3

In the previous chapters, we have seen that, while deep learning has led to remarkable advances across a wide range of tasks, its models are typically difficult to interpret. Neural networks typically operate as opaque models, capable of producing highly accurate predictions while offering little insight into the reasoning behind their decisions. As discussed earlier, this lack of transparency remains an important limitation in many real-world applications, such as medicine, healthcare, criminal justice, and education, where understanding the reasoning behind a model's predictions is essential for trust, accountability, and safety [Rud19].

Symbolic models, by contrast, are inherently interpretable. Rule-based classifiers and decision structures provide explicit reasoning chains and transparent decision criteria that are easy to inspect, debug, and explain. For this reason, symbolic methods are often preferred in domains that demand interpretability, even if their predictive performance sometimes falls short of that of neural networks.

However, symbolic approaches face a different set of challenges. Classical rule learning algorithms typically rely on combinatorial search or heuristic optimisation, which limits their compatibility with modern machine learning pipelines. In particular, they cannot easily benefit from the rich ecosystem developed around neural networks, including gradient-based optimisation, differentiable loss functions, advanced regularisation techniques, and hardware acceleration. Neural networks, on the other hand, naturally operate within this ecosystem, providing a flexible, well-understood framework for optimisation and training, which has made them the dominant approach in many machine learning tasks.

This contrast between the interpretability of symbolic models and the performance and scalability of neural networks raises many interesting questions. One of them is whether tools and architectures of deep learning can be used to discover symbolic structures, such as rules, decision lists, or other discrete models, so that the models remain both interpretable and scalable.

This question has motivated the development of approaches situated in the *Neuro_Symbolic* category of Kautz's taxonomy (see Section 2.2), where symbolic components are embedded within and, in some cases, learned by neural architectures.

This chapter explores this direction, specifically investigating how neural networks can be used to learn rule-based classifiers. The goal is to make rule learning differentiable and compatible with gradient-based optimisation, allowing the use of the wide range of tools developed in the neural network training literature, from loss functions and optimisation algorithms to regularisation methods and automated training procedures.

Some existing methods, such as `BinaPs` [FV21] for neural pattern set mining and `DR-Net` [QWL21] for neural rule set learning, already employ task-specific neural architectures to learn patterns and rule sets in a differentiable manner. These approaches successfully formulate rule learning as a neural optimisation problem, enabling interpretable models to be trained using standard neural tools. However, both methods are designed for binary classification and rely on architectures tailored to that setting. Extending them to multi-valued tasks, such as multi-class or multi-label classification, while preserving interpretability, will be explored in the following sections.

In this first part of the thesis, we address this multi-valued limitation by extending the Decision Rules Network (`DR-Net`) proposed by Qiao et al. [QWL21]. We incorporate the possibilities of (a) using rule precedence among the rules, hence adding the possibility of learning classifiers based on *rule lists* in addition to *rule sets*, and (b) solving multi-class classification problems. Unlike `DR-Net`, where all rules share the same fixed consequent (class label), our approach learns both the rule conditions and their associated class labels. Finally, our proposal can easily be tweaked to solve multi-label classification problems.

While our main focus here is on enabling differentiable, gradient-based rule learning, this framework also opens the way toward future extensions that integrate rule learning into larger neural systems, potentially bridging symbolic interpretability with neural scalability to high-dimensional and unstructured data.

The remainder of the chapter is organised as follows. Section 3.1 reviews related work on rule learning and gradient-based rule learning. Section 3.2 presents the architecture of our proposed interpretable Rule Learning neural Network (`RL-Net`). Section 3.3 details the experimental setup, including datasets, baselines, and evaluation metrics, and presents the results. A discussion of the perspectives and work published after the submission of this chapter is presented in Section 3.4. Finally, Section 3.5 concludes this chapter.

3.1 Related Work

In this section, we first review the main class of methods to perform rule learning. Then, existing methods for neural rule learning are discussed, along with the shortcomings of existing methods and how our work addresses them.

3.1.1 Rule Learning

Rule-based classification models form a prominent class of interpretable machine learning approaches. They provide symbolic, white-box representations of the decision process in the form of simple IF-THEN rules, which are logically equivalent to propositional logic formulas expressed in Disjunctive Normal Form (DNF), that is, a disjunction ($\vee$) of conjunctions ($\wedge$). These rule-based models offer transparent predictions, making it possible for users to trace and understand how decisions are made. In addition to interpretability, rule-based models also implicitly perform feature selection, as only a subset of features typically appears in the learned rules.

A distinction can be made between rule *sets* and rule *lists*. In a rule set, all rules are evaluated independently. If at least one rule matches the considered example, the predicted class label is determined by the activated rules; otherwise, a default class label is assigned. In contrast, a rule list imposes an order over rules, and the first rule of which all conditions in the IF-part are satisfied is used to perform a prediction. An advantage of the rule list approach is that the resulting models are also interpretable on classification tasks with more than two classes, since at all times only one rule is used to perform the prediction. Rule sets, by contrast, typically rely on voting mechanisms or are limited to binary tasks where all rules predict the same class, and the default accounts for the second. This makes these methods less interpretable for multi-class problems. For this reason, we focus on rule lists in this work.

Roughly speaking, two classes of approaches for learning rule-based models can be distinguished. A common strategy for rule learning relies on *pattern mining*. Traditional pattern mining is formulated as the problem of computing the set of all patterns

$$\mathrm{Th}(\mathcal{L}, \varphi, \mathcal{D}) = \{\pi \in \mathcal{L} | \varphi(\pi, \mathcal{D}) \text{ is true}\},$$

where $\mathcal{D}$ is the dataset, $\mathcal{L}$ is a language of patterns, and φ is a constraint, often based on support [GND11a; GND11b]. The size of the search space of this problem is exponential in the size of $\mathcal{L}$.

The number of all patterns satisfying the constraints is usually too large. Thus, patterns are often post-processed in a step-wise procedure

to become useful [GND11b]. In the first step, the patterns that meet the constraints are enumerated. In the second step, some patterns are selected and combined. Again, we have another search space of exponential size, in this case, in the size of $\mathrm{Th}(\mathcal{L}, \varphi, \mathcal{D})$.

Most methods adopt heuristics to select and combine the patterns, which is commonly the case for associative classification proposals, such as CBA [LHM+98] and CMAR [LHP01]. These methods solve one particular instance of the *pattern set mining* problem, which consists in computing

$$\mathbf{Th}(\mathcal{L}, \varphi, \psi, \mathcal{D}) = \{\Pi \subseteq \mathrm{Th}(\mathcal{L}, \varphi, \mathcal{D}) | \psi(\Pi, \mathcal{D}) \text{ is true}\},$$

where ψ expresses constraints that have to be satisfied by the overall pattern set [GND11a; GND11b]. The major drawback of the step-wise procedure is that it does not scale well.

The second class of methods scales better. Instead of first mining patterns, these approaches learn the rules themselves, also using heuristics; typically, they use a heuristic to iteratively add the most promising condition to a rule. A classical example is the RIPPER [Coh95] algorithm, which learns rules in a greedy manner. It starts with an empty rule and iteratively adds the most promising condition to the rule until no more conditions can be added without violating a user-defined threshold on a rule's properties, such as the number of covered examples or the rule's length. The algorithm then removes the covered examples from the dataset and repeats the process until no more rules can be learned. Tree-based methods, such as CART [Bre+17], can also be seen as a special case of this approach. They learn a decision tree by recursively splitting the dataset into subsets based on a heuristic that maximises the information gain. The leaves of the tree can then be interpreted as rules, where each path from the root to a leaf represents a rule condition.

While, as a consequence of using heuristics, these approaches find rules more quickly, the heuristics are often specific to one learning task and may have as effect that the algorithm overlooks good rules.

3.1.2 Neural Rule Learning

The use of neural networks for rule learning is more recent and is still in its early stages. The central idea is to make use of neural architectures and gradient-based optimisation to discover rules in a differentiable manner, avoiding exhaustive combinatorial search.

A first relevant example is the Soft Decision Trees (SDTs) introduced by Frosst and Hinton [FH17]. Although SDTs do not directly rely on a neural network architecture, they follow the same differentiable learning paradigm. Each branching decision is replaced by a sigmoid gate, producing

a soft routing of inputs through the tree. Leaf nodes store learnable output distributions, and the final prediction is a mixture of leaf outputs weighted by their path probabilities. This formulation allows end-to-end optimisation via gradient descent, yielding smoother decision boundaries but reduced interpretability, as predictions no longer correspond to a single deterministic rule path.

Based on a classical neural architecture, the work of Beck and Fürnkranz [BF21a; BF21b] learns rule sets to perform binary classification using a network structure. However, the network weights are learned using a greedy heuristic instead of a gradient-based approach.

Yang et al. [YMH18] introduced the Deep Neural Decision Tree (DNDT) method, which mimics the structure of a decision tree using a neural network architecture. In this proposal, the weights are trained with a gradient descent algorithm. The splitting value for each attribute and the rule labels are learned during the training. The model is thus suitable for binary and multi-class classification. The key limitation of their proposal is that it is not scalable with regard to the number of features. In their experiments, they could only find an accurate single tree for datasets with at most 12 features. Moreover, the limitation of a tree structure makes it impossible to learn arbitrary rule lists.

The Explainable Neural Rule Learning (ENRL) method [Shi+22] also learns rules in a differentiable manner coupled with a neural network structure. The Explainable Condition Module (ECM) is the building block of the method. It comprises a feature, an operator, and a value, learning atomic propositions such as $age \geq 18$. Based on the atomic propositions, ENRL adopts a complete binary tree topology to express multiple rules, and the problem of seeking appropriate rules is transformed into a neural architecture search. ENRL creates an ensemble of trees, and the final decision is made by a voting mechanism, which makes this method less interpretable. Also, it is limited to binary classification.

Most relevant to our work, as our proposed model is based on it, the Decision Rules Network (`DR-Net`) of Qiao et al. [QWL21] learns rule sets for binary classification. The model is composed of three layers: the input layer, the Rules layer, and the OR layer. The *Rules layer* learns the rules. The number of neurons in this layer is a user-defined parameter that sets the maximum number of rules. Its regularisation term controls the length of the rules. The *OR layer* chooses which rules to use and which to ignore. Its regularisation term controls the number of rules that will be used in the classifier. The network training is done in two alternating phases, one for the Rules layer and one for the OR layer. `DR-Net` learns rule sets, and we base our `RL-Net` architecture on it to extend it to multi-class classification.

In summary, existing neural approaches have demonstrated the feasibility of learning symbolic models within neural architectures, but they remain restricted in scope. Most focus exclusively on binary classification and are limited to rule sets or tree structures, falling short of arbitrary rule lists. In this work, we address these gaps by proposing RL-Net: a neural network architecture for fully interpretable rule-based classifiers, applicable to both binary and multi-class tasks, and readily extendable to multi-label classification.

3.2 Approach

This section introduces our Rule Learning neural Network (RL-Net), a neural architecture designed to learn interpretable rule lists for multi-class classification. The section describes the RL-Net architecture and training procedure, as well as its extension to multi-label classification.

3.2.1 RL-Net

RL-Net was conceived to learn interpretable rule lists, extending the DR-Net architecture [QWL21] from rule sets to rule lists, and from binary to both binary and multi-class classification. RL-Net employs the structure of a neural network as well as its gradient optimisation learning methods.

The network is composed of four layers, as depicted in Fig. 3.1. The first layer is the input layer that receives the dataset's features. It is connected to the rule layer, where the rules' conditions are learned. The next layer expresses the precedence among the rules, which naturally enforce that only one rule at a time is active and is necessary to learn a rule list instead of a rule set. Finally, the output layer assigns a specific class label to each rule.

Each layer is presented in more detail below. The model implementation can be found in our GitHub repository at `https://github.com/lucciledierckx/RLNet`.

Input Layer. This layer and the following rule layer are identical to those in DR-Net, while the next ones are different. As it is assumed for the inputs of DR-Net, RL-Net assumes that the features that are fed to the network are binary. Hence, numerical and categorical features require binarisation before being inputted into the network. Contrary to some other methods and as discussed in the rule layer description, there is no need to duplicate the input dataset to express the negation of a feature

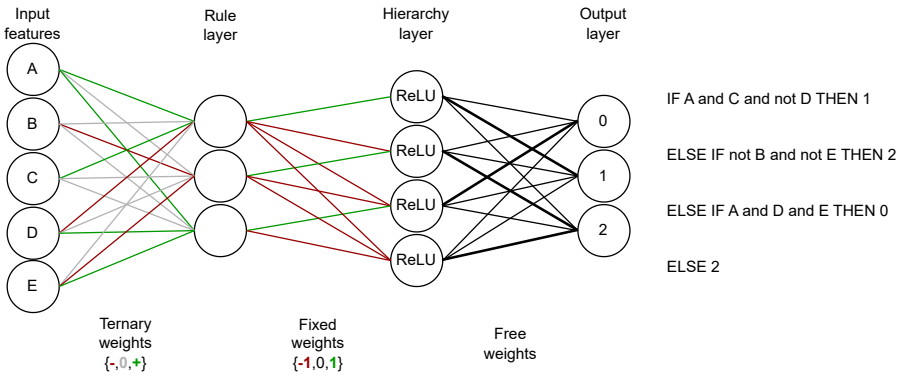


Figure 3.1: Global architecture of `RL-Net` for learning a rule list composed of three rules and the default rule (*else*) for an input dataset consisting of five binary features and three class labels. Green (resp. red, grey) weights represent weights with a positive (resp. negative, zero) value. The edges in bold represent the edges with the highest weight for each node in the hierarchy layer. Connections between neurons that are not represented are non-trainable zero weights.

because the network can express that by itself. The number of nodes n in this layer is equal to the number of binarised features of the dataset.

Rule Layer. This layer encodes rule conditions by mimicking logical conjunctions (ANDs). It is composed of r neurons, where r is a user-defined parameter that specifies the maximum number of rules to be learned. Each weight of this layer can either be positive, negative, or null to represent the fact of using a feature in the rule (+), using the negation of that feature (−), or not considering that feature for the rule (0), respectively. To make this ternary structure trainable with gradient descent, the ternary weights $W_T \in \mathbb{R}^{r \times n}$ of the layer are obtained by an element-wise product of two other matrices:

$$W_T = W_S \circ W_H \tag{3.1}$$

The weights in the matrices W_S and W_H are floating-point numbers. The weights in W_S are classic neural weights, which will converge to either a positive or a negative value during the training, thus deciding if we use the positive form of a given attribute or its negation. The weights in W_H are referred to as hidden weights. They will decide whether an attribute is used in a rule or is ignored. We ensure that these weights converge to a real 1 or a real 0 thanks to the method discussed by Louizos et al. [LWK18] to approximate binary random variables with a Bernoulli

distribution. A sparsity-based regularisation [LWK18] is added to push these hidden weights toward zero and, therefore, obtain shorter rules.

This product computation between the two weight matrices allows to have W_T weights equal to zero when the corresponding W_H weight is zero, regardless of the value of the W_S weight. But in the other cases, the W_T weight will be positive or negative depending on the sign of the W_S weight.

Now that we have described how the model can choose for each input feature whether to use it or not in a rule and whether to negate it or not, we need the neurons of the rule layer to mimic the behaviour of a logical AND, that outputs *true* (1) when all rule conditions are met or *false* (0) otherwise. This is done in two steps, as in DRNet.

In the first step, a neuron j from the rule layer performs the following operation on a binary data point x:

$$\sum_{i=0}^{n} w_{ji} \cdot x_i - \sum_{w_{ji}>0} w_{ji} + 1, \tag{3.2}$$

where $w_{ji} \in W_T$ and $x_i \in \{0, 1\}$ is the value of the binary feature i for the data sample x. In this formulation, the classical weighted sum of inputs is accompanied by a bias that has a dynamic value, which depends on the number of positive weights for the neuron. This dynamic bias is here to ensure that a result of 1 can only be obtained when all positive weights are associated with inputs equal to 1 and all negative weights are associated with inputs equal to 0. Otherwise, the computed value will always be smaller than one, making the output computed by equation (3.2) range within $(-\infty, 1]$.

The second step is the binarisation of the output computed by (3.2), which is given by

$$b(x) = \begin{cases} 1 & \text{if } x = 1 \\ 0 & \text{otherwise} \end{cases}$$

The gradient of this binarisation function is approximated using the straight-through estimator [BLC13] with gradient clipping between 0 and 1, as in `DR-Net`.

Hierarchy Layer. This layer naturally expresses the precedence among the different rules by always ensuring that a rule R_k can be activated only if all previous rules $R_1, R_2, ..., R_{k-1}$ are not active. The structure of this layer that enforce this was inspired by [ANS19] and adapted to a neural network architecture. The weights of this layer are set in the initialisation and are not trainable, as illustrated in Figure 3.1. The activation of a neuron of

this layer is given by a ReLU function. The neuron k in the rule layer represents the rule R_k in the rule list. It is connected to a neuron l of the hierarchy layer by an edge with weight w_{kl}, where

$$w_{kl} = \begin{cases} 1 & \text{when } l = k \\ -1 & \text{when } l < k \\ 0 & \text{otherwise} \end{cases}$$

The last neuron of the hierarchy layer represents the default rule (*else*). It is the only neuron of this layer that uses a bias with a value equal to 1, while the other neurons have no bias. This ensures that the default rule will be applied when none of the previous rules are applicable. Thus, the number of nodes in this layer is equal to $r+1$. The choice of weights in this layer ensures that the values computed by the neurons of this layer before the ReLU activation are always in the range $(-\infty, 1]$. Indeed, the maximum value that can be computed is 1, which happens when the corresponding rule is activated and all previous rules are not. The minimum value is unbounded, as it can happen that many previous rules are activated while the considered rule is not. The ReLU activation then ensures that the output values of this layer are either 0 or 1.

Output Layer. The last layer of our network learns the label associated with every rule condition. The number of neurons in this layer is equal to the number of class labels in the input dataset. The weights are free, and L2-regularisation is added. The activation function is the softmax since we want to perform multi-class classification. As only one rule at a time is active, only the label with the highest activation (i.e., with the highest weight for the active rule) is considered at prediction time, making the learned rules fully interpretable.

3.2.2 Training and Tuning of RL-Net

With its neural network structure, our `RL-Net` can be trained to learn rule lists just as any classical neural architecture using standard techniques of deep learning. Indeed, the Adam optimisation algorithm with the cross-entropy loss function is used. A callback on the validation loss is also applied.

Just as it is observed in standard neural networks, the performance of `RL-Net` for strongly imbalanced datasets can be improved by using a class-balanced version of the loss. Therefore, parameter e_b allows the user to specify that a balanced version of the loss is used in the first e_b epochs. This parameter can be set to zero for datasets for which there is no need for a balanced loss.

Other important parameters of RL-Net are the hidden weights W_H. These weights are initialised using a normal distribution with mean μ and variance σ^2, which directly impacts the probability of using (or not) an attribute in a rule. Thus, the choice of these parameters influences the length (specificity) of the rules in the initialisation, making the search for a good local optimum more or less hard. Other hyperparameters that must be tuned are the learning rate, the weight of the sparsity-regularisation term of the rule layer, as well as the weight of the L2-regularisation of the output layer.

3.2.3 RL-Net for Multi-label Classification

The RL-Net architecture was conceived for multi-class classification, but one of its advantages is that it can easily be transformed into a basic multi-label classifier with two minor changes. The first modification is that the activation of the output layer must be the sigmoid activation instead of the softmax, as the sum of the predictions for all classes does not have to sum to one in a multi-label setting. The second one concerns the loss function for the training of the model, which must be designed for multi-label classification, such as binary cross-entropy loss, focal loss, Huber loss, multi-label margin loss, MSE loss, or L1 loss. For our experiments, we choose the binary cross-entropy loss.

3.3 Experiments

This section experimentally evaluates RL-Net against state-of-the-art rule-based classifiers. Our experiments were designed to answer the following research questions:

(I) How does RL-Net compare against its basis, DR-Net, on binary classification?

(II) How does RL-Net compare against state-of-the-art rule-based classifiers for binary and multi-class classification?

(III) Are the proposed extensions of RL-Net for multi-label classification enough to achieve a satisfactory performance?

3.3.1 Datasets

For the experiments, 7 binary and 6 multi-class datasets were selected. All four binary datasets used in the DR-Net paper were included; these are adult census (adult), magic gamma telescope (magic), fico heloc (heloc), and home price prediction (house). To these, we added internet

advertisements (ads), king-rook vs. king-pawn (chess), and mushroom (mushroom).

For multi-class classification, we selected car evaluation (car), nursery (nursery), contraceptive method choice (contraceptivemc), page blocks classification (pageblocks), pen-based recognition of handwritten digits (pendigits), and sensorless drive diagnosis (drive). We kept the different classes of the multi-class datasets untouched except for nursery, where we merged the classes “very_recom" and “recommend" as they represented, respectively, 2.531% and 0.015% of the class distribution.

The multi-label experiments were made with the yeast (yeast) and scene (scene) datasets.

The datasets all come from the UCI Repository[1] [DG17] except heloc[2], house[3], yeast[4], and scene[5].

3.3.2 Data Preprocessing

Regarding the data used for training, the first step was to remove the data samples for which the percentage of missing values was higher than or equal to 40%. Similarly, the features for which the percentage of missing values was higher than or equal to 40% were removed. The remaining missing values were replaced by the most frequent value in the case of categorical attributes, and by the mean value for numerical features. Lastly, the same feature binarisation as the one chosen for DR-Net was applied. This binarisation applies one-hot encoding to the categorical attributes, and quantile discretisation to the numerical ones, followed by ordinal scaling [GW12]. Keeping the same binarisation allows a better comparison to the contributions and experiments that were presented in the original paper.

The main properties of each dataset before and after the preprocessing are presented in Table 3.1 for the binary and multi-class datasets, and in Table 3.2 for the multi-label ones. The preprocessed datasets are the ones given as input to every single method used in our experiments. This ensures that any observed difference in performance comes from the model itself and not from the data preprocessing variations.

[1] https://archive.ics.uci.edu/

[2] https://community.fico.com/s/explainable-machine-learning-challenge

[3] https://www.openml.org/d/821

[4] https://www.uco.es/kdis/mllresources/#YeastDesc

[5] https://www.uco.es/kdis/mllresources/#SceneDesc

Table 3.1: Characteristics of the different binary and multi-class datasets: The column #Attributes binarised presents the number of attributes after the different preprocessing steps while all the other columns are computed from the unprocessed dataset.

Binary and multi-class datasets	#Rows	#Attributes	#Attributes binarised	Proportion of each class
Adult	48842	14	128	0.24, 0.76
Magic	19020	10	90	0.35, 0.65
House	22784	16	132	0.70, 0.30
Heloc	10459	23	147	0.48, 0.52
Mushroom	8124	22	111	0.52, 0.48
Chess	3196	36	38	0.52, 0.48
Ads	3279	1559	1577	0.86, 0.14
Nursery	12960	8	26	0.33, 0.03, 0.33, 0.31
Car	1728	7	21	0.70, 0.22, 0.04, 0.04
Pageblocks	5473	10	88	0.90, 0.06, 0.01, 0.02, 0.02
Pendigits	10992	16	135	10 classes with equal proportions
Contraceptivemc	1473	9	34	0.43, 0.35, 0.23
Drive	58509	48	432	11 classes with equal proportions

Table 3.2: Characteristics of the different multi-label datasets: The column #Attributes binarised presents the number of attributes after the different preprocessing steps while all the other columns are computed from the unprocessed dataset. The cardinality is the average number of labels per example and the density is the cardinality divided by the number of labels. The distinct column indicates the number of distinct label combinations in the dataset.

Multi-label datasets	#Rows	#Attributes	#Attributes binarised	#Labels	Cardinality	Density	Distinct
Yeast	2417	103	927	14	4.237	0.303	198
Scene	2407	294	2646	6	1.074	0.179	15

3.3.3 Algorithms

We compare RL-Net against three state-of-the-art rule-based classifiers. From the family of tree-based methods, which also enable to learn rules in a training setting, we consider CART [Bre+17], and from classical heuristics-based approaches, we consider RIPPER [Coh95]. These methods were selected due to their widespread use in practice and because they already served as baselines in the evaluation of DR-Net. Finally, we compare against DR-Net itself, which constitutes the basis for the development of RL-Net.

Regarding implementations, we used CART from the *scikit-learn* library and RIPPER from the *Weka* package (*JRip*). For DR-Net, we relied on the

official implementation provided by the authors[6].

3.3.4 Protocol

Some of the datasets (namely, adult, pendigits, yeast, and scene) have a pre-defined test set. In this case, it was used as the test dataset. Otherwise, a test dataset was created by selecting 25% of the data samples in a stratified fashion, so that each class was represented in the test set in proportion to its frequency in the full dataset, and hence preserve the dataset class distribution.

The hyperparameters for RL-Net and DR-Net were tuned using stratified 10-fold cross-validation. The number of rules ranges from 2 to 20 in our experiments, but was fixed to 10 during the hyperparameter tuning for a matter of time. For both algorithms, the following hyperparameters were tuned: the number of epochs, the learning rate, and the weight of the sparsity-regularisation term. The batch size was set to 5% of the dataset training size.

For RL-Net, we also tuned the weight of the L2-regularisation of the output layer, the number of epochs for the balanced loss e_b, and the mean value μ of the normal distribution used for the initialisation of the hidden weights W_H.

The different algorithms are compared using the same number of rules. Therefore, the OR layer of DR-Net does not need to be trained, as it learns to activate or deactivate some of the learned rules. Accordingly, we set the weight of the OR layer regularisation term to zero, and the network training was focused on the Rules layer. For CART, the number of rules is controlled through the user-defined parameter *max_leaf_nodes*. There is no user-defined parameter to control the number of rules in RIPPER's implementation. However, the *minimal weights of instances within a split* parameter influences the number of rules in the classifier. So, we varied this parameter to find the desired number of rules.

For CART, we also tuned the *criterion* function, measuring the split quality, using stratified 10-fold cross-validation.

The remaining hyperparameters of all algorithms were left to their default values. For ease of reproduction, all details about the hyperparameter tuning are available in our GitHub repository[7].

To evaluate the performance of the different algorithms, we used accuracy as the main evaluation metric for both binary and multi-class classification. For multi-label classification, we reported 1 - Hamming loss, where the Hamming loss measures the fraction of labels that are incorrectly

[6] https://github.com/Joeyonng/decision-rules-network

[7] https://github.com/luciledierckx/RLNet

predicted. Consequently, 1 - Hamming loss represents the average fraction of correctly predicted labels.

3.3.5 Results

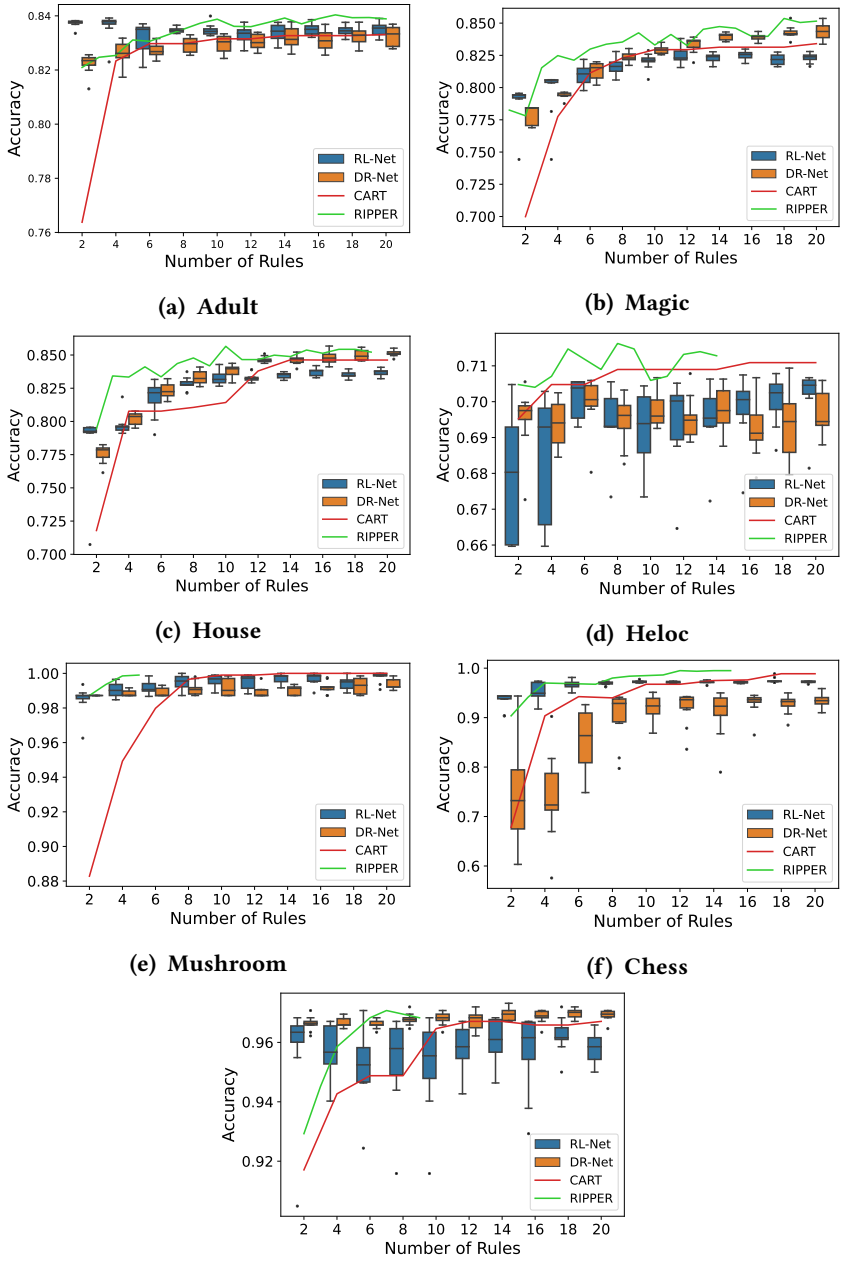
The performance of RL-Net and its competitors is presented in Figure 3.2 for the binary datasets and in Figure 3.3 for the multi-class and multi-label datasets. To ensure interpretability, we limited the number of rules to a range between 2 and 20. Since both RL-Net and DR-Net may converge to poor local optima depending on the random initialisation, each experiment was repeated 10 times for each dataset. The results of these runs are exhibited in a box-plot format.

For **binary classification**, presented in Figures 3.2 (a)-(g), the comparison between RL-Net and DR-Net reveals that it is not possible to say that one of them always performs better than the other. It actually depends on the dataset. RL-Net performs better on datasets such as adult, mushroom, and chess, while for others like ads, it is the other way around. In other cases, such as for magic, house, and heloc, the better-performing method varies with the number of rules considered. Regarding the stability of the methods, RL-Net has a large performance variability on heloc and ads datasets (with a maximal variability of 10%). In contrast, DR-Net has a large performance variability on the chess dataset (with a maximal variability of 35%). Such instability is a known drawback of neural networks with non-standard layers [Hub+16], yet both methods remain competitive when training avoids poor local optima.

Among the classical baselines, RIPPER consistently performs well and usually outperforms both neural approaches, though the margin is generally small. RL-Net outperforms CART in a wide range of cases. CART is particularly affected when the number of rules is small, as the rule length in the tree is constrained by the `max_leaf_nodes` parameter used to limit the number of learned rules, whereas the other algorithms impose no dependency between rule length and rule count.

For **multi-class classification**, presented in Figures 3.3 (a)-(f), RL-Net achieves accuracy comparable to RIPPER on datasets such as nursery, car, pendigits, and drive. RIPPER outperforms RL-Net for the pageblocks dataset, and when very few rules (2 to 4) are allowed with the contraceptivemc dataset. Compared with CART, RL-Net can achieve identical performance even though some runs get stuck in poor local optima, as for the binary case. From these results, we can thus clearly see that RL-Net proves effective in the multi-class classification setting.

Finally, for **multi-label classification**, presented in Figures 3.3 (g)-(h), we compare RL-Net not only against CART, but also against a baseline that,



(a) Adult

(b) Magic

(c) House

(d) Heloc

(e) Mushroom

(f) Chess

(g) Ads

Figure 3.2: Comparison of performances across models: our proposed `RL-Net` (blue), its base model `DR-Net` (orange), `RIPPER` (green), and `CART` (red). Results are shown with varying numbers of rules on binary datasets (a–g).

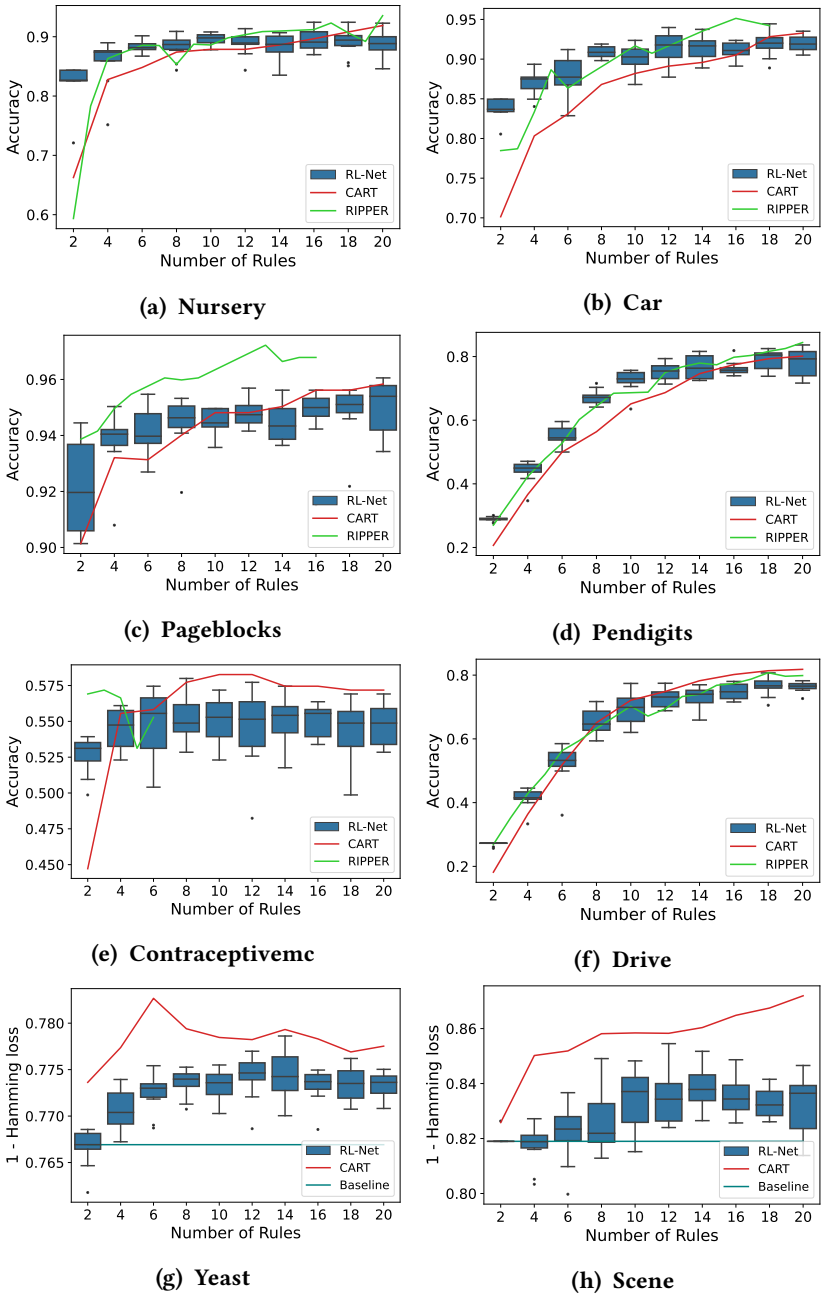


Figure 3.3: Comparison of performances across models: our proposed `RL-Net` (blue), its base model `DR-Net` (orange), `RIPPER` (green), and `CART` (red). Results are shown with varying numbers of rules on multi-class datasets (a–f) and multi-label datasets (g–h).

for each class label, predicts the most frequent value (true or false) in the training set. For both datasets, RL-Net has a lower performance than CART. For the yeast dataset, RL-Net follows CART's performance but scores 1 to 1.5% lower. With only two rules, RL-Net performs similarly to the baseline, but performance improves steadily with more rules, eventually surpassing the baseline by up to 1.2%. On the scene dataset, RL-Net sometimes obtains a lower performance than the baseline when it gets stuck in a poor local minimum, but otherwise clearly outperforms the baseline. CART, however, achieves significantly stronger results than RL-Net for the scene dataset. These findings suggest that RL-Net holds promise for multi-label classification, though its performance is not yet competitive with state-of-the-art methods. Rather, this experiment serves as a proof of concept, pointing to the potential of further exploration in this direction.

3.4 Perspectives

Our proposed method, RL-Net, provides a strong baseline for ordered rule list learning in multi-class classification. Nevertheless, several directions remain open for improving its flexibility, robustness, and integration with other learning paradigms. Some of these directions have also been explored in works that have been published after the submission of our contribution.

Processing Numerical and Categorical Features. A first promising extension concerns the direct handling of numerical and categorical features, thereby eliminating the need for pre-processing. This could be achieved by introducing an additional layer before the current input layer, to learn both the discretisation of numerical features and the enforcement of mutual exclusivity for categorical ones. For instance, numerical discretisation could be implemented via a layer using Rectified Linear Unit (ReLU) activations, where the offset parameters, deciding the binarisation thresholds, are learned during training. This would allow the discretisation thresholds to be optimised jointly with the rest of the model.

Handling categorical features presents additional challenges. Mutual exclusivity must be preserved, meaning that each categorical feature should appear at most once in its positive form within a rule. However, since categorical values may also occur in negated form, the framework should still permit multiple values of the same feature to be included in their negated form in a rule condition. Incorporating these mechanisms would enable RL-Net to process raw numerical and categorical inputs, thereby reducing pre-processing overhead and ensuring that the resulting feature binarisation is both relevant and informative for the learned rules.

Several recent works have explored solutions in this direction. For instance, the Convolutional Rule Neural Network (CR2N) [CK22; Col+23] proposes to use a StackedOR layer to process categorical features directly, avoiding one-hot encoding. This, however, implies that a rule condition cannot contain more than one value of a categorical feature and can not be made of the negation of multiple values of that categorical feature. For the handling of numerical features, both the Relational Rule Network (R2N) [Kus+22] and NeuRules [XWV24] methods propose to learn the discretisation of numerical features and their combination into conjunctive rules directly from raw features, thus bypassing pre-processing or additional constraints.

Dependency on Initialisation. Another direction is to improve the robustness of `RL-Net` to its random initialisation. Indeed, `RL-Net` performance varies a lot depending on the initialisation of the hidden weights W_H, a sensitivity likely amplified by the ordered rule structure. Because the first learned rule strongly influences subsequent ones, poor initialisation can lead to suboptimal local optima. A possible solution would be to relax the precedence at the beginning of training, progressively enforcing it later, and thus allowing the model to explore a broader range of rule combinations initially before gradually being pushed in a rule list setting. This idea of progressive imposition of the ordered structure was explored in the NeuRules [XWV24] method, which initially relaxes the ordered structure of the rule list and progressively enforces it during the training with temperature annealing.

Integration with Broader Paradigms. Finally, learning interpretable rules with `RL-Net` could be of interest in other learning paradigms. For instance, combining neural rule learning with transfer learning, semi-supervised learning, or active learning could be explored to discover if these paradigms can efficiently complement `RL-Net` and improve both performance and scalability, especially in settings where labelled data are scarce or costly to obtain.

Other Approaches. Beyond rule lists, several recent neuro-symbolic methods have explored interpretable rule-based multi-class classification. `DiffNaps` [WFV24] extends `BinaPs` [FV21], presented earlier and based on a binary autoencoder for pattern set mining, by adding a classification layer on top of the binary autoencoder. `DiffVersify` [Cha+24] in turns builds on `DiffNaps`, introducing a novel regularisation mechanism that promotes pattern diversity and coverage. Together, these works illustrate

the growing diversity of neural frameworks for interpretable, rule-based learning.

3.5 Conclusion

This chapter has addressed one approach to reconcile the interpretability of symbolic models with the scalability and optimisation power of neural networks. In particular, we have investigated the use of neural architectures to learn rule-based classifiers, aiming to combine the transparency of symbolic rules with the flexibility of differentiable learning.

To this end, we introduced `RL-Net`, an interpretable neural model that extends earlier work on differentiable rule learning by supporting rule lists and multi-class classification. We further demonstrated that `RL-Net` can be adapted to multi-label tasks, highlighting its versatility. Our experiments across a range of datasets show that `RL-Net` is competitive with established state-of-the-art rule-based methods. While it does not always yield the very best predictive accuracy, its performance is consistently close to the strongest baselines, and it succeeds in preserving interpretability while using neural models, the latter being an essential property in many real-world domains where transparency and accountability are as critical as accuracy.

These results suggest that gradient-based optimisation can indeed be used to discover symbolic rule structures, providing an alternative to traditional combinatorial search strategies. At the same time, challenges remain. `RL-Net` is still sensitive to weight initialisation, and its performance on multi-label classification does not yet match the best available methods. Nonetheless, its neural architecture makes it well-suited to further integration with recent advances in neural learning, including transfer learning, semi-supervised learning, and active learning, which may help close this gap. Moreover, this differentiable formulation opens the door to the integration with other neural architectures, similar to the ideas of concept bottleneck models, enabling rule learning over higher-dimensional data, a direction that will be further explored in the general conclusion of this thesis. In future work, the hierarchical layer of the model could also be adapted to encode other forms of structural constraints between learned rules, such as superclass and subclass relations or domain-specific hierarchies.

Overall, this work contributes to the broader effort of neuro-symbolic learning by showing how neural networks can serve for learning interpretable symbolic classifiers.

Part II

Learning from Complex Symbolic Constraints

Learning from Symbolic Constraints

4

As discussed in Chapter 2, deep learning excels at low-level perception tasks but struggles with high-level reasoning and decision-making. For instance, while neural models can reliably detect traffic lights or pedestrians in images, deciding when a vehicle should brake involves higher-level reasoning that is difficult to capture end-to-end. Training a neural model to perform this task directly raises concerns regarding interpretability and generalisation to unseen situations.

In many domains, relevant high-level knowledge already exists in symbolic form. For instance, it is known that a vehicle must brake when a red light or a pedestrian near a crossing is detected. Rather than trying to learn such rules from data, a more interpretable approach is to let a neural network handle perceptual recognition, while letting symbolic knowledge manage the high-level decision-making. Recent work shows that symbolic knowledge can also guide the learning directly, rather than serving only as a post-processing layer [Man+18; Giu+25]. Incorporating symbolic rules as soft or hard constraints can reduce the reliance on labelled data [Man+21a] and improve model generalisation through domain-consistent supervision.

This combination of neural perception and symbolic reasoning is an important part of neuro-symbolic artificial intelligence. The setting studied throughout this part of this thesis corresponds to the *Neuro[Symbolic]* subclass in Kautz's taxonomy, introduced in Section 2.2. In this subclass, symbolic constraints are not embedded directly within the architecture but guide the learning process as an external, interpretable source of supervision.

The core idea explored in this second part of the thesis, as described in Chapter 1 and illustrated in Figure 4.1, is thus to train neural models under symbolic constraints. More specifically, a neural network produces probabilistic predictions over some defined perceptual features (e.g., the presence of a red light). These outputs are then interpreted as probabilities over logical variables. The symbolic knowledge, expressed as logical formulas over these variables, defines valid configurations (e.g., when braking is required). The degree to which the neural predictions satisfy these constraints is computed and is used as feedback to guide parameter

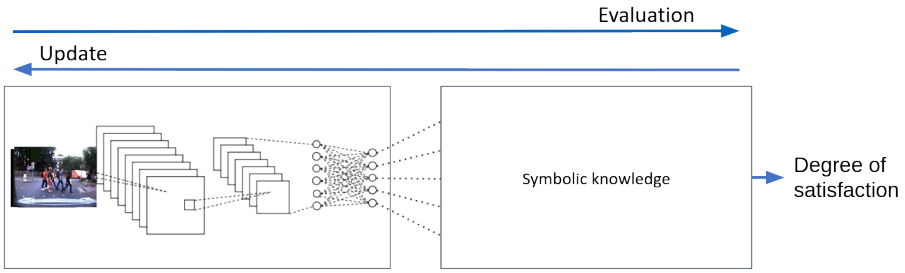


Figure 4.1: General neuro-symbolic approach. Neural predictions serve as probabilistic inputs to a symbolic reasoning module that evaluates constraint satisfaction and provides feedback to improve consistency.

updates, helping to align the model with logical consistency.

More technically, this work focuses on symbolic knowledge that can be represented in propositional logic, specifically in Conjunctive Normal Form (CNF). This representation is common to describe various types of knowledge, including Bayesian networks, probabilistic graphs, and probabilistic logic programs. These different types of knowledge can all be used to assess whether the outputs of a neural model are consistent with some expressed domain knowledge. This consistency is evaluated by computing the satisfaction degree of the formula, given the neural model outputs interpreted as probabilities over the logical variables. To compute this satisfaction degree, a certain number of approaches can be used, but we focus on an important class of such methods, which are based on *Weighted Model Counting* (WMC) and its variants such as *Projected Weighted Model Counting* (PWMC). To train models under these constraints, it is necessary to compute the gradients of the WMC with respect to the neural network parameters. As these computations are performed repeatedly during training, efficiency becomes crucial. *Knowledge Compilation (KC)* addresses this issue by transforming WMC problems into *Arithmetic Circuits (ACs)*, which allow efficient repeated evaluations for both inference and differentiation. These components and their interactions within the overall framework are presented below.

This chapter lays the conceptual and technical foundation for the neuro-symbolic learning framework developed in the second part of this thesis. Additional technical details specific to our contributions will be provided in subsequent chapters. Section 4.1 introduces different examples of symbolic knowledge that can be integrated into the framework. Section 4.2 formalises the learning setting and details the roles of WMC and PWMC. Section 4.3 reviews the core algorithms commonly used for each stage of the framework, including knowledge compilation and gradient computation. Section 4.4

summarises the full neuro-symbolic pipeline and discusses an important limitation of the general setting. Finally, Section 4.5 surveys related work, while Section 4.6 presents `Schlandals`, the solver used and extended in this work to perform (P)WMC, knowledge compilation, and parameter learning. The chapter concludes with a summary of the main aspects of the neuro-symbolic learning setting that will be used in this second part of the thesis.

4.1 Symbolic Knowledge

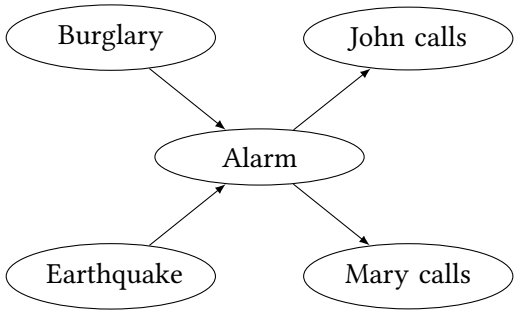
Symbolic knowledge encodes structured relationships between facts and can take various forms. In the neuro-symbolic learning framework considered in this work, where the degree of knowledge satisfaction is evaluated through WMC on formulas in conjunctive normal form (CNF), which is, as discussed earlier, an important language for expressing domain knowledge [CD08; Fie+12; Fie+15].

A wide variety of structured knowledge sources can be represented in propositional logic, including Bayesian networks [Fie+15; Bar+16], reliability networks [Vla+16], probabilistic logic programs [Fie+15; Vla+16], and other domain-specific constraints directly encoded in CNF.

4.1.1 Bayesian Networks

Bayesian Networks are probabilistic graphical models that encode *conditional dependencies* between random variables. They are typically structured as directed acyclic graphs (DAGs), where each node represents a variable (e.g., an event), and edges indicate direct probabilistic influences between them. Each node is associated with a conditional probability table (CPT) specifying the likelihood of a node's value given those of its parents nodes.

A classical example is the alarm scenario, illustrated in Figure 4.2. This model captures how events like burglaries or earthquakes influence the probability of a house alarm being triggered, and how this, in turn, affects whether the homeowners call security. The nodes of the graph represent the variables (e.g., the alarm rings, there is an earthquake,...), and the edges denote the dependencies between them. The probabilities of the variables are defined given their parents' values (e.g., the probability that the alarm rings depends on the presence of a thief and/or an earthquake) and are encoded in conditional probability tables (CPTs). Let us notice that the variables may be binary (e.g., there is a burglary or not) or *multi-valued* (e.g., there is a strong, mild, or no earthquake).



Burglary	
yes	no
0.01	0.99

	John calls	
Alarm	yes	no
no	0.01	0.99
yes	0.8	0.2

		Alarm	
Burg.	Earth.	yes	no
no	no	0.01	0.99
no	mild	0.3	0.7
no	strong	0.6	0.4
yes	no	0.95	0.05
yes	mild	0.97	0.03
yes	strong	0.99	0.01

Earthquake		
strong	mild	no
0.01	0.09	0.9

	Mary calls	
Alarm	yes	no
no	0.01	0.99
yes	0.8	0.2

Figure 4.2: Bayesian Network for the alarm problem. Each node is a variable representing an event, and the directed edges represent the dependencies between them. The conditional probability tables (CPTs) quantify the influence of parent nodes on each variable.

Example 4.1.1. (Queries in a Bayesian Network)
A typical query in Bayesian Networks is to compute the probability of a specific variable taking a given value. For instance, using the network in Figure 4.2, we might query the probability that the alarm rings, $P[Alarm = yes]$. This probability is obtained by summing the probabilities of all the scenarios in which the alarm is true and thus

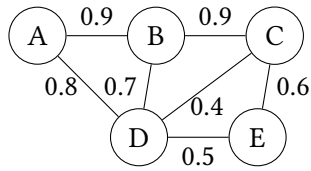


Figure 4.3: Reliability network for an electrical grid.

will be ringing:

$$\begin{aligned} P(Alarm = yes) = {} & P(Alarm = yes, Burglary = no, Earthquake = no) \\ & + P(Alarm = yes, Burglary = no, Earthquake = mild) \\ & + P(Alarm = yes, Burglary = no, Earthquake = strong) \\ & + P(Alarm = yes, Burglary = yes, Earthquake = no) \\ & + P(Alarm = yes, Burglary = yes, Earthquake = mild) \\ & + P(Alarm = yes, Burglary = yes, Earthquake = strong) \end{aligned}$$

The probability of each of these models is computed using the conditional probability tables (CPTs) associated with the variables. For instance, $P[Alarm = yes, Burglary = yes, Earthquake = no] = P(Alarm = yes | Burglary = yes, Earthquake = no) * P(Burglary = yes) * P(Earthquake = no)$.

4.1.2 Reliability Networks

Probabilistic graphs are structured graphical models that represent a system (e.g., an electrical grid, a water network) on which one can assess reliability. We will refer to these types of graphs as *reliability networks* in this manuscript. A probabilistic graph presents itself as a set of nodes and edges, as illustrated in Figure 4.3. The nodes represent the components of the system, while the edges represent the connections between them. Each edge has an associated probability that represents the likelihood of the connection being operational. Reliability Networks can be used to compute the probability for two nodes to be connected and to identify critical components in the system.

Example 4.1.2. (Queries in Reliability Networks)
A typical query in a Reliability Networks is the probability for a node to be able to reach a second one. For instance, using the network of Figure 4.3, we can query the probability that node D is reachable from node A, denoted $P[R_{A \to D}]$. Computing the probability of this

query is equivalent to computing $1 - P[\bar{R}_{A\to D}]$, with $P[\bar{R}_{A\to D}]$ the probability that no such path exists between A and D, which is easier to compute for some probability inference methods. The probability of the query can thus be computed by adding the probabilities of all the scenarios (models) in which node D is disconnected from node A.

The cases where there is no path between A and D can be enumerated as:

- At least links AD and AB are both down.
- At least links AD, BD, and BC are all down.
- At least links AD, BD, CD, and CE are all down.
- At least links AD, BD, CD, and DE are all down.

The disconnection probability is computed by summing the weights of such scenarios:

$$\begin{aligned} P[R_{A\to D}] &= 1 - P[\bar{R}_{A\to D}] \\ &= 1 - (\neg P(AD)(\neg P(AB) + P(AB)\neg P(BD)\cdot \\ &\quad (\neg P(BC) + P(BC)\neg P(CD)(\neg P(CE) + P(CE)\neg P(DE))))) \end{aligned}$$

With $P(XY)$ being the probability that the link between node X and Y is functional.

4.1.3 Probabilistic Logic Programs

Probabilistic logic programs, such as ProbLog [DKT07], extend logic programming by annotating facts with probabilities. For example, a program can define the result Z of the addition of two digits X and Y using probabilistic facts and rules.

Example 4.1.3. (ProbLog Program for the Digits Addition)

```
0.15::digit(val0,0);0.05::digit(val0,1);0.3::digit(val0,2);
0.04::digit(val0,3);0.02::digit(val0,4);0.1::digit(val0,5);
0.03::digit(val0,6);0.25::digit(val0,7);0.01::digit(val0,8);
0.05::digit(val0,9).

0.13::digit(val1,0);0.06::digit(val1,1);0.1::digit(val1,2);
0.21::digit(val1,3);0.2::digit(val1,4);0.04::digit(val1,5);
0.1::digit(val1,6);0.06::digit(val1,7);0.01::digit(val1,8);
0.09::digit(val1,9).

addition(X,Y,Z) :- digit(X,X2), digit(Y,Y2), Z is X2+Y2.
```

The first part of this example contains probabilistically annotated facts, here describing the chance for the two digits (`val0` and `val1`) to take values 0 to 9. The facts presented here have an additional specificity that they encode what is called an *annotated disjunction* in ProbLog. Annotated disjunctions are used to express *multi-valued* facts, for which exactly one value should be considered at all times. This is similar to the multi-valued facts observed for Bayesian Networks. The second part contains the rule defining how facts combine to produce, in this case, the value resulting from the addition.

For such a ProbLog program, a query can be to infer the probability of obtaining a specific result from the addition.

Example 4.1.4. (Query in ProbLog)
In the previous example, we can query the probability of obtaining 2 as the result of the addition of the two digits. The query is expressed in ProbLog as:

```
query(addition(val0,val1,2)).
```

The query is then evaluated by computing the probability of the scenarios (called *models*) that satisfy the query. The total probability of the sum being 2 is computed as: $P[sum = 2] = P(0, 2) + P(1, 1) + P(2, 0)$, where $P(i, j)$ is the probability of the two digits taking the values i and j respectively. The probability of each of these models is computed using the probabilistic facts defined in the program. For instance, $P(0, 2) = P(\texttt{val0} = 0) \cdot P(\texttt{val1} = 2)$.

4.1.4 Other Forms of Knowledge

Some forms of domain-specific symbolic knowledge are given directly as propositional constraints, typically in CNF. A notable example is the ROad event Awareness Dataset with logical Requirements (ROAD-R) [Giu+23], which focuses on road traffic prediction. This dataset is composed of a set of images taken from a vehicle in traffic. The images each come with bounding boxes identifying the road events present in the scene. Each bounding box must be labelled, in a multi-label fashion, from a set of 41 labels. The dataset also comes with logical requirements, representing common knowledge about the road traffic. For instance, a traffic light cannot be red and green at the same time, or if an agent pushes an object, then it is a pedestrian. The knowledge is directly expressed in CNF.

Example 4.1.5. (ROAD-R Constraints in CNF)
The ROAD-R dataset contains a set of logical constraints that can

be expressed in CNF. For instance, the constraint that a traffic light cannot be red and green at the same time, and the constraint that if an agent pushes an object, then it is a pedestrian, can be expressed together as the formula:

$$F = (Pedestrian \vee \neg PushObj) \wedge (\neg Red \vee \neg Green)$$

The formula in CNF is made of facts (e.g., $Pedestrian$, $PushObj$, Red, $Green$) which can be either true or false and will cause the global formula to be true or false. However, such a formula can also be used in a probabilistic setting, where each fact f is associated with a probability $P(f)$ to be true.

Knowing the probability for each fact to be true, we can compute $P^*[F]$, the probability for the formula to be true. This probability represents the degree to which the labels associated with a bounding box are consistent with the road traffic knowledge. This is done by enumerating the different assignments that make the formula true, and summing their respective probability of happening. The formula is true in the following cases:

- $Pedestrian$ is true, $PushObj$ is false, Red is false and $Green$ is false
- $Pedestrian$ is true, $PushObj$ is false, Red is false and $Green$ is true
- $Pedestrian$ is true, $PushObj$ is false, Red is true and $Green$ is false
- $Pedestrian$ is true, $PushObj$ is true, Red is false and $Green$ is false
- $Pedestrian$ is true, $PushObj$ is true, Red is false and $Green$ is true
- $Pedestrian$ is true, $PushObj$ is true, Red is true and $Green$ is false
- $Pedestrian$ is false, $PushObj$ is false, Red is false and $Green$ is false
- $Pedestrian$ is false, $PushObj$ is false, Red is false and $Green$ is true

- *Pedestrian* is false, *PushObj* is false, *Red* is true and *Green* is false

The probability of each of these scenarios is computed by multiplying the probabilities associated with each individual fact. For instance, $P(Pedestrian = true, PushObj = false, Red = false, Green = false) = P(Pedestrian = true) \cdot P(PushObj = false) \cdot P(Red = false) \cdot P(Green = false)$.

4.2 Neuro-Symbolic Learning Specification

As described in the previous section, various types of symbolic knowledge can be integrated into our considered neuro-symbolic learning framework. The common features shared across these different knowledge types are that they can be expressed in *propositional logic*, specifically in CNF, that they can be evaluated to determine the *probability that a query is satisfied*, and that their contribution to the learning lies in the *logical structure* connecting different probabilistic facts. In contrast, the probabilities associated with the logical facts are parameters that will be learned by the neuro-symbolic task, via a sub-symbolic component, for each sample in the dataset.

With that in mind, we now define more formally the neuro-symbolic learning setting considered in this work, which is also widely studied in the literature [Man+18; MMD21; Man+21a; Man+21b; MDD24]. We also describe the various concepts that are employed during the learning process.

4.2.1 General Problem Setting

The general problem setting is illustrated in Figure 4.1, with its main components described below. The goal of the learning task is to learn a sub-symbolic model whose output probabilities are consistent with the symbolic knowledge.

Dataset. We consider a training dataset $\mathcal{D} = \{d_1, ..., d_m\}$ composed of high-level data samples (e.g., images, text, etc.). Each sample d_i is associated with a logical *query* $F_i \in C = \{F_1, ..., F_m\}$. All queries are defined over a common set of propositional variables $\mathcal{X} = \{x_1, ..., x_n\}$, representing the symbolic facts that constitute the background knowledge.

Queries. A *query* is a propositional logical formula F, in *Conjunctive Normal Form (CNF)*, defined over the set $\mathcal{X}$ of variables. A literal over a variable $x \in \mathcal{X}$ is an assignment of x to one of its possible values v

(two for binary variables, more for multi-valued ones). In the probabilistic setting, F is weighted on its literals: each literal is assigned a weight $w_{x=v} \in [0, 1]$, indicating the probability that the literal is true. The set of all literal weights is denoted by $\mathcal{W}$. All queries in C are constructed using (a subset of) the same n probabilistic variables in $\mathcal{X}$.

To illustrate the various concepts introduced in this second part of the thesis, we consider two running examples. The first one, in Example 4.2.1, is a simple CNF formula that will be helpful to study the properties of the gradients that can be derived from it. The second one, in Example 4.2.2, is a longer CNF formula that will be used to illustrate the different concepts of knowledge compilation.

Example 4.2.1. (Running Example 1: Simple CNF Formula)
The CNF formula used throughout this part of the thesis for gradient analysis is as follows.

$$F = (\neg x_1 \vee x_3) \wedge (x_2 \vee x_3)$$

Example 4.2.2. (Running Example 2: Longer CNF Formula)
The CNF formula used throughout this part of the thesis to illustrate the different concepts of knowledge compilation is as follows.

$$F = C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5 \wedge C_6 \wedge C_7 \wedge C_8$$

$$C_1 = (\neg x_0 \vee \neg x_1 \vee \neg x_3 \vee x_8) \qquad C_5 = (x_0 \vee \neg x_1 \vee \neg x_4 \vee x_8)$$

$$C_2 = (\neg x_0 \vee \neg x_1 \vee x_3 \vee x_9) \qquad C_6 = (x_0 \vee \neg x_1 \vee x_4 \vee x_9)$$

$$C_3 = (\neg x_0 \vee x_1 \vee \neg x_2 \vee x_8) \qquad C_7 = (x_0 \vee x_1 \vee \neg x_5 \vee x_4)$$

$$C_4 = (\neg x_0 \vee x_1 \vee x_2 \vee x_9) \qquad C_8 = (x_0 \vee x_1 \vee x_5 \vee x_9)$$

Probabilistic Inference. Given a query F_i and a set of literal weights $\mathcal{W}$, we denote the probability of the query being satisfied as $P^*_{\mathcal{W}}[F_i]$. Computing this probability, by summing the weights of all models that satisfy F_i, is referred to as *probabilistic inference*, and can be performed using *Weighted Model Counting (WMC)* or *Projected WMC (PWMC)*.

Learning Task. Each data sample $d_i \in \mathcal{D}$ is associated with a query F_i and a ground-truth label $y_i \in [0, 1]$ and the set $\mathcal{Y} = \{y_1, ..., y_m\}$, representing the expected probability that F_i is satisfied. We can represent the dataset as a set of triples $\{(d_1, F_1, y_1), ..., (d_m, F_m, y_m)\}$ associating an input data with its related query and expected output of the query.

The learning task consists of training a neural model, M_θ, with parameters θ, which maps each input sample d_i to a vector of literal probabilities

$\mathcal{W}$:

$$M_\theta : \mathbb{R}^{|d_i|} \rightarrow [0,1]^l,$$

where l is the total number of literals used in the background knowledge. The parameters θ are optimised such that the predicted probabilities $P^*_{M_\theta(d_i)}[F_i]$ align with the expected probability y_i and, therefore, pushing the neural model to be more consistent with the symbolic knowledge.

Learning Pipeline. The sub-symbolic model and the logical knowledge are integrated through a learning pipeline, illustrated in Figure 4.1, and described as follows:

- The neural model M_θ processes the input data $\mathcal{D}$ and predicts a set of probabilities $\mathcal{W}$ for each sample.
- For each query F_i, the success probability $P^*_{\mathcal{W}}[F_i]$ is inferred using the predicted probabilities $\mathcal{W}$ for the literals.
- The predicted probability is compared to the ground-truth label $y_i \in \mathcal{Y}$ and is given as feedback for the neural model update.
- The model parameters θ are updated to reduce the discrepancy between the expected and predicted value.
- The process is repeated until convergence of parameters θ or until a predefined stopping criterion is met.

Developed Concepts. This learning framework builds upon several core components: the probabilistic inference on the logical knowledge, formulated as the (projected) weighted model counting ((P)WMC) problem, and the definition of a learning feedback, often referred to as a loss function to compare predicted and expected probabilities. These aspects are discussed in detail in the following subsections. The different learning pipeline elements are then integrated with the symbolic knowledge presented in Section 4.1 to provide a clear view of the entire neuro-symbolic picture considered here.

4.2.2 (Projected) Weighted Model Counting

Previous sections have described how the probability of a query being satisfied needs to be evaluated within the neuro-symbolic setting. This evaluation is based on both the symbolic knowledge and the output of the neural model, which provides probabilistic weights. The inferred probability is then compared against the expected one, allowing for the adjustment of

the sub-symbolic model's parameters. One common way to compute the probability of a query is to enumerate all possible scenarios (also called models) that satisfy it. This is formalised through the concept of *weighted model counting*, which sums the weights of all satisfying models [CD08]. A variant of WMC, called *projected WMC* (PWMC), is also useful for certain tasks, as it allows focusing on a subset of the variables when computing the probability, making the computation more efficient. In Section 4.2.4, we will illustrate how WMC is used for probabilistic inference in Bayesian Networks and ProbLog, while PWMC is employed for reliability networks.

We now formally define the concepts of interpretation, model counting (MC), weighted model counting (WMC), and projected weighted model counting (PWMC).

Interpretation. An *interpretation* $I = \{l_1, \ldots, l_k\}$ for a formula F is an assignment of truth values to all variables (i.e., a choice of one literal per variable). We write $F[I]$ to denote the evaluation of F under interpretation I, and let $\mathcal{I}_F$ denote the set of all interpretations for formula F. If $F[I] = \top$ (i.e., F evaluates to true under I), then I is called a model of F.

Model Counting. Given $\mathcal{I}_F$, the set of all interpretations for formula F. The model counting problem, denoted as $MC(F)$, counts the *number of models* that satisfy F as follows.

$$\mathrm{MC}(F) = |\{I \in \mathcal{I}_F \mid F[I] = \top\}|$$

Weighted Model Counting. In the *Weighted Model Counting* (WMC) setting, each literal l in the formula F is associated with a weight $w_l \in [0, 1]$, representing the probability that the literal is true. The weight of an interpretation I is given by the product of the weights of the literals it contains: $\prod_{l \in I} w_l$.

The WMC of formula F under weight assignment $\mathcal{W}$ is defined as the sum of the weights of all models of F:

$$\mathrm{WMC}_{\mathcal{W}}(F) = \sum_{I \mid F[I]=\top} \prod_{l \in I} w_l$$

Given the weights $\mathcal{W}$, we denote the inferred probability of F as $P^*_{\mathcal{W}}[F] = \mathrm{WMC}_{\mathcal{W}}(F)$.

> **Example 4.2.3.** (Weighted Model Counting)
> Let F be the CNF formula of our simple running example (Example 4.2.1)

over three binary variables $\mathcal{X} = \{x_1, x_2, x_3\}$, defined as follow.

$$F = (\neg x_1 \vee x_3) \wedge (x_2 \vee x_3)$$

Suppose the associated weights for the variables to be true are $w_{x_1} = 0.99, w_{x_2} = 0.5, w_{x_3} = 0.65$. An example interpretation is $I = \{x_1, \neg x_2, x_3\}$, which satisfies F and has a weight of $0.99 \cdot (1-0.5) \cdot 0.65 = 0.32175$. Summing the weights over all satisfying interpretations yields $\mathrm{WMC}_{\mathcal{W}}(F) = 0.65175$.

Projected Weighted Model Counting. The modelling of some problems sometimes requires the introduction of additional variables that are auxiliary to the main task. These variables are not the focus of the learning task and are not relevant to the count. To handle such cases, the concept of *projected weighted model counting* (PWMC) is introduced [Azi+15].

In Projected Weighted Model Counting, the variables are partitioned into two sets: the *priority variables*, $\mathcal{X}_P \subseteq \mathcal{X}$, and the *non-priority variables*, $\mathcal{X}_{NP} = \mathcal{X} \setminus \mathcal{X}_P$. The priority variables are the focus of the learning task and are probabilistically weighted. The non-priority variables are auxiliary and unweighted, used to express the problem more compactly.

Let $\pi_{\mathcal{X}_P}(I)$, denote a *projected interpretation*, that is an interpretation restricted to the variables in $\mathcal{X}_P$. Similarly, let $\pi_{\mathcal{X}_{NP}}(I)$ be the interpretation restricted to the variables in $\mathcal{X}_{NP}$.

Let $\pi_{\mathcal{X}_P}(F)$ be the set of all projected interpretations of F over the variables in $\mathcal{X}_P$. We define the set of projected satisfying interpretations over $\mathcal{X}_P$ as those for which some assignment to $\mathcal{X}_{NP}$ satisfies F, which can be counted as:

$$PMC(F) = |\{\pi_{\mathcal{X}_P}(I) \in \pi_{\mathcal{X}_P}(F) \mid \exists \pi_{\mathcal{X}_{NP}}(I) : F[\pi_{\mathcal{X}_P}(I) \cup \pi_{\mathcal{X}_{NP}}(I)] = \top\}|.$$

In other words, $PMC(F)$ counts the number of distinct projected interpretations of F over the variables in $\mathcal{X}_P$, that can be extended to full models of F.

Note that

$$PMC(F) \leq MC(F),$$

since different models may collapse to the same projection when only priority variables are considered.

The *projected weighted model counting* (PWMC), computes the sum of the weights of all such projected models, using weights only on the priority variables. Hence, PWMC calculates:

$$PWMC_{\mathcal{W}}(F) = \sum_{I \in \pi_{\mathcal{X}_P}(F) \mid \exists \pi_{\mathcal{X}_{NP}}(I) : F[\pi_{\mathcal{X}_P}(I) \cup \pi_{\mathcal{X}_{NP}}(I)] = \top} \prod_{l \in I \mid l \in \mathcal{X}_P} w_l$$

This yields the total probability mass over the priority variable assignments that can be extended to satisfying full interpretations.

Example 4.2.4. (Projected Weighted Model Counting)
Let us use the simple CNF formula from Example 4.2.1, $F = (\neg x_1 \vee x_3) \wedge (x_2 \vee x_3)$, and consider the following partition of the variables: $\mathcal{X}_P = \{x_1, x_3\}$ and $\mathcal{X}_{NP} = \{x_2\}$.

To find the number of projected interpretations, we consider the interpretations over $\mathcal{X}_P$ for which there exists an assignment to $\mathcal{X}_{NP}$ that satisfies F. The priority variables can take the following values:

- $I_1 = \{x_1 = \top, x_3 = \top\}$, in which case the formula is satisfied for any value of x_2. The probability of this projected interpretation is $P^*_{\mathcal{W}}[I_1] = w_{x_1} w_{x_3}$.
- $I_2 = \{x_1 = \top, x_3 = \bot\}$, in which case the formula is never satisfied, regardless of the value of x_2. This projected interpretation is thus not counted.
- $I_3 = \{x_1 = \bot, x_3 = \top\}$, in which case the formula is satisfied for any value of x_2. The probability of this projected interpretation is $P^*_{\mathcal{W}}[I_3] = (1 - w_{x_1}) w_{x_3}$.
- $I_4 = \{x_1 = \bot, x_3 = \bot\}$, in which case the formula is satisfied for $x_2 = \top$ only. The probability of this projected interpretation is $P^*_{\mathcal{W}}[I_4] = (1 - w_{x_1})(1 - w_{x_3})$. Indeed, the non-priority variables are auxiliary and are thus not associated with any weight.

The total projected weighted model count is then computed by summing the probabilities of the projected interpretations for which F can be satisfied.

4.2.3 Learning Feedback

In standard machine learning, sub-symbolic models are trained by minimising a defined *loss function*, which quantifies the discrepancy between predicted and expected values. The choice of loss function depends on the nature of the task and the desired properties of the optimisation process.

Two commonly used loss functions are the Mean Squared Error (MSE) and the Mean Absolute Error (MAE). The MSE loss function is defined as the average of the squared differences between the predicted and expected values, while the MAE loss function is defined as the average of the absolute differences. The two loss functions are defined as follows:

$$\mathcal{L}_{MSE}(\hat{y}, y) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2, \quad \mathcal{L}_{MAE}(\hat{y}, y) = \frac{1}{n}\sum_{i=1}^{n}|\hat{y}_i - y_i|,$$

where $\hat{y}_i$ is the predicted value, y_i is the expected value, and n is the number of samples. The MSE penalises larger deviations more strongly than smaller ones due to the squaring operation, while the MAE applies a uniform penalty across all errors.

4.2.4 Combined Pipeline

Building upon the high-level learning process outlined in Section 4.2, we can now detail the full workflow shown in Figure 4.4, which provides a more technical view of the neuro-symbolic learning pipeline:

- The neural model M_θ processes the input data $\mathcal{D}$ and predicts a set of probabilities $\mathcal{W}$ for each sample.
- For each query F_i, the success probability $P^*_{\mathcal{W}}[F_i]$ is inferred using the predicted probabilities $\mathcal{W}$ for the literals. The probability for F to be satisfied, $P^*_{\mathcal{W}}[F_i]$, is in practice computed with the (projected) weighted model counting ((P)WMC).
- The predicted probability is compared to the ground-truth label $y_i \in \mathcal{Y}$ using a loss function $\mathcal{L}$, which penalises discrepancies between the predicted and expected values. The loss serves as feedback for the neural model update.
- The model parameters θ are updated to minimise the loss value.
- The process is repeated until convergence of parameters θ or until a predefined stopping criterion is met.

Having established how neural predictions, symbolic knowledge, WMC, and the loss function interact within the learning loop, it remains to understand how each of the symbolic knowledge types from Section 4.1 can be integrated in the neuro-symbolic learning framework.

Bayesian Networks. The first step to enable the integration of Bayesian networks in a neuro-symbolic learning framework is to encode the graphical model of the Bayesian network into a CNF formula. The knowledge represented in the graphical model of the Bayesian network can be encoded in a CNF using different encoding methods. The most commonly used encodings include ENC1 [Dar02], ENC3 [CD05], ENC4 [CD06], and ENC4linp [Bar+16].

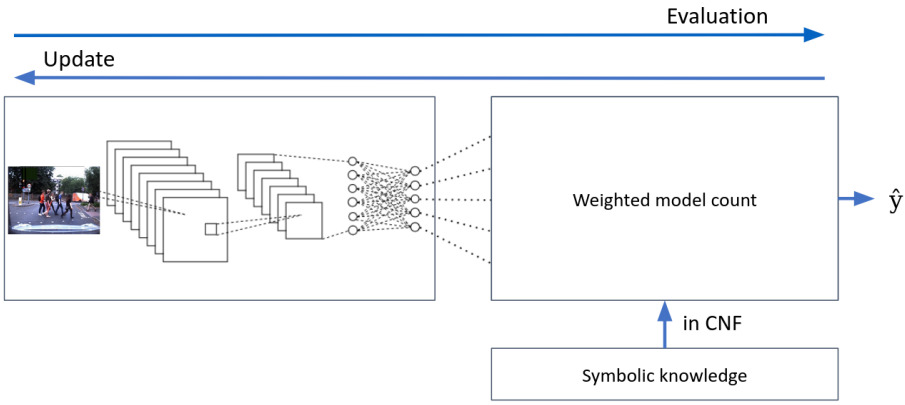


Figure 4.4: Combined neuro-symbolic learning pipeline. The neural network outputs probabilities for the literals of a CNF-encoded probabilistic logic model. The probability that the knowledge is satisfied, $\hat{y}$, is computed through weighted model counting (WMC) and compared to the target value y using a loss function $\mathcal{L}$. The resulting feedback is used to update the neural network parameters to improve consistency with the symbolic knowledge.

The ENC1 [Dar02] encoding is the most straightforward. For each value x_i of a variable x, an indicator variable v^{x_i} is introduced. These indicator variables are constrained to be mutually exclusive, using clauses of the form:

$$v^{x_1} \vee \ldots \vee v^{x_m}$$

$$\neg v^{x_i} \vee \neg v^{x_j}, \ \forall i < j$$

These clauses ensure that exactly one of the indicator variables is true. Additionally, for each conditional probability $P(x_i|u)$, where u is a specific assignment to the parents of x, a parameter variable $p_u^{x_i}$ is introduced. A clause links this parameter to the appropriate context via:

$$v^{u_1} \wedge \ldots \wedge v^{u_n} \wedge v^{x_i} \Leftrightarrow p_u^{x_i}$$

where $v^{u_1}, \ldots, v^{u_n}$ are the indicator variables for the parent assignments.

The ENC3 [CD05] encoding is a more compact version of the ENC1 encoding. In this encoding, the main improvement is that parameter variables are reused for identical probability values across the CPTs, reducing the total number of parameters and clauses.

ENC4 [CD06] aims to simplify the representation by removing irrelevant variables, subformulas, and by identifying variables that are actually implied. This often leads to a more decomposable CNF, which can significantly reduce the time and memory usage for the probabilistic inference. Lastly,

ENC4linp [Bar+16] builds further on ENC4 by encoding the frequent probability value of each CPT implicitly. It also introduces a log-encoding for variable values, making the CNF representation more compact.

While these optimisations are beneficial for inference, they are not always compatible with being used in a learning setting. For instance, ENC3's parameters are reused across identical CPT entries, which assumes fixed probabilities. This is not possible in a learning setting, as the parameters are to be learned. Moreover, let us notice that the variables of Bayesian Networks can be binary (e.g., there is a burglary or not) or *multi-valued* (e.g., there is a strong, mild, or no earthquake). This multi-valued aspect can be encoded in CNF using the classical encoding methods [Fie+15; Bar+16], but these are not especially optimised to exploit the multi-valued structure and require additional clauses, expressing their mutual exclusivity, to ensure their correct representation.

Once the Bayesian Network has been encoded in CNF, the learning framework requires the probability of a query being satisfied given the weights assigned to its literals to be inferred. The computation of the probability of a given query, i.e., for a variable to take a specific value, can thus be done by summing the probabilities of all models that satisfy the query, which is, in practice, performed by computing weighted model counting.

In a neuro-symbolic framework, the symbolic structure provided to the learning is *the topology of the Bayesian Network*. The associated probabilities in the CPTs are instead inferred from sub-symbolic models and thus *learned* from the data.

> **Example 4.2.5.** (Bayesian Network in Neuro-Symbolic Learning)
> In the alarm setting of example 4.1.1, a neural model may process camera images and seismic activity data to predict the probability of a burglary and earthquake. These predictions are then used to populate the CPTs, and the symbolic structure is used to infer the probability of the queried variable using WMC. Finally, discrepancies between the predicted and expected probabilities, computed with the loss function, are used to update the neural model, to ensure it learns to predict probabilities that are more consistent with the symbolic structure.

Reliability Networks. For reliability networks, the first step to be used in neuro-symbolic frameworks is to encode symbolic knowledge described by the graphical model into a CNF formula. This CNF encoding can be done using classical encoding methods [Vla+16], but it has been shown that these problems can be more efficiently encoded using projection-based encodings, requiring performing *Projected Weighted Model Counting*

(PWMC) to compute the queries probabilities [Due+17].

Once the reliability network is encoded in CNF, the probability of a query being satisfied given the weights assigned to its edges has to be inferred for the learning framework to be able to update the sub-symbolic model parameters. The probability of a query being satisfied is computed by summing the probabilities of all models that satisfy the query, which is done by computing projected weighted model counting (PWMC).

In a neuro-symbolic setting, the symbolic structure provided to the learning is the *reliability network topology*, while edge probabilities are predictions made by a sub-symbolic model and *are to be learned*.

> **Example 4.2.6.** (Reliability Network in Neuro-Symbolic Learning)
> In the case of an electrical network as the one presented in example 4.1.2, the neural model may, for instance, receive data from various sensors and use them to predict whether a given cable is still functional. The cable functionality predictions are then used as input probabilities for the edges in the reliability network. The latter then computes the probability of a given query using PWMC. The probability obtained using the sub-symbolic model predictions can be compared to the expected probability for that query using a loss function. This comparison allows the sub-symbolic model to be trained to predict probabilities for its defined task (link failure prediction) in a way that remains consistent with the symbolic knowledge encoded in the reliability network.

Probabilistic Logic Programs. As for the two previous knowledge types, the requirement to assess the WMC of a logical formula in our training setting is to have it expressed in CNF. However, ProbLog programs are expressed in first-order logic, which allows for the expression of more complex relationships between facts.

Hence, to be used in a neuro-symbolic learning framework and apply WMC for probabilistic inference, the ProbLog program needs to be grounded and transformed into a CNF formula. This grounding process replaces the variables in the program with their corresponding values, resulting in a propositional logic formula. Works have shown how to ground a ProbLog program and transform it into CNF [Fie+12; Fie+15; Vla+16]. Concerning the annotated disjunctions that express multi-valued facts, these are encoded using additional constraints for mutual exclusivity [DK15; Man+18], just as done for the Bayesian Networks.

Example 4.2.7. (Grounding of ProbLog Program)
After grounding, the ProbLog program representing the query for the digits to sum up to two, introduced in Example 4.1.4, becomes:

```
0.15::digit(val0,0);0.05::digit(val0,1);0.3::digit(val0,2);
0.04::digit(val0,3);0.02::digit(val0,4);0.1::digit(val0,5);
0.03::digit(val0,6);0.25::digit(val0,7);0.01::digit(val0,8);
0.05::digit(val0,9).

0.13::digit(val1,0);0.06::digit(val1,1);0.1::digit(val1,2);
0.21::digit(val1,3);0.2::digit(val1,4);0.04::digit(val1,5);
0.1::digit(val1,6);0.06::digit(val1,7);0.01::digit(val1,8);
0.09::digit(val1,9).

addition(val0,val1,2) :- digit(val0,0), digit(val1,2), true.
addition(val0,val1,2) :- digit(val0,1), digit(val1,1), true.
addition(val0,val1,2) :- digit(val0,2), digit(val1,0), true.
```

In a neuro-symbolic context, the program's *structure is fixed*, while the probabilities of facts (e.g., digit values) are derived from neural prediction (e.g., using MNIST [LeC+98] images), and *are to be learned*. This integration of ProbLog and neural model has been implemented in DeepProbLog [Man+18], which trains the neural component to be more consistent with the symbolic one.

Other Forms of Knowledge. As this category covers the cases in which the symbolic knowledge is provided in a CNF formula, the integration of this symbolic knowledge in a neuro-symbolic learning framework is straightforward. The symbolic knowledge can directly be used to compute the probability of a query being satisfied, using WMC. The obtained probability can then be compared to the expected one using a loss function, allowing for the update of the sub-symbolic model parameters to be more consistent with the symbolic knowledge.

In this neuro-symbolic setting, the knowledge lies in the *structure between the facts from the CNF formula*. The probabilities associated with the different facts are obtained from the output of sub-symbolic models and *are to be learned*.

Example 4.2.8. (Other Forms of Knowledge in Neuro-Symbolic Learning)
In the ROAD-R example, the probabilities associated to the different facts are obtained from the output of a deep learning model processing the images and bounding boxes and performing a multi-label classification on the 41 possible labels. The knowledge is then evaluated with

> WMC on the predicted probabilities, and the computed probability can be compared to the expected one using the loss function, then allowing for the learning of a sub-symbolic model that is consistent with the domain knowledge.

4.3 Algorithmic Tools

In the previous sections, we have introduced the general learning setting and discussed various forms of symbolic knowledge that can be integrated into a neuro-symbolic learning framework. We also described how the probability of a query being satisfied can be computed using (P)WMC. This inferred probability $P^*_{\mathcal{W}}[F_i]$ is then compared to the expected one y_i via a loss function, which enables the parameters θ of the sub-symbolic model to be updated accordingly. This section focuses on the algorithmic tools required to perform these different steps, specifically the learning of the neural model parameters θ and the efficient computation of the gradients through the WMC of the symbolic formula.

4.3.1 Gradient Descent.

Parameter updates in a deep-learning model M_θ are typically performed via *gradient descent*, an iterative optimisation algorithm. The basic steps of gradient descent are:

- Compute the model's output for each input sample: $M_\theta(d_i)$.
- Compute the loss over the dataset:

$$\mathcal{L}(M_\theta(d_i), y_i).$$

- Compute the gradient of the loss with respect to the model parameters:

$$\frac{\partial \mathcal{L}(M_\theta(d_i), y_i)}{\partial \theta} = \frac{\partial \mathcal{L}(M_\theta(d_i), y_i)}{\partial M_\theta(d_i)} \frac{\partial M_\theta(d_i)}{\partial \theta}.$$

- Update the model parameters using learning rate α in the opposite direction of the gradients, which minimises the loss:

$$\theta \leftarrow \theta - \alpha \cdot \frac{\partial \mathcal{L}(M_\theta(d_i), y_i)}{\partial \theta}.$$

- Repeat the process until convergence of the parameters θ, or until another stopping criterion is satisfied.

This gradient descent process is commonly used in deep learning to iteratively adjust the model parameters θ to minimise the loss function $\mathcal{L}$, which quantifies the difference between the predicted and expected outputs. It is implemented in most deep learning frameworks, such as PyTorch [Pas+19] and TensorFlow [Aba+16], which provide automatic differentiation capabilities to compute the gradients efficiently. However, our considered learning pipeline does not directly compute the loss based on the output of the neural model. The neural model outputs are instead passed through the (P)WMC computation, whose result is, in turn, used to compute the loss. Hence, we need to be able to compute the gradient of the loss with respect to the neural model parameters θ through the (P)WMC computation.

4.3.2 Differentiation Through Symbolic Models.

As described in Section 4.2.4, the training loss depends not directly on the neural outputs, but on the (P)WMC computed over the symbolic knowledge, where the outputs define the literal probabilities. The loss is thus expressed as:

$$\mathcal{L}\left(P^*_{M_\theta(d)}[F], y\right),$$

with $P_{M_\theta(d)}[F]$ denoting the probability that the formula F is satisfied given the model outputs.

To update θ via gradient descent, we must differentiate this composite function. By applying the chain rule, the gradient is decomposed as follows:

$$\frac{\partial \mathcal{L}\left(P^*_{M_\theta(d)}[F], y\right)}{\partial \theta} = \frac{\partial \mathcal{L}\left(P^*_{M_\theta(d)}[F], y\right)}{\partial P^*_{M_\theta(d)}[F]} \frac{\partial P^*_{M_\theta(d)}[F]}{\partial M_\theta(d)} \frac{\partial M_\theta(d)}{\partial \theta}. \tag{4.1}$$

Thus, a central step in neuro-symbolic learning is to compute the gradient of $P^*_{M_\theta(d)}[F]$ with respect to the neural outputs, that is, to differentiate the WMC. This requires tractable representations of the WMC of the symbolic knowledge. A common approach is to compile the logical formula into an *Arithmetic Circuit* (AC) [Vla+16], a directed acyclic graph encoding the symbolic formula as a sequence of arithmetic operations (typically additions and multiplications). The circuit can be evaluated bottom-up to compute the WMC and differentiated top-down via backpropagation. This transformation of logical formulas into differentiable arithmetic circuits is known as *knowledge compilation* and is detailed below.

4.3.3 Knowledge Compilation

A logical formula can be represented in various syntactic (e.g., propositional logic or first-order logic) and structural forms (e.g., binary decision diagrams,

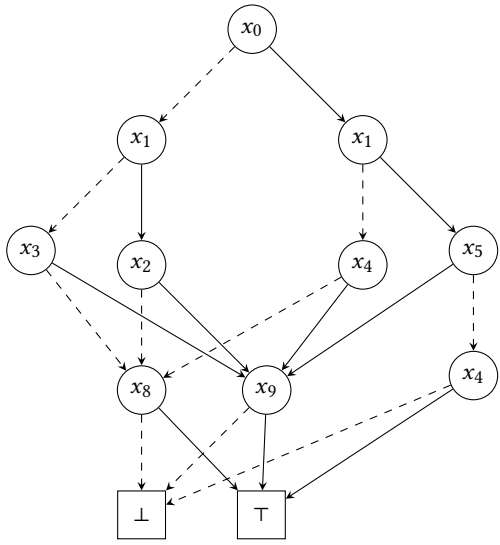


Figure 4.5: Example of a binary decision diagram (BDD) for the running example 4.2.2. The BDD consists of nodes representing variables. Nodes represent variables, with solid edges corresponding to assignments to $\top$ and dashed edges to $\bot$. The leaf $\top$ indicates that the formula is satisfied along that path, while $\bot$ indicates that it is not satisfied.

sentential decision diagrams). Structural representations are particularly advantageous, as they support efficient reasoning and repeated inference over the same formula with varying variable assignments. Knowledge Compilation (KC) is the process of transforming a logical formula into such a structural form. The key target representations are described in Section 4.5.2.

One common representation is the *Binary Decision Diagram (BDD)*, a directed acyclic graph encoding a formula as a series of binary decisions. Each node represents a variable, and each edge represents one of its possible assignments (true or false). Figure 4.5 illustrates a BDD for the running example introduced in Example 4.2.2.

Another structural form is the *Boolean Circuit*, a directed acyclic graph in which internal nodes correspond to logical operations (e.g., conjunctions and disjunctions), and leaves correspond to literals. The circuit can be evaluated bottom-up to determine whether a particular variable assignment satisfies the formula. A BDD can be transformed into a Boolean circuit by following all paths leading the model to the $\top$ leaf and adding a disjunction for variable assignments along the way and a conjunction for each edge leading to a new variable. The resulting circuit for Example 4.2.2 is shown in Figure 4.6.

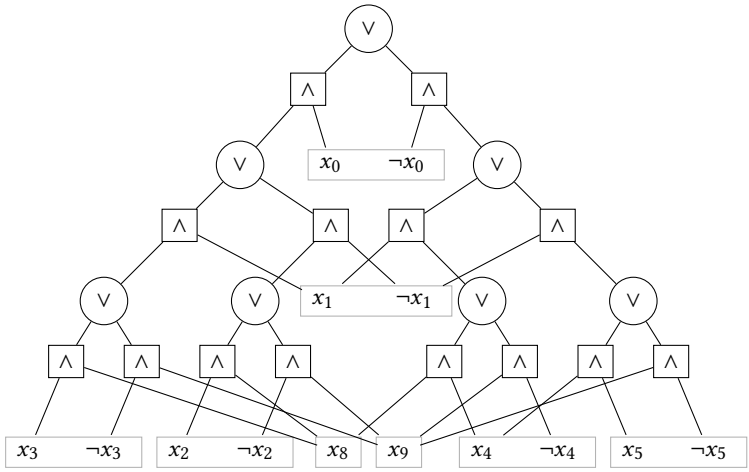


Figure 4.6: Example of a Boolean circuit for the running example 4.2.2. Internal nodes correspond to logical operations (AND/OR), while leaves represent literals.

Boolean circuits allow the evaluation of satisfiability given a binary variable assignment. However, in the probabilistic setting, variables x_i are associated with probabilities w_{x_i} of being true. In this case, Boolean circuits are generalised into *Arithmetic Circuits (ACs)*.

An arithmetic circuit is a directed acyclic graph (DAG) representing the logical formula using arithmetic operations. Three types of nodes are usually found in ACs: sum nodes, representing logical disjunctions, product nodes, representing logical conjunctions, and leaf nodes, corresponding to literals and their associated weights. ACs are particularly useful as they support both efficient evaluation and differentiation, making them well-suited to neuro-symbolic learning. Figure 4.7 shows an AC corresponding to Example 4.2.2. Evaluating it bottom-up yields the WMC of F, i.e., the probability that F is satisfied given a set of literal weights. Conceptually, an AC mirrors the Boolean circuit of Figure 4.6, with logical operations replaced by differentiable arithmetic ones.

In our neuro-symbolic framework, ACs are especially advantageous: once a query is compiled into an AC, it can be evaluated efficiently for different weights during training, enabling extensive reuse of the compiled structure. Moreover, as ACs consist entirely of differentiable operations, they are compatible with gradient-based optimisation. In particular, a top-down differentiation pass through the circuit enables the computation of the gradient of the circuit's output with respect to its inputs, i.e.:

$$\frac{\partial P^*_{M_\theta(d)}[F]}{\partial M_\theta(d)},$$

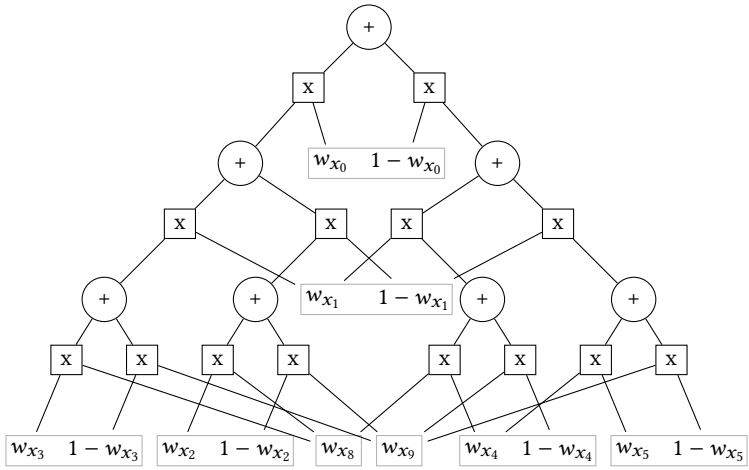


Figure 4.7: Example of an arithmetic circuit (AC) for the running example 4.2.2. Circles denote sum nodes, squares denote product nodes, and leaf nodes represent literals with their associated weights. The complements of the weights w_{x_8} and w_{x_9} are not included as they are not directly used for the AC evaluation.

as required in Equation 4.1. This gradient can then be propagated further into the sub-symbolic model, enabling end-to-end backpropagation across both symbolic and neural components. Consequently, the parameters θ of the neural model are updated under the structured guidance of the symbolic knowledge encoded within the AC.

4.4 Limitation of the General Setting

In this chapter, we have introduced the general neuro-symbolic learning setting, which combines symbolic knowledge with sub-symbolic models to learn from data. We have discussed the key components of this setting, including the compilation of symbolic knowledge into arithmetic circuits, the computation of probabilities using weighted model counting and projected WMC, and the use of loss functions to guide the learning process. The overall process for training a neural model under symbolic constraints can be summarised as follows and is illustrated in Figure 4.8.

The first step consists of compiling the symbolic query associated with each data sample into an arithmetic circuit (AC). Once compiled, the learning of the sub-symbolic model parameters proceeds in a gradient-based manner. At each iteration:

- The sub-symbolic model M_θ is evaluated on the input data d_i, producing a set of weights $M_\theta(d_i)$ corresponding to the literals

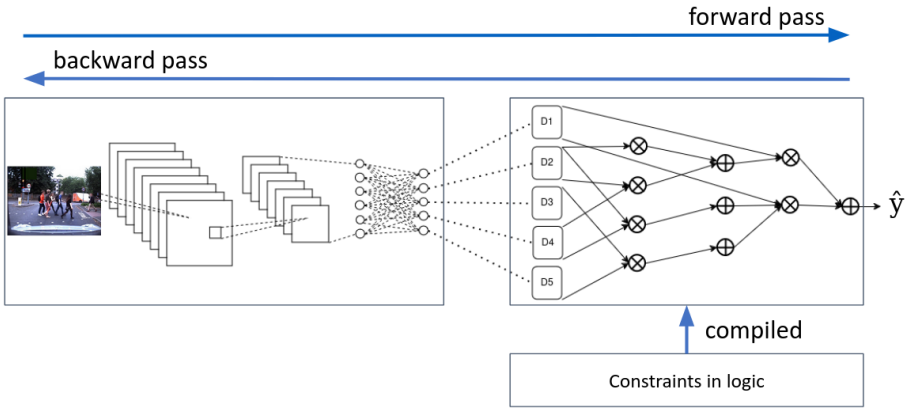


Figure 4.8: General learning setting combining a differentiable model with symbolic constraints. A differentiable model, here a convolutional neural network (CNN), processes input data and outputs weights for the probabilistic variables of a logical model. The logical model is compiled into an arithmetic circuit (AC), which can be evaluated in a single bottom-up pass. Backpropagation is used to compute the gradient of the AC inputs, which are then propagated through the differentiable model.

of the AC of query F_i.

- The AC is then evaluated in a bottom-up pass using these predicted weights, yielding the WMC of the query $P^*_{M_\theta(d_i)}[F_i]$, i.e., the probability that the query is satisfied given the predicted weights.
- A loss function $\mathcal{L}(P^*_{M_\theta(d_i)}[F_i], y_i)$ is computed between the predicted and expected probabilities.
- Gradients of the loss with respect to the sub-symbolic model parameters are computed via backpropagation: first through the AC in a top-down pass, then through the sub-symbolic model.
- The model parameters are updated accordingly, and the process repeats until convergence or another stopping criterion is met.

This section now discusses a central challenge of the presented framework and introduces a simplified learning setting that isolates the affected components. The next chapters will build on this simplified formulation to present methods addressing the identified limitations.

4.4.1 Challenge

One of the main limitations of the defined framework lies in the *compilation of symbolic knowledge into ACs*, a process known to be #*P*-hard [Val79;

Rot96]. Consequently, compilation can exhibit exponential complexity in the worst case. For complex formulas, compilation may become prohibitively expensive in both time and memory, and in some cases may fail to complete under realistic computational constraints.

The following chapters of this thesis focus on learning under these challenging conditions. Specifically, they explore methods based on *partial compilation* and *approximate inference*, which aim to mitigate the bottleneck introduced by learning from the full AC compilation. Since this limitation arises from the compilation step itself and has no link with the neural component of the model, it only impacts the computation of gradients through the ACs. To study this aspect in isolation, we introduce a *simplified learning setting*, centred on probabilistic inference and backpropagation through the ACs, as described in the next subsection.

4.4.2 Simplified Learning Setting

To study the challenges associated with *compilation and learning under complex logical constraints*, we adopt a simplified setting, illustrated in Figure 4.9.

Instead of relying on a differentiable neural model to compute the literal weights, we directly optimise a set of *trainable parameters* $\mathcal{W}$, representing the outputs that a neural model would normally produce. Consequently, rather than performing probabilistic inference $P^*_{M_\theta(d)}[F]$ using weights predicted by a neural model M_θ, we use the trainable parameters $\mathcal{W}$ directly as the weights for the literals within the AC. The learning objective then becomes to optimise $\mathcal{W}$ such that the resulting output probabilities $P^*_{\mathcal{W}}[F]$ align with the target values associated with the training queries.

In standard neuro-symbolic settings, the literal weights computed by $M_\theta(d)$ typically represent probabilities and must hence satisfy normalisation constraints (e.g., summing to one for multi-valued distributions). This is usually enforced automatically via a softmax activation at the network's output layer. In our simplified setting, since $\mathcal{W}$ is learned directly, these constraints must be enforced explicitly. To ensure valid probability distributions in the simplified setting, we optimise softmax parameters rather than the raw distribution values themselves. The gradient computed through the arithmetic circuit is thus adjusted to account for the softmax transformation, ensuring that each multi-valued distribution remains properly normalised.

Although this setting removes the complexity of a full end-to-end differentiable model, it isolates the essential symbolic learning dynamics under complex logical constraints. As such, the results obtained in this simplified configuration are expected to remain valid when integrated into a complete neuro-symbolic pipeline, where the trainable parameters $\mathcal{W}$

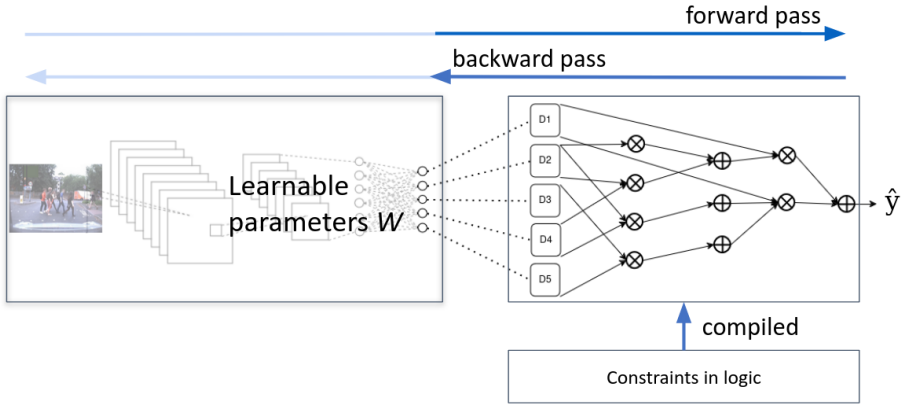


Figure 4.9: Simplified learning setting in which the neural network, whose output would normally provide the weights of the logical model variables, is replaced by a set of trainable parameters $\mathcal{W}$.

are produced by a neural component.

Having defined this simplified learning setting and its main components, the next section reviews related work relevant to the various aspects of the learning process.

4.5 Related Work

This section provides a structured overview of related work relevant to our problem setting, covering main aspects such as weighted model counting computation and the compilation of logical formulas into arithmetic circuits. Additionally, it gives an overview of some neuro-symbolic approaches that have been integrated into similar learning frameworks.

4.5.1 (Weighted) Model Counting

As defined in Section 4.2.2, in WMC, each literal l of a logical formula F is associated with a weight w_l, and the goal is to compute the sum of the weights of all satisfying assignments for F. The weight of an assignment is typically defined as the product of the weights of the literals that are true in that assignment.

The dominant approach for exact model counting is based on SAT-solving techniques, particularly the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [DLL62], a backtracking search algorithm. Modern WMC solvers extend the DPLL framework by incorporating advanced search mechanisms such as component caching, clause learning, and universal hashing. In Algorithm 1, presented in the next section, we describe

Schlandals, the DPLL-based weighted model counter used in this work, and provide an overview of the general principles of DPLL-based model counting.

Several works have proposed improvements to enhance the efficiency of model counting. Cachet, introduced by Sang et al., was the first model counter combining component caching with clause learning [San+04; SBK05]. Thurley introduced a new caching scheme in sharpSAT [Thu06], allowing for a significant reduction in the cache size. More recently, Ganak [Sha+19] used universal hashing techniques to outperform previous model counters.

4.5.2 Knowledge Compilation

Knowledge Compilation (KC) refers to the transformation of a logical formula into a more tractable representation, such as an AC, that supports efficient inference. The choice of a target representation determines the tractability of various operations, such as the WMC.

Many target languages have been proposed for KC, most being subsets of the deterministic Decomposable Negation Normal Form (d-DNNF) language [DM02]. These languages allow compact representation and efficient computation. The properties of d-DNNF are that it is in negation normal form (NNF) (formula made only of $\wedge$ and $\vee$ operators, negations applied to variables directly), decomposable (any conjunction is over disjoint set of variables) [Dar99; Dar01a], and deterministic (mutually exclusive disjunctions) [Dar01b]. It has been shown that the trace of exact model counting search algorithms corresponds to a subset of d-DNNF language [Dar01b; HD07].

Several subclasses of d-DNNF have been introduced, each balancing the trade-off between the size of the representation and the efficiency of the operations. Among the commonly used ones, Ordered Binary Decision Diagrams (OBDDs) [Bry86] give compact representations for a fixed order of variables, while ensuring the properties of d-DNNF, of canonicity (unique representation for a fixed variable order), and of polynomial-time computation for Boolean operations. Sentential Decision Diagrams (SDDs) [Dar11] also ensure d-DNNF properties, canonicity (for a fixed variable tree), and polynomial time support, while having a tighter bound on their size than OBDDs. Building upon SDDs, Probabilistic Sentential Decision Diagrams (PSDDs) [Kis+14] add probability distributions over the models. More recently, Constrained Conjunction and Decision Diagrams (CCDD) were introduced as a generalisation of d-DNNF [LMY21]. CCDD has properties of being complete (every formula can be represented) and of supporting model counting in linear time.

Many compilers have been developed to compile input formulas into these target languages. Most focus on producing an exact representation of the formula. The dsharp compiler [Mui+10; Mui+12] uses d-DNNF as a target language to propose a fast compilation based on the sharpSAT solver [Thu06]. Oztok and Darwiche propose a top-down compiler [OD15] generating SDDs [Dar11]. The d4 compiler also targets the d-DNNF language using a dynamic decomposition approach and simplification rules allowing a reduction in the compilation time and output size [LM17]. Recently, ExactMC was proposed as a compiler for CCDDs [LMY21].

Finally, the size of the compiled representation is a critical factor for efficiency and can vary significantly depending on the variable ordering used during compilation. Selecting an appropriate variable ordering is therefore essential for achieving compact and tractable compiled forms [Bry86; BW96; Vla+14].

4.5.3 Combining Neural Methods with Symbolic Constraints for Learning

As discussed in Chapter 2, several taxonomies have been proposed to classify the wide range of approaches that combine neural networks with symbolic reasoning. While these surveys capture the breadth of the neuro-symbolic AI landscape, the second part of this thesis adopts a narrower and more principled interpretation of neuro-symbolic integration.

In particular, we focus on approaches that align with what [GL23] describe as the original vision of neuro-symbolic AI: "research that integrates in a principled way neural network-based learning with symbolic knowledge representation and logical reasoning". This section, therefore, reviews methods that embody this original formulation, where symbolic constraints play an active role in guiding neural learning.

Existing Surveys. Several surveys have analysed this particular NeSy setting from complementary perspectives. Manhaeve et al. [Man+21b] outline the logical, probabilistic, and neural paradigms underlying NeSy methods, distinguishing between approaches that use logic as a regulariser and those that integrate it within neural constructs. De Raedt et al. [De +19] emphasise the importance of preserving the core properties of the combined paradigms, assessing whether existing frameworks maintain both neural and symbolic components as independently functional cases.

Giunchiglia et al. [GSL22] classify approaches by how background knowledge is represented and used, while Dash et al. [Das+22] identify three strategies for embedding knowledge into neural models: transforming the data, the loss, or the model itself. Vermeulen et al. [VMM23] focus on the

categorisation and benchmarking of NeSy systems and tasks, and Marra et al. [Mar+24] review existing methods along seven axes, including inference type, logic syntax, and learning task. Finally, De Smet and De Raedt [DD25] propose a unifying view of neuro-symbolic inference as an "aggregation of logically selected interpretations from a neural belief", linking diverse approaches through shared principles of language, semantics, belief, and logic.

Recent Contributions. Recent work has introduced numerous methods for integrating neural networks with symbolic reasoning. Following the categorisation proposed by De Smet and De Raedt [DD25], we distinguish approaches based on their logic semantics, whether they rely on fuzzy (continuous relaxations of logical operators) or Boolean (discrete truth-valued) reasoning.

The first category covers works that use fuzzy logic semantics to evaluate the logical constraints. Logic Tensor Networks [SG16; DSG17; Bad+22], map logical predicates and functions to neural networks, interpreting logical rules as soft constraints through fuzzy semantics. DL2 (Deep Learning with Differentiable Logic) translates logical knowledge into differentiable loss functions using smooth relaxations of formulas that quantify constraint violations, enabling generalisation to unseen inputs such as adversarial examples. TensorLog [CYM17] represents first-order logic operators as matrix operations, compiling logical rules into differentiable computation graphs that perform soft, fuzzy-like reasoning.

The second category consists of methods that use Boolean logic semantics to evaluate the logical constraints. DeepProbLog [Man+18; Man+21a], a well-known example fully aligned with the framework considered in this work, compiles probabilistic logic programs into arithmetic circuits, which are then combined with neural networks, enabling end-to-end gradient-based training. Xu et al. [Xu+18] propose the Semantic Loss, which compiles complex constraints into ACs to evaluate in a loss term, how well neural predictions satisfy logical rules. Semantic Probabilistic Layers [Ahm+22b] introduce a differentiable layer that enforces logical constraints on neural outputs by combining a probabilistic circuit and a constraint circuit, allowing for the generation of predictions that satisfy some specified logical properties at all times.

Beyond these approaches, efforts have been made to propose optimised frameworks, unified languages, and benchmark suites for NeSy tasks. KLay [MDM24] introduces GPU-optimised arithmetic circuits for efficient computation. Kompyle [MD25] generalises algebraic model counting to support efficient gradient computation across semirings. ULLER [Kri+24] provides a unified language for learning and reasoning. Benchmarking ini-

tiatives include the suite by Vermeulen et al. [VMM23], which covers diverse NeSy task categories, and the benchmark of Bortolotti et al. [Bor+25], which targets the identification and mitigation of reasoning shortcuts [Mar+23; Kri+25].

4.6 Schlandals

`Schlandals` [DSN23] is a solver designed for both weighted model counting and arithmetic circuit compilation. It computes the total weight of a formula's models, $WMC_W(F)$, by exploring the space of interpretations through a DPLL-style backtracking search.

In this work, `Schlandals` is used both to compute query probabilities and to compile the corresponding ACs. In addition to the fact that it was developed in our lab, `Schlandals` was selected over other solvers and compilers for several reasons:

- `Schlandals` supports PWMC, which is a strength for the handling of reliability networks, as identified in Section 4.2.4. Unlike projected solvers such as Ganak, which branch on all variables, `Schlandals` branches only on the projection (priority) variables, resulting in greater efficiency [DSN23].

- `Schlandals` natively handles multi-valued distributions, without the need for auxiliary constraints. The use of multi-valued variables has been made in our Bayesian network and ProbLog examples (Sections 4.1.1 and 4.1.3), and is particularly present in neuro-symbolic learning contexts, where the sub-symbolic model outputs often correspond to distributions over multi-valued categorical variables.

- `Schlandals` provides mechanisms for approximate inference, enabling learning with complex queries that cannot be fully compiled into exact ACs due to computational constraints, as discussed in Section 4.4.1.

The remainder of this section presents the key features of `Schlandals` relevant to our contribution.

4.6.1 Weighted Model Counting Solver and Arithmetic Circuit Compilation

Like many existing solvers and compilers, `Schlandals` can perform WMC and compile formulas into arithmetic circuits.

Algorithm 1 DPLL-Style Weighted Model Counter

Require: A Boolean formula F over variables $\mathcal{X}$ and literal weights $\mathcal{W}$
Ensure: $P^*_{\mathcal{W}}[F]$ the weighted model count of F given weights $\mathcal{W}$

1: **procedure** $\text{WMC}_{\mathcal{W}}(F)$
2: **if** $|\mathcal{X}| = 0$ **then return** 1
3: **if** $F \mapsto p$ in cache **then return** p
4: $P^*_{\mathcal{W}}[F] \leftarrow 0$
5: $x \leftarrow$ choose_variable()
6: **for** $v \in \{\top, \bot\}$ **do**
7: **if** $F[x = v] = \bot$ **then continue** ▷ Applies propagation
8: $C \leftarrow$ components of $F[x = v]$ ▷ Independent components
9: **for** $F_c \in C$ **do** $p^c \leftarrow \text{WMC}_{\mathcal{W}}(F_c)$
10: $prop \leftarrow$ literals forced to $\top$ by BUP
11: $p_{prop} \leftarrow \prod_{l \in prop} w_l$ ▷ Weights of forced literals
12: $P^*_{\mathcal{W}}[F]$ += $w_{x=v} \times p_{prop} \times \prod_{c \in C} p^c$
13: Add $F \mapsto P^*_{\mathcal{W}}[F]$ in cache
14: **return** $P^*_{\mathcal{W}}[F]$

Weighted Model Counting. Classical DPLL-style model counters compute the WMC of a formula by selecting a variable $x \in \mathcal{X}$, evaluating both truth values, propagating assignments using *Boolean Unit Propagation* (BUP), and recursively exploring the resulting residual formulas. To be efficient, such counters integrate a caching system: once the count of a sub-formula F' is computed, it is stored as $C[F'] \mapsto \text{WMC}(F')$. This allows to avoid recomputing the same sub-formula multiple times.

The search algorithm of `Schlandals`, presented in Algorithm 1, follows the same structure as common model counters. The probability $P^*_{\mathcal{W}}[F]$ is computed recursively. First, a variable x is selected to branch on (line 5). The two possible values for the variable x, $v = \top$ and $v = \bot$ are explored (lines 6-12). For each assignment to x, boolean unit propagation (BUP) is applied (line 7). If the resulting formula is not unsatisfiable, the resulting formula is decomposed into independent components (line 8) that are solved recursively (line 9).

Arithmetic Circuit Compilation. It is known that the traces of exhaustive DPLL search, with decomposition, correspond to a subset of d-DNNF language [Dar01b; HD05; HD07] and can be converted to ACs. This principle has been used in popular compilers such as `dsharp` or `d4` [Mui+10; LM17]. Those adapt the behaviour of a classical DPLL model

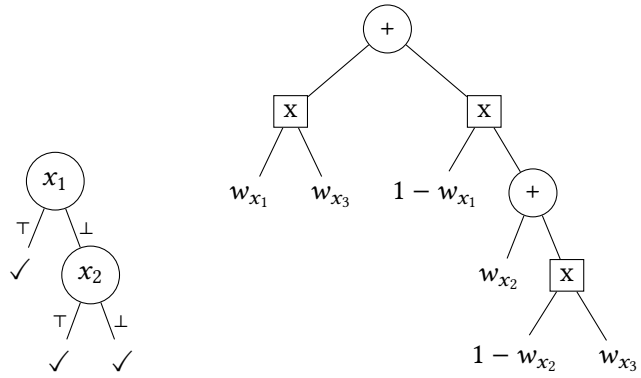


Figure 4.10: Left: A search tree for Example 4.2.1, $F = (\neg x_1 \vee x_3) \wedge (x_2 \vee x_3)$**. Leaves corresponding to a model are marked with ✓. Right: An AC computing WMC$_{\mathcal{W}}(F)$, corresponding to the polynomial** $w_{x_1} w_{x_3} + (1 - w_{x_1})(w_{x_2} + (1 - w_{x_2}) w_{x_3})$**.**

counter to generate a d-DNNF diagram instead of directly computing the WMC. A d-DNNF diagram is built by replacing the sum with disjunction and the product with conjunction nodes. Accordingly, instead of storing the WMC of a sub-formula, the cache in these compilers retains the root of the corresponding d-DNNF representation.

> **Example 4.6.1.** (Search Tree and AC for WMC)
> Figure 4.10 shows, on the left, a search tree for our simple running example $F = (\neg x_1 \vee x_3) \wedge (x_2 \vee x_3)$ (Example 4.2.1) and, on the right, a corresponding arithmetic circuit computing WMC$_{\mathcal{W}}(F)$ as derived by a traditional compiler.
>
> At the root, setting $x_1 = \top$ (left branch) reduces the first clause to x_3, forcing $x_3 = \top$. This latter assignment satisfies the second clause, simplifying F to $\top$. The two models considered in this branch are $\{x_1, x_2, x_3\}$ and $\{x_1, \neg x_2, x_3\}$ and their weighted count is $w_{x_1} w_{x_2} w_{x_3} + w_{x_1} w_{\neg x_2} w_{x_3} = w_{x_1} w_{x_3}$, which corresponds to the AC's left-most part.

Since `Schlandals` is a DPLL-style search-based model counter, its search trace can be used to compile an AC corresponding to the search that has been computed [HD07]. Instead of modifying the search algorithm to produce an AC, `Schlandals` stores additional information in the cache and constructs the AC as a post-processing step.

Intuitively, the cache stores information about the satisfying assignments encountered during the search. More formally, let F be a boolean formula, x a variable of F, and $F[x]$ and $F[\neg x]$ the two sub-formulas obtained by branching on x. We denote $\text{prop}_x^{\top}$ (resp. $\text{prop}_{\neg x}^{\top}$) as the set of variables

(excluding x) forced to $\top$ when applying BUP with $x = \top$ (resp. $x = \bot$). The cache entry has the following structure:

$$C[F] \mapsto \left(x, (F[x], \text{prop}_x^\top), (F[\neg x], \text{prop}_{\neg x}^\top)\right) \tag{4.2}$$

These $\text{prop}^\top$ sets record the interpretations built alongside a branch. The sub-formulas $F[x]$ and $F[\neg x]$ are not stored as-is as formula objects in the cache; they are rather stored using a hashable representation. For instance, the cache entry at the root node of the search tree in Figure 4.10 would be as follows:

$$(x_1, (\top, [x_3]), (x_2 \vee x_3, [\ ]))$$

Algorithm 2 shows how the cache can be parsed to produce an AC from the models seen during the search. When computing the (sub-)circuit for a formula F, Algorithm 2 first creates a sum node that will link the two assignment branches (line 2). Then, it considers both branches of the search tree (lines 4–9), producing a product node (line 5) with the variables propagated to $\top$ during the search (line 7) and the sub-circuit computing the WMC of the residual formula (line 8). For simplicity, static decomposition is omitted in Equation (4.2) and Algorithm 2. However, the framework defined by Equation (4.2) naturally extends to such cases by storing a list of sub-formulas (i.e., $F^1[x], F^2[x], \ldots$) instead of a single residual formula. Since $\text{prop}^\top$ sets correspond to edges in a d-DNNF representation, this caching mechanism is as costly as constructing a d-DNNF during the search. Indeed, the sets $\text{prop}^\top$ correspond to pointers to leaves in a d-DNNF; hence, compilers constructing the diagram directly must store as many edges as there are elements in the $\text{prop}^\top$ sets.

The search trace and its corresponding AC of our simple running example (Example 4.2.1) are shown in Figure 4.10. The circuit obtained can be expressed as $w_1 w_3 + (1 - w_1)(w_2 + (1 - w_2)w_3)$ and computes the WMC of the formula.

4.6.2 Encoding in Schlandals

The encoding of background knowledge in `Schlandals` is done using propositional logic in conjunctive normal form (CNF), restricted to Horn clauses. Horn clauses have the following form:

Example 4.6.2. (Horn Clause)

$$x_1 \wedge \ldots \wedge x_n \Rightarrow x_h.$$

Algorithm 2 Compilation Algorithm from a Search Trace

Require: Cache C with entries as defined by Equation (4.2)
Ensure: An AC computing $\text{WMC}_{\mathcal{W}}(F)$

```
1: procedure COMPILE(F, C)
2:     N⁺ = new sum node
3:     (x, (F[x], prop_x^⊤), (F[¬x], prop_¬x^⊤)) = C[F]
4:     for l ∈ {x, ¬x} do
5:         N× = new product node
6:         add w_l as child of N×
7:         ∀x' ∈ prop_l^⊤ add w_x' as child of N×
8:         add Compile(F[l], C) as child of N×
9:         add N× as child of N⁺
10:    return N⁺
```

Which can be rewritten in CNF as:

$$\neg x_1 \vee \ldots \vee \neg x_n \vee x_h,$$

where x_i are variables, $\top$, or $\bot$. Here, x_h is the head of the clause and $x_1, \ldots, x_n$ are the implicants of the clause. The head of the clause is a single atom, and the implicant is a conjunction of atoms. The head represents the conclusion of the clause, and the implicant represents the conditions under which the conclusion is true.

Additionally, `Schlandals` and its encoding differ from other solvers by leveraging projected weighted model counting [DSN23]. As described in Section 4.2.2, with PWMC, variables are divided into priority and non-priority sets. Only priority variables, which have probabilistic weights, are considered during the search; non-priority variables have a weight of one. This approach reduces the search space, since branching occurs only on priority variables, and can encode certain WMC problems in polynomial space (e.g., when cycle breaking is required [Vla+16]), while also enabling additional problem simplifications.

4.6.3 Multi-Valued Distributions

A specific feature of `Schlandals` is its direct support for multi-valued distributions without requiring additional mutually exclusive constraints, and its efficient handling of these multi-valued distributions.

In this setup, binary variables are regrouped in distributions, with each variable x belonging exclusively to one distribution $Dist_i \in \mathcal{D}ist$, where $\mathcal{D}ist$ denotes the set of all distributions; $dom(Dist_i) \subseteq \mathcal{X}$ represents the variables in distribution $Dist_i$. This implicitly implies that if an

interpretation I does not fix *exactly* one variable $x \in dom(Dist_i)$ to $\top$ in each distribution $Dist_i$, then we have $F[I] = \bot$.

This allows, for instance, to represent the alarm Bayesian Network in 4.2 in the following form:

Example 4.6.3. (`Schlandals` Encoding of the Alarm Bayesian Network, Using Multi-Valued Distributions)

$$
\begin{aligned}
p^{B_{yes}} &\Rightarrow v^{B_{yes}} & v^{B_{yes}} \wedge v^{E_{mild}} \wedge p^{A_{yes}}_{B_{yes}E_{mild}} &\Rightarrow v^{A_{yes}}\\
p^{B_{no}} &\Rightarrow v^{B_{no}} & v^{B_{yes}} \wedge v^{E_{mild}} \wedge p^{A_{no}}_{B_{yes}E_{mild}} &\Rightarrow v^{A_{no}}\\
p^{E_{strong}} &\Rightarrow v^{E_{strong}} & v^{B_{yes}} \wedge v^{E_{strong}} \wedge p^{A_{yes}}_{B_{yes}E_{strong}} &\Rightarrow v^{A_{yes}}\\
p^{E_{mild}} &\Rightarrow v^{E_{mild}} & v^{B_{yes}} \wedge v^{E_{strong}} \wedge p^{A_{no}}_{B_{yes}E_{strong}} &\Rightarrow v^{A_{no}}\\
p^{E_{no}} &\Rightarrow v^{E_{no}} & v^{A_{no}} \wedge p^{J_{yes}}_{A_{no}} &\Rightarrow v^{J_{yes}}\\
v^{B_{no}} \wedge v^{E_{no}} \wedge p^{A_{yes}}_{B_{no}E_{no}} &\Rightarrow v^{A_{yes}} & v^{A_{no}} \wedge p^{J_{no}}_{A_{no}} &\Rightarrow v^{J_{no}}\\
v^{B_{no}} \wedge v^{E_{no}} \wedge p^{A_{no}}_{B_{no}E_{no}} &\Rightarrow v^{A_{no}} & v^{A_{yes}} \wedge p^{J_{yes}}_{A_{yes}} &\Rightarrow v^{J_{yes}}\\
v^{B_{no}} \wedge v^{E_{mild}} \wedge p^{A_{yes}}_{B_{no}E_{mild}} &\Rightarrow v^{A_{yes}} & v^{A_{yes}} \wedge p^{J_{no}}_{A_{yes}} &\Rightarrow v^{J_{no}}\\
v^{B_{no}} \wedge v^{E_{mild}} \wedge p^{A_{no}}_{B_{no}E_{mild}} &\Rightarrow v^{A_{no}} & v^{A_{no}} \wedge p^{M_{yes}}_{A_{no}} &\Rightarrow v^{M_{yes}}\\
v^{B_{no}} \wedge v^{E_{strong}} \wedge p^{A_{yes}}_{B_{no}E_{strong}} &\Rightarrow v^{A_{yes}} & v^{A_{no}} \wedge p^{M_{no}}_{A_{no}} &\Rightarrow v^{M_{no}}\\
v^{B_{no}} \wedge v^{E_{strong}} \wedge p^{A_{no}}_{B_{no}E_{strong}} &\Rightarrow v^{A_{no}} & v^{A_{yes}} \wedge p^{M_{yes}}_{A_{yes}} &\Rightarrow v^{M_{yes}}\\
v^{B_{yes}} \wedge v^{E_{no}} \wedge p^{A_{yes}}_{B_{yes}E_{no}} &\Rightarrow v^{A_{yes}} & v^{A_{yes}} \wedge p^{M_{no}}_{A_{yes}} &\Rightarrow v^{M_{no}}\\
v^{B_{yes}} \wedge v^{E_{no}} \wedge p^{A_{no}}_{B_{yes}E_{no}} &\Rightarrow v^{A_{no}}
\end{aligned}
$$

with the distributions defined as:

$$
\begin{aligned}
Dist_{Burglary} &= \{p^{B_{yes}}, p^{B_{no}}\} & Dist_{Alarm_5} &= \{p^{A_{yes}}_{B_{yes}E_{strong}}, p^{A_{no}}_{B_{yes}E_{strong}}\}\\
Dist_{Earthquake} &= \{p^{E_{strong}}, p^{E_{mild}}, p^{E_{no}}\} & Dist_{Alarm_6} &= \{p^{A_{yes}}_{B_{no}E_{strong}}, p^{A_{no}}_{B_{no}E_{strong}}\}\\
Dist_{Alarm_1} &= \{p^{A_{yes}}_{B_{yes}E_{no}}, p^{A_{no}}_{B_{yes}E_{no}}\} & Dist_{John_1} &= \{p^{J_{yes}}_{A_{no}}, p^{J_{no}}_{A_{no}}\}\\
Dist_{Alarm_2} &= \{p^{A_{yes}}_{B_{no}E_{no}}, p^{A_{no}}_{B_{no}E_{no}}\} & Dist_{John_2} &= \{p^{J_{yes}}_{A_{yes}}, p^{J_{no}}_{A_{yes}}\}\\
Dist_{Alarm_3} &= \{p^{A_{yes}}_{B_{yes}E_{mild}}, p^{A_{no}}_{B_{yes}E_{mild}}\} & Dist_{Mary_1} &= \{p^{M_{yes}}_{A_{no}}, p^{M_{no}}_{A_{no}}\}\\
Dist_{Alarm_4} &= \{p^{A_{yes}}_{B_{no}E_{mild}}, p^{A_{no}}_{B_{no}E_{mild}}\} & Dist_{Mary_2} &= \{p^{M_{yes}}_{A_{yes}}, p^{M_{no}}_{A_{yes}}\}
\end{aligned}
$$

Instead of introducing mutual exclusivity constraints to enforce multi-valued distributions, the practical handling of multi-valued distributions in `Schlandals` is done by introducing some adaptations to the classical DPLL-based search and compilation algorithm. Specifically: In the search

algorithm (Algorithm 1), branching is done over all values $v \in dom(D_i)$ for each variable x_i, rather than binary choices (lines 6). In the compilation algorithm (Algorithm 2), each cache entry is indexed by the selected value from each distribution (line 3), and the recursive decomposition explores each possible value of the variables (line 4).

4.7 Conclusion

This chapter provided an overview of the learning task, backpropagation, weighted model counting, knowledge compilation, and their roles in a neuro-symbolic learning framework. We introduced `Schlandals`, the solver and compiler used in this work, and outlined its main features, including its support for multi-valued distributions and projected inference.

A central limitation of the presented setting lies in the inherent difficulty of compiling background knowledge, as this task is #P-hard and becomes intractable when dealing with complex logical constraints. In the following chapters, we focus on addressing this challenge by exploring the potential for partial compilation of arithmetic circuits, computing an approximation of the true WMC. In particular, we will investigate whether learning guarantees can be obtained when operating on such partially compiled representations.

Finally, while the current chapter has focused on exact WMC and AC compilation, it is important to note that `Schlandals` and other existing methods also support approximate inference. As the next two chapters will focus on partial compilation of arithmetic circuits and approximate WMC, the choice was made to present the state of the art of these specific aspects in the next two chapters.

Learning from Partial Arithmetic Circuits

5

The previous chapter introduced the neuro-symbolic learning framework adopted in the second part of this thesis, where symbolic constraints guide the training of a neural network to ensure greater compliance with background knowledge [Man+18; Man+21a]. In this setting, probabilistic predictions from a sub-symbolic model are interpreted as literal weights used to evaluate the satisfaction of logical constraints (Figure 4.8). This evaluation is typically performed through weighted model counting (Section 4.2.2), and the resulting inferred probability guides the learning of the sub-symbolic model's parameters. A common way to combine symbolic knowledge with neural learning is through knowledge compilation, which converts logical constraints into arithmetic circuits (see Section 4.3.3). These circuits enable efficient WMC evaluation and are differentiable, allowing gradient-based optimisation via backpropagation [Dar03].

However, as discussed in Section 4.4.1, a key limitation of this framework lies in the computational complexity of WMC, which is #*P*-hard [Val79; Rot96]. Since ACs are compiled from the trace of a model-counting search procedure (Section 4.5.1), the compilation process itself inherits this exponential complexity. Moreover, Maene et al. [MDD24] showed that computing exact gradients is also #*P*-hard. These factors make exact compilation infeasible for large or complex logical constraints, severely restricting the scalability of neuro-symbolic learning in practice.

To address the limitation associated with exact WMC inference, approximate probabilistic inference methods have been proposed to estimate the WMC of logical formulas that are otherwise too complex for exact evaluation. A variety of such approximate methods exist [Gom+07; Cha+14; CMV16; SM19; SGM20; LMY23; DSN24], some of which offer formal guarantees on the approximation quality. However, these approaches typically do not follow a DPLL-based search strategy [DLL62] or are not designed for *weighted* formulas, making them incompatible with arithmetic circuit compilation [HD07]. Consequently, while approximate inference is well studied, the development of approximate methods for AC compilation in a learning context remains limited, posing a significant challenge for learning with complex logical constraints.

In this chapter, we introduce a first approach for learning with such challenging instances. The core idea is to compile arithmetic circuits only *partially*, by replacing specific sub-circuits with *approximate WMC constants*, as illustrated in Figure 5.1. This partial compilation strategy reduces both the size and the compilation time of the circuits, while still allowing for parameter learning. Although we focus here on a particular criterion for constructing partial ACs (namely, the coverage of distributions), the proposed method is flexible and can be adapted to other circuit properties (e.g., depth, number of explored nodes, ...).

Approximating sub-circuits naturally introduces deviations between the gradients computed through the partial circuit and those obtained from a fully compiled one. It seems, therefore, intuitive that the quality of these approximations will influence the learning dynamics. To enable effective training with partial circuits, it is desirable to establish guarantees on the quality of the approximated gradients, ensuring they remain sufficiently close to the exact ones to support consistent learning with respect to the underlying knowledge.

After reviewing related work on approximate inference in neuro-symbolic learning and on existing approximate WMC methods, this chapter introduces a first setting for studying learning with partial ACs, alongside a concrete partial compilation approach. We then derive a mathematical bound on the gradient error for the specific setting based on ε-approximations for constant nodes computation. In the experimental section, we study several questions concerning the impact of partial compilation and approximation on the learning process:

- Does the approximation quality affect the parameter learning process, as intuitively expected above?
- How does the approximation quality influence the generalisation performance?
- Since approximation enables the inclusion of more complex training instances, at the cost of reduced approximation precision, is it always beneficial to increase the number of training queries as they are more strongly approximated?

These experimental results are followed by a discussion on the limitations of the proposed method. While the simplified setting provides a useful environment for studying the impact of approximation guarantees on learning, it has some restrictions that must be addressed before being able to apply the method in practical settings. Whereas this chapter focuses on a theoretical and experimental analysis of partial compilation and approximation impact in a controlled setup, the next chapter presents a fully

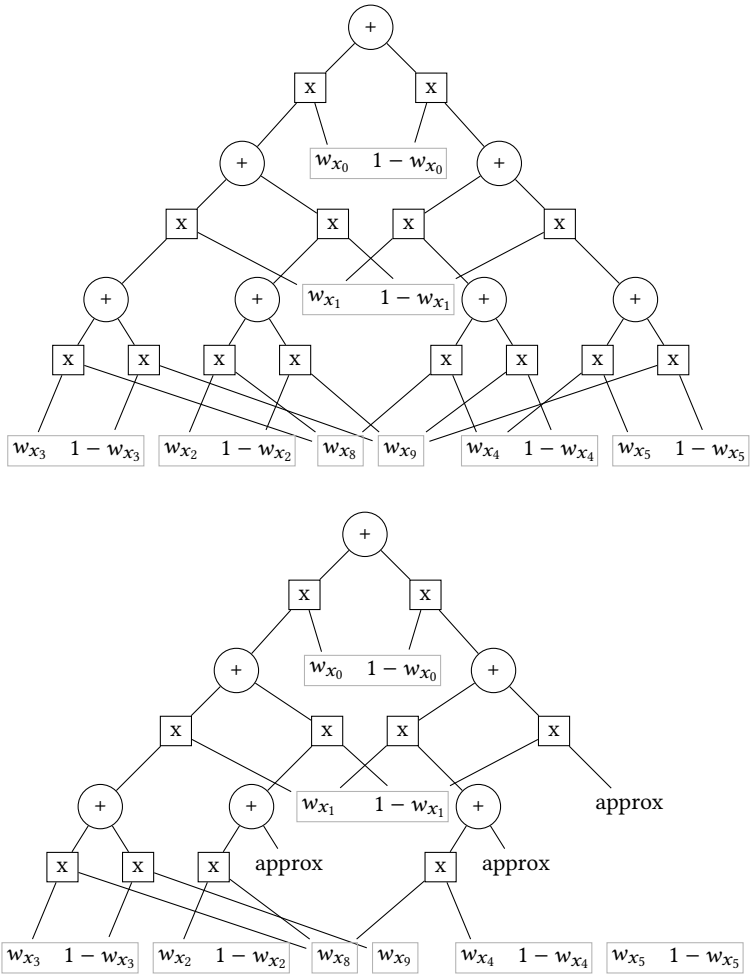


Figure 5.1: Top: Fully compiled arithmetic circuit, with leaf nodes grouped by distribution (grey boxes). Bottom: Corresponding partially compiled circuit, where some subtrees are replaced by constant approximation nodes.

functional learning framework, capable of handling complex constraints in real-world applications.

5.1 Related Work

This section outlines how the computational hardness of weighted model counting inference and arithmetic circuit compilation has been identified as an important challenge in neuro-symbolic learning. It also presents existing approximate WMC methods, which can be used within our proposed framework for compiling partial ACs, along with the theoretical guarantees they provide.

5.1.1 Scalability of WMC Inference and Compilation

Studies have recognised the computational cost of WMC inference and AC compilation as limiting factors in the scalability of neuro-symbolic learning systems.

In [Man+21b], the authors highlight scalability as a central challenge, noting that: *"One challenge to overcome is the scalability of neuro-symbolic systems. Especially the inference of systems that have a separate logical inference step can be prohibitively expensive."*

Similarly, [Mar+24] emphasizes the importance of scalable inference for NeSy approaches: *"Scalable inference is a major challenge for StarAI and therefore also for NeSy approaches with an explicit logical or probabilistic reasoning component. Investigating to what extent neural methods can help with this challenge by means of lifted (exploiting symmetries in models) or approximate inference, as well as reasoning from intermediate representations, are promising future research directions."*

The challenge is also discussed in the work of the semantic loss [Xu+18], where the authors resort to compiling ACs for relatively big constraints and reflect on ways to handle even more complex knowledge: *"An interesting direction for future work is to come up with effective approximations of semantic loss, for settings where even the methods we have described are not sufficient. There are several potential ways to proceed with this, including hierarchical abstractions, relaxations of the constraints, or projections on random subsets of variables."*

While few works have explored this important aspect of learning with complex constraints, recent efforts have proposed alternative strategies for handling them, in parallel with our own work. These include [MMD21; Hua+21; Kri+23; MDD24; Ver+24], which will be discussed in the next chapter alongside our own practical method for learning in such settings. However, these methods do not provide guarantees on the gradient error

(except for [MDD24] discussed in the next chapter), and none of them analyse how the quality of the approximation affects the learning process, which is the main focus of this chapter.

5.1.2 Approximate Weighted Model Counters

To support learning with complex constraints, we propose a method for compiling partial arithmetic circuits, in which selected subtrees are replaced by constant nodes approximating the corresponding subformula (as shown in Figure 5.1). These approximations can be computed using existing approximate WMC methods. Below, we summarise the main categories of such methods and the guarantees they provide.

While traditional model counters are designed for exact inference, numerous approximate WMC algorithms have been developed, each offering specific formal guarantees. We distinguish approximation methods giving no formal guarantees on the approximation, methods giving (ε, δ)-approximations, and, as a special case, methods providing ε-approximations (equivalent to $(\varepsilon, 0)$-approximations).

The first category includes methods such as sampling-based ones (SampleCount [Gom+07], PartialKC [LMY23]) that offer empirical convergence over time but no formal error bounds. SampleCount works by iteratively reducing the solution space by two through variable assignments and sampling, until the solution space is small enough to perform exact inference. The method then scales the result based on the reduction path. PartialKC is also based on sampling, but computes model counts for unweighted formulas.

The second category consists of methods that provide (ε, δ)-guarantees such as `WeightMC` [Cha+14] and `ApproxMC` [CMV16; SM19; SGM20]. `ApproxMC` targets unweighted formulas, while `WeightMC` extends this principle to weighted settings. These methods offer stochastic, *probably approximately correct* (PAC) guarantees: with probability of at least $1-\delta$, the computed approximation $\tilde{P}_{\mathcal{W}}[F]$ of formula F evaluated on literal weights $\mathcal{W}$ lies within a bound, defined by ε, of the exact count $P^*_{\mathcal{W}}[F]$. Formally, they ensure that:

$$P\left(\frac{P^*_{\mathcal{W}}[F]}{1+\varepsilon} \leq \tilde{P}_{\mathcal{W}}[F] \leq (1+\varepsilon)P^*_{\mathcal{W}}[F]\right) \geq 1-\delta$$

The final category includes methods offering deterministic, ε-approximations, i.e., approximations that always lie within the bound specified by ε of the true value, with no probabilistic component (equivalent to $(\varepsilon, 0)$-approximations). The only known method in this category, to our knowledge, is `Schlandals` [DSN24], which provides hard guarantees on

approximation bounds and is, therefore, particularly suited to our goal of bounding gradient errors during learning.

5.2 Partial Compilation

The compilation of arithmetic circuits belongs to the class of #*P*-hard problems, which limits the complexity of queries and formulas that can be used in neuro-symbolic learning using reasonable computing resources. To overcome this limitation, we seek methods that allow inference over complex queries in an approximate manner. Moreover, methods that provide guarantees on the quality of the computed gradient can contribute to ensuring consistent parameter learning.

In this section, we propose a first method for learning with *partial ACs*, which can be compiled within reasonable space and time constraints. These partial circuits make use of approximate weighted model counting methods to substitute parts of the AC with constant-valued nodes. We also analyse how the quality of the approximation influences the gradient computation and the learning process.

We begin by introducing a simplified setting, where the parameters used for the formula's probabilities are separated into trainable and non-trainable sets. We then describe how partial ACs are constructed in this setting, and finally present a theoretical bound on the gradient error when ε-guarantees are used for the approximation.

5.2.1 Simplified Setting

As a first step towards learning from complex queries, we introduce a simplified learning scenario where input parameters are divided into two disjoint sets: those to be learned (*trainable*) and those assumed to be known (*non-trainable*). The trainable parameters form the set $\mathcal{D}ist_T$, while the non-trainable parameters form the set $\mathcal{D}ist_{NT}$. These parameters correspond to the inputs of the logical formula, i.e., the outputs a neural network would normally produce in a full neuro-symbolic setting, and should not be confused with the neural model's internal weights.

In this new setup, the ACs used for gradient computation only need to include the trainable parameters, since not all parameters need to be learned anymore. Consequently, a full compilation over all parameters is unnecessary. Subparts of the circuit that depend solely on non-trainable parameters can safely be replaced by constant nodes: although their total weight contributes to the gradient of the trainable parameters, their internal structure is irrelevant.

The motivation behind this simplified setting is to test whether meaningful learning behaviour can already be achieved under partial compilation. If so, the approach could be generalised further to a general case, where all parameters are considered trainable. In practice, a training dataset involves many training queries, and the set of trainable parameters could be varied across them instead of being identical for all queries. It would then be possible to cover the learning of all parameters globally while only partially compiling each individual AC.

Returning to the full neuro-symbolic setting defined in the previous chapter, this split can be interpreted differently depending on the task (see Section 4.1). In reliability analysis, for instance, the reliability of certain connections could be known, while others must be learned. Similarly, in the alarm problem, the probabilities of some events can be known while the others are still to be learned.

In contrast, in applications such as the digit addition problem, all the probabilities are predicted by the same neural network, which maps images to digit probabilities. In this case, it would be unrealistic to assume that some digit probabilities are known while others are not. For such cases, all parameters must be considered as trainable at a high level, while considering some of them as non-trainable at a sample level. In that way, some image predictions would be considered as "not-learnable truths" while actually being dependent on the neural network weights update. This adaptation to cover the whole set of parameters is further discussed in Section 5.4 after having described the compilation method, the theoretical guarantees, and the experimental results.

5.2.2 Creating Partially Compiled Arithmetic Circuits

The core idea for creating a *partially* compiled AC from trainable and non-trainable sets of parameters is that, to compute gradients with respect to the trainable parameters, it is sufficient to compile only those parts of the AC that involve them. Subtrees that involve only non-trainable parameters can be regrouped and approximated by constant nodes without affecting the gradient computation for the parameters of interest.

A naive approach for building such an AC is illustrated in Figure 5.2 and proceeds as follows:

- Compile a complete AC for the formula F involving all parameters;
- Fill in all non-trainable parameters;
- Simplify the resulting AC, such that leaves are only either trainable parameters or constants regrouping the contribution of a subtree of non-trainable parameters;

Algorithm 3 DPLL-style Approximate Weighted Model Counter

Require: A Boolean formula F over variables $\mathcal{X}$ and literal weights $\mathcal{W}$
Require: The sets $\mathcal{D}ist_T$ of trainable and $\mathcal{D}ist_{NT}$ of non-trainable variables
Ensure: $\tilde{P}_{\mathcal{W}}[F]$ the approximated WMC of F given weights $\mathcal{W}$

1: **procedure** Approx_WMC$_{\mathcal{W}}(F)$
2: **if** $|\mathcal{X}| = 0$ **then return** 1
3: **if** $\mathcal{X} \cap \mathcal{D}_T = \emptyset$ **then return** approx(F)
4: **if** $F \mapsto p$ in cache **then return** p
5: $\tilde{P}_{\mathcal{W}}[F] \leftarrow 0$
6: $x \leftarrow$ choose_variable()
7: **for** $v \in \{\top, \bot\}$ **do**
8: **if** $F[x = v] = \bot$ **then continue** ▷ Applies propagation
9: $C \leftarrow$ components of $F[x = v]$ ▷ Independent components
10: **for** $F_c \in C$ **do** $p^c \leftarrow$ Approx_WMC$_{\mathcal{W}}(F_c)$
11: $prop \leftarrow$ literals forced to $\top$ by BUP
12: $p_{prop} \leftarrow \prod_{l \in prop} w_l$ ▷ Weights of forced literals
13: $\tilde{P}_{\mathcal{W}}[F]$ += $w_{x=v} \times p_{prop} \times \prod_{c \in C} p^c$
14: Add $F \mapsto \tilde{P}_{\mathcal{W}}[F]$ in cache
15: **return** $\tilde{P}_{\mathcal{W}}[F]$

- Use the resulting simplified AC for gradient computation.

However, this naive approach still requires a complete and exact AC compilation in the first step, which is what we wish to avoid. Instead, we propose a more efficient method that directly constructs the same kind of partial AC using approximate weighted model counters.

For this purpose, we modify the classical DPLL-style search-based WMC algorithm (presented in Algorithm 1 of the previous chapter) to stop recursive decomposition when only non-trainable parameters remain. This modified version, described in Algorithm 3, is very close to the original one: The approximate probability $\tilde{P}_{\mathcal{W}}[F]$, allowing to construct the partial circuit using the cache, is computed recursively. First, a variable x is selected to branch on (line 6), and its possible values are explored (lines 7-13). For each assignment to x, boolean unit propagation is applied (line 8), and if the resulting formula is not unsatisfiable, the residual formula is decomposed into independent components (line 9) that are solved recursively (line 10).

The important difference lies in line 3: when there are no more variables whose weight must be learned (i.e., only variables from $\mathcal{D}_{NT}$ remain in the formula), then we stop the search for that formula, and determine a constant that is put in the AC. A second core idea is that to calculate this constant,

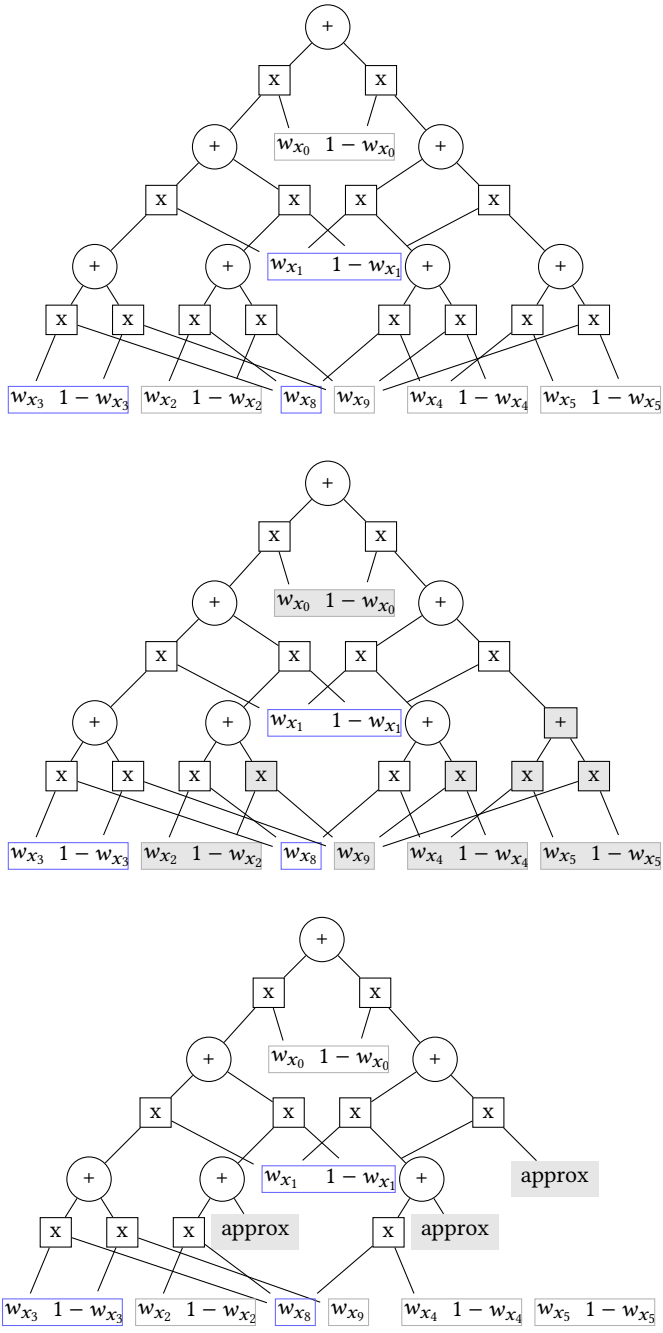


Figure 5.2: Naive approach to obtain a partially compiled AC, given that $Dist_1$, $Dist_3$, and $Dist_8$ are trainable parameters (blue frames) while $Dist_0$, $Dist_2$, $Dist_4$, $Dist_5$, and $Dist_9$ are non-trainable (grey frames). From a fully compiled AC (top), the non-trainable components are identified (centre, in grey), and replaced with constants (bottom, in grey).

we can also use any *approximate* model counter `Approx-WMC` from the literature to calculate an approximation $\tilde{P}_{\mathcal{W}}[F]$. If an exact counter is used, this value equals the true count $P^*_{\mathcal{W}}[F]$, but any approximate WMC method may be employed, including those offering formal guarantees.

5.2.3 Approximate Gradient

Partial compilation allows the use of more complex knowledge during training by reducing compilation complexity. However, since the constant nodes in a partial AC approximate the subcircuits they replace, this may introduce discrepancies between the computed and true gradients obtained from a fully compiled circuit. It is therefore important to analyse how such approximations affect gradient quality.

We prove that, under certain assumptions on $\tilde{P}_{\mathcal{W}}[F]$, one can obtain guarantees on the quality of the computed gradients. Specifically, if the partial circuit computes $\tilde{P}_{\mathcal{W}}[F]$ such that it is an ε–approximation of the true WMC $P^*_{\mathcal{W}}[F]$, i.e., if it holds that:

$$\frac{P^*_{\mathcal{W}}[F]}{1+\varepsilon} \leq \tilde{P}_{\mathcal{W}}[F] \leq P^*_{\mathcal{W}}[F](1+\varepsilon),$$

then, Theorem 1 provides a bound on the error distance between the approximate gradient $\partial\tilde{P}_{\mathcal{W}}[F]/\partial w$ and the exact gradient $\partial P^*_{\mathcal{W}}[F]/\partial w$. The larger the ε, the larger the distance from the true gradient can be.

Theorem 1. Let $\tilde{P}_{\mathcal{W}}[F]$ represent $P^*_{\mathcal{W}}[F]$ calculated by approximating subformulas of F over non-trainable parameters with error rate ε. Then, for any trainable weight w:

$$\left|\frac{\partial\tilde{P}_{\mathcal{W}}[F]}{\partial w}\right| \leq \left|\frac{\partial P^*_{\mathcal{W}}[F]}{\partial w}\right| + \varepsilon\left(\frac{3}{4}+\frac{1}{2}\varepsilon\right)$$

Before proving Theorem 1, we introduce the following intermediate result, which provide a procedural way for ensuring an ε-bounded approximation at node (n) in an arithmetic circuit.

Let us first introduce the notation $F^{(n)}$ that denotes the formula of the sub-problem rooted at node n of an AC.

Theorem 2. Let (n) be a node in an arithmetic circuit and $(c_1), \ldots, (c_k)$ be its children. If it holds that $\forall i = 1, \ldots k$,

$$\begin{cases} \frac{P^*_{\mathcal{W}}[F^{(c_i)}]}{1+\varepsilon} \leq \tilde{P}_{\mathcal{W}}[F^{(c_i)}] \leq P^*_{\mathcal{W}}[F^{(c_i)}](1+\varepsilon) & \text{if } n \text{ is a } \texttt{sum} \text{ node} \\ \frac{P^*_{\mathcal{W}}[F^{(c_i)}]}{\sqrt[k]{1+\varepsilon}} \leq \tilde{P}_{\mathcal{W}}[F^{(c_i)}] \leq P^*_{\mathcal{W}}[F^{(c_i)}]\sqrt[k]{1+\varepsilon} & \text{if } n \text{ is a } \texttt{product} \text{ node} \end{cases}$$

Then, the approximation at node (n) satisfies the following ε-bounded guarantee:

$$\frac{P^*_{\mathcal{W}}[F^{(n)}]}{1+\varepsilon} \leq \tilde{P}_{\mathcal{W}}[F^{(n)}] \leq P^*_{\mathcal{W}}[F^{(n)}](1+\varepsilon).$$

Proof. Let us consider the two types of nodes separately

Case (n) is a `sum` node:
Let $v_1, \ldots, v_k$ be the values of the distribution being branched on (line 6 of Algorithm 3) at sum node n, and let $P(v_i)$ be the probability associated with value v_i (with $P(v_i) \in [0,1]\ \forall i = 1, \ldots, k$).
Then we can compute $P^*_{\mathcal{W}}\left[F^{(n)}\right]$ and $\tilde{P}_{\mathcal{W}}\left[F^{(n)}\right]$ as

$$P^*_{\mathcal{W}}\left[F^{(n)}\right] = \sum_{i=1}^{k}\left(P(v_i)\cdot P^*_{\mathcal{W}}\left[F^{(c_i)}\right]\right) \text{ and } \tilde{P}_{\mathcal{W}}[F^{(n)}] = \sum_{i=1}^{k}\left(P(v_i)\cdot \tilde{P}_{\mathcal{W}}[F^{(c_i)}]\right)$$

We can now derive an upper-bound on $\tilde{P}_{\mathcal{W}}[F^{(n)}]$:

$$\begin{aligned}\tilde{P}_{\mathcal{W}}[F^{(n)}] = \sum_{i=1}^{k}\left(P(v_i)\cdot \tilde{P}_{\mathcal{W}}[F^{(c_i)}]\right) &\leq \sum_{i=1}^{k}\left(P(v_i)P^*_{\mathcal{W}}[F^{(c_i)}](1+\varepsilon)\right)\\ &= (1+\varepsilon)\sum_{i=1}^{k}\left(P(v_i)P^*_{\mathcal{W}}[F^{(c_i)}]\right) = (1+\varepsilon)P^*_{\mathcal{W}}[F^{(n)}]\end{aligned}$$

Similarly, a lower bound on $\tilde{P}_{\mathcal{W}}[F^{(n)}]$ is given by

$$\tilde{P}_{\mathcal{W}}[F^{(n)}] = \sum_{i=1}^{k}\left(P(v_i)\cdot \tilde{P}_{\mathcal{W}}[F^{(c_i)}]\right) \geq \sum_{i=1}^{k}\left(P(v_i)\cdot \frac{P^*_{\mathcal{W}}[F^{(c_i)}]}{1+\varepsilon}\right) = \frac{1}{1+\varepsilon}P^*_{\mathcal{W}}[F^{(n)}]$$

Case (n) is an `product` node:
We can compute $P^*_{\mathcal{W}}[F^{(n)}]$ and $\tilde{P}_{\mathcal{W}}[F^{(n)}]$ as

$$P^*_{\mathcal{W}}[F^{(n)}] = \prod_{i=1}^{k} P^*_{\mathcal{W}}[F^{(c_i)}] \text{ and } \tilde{P}_{\mathcal{W}}[F^{(n)}] = \prod_{i=1}^{k} \tilde{P}_{\mathcal{W}}[F^{(c_i)}]$$

An upper bound for $\tilde{P}_{\mathcal{W}}[F^{(n)}]$ is

$$\begin{aligned}\tilde{P}_{\mathcal{W}}[F^{(n)}] = \prod_{i=1}^{k} \tilde{P}_{\mathcal{W}}[F^{(c_i)}] \leq \prod_{i=1}^{k}\left(P^*_{\mathcal{W}}[F^{(c_i)}]\cdot \sqrt[k]{1+\varepsilon}\right) &= (1+\varepsilon)\prod_{i=1}^{k} P^*_{\mathcal{W}}[F^{(c_i)}]\\ &= (1+\varepsilon)\cdot P^*_{\mathcal{W}}[F^{(n)}]\end{aligned}$$

Similarly, a lower bound is given by

$$\tilde{P}_{\mathcal{W}}[F^{(n)}] = \prod_{i=1}^{k} \tilde{P}_{\mathcal{W}}[F^{(c_i)}] \geq \prod_{i=1}^{k} \frac{P^*_{\mathcal{W}}[F^{(c_i)}]}{\sqrt[k]{1+\varepsilon}} = \frac{1}{1+\varepsilon}P^*_{\mathcal{W}}[F^{(n)}]$$

□

Having established a procedural way of ensuring an ε-bounded circuit in Theorem 2, we can now prove Theorem 1.

Theorem 1 states that, when using ε-approximations, the error on the gradient of the approximated probability is bounded by

$$\left|\frac{\partial \tilde{P}_{\mathcal{W}}[F]}{\partial w}\right| \leq \left|\frac{\partial P^{*}_{\mathcal{W}}[F]}{\partial w}\right| + \varepsilon\left(\frac{3}{4} + \frac{1}{2}\varepsilon\right)$$

The proof of Theorem 1, provided below, assumes, without loss of generality, that the partial AC is produced by `Schlandals`, introduced in Section 4.6. As discussed in subsection 4.6.3, one of the main difference between `Schlandals` and the other commonly used compilers lies in the fact that the branching is performed over multi-valued distributions and, therefore, sum nodes can have more than two children. The results and proof, however, can straightforwardly be adapted to classical binary variables-based AC compilation algorithms.

Proof. In a partial arithmetic circuit, the output probability of a node (n) with children $(c_1), ..., (c_k)$ is defined recursively as:

$$\tilde{P}_{\mathcal{W}}[F^{(n)}] = \begin{cases} \sum_{i=1}^{k} \tilde{P}_{\mathcal{W}}[F^{(c_i)}] & \text{if (n) is an } \texttt{sum} \text{ node} \\ \prod_{i=1}^{k} \tilde{P}_{\mathcal{W}}[F^{(c_i)}] & \text{if (n) is an } \texttt{product} \text{ node} \\ P(w) & \text{if (n) is a leaf node with parameter } w \end{cases}$$

For any parameter w from distribution $Dist_i$ present in the partially compiled circuit, we are interested to compute

$$\frac{\partial \tilde{P}_{\mathcal{W}}[F]}{\partial w}$$

The proof on the bound of the approximate gradient will start by developing the expression of the latter, and then by bounding the different terms of the obtained expression.

Approximate Gradient Expression. Assuming that the root node (r) of the circuit is a `sum` node, and without loss of generality, we have:

$$\frac{\partial \tilde{P}_{\mathcal{W}}[F^{(r)}]}{\partial w} = \frac{\partial \sum_{i=1}^{k} \tilde{P}_{\mathcal{W}}[F^{(c_i)}]}{\partial w} = \sum_{i=1}^{k} \frac{\partial \tilde{P}_{\mathcal{W}}[F^{(c_i)}]}{\partial w}$$

with $(c_i), 1 \leq i \leq k$ the children of (r).

Since the partial derivative of a sum term that does not contain w is zero, we can ignore the other children of (r) in the gradient computation.

Hence, let $(c_1), ..., (c_{k'})$ be the children of (r) for which the subcircuit rooted at $c_i, 1 \leq i \leq k'$ contains an input node sending a value of distribution $Dist_i$. The gradient can be expressed as

$$\sum_{i=1}^{k} \frac{\partial \tilde{P}_{\mathcal{W}}[F^{(c_i)}]}{\partial w} = \sum_{i=1}^{k'} \frac{\partial \tilde{P}_{\mathcal{W}}[F^{(c_i)}]}{\partial w} \tag{5.1}$$

As the circuit alternates between `product` and `sum` nodes, the children c_i are `product` nodes and the partial derivatives becomes

$$\sum_{i=1}^{k'} \frac{\partial \tilde{P}_{\mathcal{W}}[F^{(c_i)}]}{\partial w} = \sum_{i=1}^{k'} \frac{\partial \prod_{j=1}^{k_i} \tilde{P}_{\mathcal{W}}[F^{(c_j^i)}]}{\partial w} = \sum_{i=1}^{k'} \sum_{j=1}^{k_i} \left(\frac{\partial \tilde{P}_{\mathcal{W}}[F^{(c_j^i)}]}{\partial w} \prod_{l=1, l\neq j}^{k_i} \tilde{P}_{\mathcal{W}}[F^{(c_l^i)}] \right) \tag{5.2}$$

with $(c_j^i), 1 \leq j \leq k_i$ the children of (c_i).

From Equations 5.1 and 5.2, we can see that the expression of the gradient will only be impacted by the *paths in the circuit that contain the parameters of* $Dist_i$ (from Equation 5.1) and that the contributions of these paths will be *weighted by the values of the other children of the* `product` *nodes* along these paths (from Equation 5.2). Additionally, the gradient expression will contain the *partial derivatives of the softmax operations* that are used to model the distributions consistently (see Section 4.4.2).

Our proof will now derive detailed expressions for the partial derivation of the softmax function and for the contributions of the other children of the `product` nodes along the path, which we will refer to as the *multiplicative factor of the path*.

Concerning the partial derivative of the softmax, we have that

$$\frac{\partial \tilde{P}_{\mathcal{W}}[F^{(c_j^i)}]}{\partial w} = \begin{cases} P(w) \cdot (1 - P(w)) & \text{if } c_j^i = w \\ -P(w) \cdot P(c_j^i) & \text{if } c_j^i \neq w \end{cases} \tag{5.3}$$

where, c_j^i represents the node containing values of distribution $Dist_i$, to which w belongs. Additionally, by abuse of notation, we denote $P(c_j^i)$ as the probability of the parameter in node (c_j^i) to be true.

In the case where root node (r) directly branches on distribution $Dist_i$, the partial derivative of the circuit with respect to a parameter w of $Dist_i$ can be directly computed, using Equation 5.3, together with the fact that the parameters of $Dist_i$ sum up to 1, as

$$\begin{aligned}\frac{\partial \tilde{P}_{\mathcal{W}}[F^{(r)}]}{\partial w} &= P(w)(1-P(w)) \cdot \tilde{P}_{\mathcal{W}}[F|_w] - P(w) \cdot \sum_{\substack{u \in dom(Dist_i) \\ u \neq w}} P(u) \cdot \tilde{P}_{\mathcal{W}}[F|_u] \\ &= P(w) \cdot \left(\sum_{\substack{u \in dom(Dist_i) \\ u \neq w}} P(u) \cdot \tilde{P}_{\mathcal{W}}[F|_w] - \sum_{\substack{u \in dom(Dist_i) \\ u \neq w}} P(u) \cdot \tilde{P}_{\mathcal{W}}[F|_u] \right) \\ &= P(w) \cdot \sum_{\substack{u \in dom(Dist_i) \\ u \neq w}} P(u)(\tilde{P}_{\mathcal{W}}[F|_w] - \tilde{P}_{\mathcal{W}}[F|_u])\end{aligned} \tag{5.4}$$

with $\tilde{P}_{\mathcal{W}}[F|_w]$ being the value of the subcircuit evaluation, after branching on the parameter w of distribution $Dist_i$, or in other words, the constant value containing the evaluation of the subformula that remains to be explored after having branched on w.

Let us now consider the more general case in which the parameter w is found at a lower level than at the root in the compiled circuit. We now have to take into account the contribution of the paths leading to the parameters of $Dist_i$, as identified with Equation 5.2.

Let us define, for any parameter $u \in dom(Dist_i)$, $C_v^i =< n_1, ..., n_m >$ the sequence of nodes on the i-th path from n_1 (being the root (r)) to node n_m, the `sum` node branching on (u) in that path, and C_{Dist_i} the set of all paths going from (r) to any of the parameters of $Dist_i$. Let $\tilde{pv} : C_{Dist_i} \mapsto [0;1]$ be a function that maps a path to what we define as the *multiplicative factor of the path*, evaluated as

$$\tilde{pv}(C_v^i) = \prod_{i=1}^{m} \begin{cases} 1 & \text{if } n_i \text{ is an } \texttt{sum} \text{ node} \\ \prod_{\substack{c \in children(n_i) \\ c \neq n_{i+1}}} \tilde{P}_{\mathcal{W}}[F^{(c)}] & \text{if } n_i \text{ is an } \texttt{product} \text{ node} \end{cases}$$

This multiplicative factor corresponds to the product of all the siblings of the nodes followed in the path, at each `product` node, resulting from the partial derivative of `product` nodes, observed earlier in Equation 5.2. Similarly, $pv(C_v^i)$ is the analogous function in the fully compiled circuit, with no approximation.

Using the result obtained in Equation 5.4 and the expression of the path leading to the parameter we wish to derive, the derivative of the AC circuit with respect to w, which is the expression for which we would like to derive a bound, can thus be expressed as

$$\frac{\partial \tilde{P}_{\mathcal{W}}[F]}{\partial w} = \sum_{C \in C_{Dist_i}} \left(\tilde{pv}(C) P(w) \sum_{\substack{u \in dom(Dist_i) \\ u \neq w}} P(u) (\tilde{P}_{\mathcal{W}}[F|_{C,w}] - \tilde{P}_{\mathcal{W}}[F|_{C,u}]) \right) \tag{5.5}$$

with $P[F|_{C,w}]$ being defined as value of the subcircuit evaluation, after following path C and branching on the parameter w of distribution $Dist_i$.

Having derived the expression of the gradient, we can now proceed to bound it.

Bounding the Gradient Expression. We will now show that this expression of the partial derivative can be bounded as

$$\left| \frac{\partial \tilde{P}_{\mathcal{W}}[F]}{\partial w} \right| \leq \left| \frac{\partial P^*_{\mathcal{W}}[F]}{\partial w} \right| + \varepsilon \left(\frac{3}{4} + \frac{1}{2}\varepsilon \right).$$

To do so, we will focus on the different terms of Equation (5.5) and describe how they can each be bounded. We start by deriving an upper bound for the multiplicative factor of the path $\tilde{pv}(C)$.

We know that $\tilde{pv}(C)$ is a product of the children of the *product* nodes in path C. Additionally, Theorem 2 tells us that at each *product* node, the error bound is reduced to $(1+\varepsilon)^{1/k}$ with k being the number of independent subproblems solved at that node during compilation. Therefore, if our currently considered path is composed of m *product* nodes n_i each solving k_i independent subproblems, we have that the approximation upper bound for $\tilde{pv}(C)$ is of the form

$$\begin{aligned}
\tilde{pv}(C_v^i) &\leq \prod_{i=1}^{m} \prod_{\substack{c \in children(n_i) \\ c \neq n_{i+1}}} \left(P_{\mathcal{W}}[F^{(c)}] \cdot (1+\varepsilon)^{\prod_{j=1}^{i} \frac{1}{k_j}} \right) \\
&= pv(C_v^i) \prod_{i=1}^{m} \prod_{\substack{c \in children(n_i) \\ c \neq n_{i+1}}} (1+\varepsilon)^{\frac{1}{\prod_{j=1}^{i} k_j}} \\
&= pv(C_v^i) \prod_{i=1}^{m} (1+\varepsilon)^{\frac{k_i - 1}{\prod_{j=1}^{i} k_j}} = pv(C_v^i)(1+\varepsilon)^{\sum_{i=1}^{m} \frac{k_i - 1}{\prod_{j=1}^{i} k_j}} \\
&= pv(C_v^i)(1+\varepsilon)^{\sum_{i=1}^{m} \left(\frac{k_i}{\prod_{j=1}^{i} k_j} - \frac{1}{\prod_{j=1}^{i} k_j} \right)} = pv(C_v^i)(1+\varepsilon)^{\sum_{i=1}^{m} \left(\frac{1}{\prod_{j=1}^{i-1} k_j} - \frac{1}{\prod_{j=1}^{i} k_j} \right)} \\
&= pv(C_v^i)(1+\varepsilon)^{1 - \frac{1}{\prod_{j=1}^{m} k_j}} \quad \text{(using telescopic sum properties)} \\
&\leq pv(C_v^i)(1+\varepsilon)
\end{aligned}$$

With this bound on the multiplicative factor of the path, we can already bound the whole expression of the gradient in Equation 5.5 as

$$\frac{\partial \tilde{P}_{\mathcal{W}}[F]}{\partial w} \leq \sum_{C \in C_{Dist_i}} \left(pv(C)(1+\varepsilon)P(w) \sum_{\substack{u \in dom(Dist_i) \\ u \neq w}} P(u)(\tilde{P}_{\mathcal{W}}[F|_{C,w}] - \tilde{P}_{\mathcal{W}}[F|_{C,u}]) \right) \tag{5.6}$$

Let us consider the case in which $\forall u \in dom(Dist_i) : P_{\mathcal{W}}[F|_{C,w}] \geq P_{\mathcal{W}}[F|_{C,u}]$:

Let us remember that

$$\frac{P^*_{\mathcal{W}}[F|_{C,w}]}{1+\varepsilon} \leq \tilde{P}_{\mathcal{W}}[F|_{C,w}] \leq P^*_{\mathcal{W}}[F|_{C,w}](1+\varepsilon) \tag{5.7}$$

Let us define AC as $\frac{\partial P^*_{\mathcal{W}}[F]}{\partial w}$ and $\tilde{AC}$ its equivalent using an approximate circuit. We can then express an upper bound for $|\tilde{AC}|$. Using equation (5.6) and the inequalities (5.7) we have

$$\begin{aligned} |\tilde{AC}| &\leq \left| \sum_{C \in C_{Dist_i}} \left(pv(C)(1+\varepsilon)P(w) \sum_{\substack{u \in dom(Dist_i) \\ u \neq w}} P(u) \left(\tilde{P}_{\mathcal{W}}[F|_{C,w}] - \tilde{P}_{\mathcal{W}}[F|_{C,u}] \right) \right) \right| \\ &\leq \left| \sum_{C \in C_{Dist_i}} \left(pv(C)(1+\varepsilon)P(w) \sum_{\substack{u \in dom(Dist_i) \\ u \neq w}} P(u) \left(P^*_{\mathcal{W}}[F|_{C,w}](1+\varepsilon) - \frac{P^*_{\mathcal{W}}[F|_{C,u}]}{1+\varepsilon} \right) \right) \right| \end{aligned}$$

By distributing $(1+\varepsilon)$ in the second sum and by adding $-P^*_{\mathcal{W}}[F|_{C,u}] + P^*_{\mathcal{W}}[F|_{C,u}]$ (hence not changing the total value) we have

$$\begin{aligned} |\tilde{AC}| \leq \Bigg| \sum_{C \in C_{Dist_i}} \Bigg(& pv(C)(1+\varepsilon)P(w) \cdot \\ & \sum_{\substack{u \in dom(Dist_i) \\ u \neq w}} P(u) \left(P^*_{\mathcal{W}}[F|_{C,w}] + \varepsilon P^*_{\mathcal{W}}[F|_{C,w}] - P^*_{\mathcal{W}}[F|_{C,u}] + \frac{(1+\varepsilon)-1}{1+\varepsilon} P^*_{\mathcal{W}}[F|_{C,u}] \right) \Bigg) \Bigg| \\ = \Bigg| \sum_{C \in C_{Dist_i}} \Bigg(& pv(C)(1+\varepsilon)P(w) \cdot \\ & \sum_{\substack{u \in dom(Dist_i) \\ u \neq w}} P(u) \left(P^*_{\mathcal{W}}[F|_{C,w}] - P^*_{\mathcal{W}}[F|_{C,u}] + \varepsilon \left(P^*_{\mathcal{W}}[F|_{C,w}] + \frac{P^*_{\mathcal{W}}[F|_{C,u}]}{1+\varepsilon} \right) \right) \Bigg) \Bigg| \end{aligned}$$

Each probability is always bounded by 1, in particular, $P^*_{\mathcal{W}}[F|_{C,w}]$ and $P^*_{\mathcal{W}}[F|_{C,u}]$ can be replaced by 1 in the upper bound

$$
\begin{aligned}
|\tilde{AC}| \leq & \left| \sum_{C \in C_{Dist_i}} \left(pv(C)(1+\varepsilon)P(w) \cdot \right. \right. \\
& \left. \left(\sum_{\substack{u \in dom(Dist_i) \\ u \neq w}} P(u) \left(P^*_{\mathcal{W}}[F|_{C,w}] - P^*_{\mathcal{W}}[F|_{C,u}] + \varepsilon \left(1 + \frac{1}{1+\varepsilon} \right) \right) \right) \right| \\
= & \left| \sum_{C \in C_{Dist_i}} pv(C)(1+\varepsilon)P(w) \left(\sum_{\substack{u \in dom(Dist_i) \\ u \neq w}} P(u)(P^*_{\mathcal{W}}[F|_{C,w}] - P^*_{\mathcal{W}}[F|_{C,u}]) \right) \right| \\
& + \left| \sum_{C \in C_{Dist_i}} pv(C)(1+\varepsilon)P(w) \left(\sum_{\substack{u \in dom(Dist_i) \\ u \neq w}} P(u)\varepsilon \left(\frac{2+\varepsilon}{1+\varepsilon} \right) \right) \right|
\end{aligned}
$$

We observe that the first sum can be rewritten as

$$
\begin{aligned}
& \sum_{C \in C_{Dist_i}} pv(C)(1+\varepsilon)P(w) \left(\sum_{\substack{u \in dom(Dist_i) \\ u \neq w}} P(u)(P^*_{\mathcal{W}}[F|_{C,w}] - P^*_{\mathcal{W}}[F|_{C,u}]) \right) \\
= & \sum_{C \in C_{Dist_i}} pv(C)P(w) \left(\sum_{\substack{u \in dom(Dist_i) \\ u \neq w}} P(u)(P^*_{\mathcal{W}}[F|_{C,w}] - P^*_{\mathcal{W}}[F|_{C,u}]) \right) \\
+ & \sum_{C \in C_{Dist_i}} pv(C)\varepsilon P(w) \left(\sum_{\substack{u \in dom(Dist_i) \\ u \neq w}} P(u)(P^*_{\mathcal{W}}[F|_{C,w}] - P^*_{\mathcal{W}}[F|_{C,u}]) \right)
\end{aligned}
$$

Remark that the first term corresponds to Equation (5.5) in the non-partial case. Therefore, $\tilde{AC}$ is bounded by

$$
\begin{aligned}
|\tilde{AC}| \leq & \left| \left(AC + \sum_{C \in C_{Dist_i}} pv(C)\varepsilon P(w) \sum_{\substack{u \in dom(Dist_i) \\ u \neq w}} P(u)(P^*_{\mathcal{W}}[F|_{C,w}] - P^*_{\mathcal{W}}[F|_{C,u}]) \right) \right| \\
& + \left| \left(\sum_{C \in C_{Dist_i}} pv(C)(1+\varepsilon)P(w) \sum_{\substack{u \in dom(Dist_i) \\ u \neq w}} P(u)\varepsilon \left(\frac{2+\varepsilon}{1+\varepsilon} \right) \right) \right|
\end{aligned}
$$

As we are in the case where $\forall u \in dom(Dist_i) : P_{\mathcal{W}}[F|_{C,w}] \geq P_{\mathcal{W}}[F|_{C,u}]$, the biggest difference that we can have between $P_{\mathcal{W}}[F|_{C,w}]$ and $P_{\mathcal{W}}[F|_{C,u}]$ is 1:

$$|\tilde{AC}| \leq \left| \left(AC + \sum_{C \in C_{Dist_i}} pv(C)\varepsilon P(w) \sum_{\substack{u \in dom(Dist_i) \\ u \neq w}} P(u) \right. \right.$$
$$\left. \left. + \sum_{C \in C_{Dist_i}} pv(C)(1+\varepsilon)P(w) \sum_{\substack{u \in dom(Dist_i) \\ u \neq w}} P(u)\varepsilon \left(\frac{2+\varepsilon}{1+\varepsilon} \right) \right) \right|$$

Since $\sum_{u \in dom(Dist_i)} P(u) = 1$, we have that $\sum_{\substack{u \in dom(Dist_i) \\ u \neq w}} P(u) = 1 - P(w)$. It follows that

$$|\tilde{AC}| \leq \left| \left(AC + \sum_{C \in C_{Dist_i}} pv(C)\varepsilon P(w)(1-P(w)) + pv(C)(1+\varepsilon)\varepsilon(1-P(w)) \frac{2+\varepsilon}{1+\varepsilon} \right) \right|$$

The highest value that $P(w)(1-P(w))$ can reach is when $P(w)$ is 0.5, which gives a value of 0.25 to the product. Hence,

$$|\tilde{AC}| \leq \left| \left(AC + \sum_{C \in C_{Dist_i}} pv(C)\frac{1}{4}\varepsilon + pv(C)(1+\varepsilon)\varepsilon \frac{1}{4} \frac{2+\varepsilon}{1+\varepsilon} \right) \right|$$

Given that $0 \leq \varepsilon$, the highest value that $\frac{2+\varepsilon}{1+\varepsilon}$ can take is when $\varepsilon = 0$, which gives an upper bound of

$$|\tilde{AC}| \leq \left| \left(AC + \sum_{C \in C_{Dist_i}} pv(C)\frac{1}{4}\varepsilon + pv(C)(1+\varepsilon)\varepsilon \frac{1}{4} 2 \right) \right| = |AC| + \sum_{C \in C_{Dist_i}} pv(C)\varepsilon \left(\frac{3}{4} + \frac{\varepsilon}{2} \right)$$

Each path C from the root to a leaf in the circuit can be seen as a partial interpretation of the formula, with a probability p_C equal to the product of the decisions on the path. As p_C is a subproduct of $pv(C)$ and the additional terms in $pv(C)$ are less than 1, then $pv(C) \leq p_C$. In addition to that, the sum of all interpretations' probability of F is no greater than 1. Hence,

$$\sum_{C \in C_{Dist_i}} pv(C) \leq \sum_{C \in C_{Dist_i}} p_C \leq 1$$

Which finally gives

$$\left| \frac{\partial \tilde{P}_{\mathcal{W}}[F]}{\partial w} \right| \leq \left| \frac{\partial P^*_{\mathcal{W}}[F]}{\partial w} \right| + \varepsilon \left(\frac{3}{4} + \frac{1}{2}\varepsilon \right)$$

The same formula can be derived from the cases in which $\forall u \in dom(Dist_i) : P_{\mathcal{W}}[F|_{C,w}] \geq P_{\mathcal{W}}[F|_{C,u}]$ does not hold following the same reasoning. □

5.3 Experimental Results

This section addresses the questions raised in the introduction of this chapter, concerning the impact of using approximations within a learning framework. We first present the datasets and training protocol, and then discuss the following questions:

(I) What is the impact of the partial compilation on convergence?

(II) How well does the approximate learning process generalise to unseen examples?

(III) How does the number of queries used for training affect the learning?

5.3.1 Datasets

Our experimental setting follows the one described in Section 4.4.2 of the previous chapter, for which the aim is to learn the parameters of a set of distributions $\mathcal{D}ist$ from a set of training queries. Two types of probabilistic queries are considered in our experiments: marginal probabilities computation in Bayesian Networks, and reliability estimation in probabilistic graphs.

The Bayesian networks are sourced from the bnlearn R package [Scu09], which offers a variety of networks with different sizes and structures[1]. For each Bayesian network of the repository, a dataset is created by generating queries for each network variable N and each of its possible values v, asking for the marginal probability $P[N = v]$ without evidence.

For reliability estimation, we compute the probability that two nodes are connected in a graph where each edge has an associated probability of being functional/present. Two families of datasets are used for this problem. First, we use graphs extracted from the GridKit tool [Med+17; Wie16], representing high-voltage transmission networks in Europe and the USA. One dataset is created per country (Europe) or state (USA). As these graphs are undirected, the instances are created by randomly selecting source-target pairs until all nodes are ensured to be in at least one query. By doing so, we ensure that there is a possibility of learning all distributions in the networks (even if, in practice, not all queries can be compiled). Second, we use water supply networks coming from [Kli+17]. For these directed graphs, a query is created for each pair consisting of a source node (a node without a parent) and a sink node (a node without children), excluding queries in which no path exists between the nodes.

[1] https://www.bnlearn.com/bnrepository/

5.3.2 Training Protocol

To learn from the generated queries and study the impact of the approximation quality, they must first be compiled into partial ACs under various approximation guarantees. Once the partial ACs are created, we then conduct learning on these circuits. The following paragraphs describe the practical details of these steps in detail.

Partial Compilation. We construct partial circuits using a modified version of `Schlandals` [DSN23], which has been extended[2] to store its search trace and insert approximation nodes, as described in Algorithm 3. This partial compilation approach is not specific to the `Schlandals` compilation and may be applied to other DPLL-based ACs compilation methods.

Approximation Guarantees. For the partial compilation of ACs, we consider (ε, δ)-approximations, including the special case of $(\varepsilon, 0)$-approximations. Methods without formal guarantees are not considered, as they can be interpreted as a special case of $(\varepsilon, \delta)-$guarantees with a large ε.

An aspect that requires careful consideration is that the approximation guarantees provided by the partial compilation of ACs have to apply to the *root node's value*. To ensure this, the ε-bound must be adjusted for each approximated node along the circuit path. Specifically, Theorem 2 establishes how the ε-bound should be adjusted when encountering sum and product nodes. Particularly, it shows that, for each product node on the path from the root to an approximation node, the local ε-bound must be tightened by a factor of $\sqrt[k]{\varepsilon}$, where k is the node's branching factor. Some constant approximated nodes might thus be produced using a different ε-bound than the desired one for the root value. This scaling accounts for the amplification of error introduced by the product operation. All our experiments take into account the adapted ε value required at each approximation node to ensure the correct ε-bound at the root.

To focus the experiments on the impact of the approximation guarantees on the learning process, we do not use any specific existing approximation algorithm for the partial compilation. Instead, the value of each approximation node is sampled uniformly from the valid (ε, δ) interval, after first computing its exact value (i.e., a $(0, 0)$-approximation). This prevents the experiments from being biased by the intrinsic working of a specific approximation method (e.g., having a tendency to over- or under-estimate

[2]The modifications are available on `Schlandals` GitHub `https://github.com/aia-uclouvain/schlandals` on the "nesy" branch, as well as on `https://github.com/luciledierckx/Parameter_Learning_Using_Approximate_MC`.

the true count, always performing better than the announced guarantee, etc.).

Training. Each experiment is repeated 10 times, and we report the mean results for a representative subset of the datasets[3]. For training, we assume that 70% of the distributions are known, and the remaining 30% are to be learned. The datasets are limited to the queries that can be compiled in 10 minutes, because we need to compare the learning with the exact compilation setting.

The training uses the Mean Absolute Error (MAE) as a loss, and the optimiser used to update the distribution values is the Stochastic Gradient Descent (without momentum), as our study here focuses on gradient computation and not on finding the most efficient optimiser. Training runs for a maximum of 6000 epochs, with early stopping triggered if the loss improvement is below 10^{-5} for five consecutive epochs, or if the average loss falls below 10^{-4}. We also impose a one-hour time limit per training run. The learning rate is initially set to 0.3 and is reduced by a factor of 0.75 every 100 epochs.

5.3.3 Results

The following paragraphs present the results of the experiments, answering the questions raised in the introduction of this section.

I) What is the Impact of the Partial Compilation on the Convergence? We first study the convergence of the learning process when using partially compiled ACs. Table 5.1 reports the average final training loss after convergence across different (ε, δ) configurations. The baseline, $(\varepsilon, \delta) = (0, 0)$, corresponds to the fully compiled ACs behaviour. As expected, looser approximation guarantees (i.e., larger ε and δ values) lead to higher final losses. Notably, the loss tends to increase approximately linearly with ε, for all values of δ. For all values of δ, there is a greater increase in final loss.

II) How well does the Approximate Learning Process Generalise to Unseen Examples? To assess the generalization capability of the proposed method, we perform training on 80% of the queries and evaluate on the remaining 20%. This experiment is only done using the Bayesian networks as they contain multiple queries per variable (for each possible

[3]The behaviors reported generalise across the other datasets, but only a subset is included for conciseness.

Table 5.1: Average training MAE loss ± its standard deviation at convergence over 10 runs for three different datasets, using various approximation guarantees.

		$\varepsilon = 0.0$ (exact)	$\varepsilon = 0.05$	$\varepsilon = 0.3$	$\varepsilon = 0.8$
Munin	$\delta = 0.0$	0.036 ± 0.0	0.037 ± 1.7e-04	0.042 ± 1.2e-03	0.052 ± 2.6e-03
	$\delta = 0.2$		0.090 ± 9.1e-03	0.095 ± 8.9e-03	0.110 ± 9.7e-03
	$\delta = 0.5$		0.170 ± 0.010	0.170 ± 0.010	0.180 ± 0.010
	$\delta = 0.8$		0.250 ± 0.012	0.260 ± 0.012	0.270 ± 0.011
Tennessee	$\delta = 0.0$	9.1e-04 ± 1.1e-19	5.0e-03 ± 1.8e-03	0.018 ± 9.5e-03	0.036 ± 0.018
	$\delta = 0.2$		0.015 ± 0.015	0.027 ± 0.023	0.044 ± 0.031
	$\delta = 0.5$		0.066 ± 0.046	0.079 ± 0.049	0.093 ± 0.050
	$\delta = 0.8$		0.077 ± 0.037	0.091 ± 0.037	0.110 ± 0.036
Net3	$\delta = 0.0$	8.0e-04 ± 1.1e-19	2.5e-03 ± 3.5e-04	8.5e-03 ± 2.8e-03	0.017 ± 5.3e-03
	$\delta = 0.2$		8.1e-03 ± 5.9e-03	0.015 ± 6.9e-03	0.023 ± 8.7e-03
	$\delta = 0.5$		0.019 ± 0.013	0.025 ± 0.014	0.032 ± 0.014
	$\delta = 0.8$		0.035 ± 0.017	0.042 ± 0.016	0.049 ± 0.014

Table 5.2: The MAE loss computed on a test set *before* and *after* the learning of the distributions on a training set. The values are presented as: *before* → *after*.

		$\varepsilon = 0.0$ (exact)	$\varepsilon = 0.05$	$\varepsilon = 0.3$	$\varepsilon = 0.8$
Munin	$\delta = 0.0$	0.150 → 0.094	0.160 → 0.096	0.160 → 0.099	0.170 → 0.110
	$\delta = 0.2$		0.200 → 0.130	0.200 → 0.130	0.210 → 0.140
	$\delta = 0.5$		0.260 → 0.190	0.260 → 0.190	0.270 → 0.200
	$\delta = 0.8$		0.330 → 0.260	0.340 → 0.260	0.340 → 0.270
Hepar2	$\delta = 0.0$	0.051 → 5.0e-04	0.053 → 3.0e-03	0.061 → 0.012	0.072 → 0.025
	$\delta = 0.2$		0.088 → 0.062	0.096 → 0.070	0.110 → 0.082
	$\delta = 0.5$		0.160 → 0.130	0.170 → 0.140	0.170 → 0.150
	$\delta = 0.8$		0.220 → 0.210	0.230 → 0.210	0.240 → 0.220
Water	$\delta = 0.0$	0.160 → 0.024	0.160 → 0.024	0.160 → 0.025	0.160 → 0.025
	$\delta = 0.2$		0.190 → 0.080	0.190 → 0.081	0.190 → 0.100
	$\delta = 0.5$		0.250 → 0.200	0.250 → 0.200	0.250 → 0.200
	$\delta = 0.8$		0.320 → 0.310	0.330 → 0.310	0.330 → 0.320

value), which makes it feasible to split them while ensuring that we do not perform testing on distributions that were never seen during the training.

Table 5.2 shows the average MAE *on the test set* before and after having learned the model's distributions *on the train set.* Overall, the results indicate that the ε parameter has a limited impact on the generalization of the learning process, while the δ parameter has a more pronounced effect. Larger values of δ introduce additional randomness, hindering convergence and generalisation.

III) How does the Number of Queries Used for Training Affect the Learning? Finally, we explore whether increasing the number (and therefore complexity) of partially compiled queries used impacts the learn-

ing. Using the largest Bayesian network, `Munin`, we evaluate test set generalisation performance when training on a fraction of the training set. We learned the distribution for $\varepsilon = 0.0, 0.1, 0.5$, and 0.8 on, respectively, 40%, 60%, 80%, and 100% of the train dataset, selecting the queries producing the smallest ACs first.

With $\varepsilon = 0.0$ and considering only the easiest 40% instances of the training set, the learning fails to generalise, and *the test MAE* increases from 0.15 before the training to 0.17 after the model has been trained. This can be explained by the fact that only a small subset of the distributions is covered by these easiest instances, making it impossible to generalise well. Increasing ε to 0.1 and using 60% of the training set yields a small improvement in the test MAE that reduces from 0.16 before training to 0.15 after.

Stronger approximations improve generalisation more efficiently. For $\varepsilon = 0.5$ and $\varepsilon = 0.8$, the average test loss decreases from 0.16 to 0.1 and from 0.17 to 0.11 respectively. Interestingly, the learning process seems to generalise similarly for both values. This suggests that a trade-off exists: excessive approximation reduces gradient quality and offsets the benefits of including more queries. This indicates that, for a given dataset, there must be an optimal value for ε that balances query coverage and gradient reliability. Increasing ε beyond this point does not improve, and may even degrade generalisation performance.

5.4 Current Limitations

The proposed method enables to learn parameters of complex probabilistic queries, even when full compilation into arithmetic circuits is infeasible. However, it has limitations that currently restrict its applicability in practical learning settings, and that need to be addressed in future work.

The most significant limitation is that, in the current state of the method, only a subset of parameters is learned, while the remaining ones are assumed to be known. Ideally, the method should allow for learning the full set of parameters.

In theory, this limitation could be easily addressed by defining for each query its own partition of trainable and non-trainable distributions, rather than using a unique partition shared across all queries. This would ensure that each distribution appears as to be learned in a portion of the training ACs, enabling all parameters to be learned while keeping the individual ACs small. In such a scenario, the parameters considered as known in some partial ACs would actually be updated through their inclusion in other circuits. Consequently, the approximation values of the constant nodes, which depend on the fixed distribution values, may become outdated.

However, it is not straightforward to implement this idea in practice. The current partial compilation procedure relies on classical variable branching heuristics (e.g., MinInDegree, in which the first constrained distribution from the clause with the smallest number of constrained variables is selected), which does not prioritise the choice of trainable parameters over non-trainable ones. As a result, the partial AC for a given query might still require considerable depth before all trainable parameters are included, particularly if the selected trainable parameters are not strategically chosen for each query.

One possible solution would be to adapt the branching heuristic to prioritise the branching on trainable parameters, ensuring that they appear early in the partial compilation process. This would reduce the depth of the partial AC and facilitate the learning of all parameters. The main challenge with this approach is that, as noted in Section 4.5.2, the size of the AC is highly sensitive to the variable ordering [Bry86; BW96; DKT07; Vla+14]. Forcing the compiler to branch on specific variables can thus significantly increase the size and width of the circuit, which in turn raises the number of approximation nodes, increases compilation time, and may lead to memory issues. Designing a branching strategy or parameter partitioning that ensures good coverage of all trainable parameters while maintaining reasonable AC sizes is a research question that needs to be addressed.

A related issue is the validity of the constant approximation nodes when parameters are shared across queries. If the supposedly fixed parameters in a query are in fact being updated through their inclusion in other queries, the corresponding approximation nodes may become irrelevant. To maintain the correctness of the approximations and the learning, it might be necessary to periodically re-evaluate the constant nodes to reflect the updated values of the parameters. This could be done at each training epoch, at fixed intervals, or after a certain criterion is met, but doing so would increase the training time of the method.

5.5 Conclusion

In this chapter, we introduced a first approach for learning the parameters of probability distributions using partial arithmetic circuit compilation. Explored within a simplified setting, this method shows potential for enabling parameter learning even when full compilation of the underlying circuit is computationally infeasible. Our analysis highlights that the choice of the approximation level and guarantees significantly impact both convergence and generalisation capabilities of the learning process. Importantly, the use of approximate compilation allows a broader range of

training instances to be utilised, while still providing theoretical guarantees on the gradient error under ε-bounded conditions.

This work opens several paths for future research. While the current approximate inference and compilation method is still a simple extension of a DPLL-style search, enhancing this technique could allow it to handle more complex queries and improve both the approximation quality and the learning performance. Additionally, extending the gradient error bounds to incorporate the probabilistic failure parameter, i.e., to include δ for (ε, δ)-approximations methods, remains an aspect to be investigated.

An important limitation of the current method is that not all parameters are learned: a fixed subset is assumed to be known. Addressing this limitation is crucial for applying the method in practice. One promising direction is to define a separate set of trainable and non-trainable parameters for each query, allowing full parameter coverage while maintaining manageable circuit sizes. However, this strategy introduces challenges related to variable ordering and branching heuristics, as naïvely prioritising trainable parameters can lead to large arithmetic circuits and compilation issues.

In the next chapter, we address these limitations by presenting a more practical learning method that removes the need for a fixed split between trainable and non-trainable parameters, while still ensuring provable bounds on the gradient error.

Learning from Lower- and Upper-Bound Arithmetic Circuits

6

The previous chapter introduced a first approach for learning with partial arithmetic circuits while providing guarantees on the gradient error. However, as discussed in Section 5.4, that method has several limitations which currently hinder its use in practical settings. In this chapter, we present a new learning framework, called *Lower- and Upper-Bound ACs (LUBAC)*. LUBAC builds on a more widely used method for partial compilation and extends it so that it also provides deterministic guarantees on the gradient error. Doing so, LUBAC retains the ability to bound the gradient error, but without some of the practical restrictions of the method introduced in the previous chapter, such as the need for a fixed split between trainable and non-trainable parameters.

In a learning setting, when symbolic constraints become large and complex, a commonly adopted strategy is to stop the compilation process before its end, following a criterion such as a timeout, memory limit, minimal probability mass, or fixed number of explored models [MMD21; Hua+21; VDS23; Ver+24]. In this case, only part of the search space is explored, so the interrupted weighted model count of the logical formula corresponds to a *lower-bound of the exact WMC* in practice [Vla+16]. Partial compilation of this form often provides useful approximations of the true WMC and enables scaling to larger problems. However, as we will show, it has an important drawback in the context of learning. Indeed existing approaches based on partial compilation do not provide guarantees on the gradients derived from such approximations. In this chapter, we demonstrate that this can lead to gradients that diverge substantially from the true ones, which is an undesirable property in a gradient-based learning context.

To overcome this weakness, this chapter introduces the *Lower- and Upper-Bound ACs* (LUBAC) framework[1]. The core novelty in this framework

[1] Available on `Schlandals` GitHub `https://github.com/aia-uclouvain/schlandals` on the "LDS_learn" branch, as well as on `https://github.com/luciledierckx/LUBAC_IJCAI2025`.

is that we propose to *compile two ACs per query*:

- a *lower-bound* AC, representing the classical subset of models obtained through limited compilation, and
- a complementary *upper-bound* AC, which is a novel circuit encoding the subset of non-satisfying assignments (also referred to as *non-models* later) encountered during the compilation process.

We show that the overhead of constructing both circuits in a modified WMC solver is negligible in theory and practice, while together, these two ACs contain significantly more information than either alone. This dual representation enables us to derive a new form of approximate gradient, along with an *optimality gap* that quantifies the difference between the true and approximated gradients, hence bounding the error on the approximate gradient. In addition, we demonstrate how careful selection of lower- and upper-bound circuits can yield smaller ACs, further reducing complexity. Finally, we show how the resulting circuits can be seamlessly integrated into existing neuro-symbolic learning pipelines.

This chapter is structured as follows. Section 6.1 reviews existing solvers for lower-bound approximation and anytime approximate inference, discusses prior neuro-symbolic learning approaches based on partial compilation or approximate inference, and highlights specific properties of the `Schlandals` solver (introduced in Section 4.6) that are particularly relevant to our framework. Section 6.2 then describes how to compile both lower- and upper-bound ACs by modifying a classical WMC solver. Building on this, Section 6.3 demonstrates how relying on a single lower-bound AC can produce misleading gradients, and how using both lower- and upper-bound ACs allows us to derive a new form of approximate gradients. In this section, we also derive a bound on the error of this new gradient, discuss how to integrate both ACs into a learning setting, and present the limitations of the LUBAC framework. Section 6.4 presents an empirical evaluation of our framework, demonstrating its practical applicability and confirming our theoretical findings. Finally, Section 6.5 outlines some open questions and future research directions, before concluding in Section 6.6.

6.1 Related Work

This section reviews existing solvers for lower-bound and anytime approximate inference, along with the prior neuro-symbolic learning approaches that have explored learning with partial circuits or approximate inference. Finally, the section highlights specific properties of the `Schlandals` solver, previously introduced for its basic features in Section 4.6, that are particularly relevant to our framework.

6.1.1 (Learning from) Approximate Inference

As discussed in Section 5.1.1, exact WMC solvers such as those in Section 4.5.1 are often impractical for complex queries, as compiling a full arithmetic circuit can be prohibitively expensive. Section 5.1.2 reviewed several approximate WMC solvers offering different guarantees, such as empirical convergence, stochastic guarantees ((ϵ, δ)-approximation), or deterministic guarantees (ϵ-approximation). However, not all are compatible with compilation, and thus, many cannot be directly used in learning settings where the WMC is generally compiled into an AC.

In practice, many learning approaches rely on *anytime methods* that can be interrupted after a timeout of other criteria and return a partially compiled circuit. Some stop after enumerating a fixed number k of models, producing a lower-bound on the true WMC. This idea of considering only a subset of models was already explored in ProbLog [DKT07], where query probabilities were approximated by retaining only high-contribution models. Kimmig et al. [Kim+08] proposed using the k-best models, and Renkens et al. [RVN12] introduced k-optimal models, maximising total probability while avoiding overlap.

This subset-based idea inspired several approximate learning frameworks. In DeepProbLog, Manhaeve et al. introduced DPLA* [MMD21], an A*-based search guided by heuristics to find models with high explanation probability. Scallop [Hua+21] similarly trains on the k most likely models. However, these methods provide no formal guarantees on their approximations or on the gradients derived from them.

As an alternative to partial compilation, some approaches rely on sampling-based methods to extract a subset of the models. ExAL [VDS23; Ver+24] trains on a sampled subset of models with a diversity criterion to improve coverage. WeightMe [MDD24] employs approximate weighted model sampling with (ϵ, δ)-guarantees, ensuring unbiased WMC estimates. Other approaches, such as A-NeSI [Kri+23], replace explicit compilation with neural approximations of query probabilities.

Despite the scalability advantages of these different methods, only WeightME [MDD24] provides stochastic (ϵ, δ)-guarantees on the gradients it computes, but deterministic guarantees, which could be obtained from partial circuit compilation, remain lacking.

The following sections introduce the LUBAC framework, which addresses the approximate gradient limitations related to partial compilation. By exploiting not only the subset of satisfying models (to build a lower-bound circuit) but also the subset of non-satisfying assignments (to construct an upper-bound circuit), LUBAC enables the derivation of new approximate gradients together with provable deterministic bounds on their error.

Our approach makes use of the previously introduced `Schlandals` weighted model counter [DSN23], which, to our knowledge, is the only WMC solver offering deterministic ϵ-bounded lower- and upper-bounds on the true WMC [DSN24]. The specific properties of `Schlandals` relevant to our framework are discussed next.

6.1.2 Schlandals for Partial Compilation

`Schlandals` serves as the foundation for our proposed framework, LUBAC. As discussed in Section 4.6, `Schlandals` is a search-based WMC solver whose features make it particularly well suited for our approach.

In addition to the capabilities described earlier, `Schlandals` supports anytime approximate WMC, providing $(\epsilon, 0)$-guarantees on its approximations. It also implements the Limited Discrepancy Search (LDS) heuristic, which guides the search toward high-weight models while maintaining diversity.

These two features are especially valuable for our framework, as they enable efficient compilation of both lower- and upper-bound arithmetic circuits while keeping the circuit size manageable. The following paragraphs describe these properties in more detail.

Anytime Approximated Weighted Model Counting. `Schlandals` allows to perform anytime approximation when an exact WMC cannot be found within a given time limit [DSN24]. If the solver does not determine the exact WMC before the specified timeout, it will instead provide upper- and lower-bounds on the true probability it aims to infer. At each step of the search, `Schlandals` computes the weights of the discovered satisfying assignments (`models`), to increase the current lower-bound, and the weights of the discovered non-satisfying assignments (referred to as *non-models*) to lower the upper-bound.

If the search times out before finding an exact solution $P^*_{\mathcal{W}}[F]$, the solver outputs a lower-bound probability $P^{lb}_{\mathcal{W}}[F]$ as well as an upper-bound $P^{ub}_{\mathcal{W}}[F]$. These two bounds represent the partial model count and one minus the partial non-model count, respectively, such that:

$$P^{lb}_{\mathcal{W}}[F] \leq P^*_{\mathcal{W}}[F] \leq P^{ub}_{\mathcal{W}}[F].$$

From these two bounds, the approximated WMC value, $\tilde{P}_{\mathcal{W}}[F]$, is computed as:

$$\tilde{P}_{\mathcal{W}}[F] = \sqrt{P^{lb}_{\mathcal{W}}[F].P^{ub}_{\mathcal{W}}[F]}.$$

The computed bounds make it possible to quantify the approximation

degree ϵ of the inference, computed as:

$$\epsilon = \sqrt{\frac{P^{ub}_{\mathcal{W}}[F]}{P^{lb}_{\mathcal{W}}[F]}} - 1,$$

This approximation degree ensures that the estimated probability $\tilde{P}_{\mathcal{W}}[F]$ for the inference is bounded by

$$\frac{P^{*}_{\mathcal{W}}[F]}{1+\epsilon} \leq \tilde{P}_{\mathcal{W}}[F] \leq P^{*}_{\mathcal{W}}[F](1+\epsilon),$$

where $P^{*}_{\mathcal{W}}[F]$ is the true probability. This corresponds to an $(\epsilon, 0)$-approximation of the true WMC following the categories of approximation defined in Section 5.1.2.

Limited Discrepancy Search. As outlined before, `Schlandals` supports anytime search, allowing the process to stop before the full search tree is explored. Several heuristics can guide this partial exploration. By default, `Schlandals` uses a depth-first search (DFS) strategy that always follows the most probable assignment first (i.e., the leftmost branch in the search tree shown at the top of Figure 6.1). It also implements the Limited Discrepancy Search (LDS) heuristic [HG95], as described in [DSN24], illustrated at the bottom of Figure 6.1.

LDS is an incremental search strategy that prioritises the high-weight interpretations while still maintaining diversity among the explored interpretations. The main intuition behind LDS is that the branching heuristic defining the search order is generally effective, and that, therefore, if a model is not found by following this order, it is likely reachable by deviating from it a limited number of times. These controlled deviations promote diversity in the explored interpretations.

Compared to a pure DFS strategy, which may quickly focus on a region of high-weight interpretations but then spend a long time overexploring similar ones with low additional contribution to the overall WMC, LDS broadens exploration early while still ensuring that the most promising branches are explored first. Concretely, as shown in Figure 6.1, LDS starts with the leftmost path ($d = 0$), corresponding to always selecting the most probable assignment, and progressively increases the discrepancy value d to allow more deviations and explore a wider portion of the search space.

For our framework, this property is particularly valuable: since LUBAC relies on constructing both lower- and upper-bound ACs, efficiently identifying high-weight interpretations while maintaining controlled diversification ensures that the compiled circuits are informative yet manageable in size.

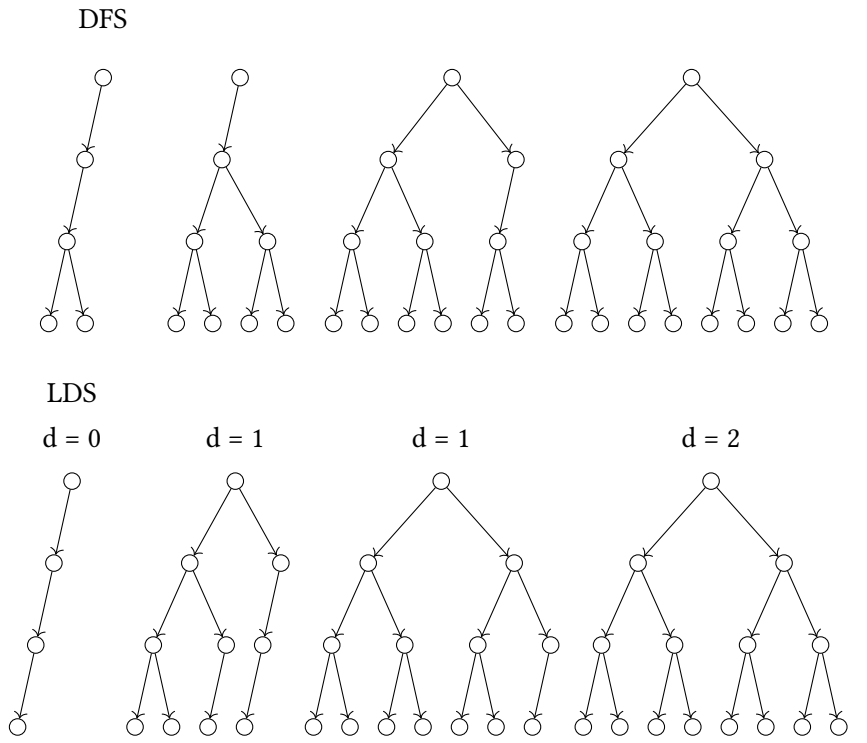


Figure 6.1: DFS (top) and LDS (bottom) search strategies. The leftmost branch corresponds to always selecting the most probable assignment at each branching point. The discrepancy value d in LDS indicates the maximum number of deviations allowed from the leftmost option along any root-to-leaf path.

6.2 Compiling Lower- and Upper-Bound Arithmetic Circuits

In this section, we describe how to compile the upper- and lower-bound circuits used in our LUBAC framework. We first briefly recall the principles of standard arithmetic circuit compilation before introducing our approach for jointly constructing both circuits.

6.2.1 From Exhaustive Search to Arithmetic Circuit

We have seen in Section 4.5.2 and with Algorithm 1 that classical DPLL-style model counters calculate $\mathrm{WMC}_{\mathcal{W}}(F)$ by selecting a variable $x \in \mathcal{X}$, exploring both truth values, propagating the assignments using *Boolean Unit Propagation* (BUP), and recursively exploring the residual formulas.

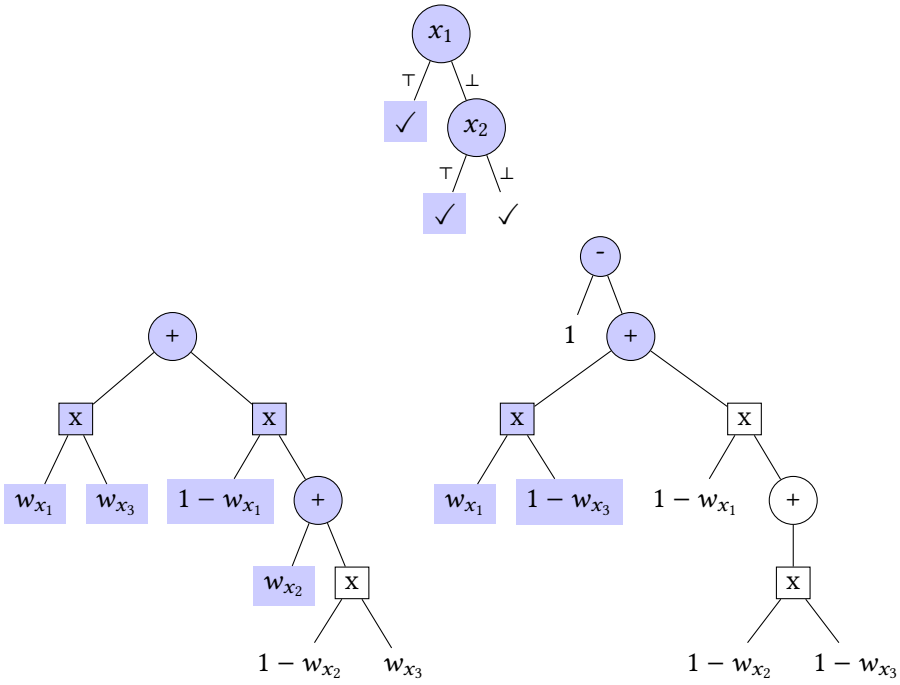


Figure 6.2: Top: A search tree for running Example 4.2.1, $F = (\neg x_1 \vee x_3) \wedge (x_2 \vee x_3)$**. Leaves corresponding to a model are marked with ✓. Bottom left: An AC computing** $\mathrm{WMC}_{\mathcal{W}}(F)$**, corresponding to the polynomial** $w_{x_1} w_{x_3} + (1 - \omega_{x_1})(w_{x_2} + (1 - w_{x_2}) w_{x_3})$**.Bottom right: An AC computing the complement of the non-satisfying assignments' WMC, corresponding to the polynomial** $1 - (w_{x_1}(1 - w_{x_3}) + (1 - w_{x_1})(1 - w_{x_2})(1 - w_{x_3}))$**. In Blue: Matching parts between the ACs and the search tree.**

> **Example 6.2.1.** (Search Tree and AC for WMC)
> Figure 6.2 shows, at the top, a search tree for the formula of our simple running Example 4.2.1 and, at the bottom left, an arithmetic circuit computing $\mathrm{WMC}_{\mathcal{W}}(F)$, as derived by a traditional compiler.

To be efficient, such counters integrate a caching system: when the count of a sub-formula F' is computed, it is stored that $C[F'] \mapsto \mathrm{WMC}_{\mathcal{W}}(F')$. Briefly, classical search-based compilers such as `dSharp` or `d4` build a d-DNNF diagram by replacing the sum with disjunction and the product with conjunction nodes. Accordingly, the cache does not store the solved sub-formula's WMC but the root of their d-DNNF diagram.

A key observation for our work is that such algorithms can easily be modified to halt during their execution, for instance, when a timeout is reached. In such a case, we can still compile a partial d-DNNF representing

models encountered until that moment, providing a circuit from which a lower-bound on the true WMC can be calculated.

> **Example 6.2.2.** (Partial Compilation)
> Figure 6.2 illustrates this idea: if we interrupt the search after considering $x_2 = \top$, we have considered the blue part of the search tree, from which we can still compile the blue part of the exact AC, at the bottom left. This partial AC computes a lower-bound on the true WMC, as it only considers a subset of all models.

However, existing compilers are not yet designed to also produce an upper-bound AC. Next, we discuss a new caching system that allows compiling both lower- and upper-bound (partial) circuits.

6.2.2 Partial Compilation as a Post-Processing Step

While Section 4.6.1 described how exact compilation is performed in `Schlandals` and other DPLL-based solvers, we now explain how this process can be adapted to partial compilation. This adaptation yields a lower-bound AC and, in turn, can be extended to also produce an upper-bound AC.

Exact Compilation Reminder. As a reminder, `Schlandals` stores additional information in its cache during the search process, which can then be post-processed to construct arithmetic circuits.

For the exact compilation, let F be a boolean formula, x a variable of F, and $F[x]$ and $F[\neg x]$ the two sub-formulas obtained by branching on x. We denote $\text{prop}_x^\top$ (resp. $\text{prop}_{\neg x}^\top$) the set of variables (except x) set to $\top$ when applying BUP with $x = \top$ (resp. $x = \bot$). Hence, the cache entry for a complete compilation has the following structure:

$$C[F] \mapsto \left(x, (F[x], \text{prop}_x^\top), (F[\neg x], \text{prop}_{\neg x}^\top)\right) \tag{6.1}$$

Here, the $\text{prop}^\top$ sets are recording the interpretations built alongside a branch.

> **Example 6.2.3.** (Cache Entry)
> The cache entry for the root node of the search tree in Figure 6.2 would be as follows:
>
> $$(x_1, (\top, [x_3]), (x_2 \vee x_3, [\ ]))$$

Algorithm 4 Partial Compilation Algorithm from a Search Trace

Require: Cache C with entries as defined by Equation (6.1)
Ensure: An AC computing a lower-bound of the $\mathrm{WMC}_{\mathcal{W}}(F)$

```
1: procedure PartialCompile(F, C)
2:     N⁺ = new sum node
3:     (x, (F[x], prop_x^⊤), (F[¬x], prop_¬x^⊤)) = C[F]
4:     for l ∈ {x, ¬x} do
5:         if F[l] = † then skip l
6:         N× = new product node
7:         add w_l as child of N×
8:         ∀x' ∈ prop_l^⊤ add w_x' as child of N×
9:         add PartialCompile(F[l], C) as child of N×
10:        add N× as child of N⁺
11:    return N⁺
```

Partial Compilation Modifications. When the search is partial (e.g., due to a compilation timeout), unexplored branches are replaced in the cache by a special marker †. This marker indicates that the corresponding branch has not been explored, and hence, no information is available about the models or propagations that would have been discovered in that branch.

> **Example 6.2.4.** (Partial Cache Entry)
> Continuing the previous example, if the search is interrupted after exploring the left branch of the root node, the cache entry becomes:
>
> $$(x_1, (\top, [x_3]), (\dagger, [\ ]))$$

Algorithm 4, a modified version of Algorithm 2, illustrates how the cache can be parsed to produce a partial, lower-bound AC from the set models discovered during the search. The central difference lies in line 5, where unexplored branches of the search tree are ignored in the compilation process. The rest of the compilation steps remain identical to those explained in Section 4.6.1.

Addition of an Upper-Bound Circuit. To also construct a (partial) circuit representing the non-satisfying assignments, the cache, originally containing information about the satisfying assignments, must be updated to additionally record the details about the non-models encountered during the search. Then, Algorithm 4 can easily be modified to compile an upper-bound AC from this enriched cache.

The main modification occurs in lines 6–8 of Algorithm 4. While for

the lower-bound, compilation creates an AC representation of

$$w_l \times \prod_{x' \in \text{prop}_l^\top} w_{x'},$$

for the upper-bound AC, by contrast, if $\text{prop}_l^\top \neq \emptyset$ we instead add an AC representation of

$$w_l \times (1 - \prod_{x' \in \text{prop}_l^\top} w_{x'})$$

to the parent summation node; and no node if $\text{prop}_l^\top = \emptyset$.

The intuition behind this formula is the following. Since $\prod_{x' \in \text{prop}_l^\top} w_{x'}$ represents the weights of the literal forced to be true by propagation, it means that they *cannot* all be part of a non-satisfying assignment. The weighted sum of the non-satisfying interpretations resulting from propagation is given by $1 - \prod_{x' \in \text{prop}_l^\top} w_{x'}$.

> **Example 6.2.5.** (Upper-Bound Compilation)
> In Figure 6.2, the AC resulting from this compilation on a fully traversed search space is illustrated at the bottom right; in blue is shown the part that is compiled when the search is interrupted before traversing the assignment $x_2 = \bot$. Note that instead of w_{x_3} in the lower-bound circuit, we have $1 - w_{x_3}$ in the upper-bound circuit.

When the formula is fully compiled, both ACs calculate exactly the same model count. However, importantly, when the search is interrupted, both circuits represent different model counts, where one corresponds to a lower-bound and the other to an upper-bound. Having both circuits can be valuable: by calculating a probability using both, we can calculate an *optimality gap* that provides information with respect to how good the approximation is.

6.3 Combining Lower- and Upper-Bounds Arithmetic Circuits for Learning

Having established how lower- and upper-bound circuits can be obtained from a single partial compilation process, we now turn to their use in learning. In this section, we motivate why combining both circuits provides a stronger basis for gradient approximation than relying on lower-bounds alone, and we derive a formal bound on the resulting gradient error. Then, the integration of both ACs into a learning setting is discussed, and the section is concluded with a discussion on the limitations of our framework.

6.3.1 Analysis of Gradients

Let us show that there are scenarios in which learning from only the lower-bound AC produces gradients that do not reflect the true gradient behaviour and that combining both lower- and upper-bound ACs can help mitigate this issue.

Example 6.3.1. (Lower- and Upper-Bounds Gradient Analysis)
Let us use our running Example 4.2.1, $F = (\neg x_1 \vee x_3) \wedge (x_2 \vee x_3)$, with weights $w_{x_1} = 0.99, w_{x_2} = 0.5, w_{x_3} = 0.65$, and consider the two partial circuits, denoted $P^{lb}_{\mathcal{W}}[F]$ and $P^{ub}_{\mathcal{W}}[F]$, highlighted in blue at the bottom left and right of in Figure 6.2. Let us also consider the complete circuit in the bottom left in blue and white, denoted $P^{*}_{\mathcal{W}}[F]$.

These circuits can be expressed as polynomials in the following way:

$$P^{lb}_{\mathcal{W}}[F] = w_{x_1}w_{x_3} + (1 - w_{x_1})w_{x_2} = 0.6485$$
$$P^{ub}_{\mathcal{W}}[F] = 1 - w_{x_1}(1 - w_{x_3}) = 0.6535$$
$$P^{*}_{\mathcal{W}}[F] = w_{x_1}w_{x_3} + (1 - w_{x_1})(w_{x_2} + (1 - w_{x_2})w_{x_3}) = 0.65175$$

Note that, as $P^{*}_{\mathcal{W}}[F]$ is obtained from the exact AC, it is the most accurate form for the WMC, but cannot always be derived within a reasonable time for complex queries.

The gradients w.r.t. parameter w_{x_1} for each circuits are:

$$\partial P^{lb}_{\mathcal{W}}[F]/\partial w_{x_1} = w_{x_3} - w_{x_2} = 0.15$$
$$\partial P^{ub}_{\mathcal{W}}[F]/\partial w_{x_1} = -1 + w_{x_3} = -0.35$$
$$\partial P^{*}_{\mathcal{W}}[F]/\partial w_{x_1} = w_{x_3} - w_{x_2} - (1 - w_{x_2})w_{x_3} = -0.175$$

This example highlights that even when the absolute difference between the lower-bound and the true WMC is small, differentiating only through one partial AC might be insufficient for learning. Specifically, in this case, learning from the lower-bound would push the parameter in the opposite direction of the true gradient. This issue can also apply to using the upper-bound alone. On the other hand, combining the gradients of both bounds through their mean results in a value closer to the true gradient behaviour.

Although the true gradient towards a parameter cannot be computed without a complete AC, the gradients derived from the two partial ACs allow to give an interval in which it lies. Theorem 3 states that the weighted sum of the partial ACs' gradients bounds the true gradient and that the size of the interval depends on the bounds' gap.

Theorem 3. Let F be a Boolean formula in CNF over a set of variables weighted by parameters $\mathcal{W}$. Let $P^*_{\mathcal{W}}[F], P^{lb}_{\mathcal{W}}[F]$, and $P^{ub}_{\mathcal{W}}[F]$ respectively be the exact, lower-, and upper-bound WMC for formula F. The following inequalities hold for every parameter $w \in \mathcal{W}$:

$$(1-w)\frac{\partial P^{lb}_{\mathcal{W}}[F]}{\partial w} + w\frac{\partial P^{ub}_{\mathcal{W}}[F]}{\partial w} - \left(P^{ub}_{\mathcal{W}}[F] - P^{lb}_{\mathcal{W}}[F]\right) \leq \frac{\partial P^*_{\mathcal{W}}[F]}{\partial w} \leq w\frac{\partial P^{lb}_{\mathcal{W}}[F]}{\partial w} + (1-w)\frac{\partial P^{ub}_{\mathcal{W}}[F]}{\partial w} + \left(P^{ub}_{\mathcal{W}}[F] - P^{lb}_{\mathcal{W}}[F]\right) \quad (6.2)$$

Proof. Let us define $P^*[F_{w=0}]$ and $P^*[F_{w=1}]$, the exact model count of query F with variable weight w set to 0 and 1. The same definition is done for $P^{lb}_{\mathcal{W}}[F_{w=0}]$, $P^{lb}_{\mathcal{W}}[F_{w=1}]$, $P^{ub}_{\mathcal{W}}[F_{w=0}]$, and $P^{ub}_{\mathcal{W}}[F_{w=1}]$ for the lower- and upper-bound count.

Given that the polynomial representing a model count is a first-order polynomial, its gradient equals its slope between any two points of its domain. Hence,

$$\frac{\partial P^*_{\mathcal{W}}[F]}{\partial w} = P^*[F_{w=1}] - P^*[F_{w=0}], \quad (6.3)$$

$$\frac{\partial P^{lb}_{\mathcal{W}}[F]}{\partial w} = P^{lb}_{\mathcal{W}}[F_{w=1}] - P^{lb}_{\mathcal{W}}[F_{w=0}], \quad (6.4)$$

and

$$\frac{\partial P^{ub}_{\mathcal{W}}[F]}{\partial w} = P^{ub}_{\mathcal{W}}[F_{w=1}] - P^{ub}_{\mathcal{W}}[F_{w=0}]. \quad (6.5)$$

We can decompose the model count as the weighted sum of all models in which a parameter w is true and of all models in which the parameter is false. This is described by

$$P^*_{\mathcal{W}}[F] = wP^*[F_{w=1}] + (1-w)P^*[F_{w=0}]. \quad (6.6)$$

Similarly, we have

$$P^{lb}_{\mathcal{W}}[F] = wP^{lb}_{\mathcal{W}}[F_{w=1}] + (1-w)P^{lb}_{\mathcal{W}}[F_{w=0}], \quad (6.7)$$

and

$$P^{ub}_{\mathcal{W}}[F] = wP^{ub}_{\mathcal{W}}[F_{w=1}] + (1-w)P^{ub}_{\mathcal{W}}[F_{w=0}]. \quad (6.8)$$

As the lower bound (resp. upper bound) count is a count that, by definition, covers a subset of all models (resp. non-models), we have that

$$P^{lb}_{\mathcal{W}}[F] \leq P^{*}_{\mathcal{W}}[F] \leq P^{ub}_{\mathcal{W}}[F], \tag{6.9}$$

$$P^{lb}_{\mathcal{W}}[F_{w=0}] \leq P^{*}[F_{w=0}] \leq P^{ub}_{\mathcal{W}}[F_{w=0}], \tag{6.10}$$

and

$$P^{lb}_{\mathcal{W}}[F_{w=1}] \leq P^{*}[F_{w=1}] \leq P^{ub}_{\mathcal{W}}[F_{w=1}]. \tag{6.11}$$

From Equations 6.10 and 6.11, we can formulate the following bound on the gradient:

$$P^{lb}_{\mathcal{W}}[F_{w=1}] - P^{ub}_{\mathcal{W}}[F_{w=0}] \leq P^{*}[F_{w=1}] - P^{*}[F_{w=0}] \leq P^{ub}_{\mathcal{W}}[F_{w=1}] - P^{lb}_{\mathcal{W}}[F_{w=0}]. \tag{6.12}$$

Using Equation 6.3, this becomes:

$$P^{lb}_{\mathcal{W}}[F_{w=1}] - P^{ub}_{\mathcal{W}}[F_{w=0}] \leq \frac{\partial P^{*}_{\mathcal{W}}[F]}{\partial w} \leq P^{ub}_{\mathcal{W}}[F_{w=1}] - P^{lb}_{\mathcal{W}}[F_{w=0}]. \tag{6.13}$$

By combining Equations 6.4 and 6.7 we obtain

$$P^{lb}_{\mathcal{W}}[F_{w=0}] = P^{lb}_{\mathcal{W}}[F] - w\frac{\partial P^{lb}_{\mathcal{W}}[F]}{\partial w} \tag{6.14}$$

and

$$P^{lb}_{\mathcal{W}}[F_{w=1}] = P^{lb}_{\mathcal{W}}[F] + (1-w)\frac{\partial P^{lb}_{\mathcal{W}}[F]}{\partial w}. \tag{6.15}$$

The same kind of expressions can be derived from Equations 6.5 and 6.8:

$$P^{ub}_{\mathcal{W}}[F_{w=0}] = P^{ub}_{\mathcal{W}}[F] - w\frac{\partial P^{ub}_{\mathcal{W}}[F]}{\partial w} \tag{6.16}$$

and

$$P^{ub}_{\mathcal{W}}[F_{w=1}] = P^{ub}_{\mathcal{W}}[F] + (1-w)\frac{\partial P^{ub}_{\mathcal{W}}[F]}{\partial w}. \tag{6.17}$$

The terms of Equation 6.13 can be replaced by the expressions found

in 6.14, 6.15, 6.16, and 6.17. This results in the inequality

$$(1-w)\frac{\partial P^{lb}_{\mathcal{W}}[F]}{\partial w} + P^{lb}_{\mathcal{W}}[F] - \left(P^{ub}_{\mathcal{W}}[F] - w\frac{\partial P^{ub}_{\mathcal{W}}[F]}{\partial w}\right) \leq \frac{\partial P^{*}_{\mathcal{W}}[F]}{\partial w} \leq (1-w)\frac{\partial P^{ub}_{\mathcal{W}}[F]}{\partial w} + P^{ub}_{\mathcal{W}}[F] - \left(P^{lb}_{\mathcal{W}}[F] - w\frac{\partial P^{lb}_{\mathcal{W}}[F]}{\partial w}\right). \quad (6.18)$$

By rearranging the terms of Equation 6.18, we obtain the expression of the theorem:

$$(1-w)\frac{\partial P^{lb}_{\mathcal{W}}[F]}{\partial w} + w\frac{\partial P^{ub}_{\mathcal{W}}[F]}{\partial w} - \left(P^{ub}_{\mathcal{W}}[F] - P^{lb}_{\mathcal{W}}[F]\right) \leq \frac{\partial P^{*}_{\mathcal{W}}[F]}{\partial w} \leq w\frac{\partial P^{lb}_{\mathcal{W}}[F]}{\partial w} + (1-w)\frac{\partial P^{ub}}{\partial w} + \left(P^{ub}_{\mathcal{W}}[F] - P^{lb}_{\mathcal{W}}[F]\right) \quad (6.19)$$

□

Example 6.3.2. (Lower- and Upper-Bounds Gradient, Continued) Let us continue with our small example, whose bounds are $P^{lb}_{\mathcal{W}}[F] = 0.6485$ and $P^{ub}_{\mathcal{W}}[F] = 0.6535$. Hence, we have that $P^{ub}_{\mathcal{W}}[F] - P^{lb}_{\mathcal{W}}[F] = 0.005$. The true gradient w.r.t. parameter $w_{x_1} = 0.99$ is bounded as follows. The lower-bound is given by

$$0.01 \times 0.15 + (0.99 \times -0.35) - 0.005 = -0.35$$

The upper-bound is given by

$$0.99 \times 0.15 + (0.01 \times -0.35) + 0.005 = 0.15$$

The true gradient, -0.175, is within the bounds.

An interesting property derived from Theorem 3 is that it is possible to bound the distance between the exact gradient and the mean of the lower- and upper-bound gradients. Corollary 1 demonstrates that this bound mainly depends on the two following elements: the bounds' gap and the difference between the partial gradients.

Corollary 1. The distance between the true gradient, $\partial P^*_{\mathcal{W}}[F]/\partial w$, and the mean of the bounds' gradients is bounded as follows:

$$\left|\frac{\partial P^*_{\mathcal{W}}[F]}{\partial w} - \frac{1}{2}\left(\frac{\partial P^{lb}_{\mathcal{W}}[F]}{\partial w} + \frac{\partial P^{ub}_{\mathcal{W}}[F]}{\partial w}\right)\right| \leq \left|(\frac{1}{2} - w)\left(\frac{\partial P^{ub}_{\mathcal{W}}[F]}{\partial w} - \frac{\partial P^{lb}_{\mathcal{W}}[F]}{\partial w}\right) + (P^{ub}_{\mathcal{W}}[F] - P^{lb}_{\mathcal{W}}[F])\right|$$

Proof. Let us first show that the centre of the bounded interval corresponds to the mean of the gradients derived from the bounds. The centre of an interval is half the sum of its two bounds, which in our case is expressed as follows:

$$\begin{aligned}\frac{1}{2}\Bigg((1-w)\frac{\partial P^{lb}_{\mathcal{W}}[F]}{\partial w} + w\frac{\partial P^{ub}_{\mathcal{W}}[F]}{\partial w} - \left(P^{ub}_{\mathcal{W}}[F] - P^{lb}_{\mathcal{W}}[F]\right) \\ + w\frac{\partial P^{lb}_{\mathcal{W}}[F]}{\partial w} + (1-w)\frac{\partial P^{ub}_{\mathcal{W}}[F]}{\partial w} + \left(P^{ub}_{\mathcal{W}}[F] - P^{lb}_{\mathcal{W}}[F]\right)\Bigg) \\ = \frac{1}{2}\left(\frac{\partial P^{lb}_{\mathcal{W}}[F]}{\partial w} + \frac{\partial P^{ub}_{\mathcal{W}}[F]}{\partial w}\right)\end{aligned}$$

Since the absolute maximal error that can be made between a bounded value and the centre of its interval can be expressed as half the difference between the upper and lower bounds of that interval, we have:

$$\begin{aligned}\left|\frac{\partial P^*_{\mathcal{W}}[F]}{\partial w} - \frac{1}{2}\left(\frac{\partial P^{lb}_{\mathcal{W}}[F]}{\partial w} + \frac{\partial P^{ub}_{\mathcal{W}}[F]}{\partial w}\right)\right| \leq \\ \Bigg|\frac{1}{2}\Bigg(w\frac{\partial P^{lb}_{\mathcal{W}}[F]}{\partial w} + (1-w)\frac{\partial P^{ub}_{\mathcal{W}}[F]}{\partial w} + \left(P^{ub}_{\mathcal{W}}[F] - P^{lb}_{\mathcal{W}}[F]\right) \\ - \left((1-w)\frac{\partial P^{lb}_{\mathcal{W}}[F]}{\partial w} + w\frac{\partial P^{ub}_{\mathcal{W}}[F]}{\partial w} - \left(P^{ub}_{\mathcal{W}}[F] - P^{lb}_{\mathcal{W}}[F]\right)\right)\Bigg)\Bigg| \\ = \left|\left(\frac{1}{2} - w\right)\left(\frac{\partial P^{ub}_{\mathcal{W}}[F]}{\partial w} - \frac{\partial P^{lb}_{\mathcal{W}}[F]}{\partial w}\right) + \left(P^{ub}_{\mathcal{W}}[F] - P^{lb}_{\mathcal{W}}[F]\right)\right|\end{aligned}$$

□

This corollary suggests that, if the gap between the bounds and/or the partial gradients is small, the mean of the partial gradients is a good approximation of the true gradient. We can exploit this in a learning algorithm, as discussed next. Note that in our running example, the

approximate gradient would be $(-0.35 + 0.15)/2 = -0.1$, which is closer to the true gradient than either the upper- or lower-bound gradient.

6.3.2 Learning from Lower- and Upper-Bound ACs

Most gradient descent-based learning algorithms iteratively update model parameters in the direction that minimises the loss. Let us first assume that we can determine an exact circuit for calculating a model count, then the loss for a given instance can be expressed as

$$\frac{\partial \mathcal{L}(y, P^*_{\mathcal{W}}[F])}{\partial w} = \frac{\partial \mathcal{L}(y, P^*_{\mathcal{W}}[F])}{\partial P^*_{\mathcal{W}}[F]} \frac{\partial P^*_{\mathcal{W}}[F]}{\partial w}. \tag{6.20}$$

where y is the target probability, $P^*_{\mathcal{W}}[F]$ is the predicted probability derived from the exact model count using the parameter weights $\mathcal{W}$, and $\mathcal{L}(y, \hat{y})$ is a loss function, for instance, $\mathcal{L}(y, \hat{y}) = (y - \hat{y})^2$.

To update the parameters, we need to compute, for every training instance, the gradient of the loss with respect to a parameter w:

$$\frac{\partial \mathcal{L}(y, P^*_{\mathcal{W}}[F])}{\partial w} = \frac{\partial \mathcal{L}(y, P^*_{\mathcal{W}}[F])}{\partial P^*_{\mathcal{W}}[F]} \frac{\partial P^*_{\mathcal{W}}[F]}{\partial w}. \tag{6.21}$$

Let us now assume we do not have an exact circuit for $P^*_{\mathcal{W}}[F]$, then Corollary 1 suggests we can approximate $\frac{\partial P^*_{\mathcal{W}}[F]}{\partial w}$:

$$\frac{1}{2} \frac{\partial \mathcal{L}(y, P^*_{\mathcal{W}}[F])}{\partial P^*_{\mathcal{W}}[F]} \frac{\partial P^{lb}_{\mathcal{W}}[F]}{\partial w} + \frac{1}{2} \frac{\partial \mathcal{L}(y, P^*_{\mathcal{W}}[F])}{\partial P^*_{\mathcal{W}}[F]} \frac{\partial P^{ub}_{\mathcal{W}}[F]}{\partial w}. \tag{6.22}$$

The challenge is that we do not know $P^*_{\mathcal{W}}[F]$; hence we also need to approximate it. One approach to deal with this is as follows. Assuming that both circuits calculate probabilities that are close, then we could assume that both $P^*_{\mathcal{W}}[F] \approx P^{ub}_{\mathcal{W}}[F]$ and $P^*_{\mathcal{W}}[F] \approx P^{lb}_{\mathcal{W}}[F]$; our expression becomes

$$\frac{1}{2} \frac{\partial \mathcal{L}(y, P^{lb}_{\mathcal{W}}[F])}{\partial P^{lb}_{\mathcal{W}}[F]} \frac{\partial P^{lb}_{\mathcal{W}}[F]}{\partial w} + \frac{1}{2} \frac{\partial \mathcal{L}(y, P^{ub}_{\mathcal{W}}[F])}{\partial P^{ub}_{\mathcal{W}}[F]} \frac{\partial P^{ub}_{\mathcal{W}}[F]}{\partial w}, \tag{6.23}$$

which corresponds to

$$\frac{1}{2} \frac{\partial \mathcal{L}(y, P^{lb}_{\mathcal{W}}[F])}{\partial w} + \frac{1}{2} \frac{\partial \mathcal{L}(y, P^{ub}_{\mathcal{W}}[F])}{\partial w}. \tag{6.24}$$

This points to a straightforward solution in which we execute the following steps:

(I) Initialise the weights of the machine learning model;

(II) Compile every query, either with a timeout, or with a constraint on the quality of the resulting circuits, providing lower-bound and upper-bound circuits;

(III) Treat every lower-bound and upper-bound circuit as a separate training instance during gradient descent.

The advantage of this approach is that any existing NeSy implementation calculating gradients from a given AC can be reused, once for every lower-bound and once for every upper-bound AC. Other more complex approaches are also made possible by our framework, which is discussed in Section 6.5.

6.3.3 Limitations of the LUBAC Approach

While the LUBAC framework provides a way to control the gradient approximation quality during the learning under symbolic constraints, it also inherits several limitations from its underlying compilation-based nature.

First, since the method is still based on model compilation, it remains to some extent subject to the inherent computational complexity of the considered problem. For highly complex constrained problems, especially the ones where the search space is large or where the proportion of satisfying assignments is small, obtaining sufficiently tight lower- and upper-bound circuits may still require significant computation time before informative sets of models and non-models can be collected. Because the bound on the gradient error depends partly on the gap between these bounds, reaching a satisfactory approximation level remains of use to obtain trustworthy gradients. This is especially relevant at the early stages of training, when weights are typically initialised around 0.5, in which case the bound on the gradient error depends solely on the difference between the two partially compiled circuits.

Second, although the framework establishes theoretical bounds on the gradient error, these guarantees do not ensure that the direction of the approximate gradient always coincides with that of the true gradient. In practice, this implies that, even if the magnitude of the gradient error is bounded, the optimisation process may still take suboptimal or even opposite steps, potentially slowing convergence or introducing instability during training.

Finally, the current gradient error bounds can be relatively loose for certain parameters, depending on the values of the weights and gradients. There is therefore scope for improving the tightness of these bounds, which could, in turn, provide better insights regarding whether the current

approximation provided by the two partial circuits is adequate for learning or if further compilation would be beneficial.

6.4 Experimental Results

This section describes the solver, datasets, and general protocol used in our experiments. Then, we answer the following questions:

(I) How well does LUBAC learning generalise?

(II) How are the initial gradients impacted by which AC is used?

(III) What is the time overhead to compile both lower- and upper-bound ACs?

6.4.1 Setting

Concerning the solver used, we implemented the caching system and compilation algorithm defined in Section 6.2.2 into the `Schlandals` solver. As described in Section 6.1.2, the latter has two advantages compared to other solvers. First, it natively supports the computation of bounds on the true WMC, making the implementation of the caching system described above relatively easy. As `Schlandals` works directly with the distributions (i.e., non-binary variables), we extended the caching system and compilation algorithm to work with such variables.

The `Schlandals` solver can generate complete or partial ACs from DFS or LDS search heuristics. The classical DFS search can be run until completion or stopped at a timeout. The LDS-based search, by nature, provides a way to compute partial ACs, as each iteration corresponds to a portion of the search space. In our experiments, both search strategies will be used, and the default configuration is using LDS. Additionally, when the DFS is used to produce exact ACs, only the queries that can be fully compiled within 10 minutes are used, excluding the others of the training set. This mimics the usage of a classical compiler in the literature that does not provide approximations or partial compilation.

As our contribution to the literature is the use of two circuits in the training process, our evaluation focuses on assessing the benefits of using these. To isolate the effect of using two circuits, we compare our approach against methods that compile only a single lower- or upper-bound circuit, while keeping all other factors, such as the Bayesian network encodings and the compilation heuristics, constant. For this reason, we exclude methods that rely on different constraint encodings or compilation algorithms, such as [MMD21; MDD24], as these differences would make it harder to isolate the impact of using two circuits versus one.

6.4.2 Datasets

As discussed in Section 4.4.2, we consider a simplified learning setting, compared to the full NeSy setting, in which we learn parameters of a probabilistic model from probabilistic queries, making an abstraction of the neural part. This allows us to focus on the quality of the gradients obtained from the partial ACs, as was also done in [MDD24]. Hence, we use sets of probabilistic queries in our experiments, which originate from Bayesian Networks for which we wish to learn parameters that lead to desired marginal probabilities. These Bayesian Networks are the same as the ones used in Chapter 5, and come from the bnlearn R package [Scu09]. For each network, one dataset is created as follows. For each value v of each network variable N, we create a query $P[N = v]$ (without evidence). These query sets, with their expected marginal probabilities, form each a dataset for parameter learning. This inference task is challenging, providing queries of various difficulties. Hence, it is possible to evaluate the LUBAC framework as some queries cannot be compiled exactly. Our results are presented on *munin* [And+87], *pigs*, and *water* [Jen+89] datasets except for the experiments in Section 6.5, covering learning improvements, in which case only the *munin* dataset is used.

6.4.3 Results

Below we present our experimental results for the LUBAC framework.

I) How well does LUBAC Learning Generalise? We first evaluate the generalisation of LUBAC learning and compare it to classical learning with exact ACs and learning with only one of the partial ACs (either lower- or upper-bound).

Training runs for up to 6000 epochs and up to one hour, with early stopping if the loss improves by less than 10^{-5} over five epochs or falls below 10^{-4}. The learning rate starts at 0.3 and decays by 0.75 every 100 epochs. We use the Mean Absolute Error (MAE) as the loss and Stochastic Gradient Descent without momentum for optimisation. Data are split 80/20 for training and testing per query group, ensuring that test queries involve only distributions observed during training.

Table 6.1 presents the MAE for the considered settings. It shows that exact ACs generalise poorly compared to partial methods, underperforming compared to 50-second partial compilations, while requiring more compilation time. This is especially noticeable for the *water* dataset, where few queries are usable in the exact setting.

Interestingly, using both ACs does not always improve generalisation; however, it cannot be said beforehand that either a lower-bound (LB)

Table 6.1: Learning generalisation on a test set, with different datasets, compilation timeouts, and AC types. For each combination, the test set MAE at the end of the training and the time taken by the learning loop are given. The first line, represents the test set MAE before training. The best final MAE for each timeout and dataset is in bold. The number of training instances is given next to the dataset name.

		Munin (1384)		Pigs (282)		Water (21)	
Time-out (s)	AC type	Final test MAE	Learn t (s)	Final test MAE	Learn t (s)	Final test MAE	Learn t (s)
Initial test MAE		0.127		0.093		0.111	
5	LB	0.125	152	0.054	1061	0.0009	51
	UB	**0.112**	987	0.083	2004	0.0179	50
	LUBAC	0.127	3614	**0.017**	2688	**0.0007**	208
50	LB	**0.036**	1331	0.020	1563	0.0110	118
	UB	0.070	3604	0.024	3601	0.0176	196
	LUBAC	0.066	3609	**0.003**	3604	**0.0074**	247
150	LB	**0.025**	2771	0.009	1998	0.0083	161
	UB	0.078	3608	0.012	3603	0.0049	530
	LUBAC	0.072	3624	**0.003**	3603	**0.0047**	798
Exact (600)	DFS	0.096	3600	0.044	1896	0.0473	1

or upper-bound (UB) circuit always performs better. While the LUBAC framework *bounds the approximate gradient's error*, the true gradient may align more closely with either of the bounds. Our method's strength is ensuring the learning gradients do not significantly deviate from the true ones, reducing the risk of being misled by gradients derived from one bound only.

Additionally, increasing compilation time does not always improve generalisation. For instance, on the *water* dataset, LUBAC learning performs better with 5 seconds of compilation, and the learning time is reduced, as calculating the gradient through smaller circuits is much faster.

II) How are the Initial Gradients Impacted by which AC is Used? Corollary 1 states that our approximate gradient's error is bounded. If we use both ACs during training, as the learning progresses, the probabilities calculated by both partial ACs align with the true label values, reducing the ACs' bounds gap and the approximate gradient's error bound. The question remains how our approximate gradient compares to exactly calculated

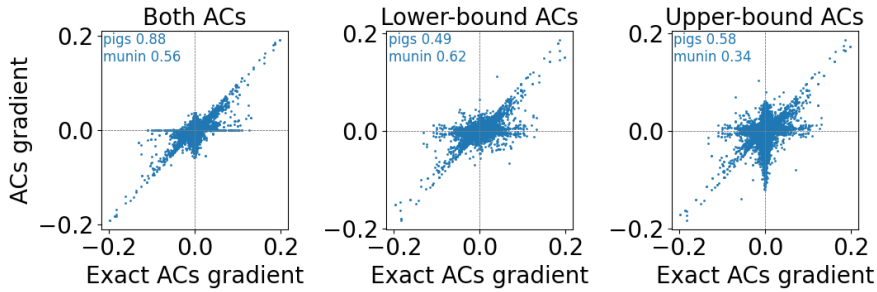


Figure 6.3: Exact and partial ACs parameter gradients for the first training epoch. The x-axis represents the exact gradient value, while the y-axis shows gradients computed from both (left), lower-bound (centre), or upper-bound (right) AC(s). Points correspond to a parameter gradient from the two datasets' queries. Points near the main diagonal are cases where the partial AC gradients closely approximate the true gradients. The Pearson correlation coefficient of the datasets is indicated in the upper left of each subfigure.

gradients early in the search, when the direction of the gradient descent is determined. Figure 6.3 shows how the approximate gradient with a 5-second timeout compares to the true gradient for parameters in the *munin* and *pigs* datasets. Each data point corresponds to a parameter. If the data point is in the lower left or upper right quarter of the graph, then the approximate gradient is in the same direction as the true one. The approximate gradient is positively correlated with the true gradient, meaning that, for most parameters, they are in the same direction. The correlation coefficient supports the finding in the previous experiment; for the *pigs* dataset, the combination of ACs works significantly better than using either bound alone, while for *munin*, the lower-bound works generally better than the other configurations.

III) What is the Time Overhead to Compile Both Lower- and Upper-Bound ACs? We evaluate the runtime overhead of transforming `Schlandals`' search algorithm into a compiler using the cache system and algorithm in Section 6.2.2. Queries solvable within 20 minutes via DFS were timed for two approaches: initial `Schlandals` search and the proposed compilation. Figure 6.4 shows that adding compilation has minimal runtime impact for most queries, with only a few requiring up to 40% more time. It can be seen that sometimes the compilation takes *less* time than the search. This variance is due to variances in hardware utilisation (e.g., L3 cache access) between the benchmark sets, which we

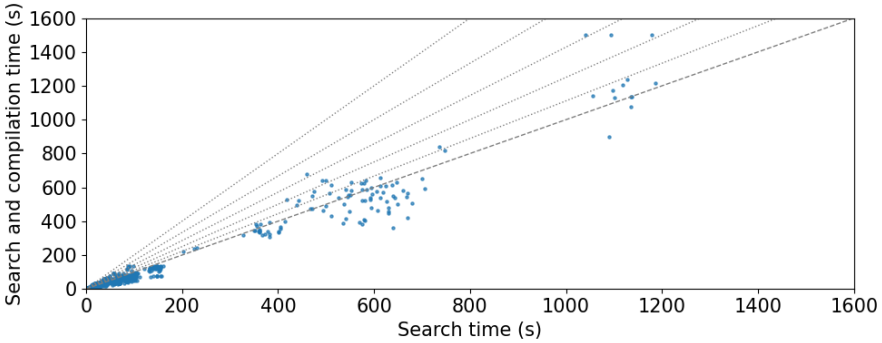


Figure 6.4: Search vs compilation time. Points are the processing time for the three datasets' queries. The x-axis presents the time needed for the classical search only, while the y-axis is the time taken to perform both the search with the new caching system and the ACs compilation. The diagonals are guidelines ranging from $y = 1x$ to $y = 1.5x$ by steps of 0.1. Points on the main diagonal mean that both the search and compilation take the same time; points above it mean the compilation takes longer.

ran in parallel. In any case, the overhead of the compilation is negligible.

6.5 Improvements of the Learning Algorithm

The LUBAC framework allows many different forms of learning algorithms in addition to the one discussed in Section 6.3. Below, we propose several different strategies for learning, as well as some early experimental results.

Loss Computation. In our baseline framework, we made a specific choice for approximating the exact model count $P^*_{\mathcal{W}}[F]$ in Equation 6.22. While this choice leads to a practical learning algorithm, other choices may be more well-founded from a theoretical perspective. For instance, in [DSN24], it was argued that a better approximation of $P^*_{\mathcal{W}}[F]$ is obtained by calculating the geometric mean $\sqrt{P^{lb}_{\mathcal{W}}[F].P^{ub}_{\mathcal{W}}[F]}$. Our experiments with this alternative showed that its results were not significantly different from the more practical approach introduced earlier. Hence, we skip further results on this alternative.

Recompilation of Lower- and Upper-Bound AC Pairs. The first step of the LUBAC framework is to compile partially two ACs. In our experiments, we used `Schlandals` with LDS, which prioritises high-weight interpretations according to the initial choice of weights, enabling faster convergence of upper- and lower-bounds on the probability. However, since

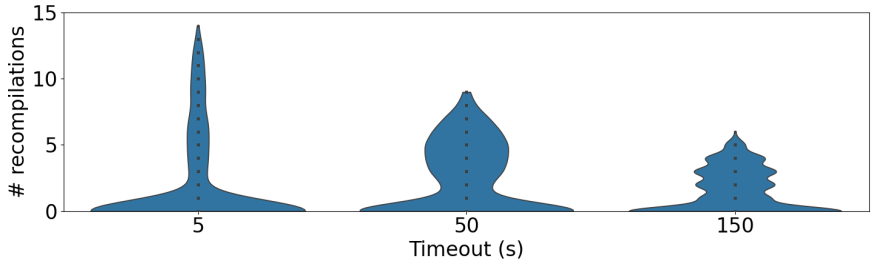


Figure 6.5: Number of times the gap between a lower- and upper-bound ACs pair increased by more than 10^{-4} for 10 consecutive epochs during the training, indicating a potential need for recompilation.

parameters change during learning, previously accurate ACs may no longer effectively approximate the true probability. This possible behaviour has also been considered in DeepProbLog and solved by considering alternative approximate compilation techniques [Man+18; Man+21a; MMD21]. Within LUBAC, maintaining both lower- and upper-bound ACs allows tracking the *optimality gap* by simply updating parameters in both circuits. If the bounds diverge significantly, recompilation might be necessary.

Figure 6.5 shows, for all pairs of corresponding lower- and upper-bound ACs, the number of times their bound difference increases by a non-negligible factor (10^{-4} in our experiment) during training for 10 consecutive epochs. The results indicate that recompilation is not necessary for most queries throughout training. The small number of queries that would benefit from it decreases as the compilation timeout increases.

Trade-Off Between Time, Space, and Bounds Gap. In our baseline approach, we use the ACs produced at the end of the partial compilation algorithm under a per-query timeout. However, these ACs can already be very large, significantly slowing gradient calculations and the learning process, in particular for higher timeouts.

In practice, longer compilation times will not always substantially change the probabilities calculated for the lower- or upper-bound and do not always produce significantly better generalisation. Hence, there are strong advantages to using smaller circuits.

An important observation is that we can modify the caching algorithm relatively easily to roll back towards a partial circuit earlier in the search if that AC has a better precision-size trade-off. By adding timestamps to cache nodes, representing when they were added, we can limit compilation to nodes discovered before a certain point. In an LDS-style search, this

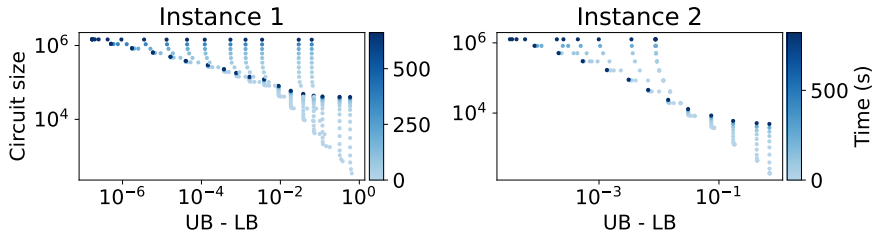


Figure 6.6: Trade-off between circuit size and bounds' gap. Points are pairs of possible discrepancies for lower- and upper-bound partial ACs, with an LDS timeout of 10 minutes. The y-axis represents the sum of the two circuits' sizes. The x-axis is the difference between the upper- and lower-bound. The colour indicates the time needed to obtain the two ACs.

timestamp can be the iteration in which the node was added. This can be done independently for the lower- and upper-bounds, choosing moments when the bounds are close enough and the ACs are small enough. This is a multi-objective optimisation problem; the goal is to find, within a reasonable time, a pair of partial circuits whose bounds' gap and size are small enough.

Figure 6.6 shows this trade-off for two representative hard instances. Each point is a possible combination of a lower-bound and upper-bound pair of partial ACs obtained using `Schlandals`' LDS search, where pairs of circuits found at different moments in time are also considered. The smallest bound gap (i.e., the leftmost point) is always achieved at the end of the search, requiring the most time (darkest point) and space (highest point). It can be seen that solutions on the Pareto front are not necessarily the ones with the highest compilation time. We leave it as future work on how to best solve this multi-objective optimisation problem; however, our result already demonstrates the potential of performing such an optimisation.

6.6 Conclusion

In this chapter, we introduced the Lower- and Upper-Bound ACs (LUBAC) framework as a new approach for learning from CNF formulas that are too complex to be fully compiled into a complete arithmetic circuit. In contrast to our first method, LUBAC does not rely on parameter splitting, making it applicable in practical settings. Moreover, unlike existing partial compilation methods, which provide only a lower-bound on the weighted model count and no guarantees on gradient quality, LUBAC makes use of both lower- and upper-bound circuits to derive a principled bound on the

gradient error.

We presented an efficient algorithm that produces these dual circuits by post-processing the cache of a weighted model counter, showing both theoretically and empirically that the additional overhead is negligible. Building on this, we proved that combining lower- and upper-bound circuits enables the derivation of an *optimality gap* between the true and approximate gradients, thereby addressing a key weakness of previous partial compilation approaches. Our experiments showed that the LUBAC framework can help the learning to generalise better than with only one partial AC. The LUBAC framework makes many different learning algorithms possible, as introduced in Section 6.5. Future work can study these in depth; this includes testing various compilation methods, loss functions, and strategies for selecting only small ACs.

Next steps for this work include comparing the proposed method with other approximate lower-bound-based compilation methods, such as DPLA* [MMD21]. Beyond the expected performance differences arising from distinct constraint encodings and compilation heuristics, we anticipate that these methods will yield results comparable to our lower-bound-only analysis. Another important direction is to investigate the impact in practice of stochastic versus deterministic guarantees on gradient quality, respectively provided by `WeightME` [MDD24] and our LUBAC approach, on both learning behaviour and computational efficiency. Such a comparison could further highlight the respective strengths and limitations of sampling-based and partial compilation-based techniques.

Finally, the LUBAC framework could be extended to a full neuro-symbolic experiment setting, in which neural networks provide probabilities for a subset of variables. From a technical standpoint, this extension is straightforward, as gradients from arithmetic circuits can be backpropagated through neural networks using standard deep learning frameworks. However, as will be discussed further in Chapter 7, identifying a suitable dataset that combines complex symbolic constraints with a meaningful neural component is not straightforward.

In summary, LUBAC retains the scalability benefits of partial compilation while overcoming its main limitation for gradient-based learning, providing a foundation for more robust and principled integration of symbolic constraints into neuro-symbolic learning.

Conclusion

7

This thesis investigated challenges in neuro-symbolic learning, with the goal of combining the strengths of neural, gradient-based approaches with symbolic, knowledge-oriented reasoning. It focused on two complementary aspects of neuro-symbolic learning.

First, this work explored how symbolic knowledge, in the form of logical rules, can be learned directly within differentiable architectures, so as to preserve interpretability while leveraging the strengths of deep learning. Second, we investigated how neural models can be trained under symbolic constraints, making use of domain knowledge to improve logical consistency during learning. A central issue in this part is the high computational cost of reasoning with complex symbolic constraints, which motivated the development of methods for learning with approximate evaluations of constraint satisfaction.

More specifically, in Part I, we introduced RL-Net, a neural architecture for learning rule lists. In Chapter 3, we demonstrated how rule lists provide a natural extension of DR-Net, an earlier rule set neural model, enabling both binary and multi-class classification, and offering promising first results for multi-label tasks. This highlighted how gradient-based optimisation methods can be used to discover interpretable symbolic structures, offering an alternative to traditional combinatorial search strategies or heuristic-defined methods.

In Part II, we focused on training neural models under symbolic constraints. Chapter 4 introduced the technical foundations, in particular the use of weighted model counting to evaluate the satisfaction of logical constraints, and the compilation of logical formulas into arithmetic circuits to support efficient and differentiable reasoning. While exact compilation ensures correctness, it also poses a scalability challenge, motivating the development of approximate methods. Chapter 5, proposed a partial compilation of the constraints, where subtrees of the arithmetic circuit are replaced by constant approximation nodes. This method provided theoretical guarantees on the approximated gradient quality, but was limited by the need to split parameters into trainable and non-trainable sets, making it challenging to learn the whole set of model parameters together.

Finally, in Chapter 6, we presented LUBAC, a novel framework for

learning under complex logical constraints using paired lower- and upper-bound arithmetic circuits. After showing how such paired circuits can be constructed, we demonstrated how relying solely on a lower-bound circuit, which is commonly done, may lead to gradients that deviate significantly from the true ones, and how the addition of an upper-bound circuit enables the derivation of bounds on this approximation error. The experimental evaluation showed how the use of both lower- and upper-bound circuits can help ensure more reliable gradient estimates, and how a trade-off has to be found between the tightness of the bounds, the compilation time, and the size of the resulting partial circuits.

While each contribution chapter concluded with a discussion of immediate technical extensions (see Section 3.4, Section 5.4, and Section 6.5), this conclusion outlines several broader research directions. We present several additional research perspectives, ranging from contribution-specific extensions to higher-level neuro-symbolic considerations.

Lower- and Upper-Bound Computation. In Chapter 6, we proposed a specific method to compute paired lower- and upper-bound arithmetic circuits, by storing all models and non-models encountered during the timed search. However, other methods for computing such paired bounds could be explored. One possible direction is to derive lower- and upper-bounds by modifying the original formula. Lower-bounds could be obtained by adding constraints to the original formula, thereby reducing the number of models, while upper-bounds could be derived by removing constraints, thus increasing the number of models.

The selection of constraints to add or remove could be guided by the goal of significantly reducing the search time, allowing for their exact contribution to be computed. Alternatively, these bounds could themselves be approximated, with the user having the flexibility to specify an independent time limit or approximation levels for each bound. This process of constraint addition or removal could be iterated multiple times, generating a set of lower- and upper-bounds, from which the most suitable pair could be selected based on criteria such as bound tightness, compilation time, or the size of the resulting circuits.

An important open question is whether such derived bounds can also be used to derive guarantees on the gradient error, as is the case for our proposed bound computation method.

Partial Compilation Heuristics Specialised for Learning. The current heuristics used during the partial compilation process in Chapter 6 are those implemented in the underlying compilation method, namely depth-first search (DFS) and limited discrepancy search (LDS). While DFS is a

standard search method, LDS is a more advanced search strategy that has been implemented in `Schlandals` for its ability to explore high-probability regions of the search space while also exploring diverse regions of the search space, instead of focusing entirely on a narrow region first. However, these heuristics were not specifically designed to be used in a learning, nor in an upper- and lower-bound circuits setting.

Indeed, once we consider a learning setting, we cannot assume that the interpretations carrying the highest weight at compilation time will remain the most likely interpretations once the model parameters have been updated. This is particularly true when the initial weights are initialised at random, as is common with neural networks. Hence, the fact that both DFS and LDS focus an important part of their search on the most likely interpretations at compilation time is maybe not necessary in this setting and could be replaced by heuristics that are more focused on diversification of the explored parts of the search space.

This is also an aspect that has been identified in the related literature. In DPLA* [MMD21], one of the partial search-based learning model, the authors also identified the impact of the initial weights on the convergence of the learning process. To mitigate this, they proposed to use curriculum learning, by starting the learning with simpler problems or providing a few labelled examples. They also studied the use of exploration strategies during the search, which worked well but did not scale to larger problems. Concerning sample-based methods, Verreet et al. [VDS23] also identified the importance of diversification in the search space and developed diversity-based sampling methods instead of focusing on the most likely interpretations.

A first idea for developing heuristics that are more focused on diversification could be to keep an idea similar to LDS, but with a shift of focus. Rather than allocating more time to the exploration of the most likely interpretations, the heuristic would ensure that exactly one interpretation is explored for each possible value of all distributions from layer l of the search tree, where l is iteratively increased. In the first iteration, and as illustrated in Figure 7.1, this would correspond to exploring one interpretation per value of the root distribution. In the next iteration, the search would expand to cover at least one interpretation for each value of the subsequent distribution, and so forth. This progressive broadening of the search would ensure that a large portion of the search space is explored, avoiding an initial bias towards the most likely interpretations, based on potentially uninformative initial weights. In the context of our LUBAC framework, this would thus still be combined with the use of both lower- and upper-bound circuits.

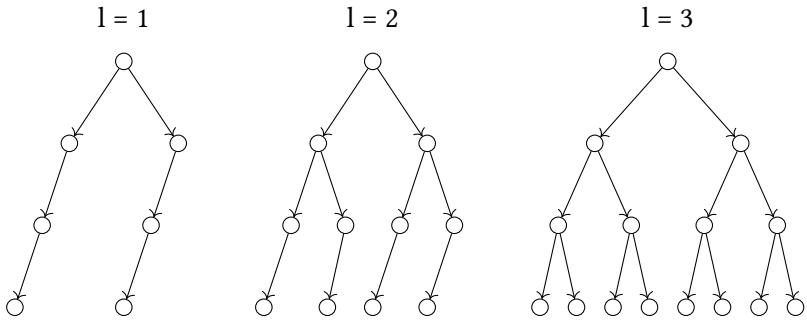


Figure 7.1: Illustration of a diversification-focused heuristic for partial compilation. Each iteration explores at least one interpretation for each value of a variable for the first l layers of the search tree, where l is progressively increased.

Using Bounds with Other Solvers and Problems. The LUBAC framework was introduced as a way to control approximation quality when learning under complex logical constraints, by relying on paired lower- and upper-bound arithmetic circuits. In this thesis, we explored this idea using `Schlandals`, a search-based compilation method specialised for Horn-clause CNF formulas. However, the framework itself is more general and is not restricted to this specific setting. The main requirement is that the compilation method follow a DPLL-style search, in which the compilation is done by exploring a search tree with backtracking. This is the case for many compilation methods, such as `sharpSAT` [Thu06], `dsharp` [Mui+10; Mui+12], and `d4` [LM17]. Since the bound computation method proposed in Chapter 6 is generic and simply relies on storing encountered models, non-models, and the applied propagation, the same principle could, in theory, be applied to these solvers. The main challenge would lie in adapting the implementation to their specific optimisations and larger codebases.

Beyond CNF formulas, there is also potential to apply the LUBAC framework to other types of logical or combinatorial constraints. For instance, global constraints such as the all-different constraint, widely used in constraint programming, allow for compact encodings and benefit from efficient propagation schemes that go beyond standard Boolean unit propagation. These types of constraints could be highly beneficial in a neuro-symbolic setting, as they may yield faster reductions in the search space compared to their propositional logic encodings. Work on compiling such constraints into arithmetic circuits [Ber+16] suggests that the LUBAC framework could be adapted to handle these constraints, provided a suitable compilation method is employed and the bound computation is tailored to

the specific characteristics of these constraints. This could open up new possibilities for learning under complex combinatorial constraints, beyond propositional logic.

Bridging the Two Explored Facets of Neuro-Symbolic Learning. This work has explored two complementary facets of neuro-symbolic learning: learning symbolic knowledge within differentiable architectures, and training neural models under symbolic constraints. While these have been studied separately in the two parts of the thesis, an interesting direction for future work is to explore how links could be established between them.

A first avenue is to investigate how a trained RL-Net model could be repurposed as a source of symbolic knowledge, in the same way that arithmetic circuits are used in the second part of the thesis, to provide logical guidance to another model. In this setting, the rules learned by RL-Net would remain fixed, and the task would be to guide the training of another neural model under the logical constraints defined by these rules.

If this gives promising results, this approach could be extended to the more ambitious goal of jointly learning both the rules and the neural model in an end-to-end fashion. Such a setting would be particularly appealing for high-dimensional data domains, such as vision or language, as RL-Net in its current form is limited to tabular data. The idea of learning intermediate concepts alongside the final prediction is closely related to the framework of concept bottleneck models [Koh+20]. In particular, since the relationship between the concepts and the final prediction would here be learned under interpretable rules, this approach would fall within the family of relational concept bottleneck models [Bar+24].

As the combination of learning both the rules and the neural model is a challenging task, a practical strategy might be to alternate between learning rules with RL-Net and training the neural model under these rules, iterating until convergence. One challenge lies in the cold-start scenario, where neither rules nor reliable model predictions are initially available. Possible solutions include pre-training the neural model on a small set of supervised examples, applying curriculum learning as in DPLA* [MMD21], or starting the process with a small number of expert-provided rules.

Full Neuro-Symbolic Benchmark and Complexity of Datasets. A central theme of this thesis was the challenge of scaling to complex logical constraints, which motivated the development of approximate compilation methods. The evaluation conducted in this work, therefore, relied on benchmarks adapted to the simplified learning setting defined in Section 4.4.2, focusing primarily on deriving gradients from the complex constraints. This design choice allowed us to isolate the impact of approximate compilation

on learning performance compared to exact compilation. Similar simplified settings and model counting instances have also been studied in [MDD24] to specifically study the influence of gradient estimation.

As evoked in the conclusion of Chapter 6, a natural next step would be to evaluate the proposed framework on full neuro-symbolic benchmarks, where the neural model is trained end-to-end alongside the symbolic component. Such experiments would provide a more realistic assessment of the framework's practical value in real-world scenarios. While the qualitative trends observed in the simplified setting are expected to hold, the complexity of the symbolic constraints will likely have a strong influence on the learning performance and computational cost. Indeed, since the method still relies on partial compilation, the complexity of the constraints will impact the tightness of the bounds that can be obtained within a given time budget, which in turn affects the quality of the computed gradients.

Regarding available benchmarks for this study, we find that the field lacks a systematic study of how neuro-symbolic applications can be categorised according to the complexity of their logical constraints.

Indeed, existing full neuro-symbolic benchmarks studied for approximation in the related work often fall into two distinct categories: either they are too simple, making exact compilation feasible, or they are very complex, making them challenging for initial explorations of approximate compilation methods. One known example of a real-world neuro-symbolic dataset is the ROAD-R dataset [Giu+23], which has been used in experiments of `WeightME` [MDD24]. To judge its constraints complexity, we compiled it with the `d4` solver. This calculation was finished in under 0.001 seconds, showing the relative simplicity of its symbolic constraints, which makes exact compilation feasible and the benchmark not particularly informative for studying approximation. If these constraints can be translated into Horn clauses for `Schlandals`, their compilation would likely remain sufficiently fast that no approximation would be required.

Another widely used toy benchmark is the multi-digit MNIST addition [LeC+98; Man+18], later reused in the experiments of `DPLA*`, `A-NeSI` and `ExAL` [MMD21; Kri+23; Ver+24] with experiments extending up to summing two numbers made of 15 digits. However, our experiments with `Schlandals`, presented in Table 7.1, show that exact compilation remains tractable even for 15-digit addition, completing in only a few seconds per query. The table also reports the compilation times required by `ProbLog`, which, while slower, remains within a reasonable range. The efficiency of both methods is likely coming from the way the addition constraints are expressed using carry variables, which allows the constraints to be expressed on each result's digit rather than enumerating all possible sum combinations. Additionally, `Schlandals`'s superior performance over

Table 7.1: Time (in seconds) taken by Schlandals and ProbLog to compute the WMC of a query on the MNIST addition task, for different values of N, the number of digits in each number.

	N=5	N=10	N=12	N=15	N=18
Schlandals	0.21	0.98	1.14	2.20	3.24
ProbLog	3.88	10.53	9.63	78.16	50.85

Table 7.2: Time (in seconds) taken by Schlandals and ProbLog to compute the WMC of a query on the Sudoku task, for different grid sizes. A timeout of an hour (3600 seconds) was set, queries that exceeded this limit to compile are indicated with >3600.

	4x4	9x9
Schlandals	0.04	>3600
ProbLog	15.56	>3600

ProbLog is likely due to its specialisation for multi-valued distributions, avoiding the additional clauses and propagation steps required to enforce single-value assignments for digit variables.

A more challenging benchmark for approximate compilation is the visual sudoku puzzle classification task [Aug+22], which has been used for experiments in A-NeSI [Kri+23], where the goal is to classify whether a given image-based filled Sudoku grid is correct under standard Sudoku constraints. As illustrated in Table 7.2, it presents a very uneven complexity distribution: the 4x4 version compiles trivially, whereas the 9x9 version becomes very hard, with a very low model-to-interpretation ratio. Consequently, partial-search methods such as LUBAC still require long compilation times before producing useful approximations. This highlights the relevance of the gradient error bound introduced in Chapter 6, which provides an explicit indication when the approximation is too coarse to support useful learning signals.

Finally, tasks such as the one proposed by the Warcraft visual path planning problem [Pog+19], used in [Kri+23; Ver+24], introduce a different limitation. Here, the symbolic component (shortest-path computation) is defined algorithmically via Dijkstra's algorithm and cannot be represented in propositional logic, restricting evaluation to sampling-based approaches.

An important direction for future work is, therefore, the development of a broader suite of neuro-symbolic benchmarks with varying levels of complexity for the logical constraints. This would help understand how the complexity of the constraints affects the learning performance when using approximate compilation methods, as well as how guarantees on the gradient error can effectively improve the learning performance.

Bibliography

[Aba+16] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. "{TensorFlow}: a system for {Large-Scale} machine learning". In: *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 2016, pp. 265–283.

[Ach+23] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, et al. "Gpt-4 technical report". In: *arXiv preprint arXiv:2303.08774* (2023).

[Ahm+22a] K. Ahmed, T. Li, T. Ton, Q. Guo, K.-W. Chang, P. Kordjamshidi, V. Srikumar, G. Van den Broeck, and S. Singh. "Pylon: A pytorch framework for learning with constraints". In: *NeurIPS 2021 Competitions and Demonstrations Track*. PMLR. 2022, pp. 319–324.

[Ahm+22b] K. Ahmed, S. Teso, K.-W. Chang, G. Van den Broeck, and A. Vergari. "Semantic probabilistic layers for neuro-symbolic learning". In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 29944–29959.

[And+87] S. Andreassen, M. Woldbye, B. Falck, and S. K. Andersen. "MUNIN: A causal probabilistic network for interpretation of electromyographic findings". In: *Proceedings of the 10th international joint conference on Artificial intelligence-Volume 1*. 1987, pp. 366–372.

[ANS19] J. O. Aoga, S. Nijssen, and P. Schaus. "Modeling Pattern Set Mining Using Boolean Circuits". In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2019, pp. 621–638.

[Aug+22] E. Augustine, C. Pryor, C. Dickens, J. Pujara, W. Wang, and L. Getoor. "Visual sudoku puzzle classification: A suite of collective neuro-symbolic tasks". In: *International Workshop on Neural-Symbolic Learning and Reasoning (NeSy)*. 2022.

[Azi+15] R. A. Aziz, G. Chu, C. Muise, and P. Stuckey. "#∃SAT: Projected model counting". In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2015, pp. 121–137.

[Bad+22] S. Badreddine, A. d. Garcez, L. Serafini, and M. Spranger. "Logic tensor networks". In: *Artificial Intelligence* 303 (2022), p. 103649.

[Bar+16] A. Bart, F. Koriche, J.-M. Lagniez, and P. Marquis. "An improved CNF encoding scheme for probabilistic inference". In: *Proceedings of the Twenty-second European Conference on Artificial Intelligence*. 2016, pp. 613–621.

[Bar+24] P. Barbiero, F. Giannini, G. Ciravegna, M. Diligenti, and G. Marra. "Relational concept bottleneck models". In: *Advances in Neural Information Processing Systems* 37 (2024), pp. 77663–77685.

[Bek+21] M. van Bekkum, M. de Boer, F. van Harmelen, A. Meyer-Vitali, and A. t. Teije. "Modular design patterns for hybrid learning and reasoning systems: a taxonomy, patterns and use cases". In: *Applied Intelligence* 51.9 (2021), pp. 6528–6546.

[Ber+16] D. Bergman, A. A. Cire, W.-J. Van Hoeve, and J. Hooker. *Decision diagrams for optimization*. Vol. 1. Springer, 2016.

[Bes+21] T. R. Besold, A. d'Avila Garcez, S. Bader, H. Bowman, P. Domingos, P. Hitzler, K.-U. Kühnberger, L. C. Lamb, P. M. V. Lima, L. de Penning, et al. "Neural-symbolic learning and reasoning: A survey and interpretation 1". In: *Neuro-Symbolic Artificial Intelligence: The State of the Art*. IOS press, 2021, pp. 1–51.

[BF21a] F. Beck and J. Fürnkranz. "An empirical investigation into deep and shallow rule learning". In: *Frontiers in Artificial Intelligence* 4 (2021).

[BF21b] F. Beck and J. Fürnkranz. "An Investigation into Mini-Batch Rule Learning". In: *arXiv preprint arXiv:2106.10202* (2021).

[BH05] S. Bader and P. Hitzler. "Dimensions of neural-symbolic integration-a structured survey". In: *arXiv preprint cs/0511042* (2005).

[BL99] H. K. Büning and T. Lettmann. *Propositional logic: deduction and algorithms*. Vol. 48. Cambridge University Press, 1999.

[BLC13] Y. Bengio, N. Léonard, and A. Courville. "Estimating or propagating gradients through stochastic neurons for conditional computation". In: *arXiv preprint arXiv:1308.3432* (2013).

[Bor+25] S. Bortolotti, E. Marconato, T. Carraro, P. Morettin, E. van Krieken, A. Vergari, S. Teso, and A. Passerini. "A neuro-symbolic benchmark suite for concept quality and reasoning shortcuts". In: *Advances in Neural Information Processing Systems* 37 (2025), pp. 115861–115905.

[Bre+17] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and regression trees*. Routledge, 2017.

[Bry86] R. E. Bryant. "Graph-based algorithms for boolean function manipulation". In: *Computers, IEEE Transactions on* 100.8 (1986), pp. 677–691.

[BW96] B. Bollig and I. Wegener. "Improving the variable ordering of OBDDs is NP-complete". In: *IEEE Transactions on computers* 45.9 (1996), pp. 993–1002.

[CD05] M. Chavira and A. Darwiche. "Compiling Bayesian networks with local structure". In: *IJCAI*. Vol. 5. 2005, pp. 1306–1312.

[CD06] M. Chavira and A. Darwiche. "Encoding CNFs to empower component analysis". In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2006, pp. 61–74.

[CD08] M. Chavira and A. Darwiche. "On Probabilistic Inference by Weighted Model Counting". In: *Artificial Intelligence* 172.6-7 (2008).

[Cha+14] S. Chakraborty, D. Fremont, K. Meel, S. Seshia, and M. Vardi. "Distribution-Aware Sampling and Weighted Model Counting for SAT". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 28. 2014.

[Cha+24] T. Chataing, J. Perez, M. Plantevit, and C. Robardet. "DiffVersify: a Scalable Approach to Differentiable Pattern Mining with Coverage Regularization". In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer. 2024, pp. 407–422.

[CK22] M. Collery and R. Kusters. "Learning Binary Classification Rules for Sequential Data." In: *Spatio-Temporal Reasoning and Learning (STRL@IJCAI)*. 2022.

[CMV16] S. Chakraborty, K. S. Meel, and M. Y. Vardi. *Algorithmic Improvements in Approximate Counting for Probabilistic Inference: From Linear to Logarithmic SAT Calls*. Tech. rep. 2016.

[Coh95] W. W. Cohen. "Fast effective rule induction". In: *Twelfth International Conference on Machine Learning*. Elsevier, 1995, pp. 115–123.

[Col+23] M. Collery, P. Bonnard, F. Fages, and R. Kusters. "Neural-based classification rule learning for sequential data". In: *International Conference on Learning Representations*. 2023.

[CYM17] W. W. Cohen, F. Yang, and K. R. Mazaitis. "Tensorlog: Deep learning meets probabilistic dbs". In: *arXiv preprint arXiv:1707.05390* (2017).

[Dan+24] A. Daniele, T. Campari, S. Malhotra, and L. Serafini. "Simple and Effective Transfer Learning for Neuro-Symbolic Integration". In: *International Conference on Neural-Symbolic Learning and Reasoning*. Springer. 2024, pp. 166–179.

[Dar01a] A. Darwiche. "Decomposable negation normal form". In: *Journal of the ACM (JACM)* 48.4 (2001), pp. 608–647.

[Dar01b] A. Darwiche. "On the tractable counting of theory models and its application to truth maintenance and belief revision". In: *Journal of Applied Non-Classical Logics* 11.1-2 (2001), pp. 11–34.

[Dar02] A. Darwiche. "A logical approach to factoring belief networks". In: *KR* 2 (2002), pp. 409–420.

[Dar03] A. Darwiche. "A differential approach to inference in Bayesian networks". In: *Journal of the ACM (JACM)* 50.3 (2003), pp. 280–305.

[Dar11] A. Darwiche. "SDD: A New Canonical Representation of Propositional Knowledge Bases". In: *Twenty-Second International Joint Conference on Artificial Intelligence*. 2011.

[Dar99] A. Darwiche. "Compiling knowledge into decomposable negation normal form". In: *IJCAI*. Vol. 99. Citeseer. 1999, pp. 284–289.

[Das+22] T. Dash, S. Chitlangia, A. Ahuja, and A. Srinivasan. "A review of some techniques for inclusion of domain-knowledge into deep neural networks". In: *Scientific Reports* 12.1 (2022), p. 1040.

[DD25] L. De Smet and L. De Raedt. "Defining neurosymbolic AI". In: *arXiv preprint arXiv:2507.11127* (2025).

[DDN24b] L. Dierckx, A. Dubray, and S. Nijssen. "Parameter Learning Using Approximate Model Counting". In: *International Conference on Neural-Symbolic Learning and Reasoning*. Springer. 2024, pp. 80–88.

[DDN25b] L. Dierckx, A. Dubray, and S. Nijssen. "Learning from Logical Constraints with Lower- and Upper-Bound Arithmetic Circuits". In: *International Joint Conference on Artificial Intelligence*. 2025.

[De +19] L. De Raedt, R. Manhaeve, S. Dumancic, T. Demeester, and A. Kimmig. "Neuro-Symbolic= Neural+ Logical+ Probabilistic". In: *NeSy@ IJCAI*. 2019.

[DG17] D. Dua and C. Graff. *UCI Machine Learning Repository*. 2017.

[DK15] L. De Raedt and A. Kimmig. "Probabilistic (logic) programming concepts". In: *Machine Learning* 100.1 (2015), pp. 5–47.

[DKT07] L. De Raedt, A. Kimmig, and H. Toivonen. "ProbLog: A probabilistic Prolog and its application in link discovery". In: *IJCAI 2007, Proceedings of the 20th international joint conference on artificial intelligence*. 2007, pp. 2462–2467.

[DLL62] M. Davis, G. Logemann, and D. Loveland. "A machine program for theorem-proving". In: *Communications of the ACM* 5.7 (1962), pp. 394–397.

[DM02] A. Darwiche and P. Marquis. "A knowledge compilation map". In: *Journal of Artificial Intelligence Research* 17 (2002), pp. 229–264.

[DSG17] I. Donadello, L. Serafini, and A. d. Garcez. "Logic tensor networks for semantic image interpretation". In: *arXiv preprint arXiv:1705.08968* (2017).

[DSN23] A. Dubray, P. Schaus, and S. Nijssen. "Probabilistic Inference by Projected Weighted Model Counting on Horn Clauses". In: *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*. 2023.

[DSN24] A. Dubray, P. Schaus, and S. Nijssen. "Anytime Weighted Model Counting with Approximation Guarantees for Probabilistic Inference". In: *30th International Conference on Principles and Practice of Constraint Programming (CP 2024)*. 2024.

[Due+17] L. Duenas-Osorio, K. Meel, R. Paredes, and M. Vardi. "Counting-Based Reliability Estimation for Power-Transmission Grids". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 31. 2017.

[DVN23b] L. Dierckx, R. Veroneze, and S. Nijssen. "Rl-net: Interpretable rule learning with neural networks". In: *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer. 2023, pp. 95–107.

[Fel+24] J. Feldstein, P. Dilkas, V. Belle, and E. Tsamoura. "Mapping the neuro-symbolic AI landscape by architectures: A handbook on augmenting deep learning through symbolic reasoning". In: *arXiv preprint arXiv:2410.22077* (2024).

[FH17] N. Frosst and G. Hinton. "Distilling a neural network into a soft decision tree". In: *arXiv preprint arXiv:1711.09784* (2017).

[Fie+12] D. Fierens, G. V. d. Broeck, I. Thon, B. Gutmann, and L. De Raedt. "Inference in probabilistic logic programs using weighted CNF's". In: *arXiv preprint arXiv:1202.3719* (2012).

[Fie+15] D. Fierens, G. Van den Broeck, J. Renkens, D. Shterionov, B. Gutmann, I. Thon, G. Janssens, and L. De Raedt. "Inference and Learning in Probabilistic Logic Programs Using Weighted Boolean Formulas". In: *Theory and Practice of Logic Programming* 15.3 (2015).

[FV21] J. Fischer and J. Vreeken. "Differentiable Pattern Set Mining". In: *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 2021, pp. 383–392.

[GAS16] M. Garnelo, K. Arulkumaran, and M. Shanahan. "Towards deep symbolic reinforcement learning". In: *arXiv preprint arXiv:1609.05518* (2016).

[Giu+23] E. Giunchiglia, M. C. Stoian, S. Khan, F. Cuzzolin, and T. Lukasiewicz. "ROAD-R: the autonomous driving dataset with logical requirements". In: *Machine Learning* 112.9 (2023), pp. 3261–3291.

[Giu+24] E. Giunchiglia, A. Tatomir, M. C. Stoian, and T. Lukasiewicz. "CCN+: A neuro-symbolic framework for deep learning with requirements". In: *International Journal of Approximate Reasoning* 171 (2024), p. 109124.

[Giu+25] E. Giunchiglia, F. Imrie, M. van der Schaar, and T. Lukasiewicz. "Machine learning with requirements: A manifesto". In: *Neurosymbolic Artificial Intelligence* 1 (2025), NAI–240767.

[GL20] E. Giunchiglia and T. Lukasiewicz. "Coherent hierarchical multi-label classification networks". In: *Advances in neural information processing systems* 33 (2020), pp. 9662–9673.

[GL21] E. Giunchiglia and T. Lukasiewicz. "Multi-label classification neural networks with hard logical constraints". In: *Journal of Artificial Intelligence Research* 72 (2021), pp. 759–818.

[GL23] A. d. Garcez and L. C. Lamb. "Neurosymbolic AI: The 3rd wave". In: *Artificial Intelligence Review* 56.11 (2023), pp. 12387–12406.

[GLG09] A. Garcez, L. Lamb, and D. Gabbay. *Neural-Symbolic Cognitive Reasoning*. Jan. 2009. ISBN: 978-3-540-73245-7. DOI: 10.1007/978-3-540-73246-4.

[GND11a] T. Guns, S. Nijssen, and L. De Raedt. "Evaluating pattern set mining strategies in a constraint programming framework". In: *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer. 2011, pp. 382–394.

[GND11b] T. Guns, S. Nijssen, and L. De Raedt. "K-Pattern set mining under constraints". In: *IEEE Transactions on Knowledge and Data Engineering* 25.2 (2011), pp. 402–418.

[Gom+07] C. P. Gomes, J. Hoffmann, A. Sabharwal, and B. Selman. "From Sampling to Model Counting." In: *IJCAI*. 2007.

[Goo+20] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. "Generative adversarial networks". In: *Communications of the ACM* 63.11 (2020), pp. 139–144.

[Gri09] S. Grimm. "Knowledge representation and ontologies". In: *Scientific data mining and knowledge discovery: Principles and foundations*. Springer, 2009, pp. 111–137.

[GS19] M. Garnelo and M. Shanahan. "Reconciling deep learning with symbolic artificial intelligence: representing objects and relations". In: *Current Opinion in Behavioral Sciences* 29 (2019), pp. 17–23.

[GSL22] E. Giunchiglia, M. C. Stoian, and T. Lukasiewicz. "Deep learning with logical constraints". In: *arXiv preprint arXiv:2205.00523* (2022).

[GW12] B. Ganter and R. Wille. *Formal concept analysis: mathematical foundations*. Springer Science & Business Media, 2012.

[Háj01] P. Hájek. *Metamathematics of fuzzy logic*. Vol. 4. Springer Science & Business Media, 2001.

[HD05] J. Huang and A. Darwiche. "DPLL with a trace: From SAT to knowledge compilation". In: *IJCAI*. Vol. 5. 2005, pp. 156–162.

[HD07] J. Huang and A. Darwiche. “The language of search”. In: *Journal of Artificial Intelligence Research* 29 (2007), pp. 191–219.

[HG95] W. D. Harvey and M. L. Ginsberg. “Limited discrepancy search”. In: *IJCAI (1)*. 1995, pp. 607–615.

[HH07] B. Hammer and P. Hitzler. *Perspectives of neural-symbolic integration*. Vol. 77. Springer, 2007.

[HS22] P. Hitzler and M. K. Sarker. *Neuro-symbolic artificial intelligence: The state of the art*. IOS press, 2022.

[Hua+21] J. Huang, Z. Li, B. Chen, K. Samel, M. Naik, L. Song, and X. Si. “Scallop: From probabilistic deductive databases to scalable differentiable reasoning”. In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 25134–25145.

[Hub+16] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. “Binarized neural networks”. In: *Advances in neural information processing systems* 29 (2016).

[Jen+89] F. Jensen, U. Kjærulff, K. Olesen, and J. Pedersen. *An expert system for control of waste water treatment—a pilot project*. Tech. rep. Technical report, Judex Datasystemer A/S, Aalborg, 1989. In Danish, 1989.

[Joh+17] J. Johnson, B. Hariharan, L. Van Der Maaten, L. Fei-Fei, C. Lawrence Zitnick, and R. Girshick. “Clevr: A diagnostic dataset for compositional language and elementary visual reasoning”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 2901–2910.

[Kah11] D. Kahneman. “Thinking, fast and slow”. In: *Farrar, Straus and Giroux* (2011).

[Kau22] H. Kautz. “The third ai summer: Aaai robert s. engelmore memorial lecture”. In: *Ai magazine* 43.1 (2022), pp. 105–125.

[Kim+08] A. Kimmig, V. Santos Costa, R. Rocha, B. Demoen, and L. De Raedt. “On the efficient execution of ProbLog programs”. In: *International Conference on Logic Programming*. Springer. 2008, pp. 175–189.

[Kis+14] D. Kisa, G. Van den Broeck, A. Choi, and A. Darwiche. “Probabilistic Sentential Decision Diagrams”. In: *Fourteenth International Conference on the Principles of Knowledge Representation and Reasoning*. 2014.

[Kli+17] K. A. Klise, M. Bynum, D. Moriarty, and R. Murray. "A Software Framework for Assessing the Resilience of Drinking Water Systems to Disasters with an Example Earthquake Case Study". In: *Environmental modelling & software* 95 (2017).

[Koh+20] P. W. Koh, T. Nguyen, Y. S. Tang, S. Mussmann, E. Pierson, B. Kim, and P. Liang. "Concept bottleneck models". In: *International conference on machine learning*. PMLR. 2020, pp. 5338–5348.

[Kri+23] E. van Krieken, T. Thanapalasingam, J. Tomczak, F. Van Harmelen, and A. Ten Teije. "A-nesi: A scalable approximate method for probabilistic neurosymbolic inference". In: *Advances in Neural Information Processing Systems* 36 (2023), pp. 24586–24609.

[Kri+24] E. van Krieken, S. Badreddine, R. Manhaeve, and E. Giunchiglia. "Uller: A unified language for learning and reasoning". In: *International Conference on Neural-Symbolic Learning and Reasoning*. Springer. 2024, pp. 219–239.

[Kri+25] E. van Krieken, P. Minervini, E. Ponti, and A. Vergari. "Neurosymbolic Reasoning Shortcuts under the Independence Assumption". In: *arXiv preprint arXiv:2507.11357* (2025).

[Kus+22] R. Kusters, Y. Kim, M. Collery, C. d. S. Marie, and S. Gupta. "Differentiable rule induction with learned relational features". In: *arXiv preprint arXiv:2201.06515* (2022).

[LBH15] Y. LeCun, Y. Bengio, and G. Hinton. "Deep learning". In: *nature* 521.7553 (2015), pp. 436–444.

[LeC+98] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[LHM+98] B. Liu, W. Hsu, Y. Ma, et al. "Integrating classification and association rule mining." In: *Kdd*. Vol. 98. 1998, pp. 80–86.

[LHP01] W. Li, J. Han, and J. Pei. "CMAR: Accurate and efficient classification based on multiple class-association rules". In: *Proceedings 2001 IEEE international conference on data mining*. IEEE. 2001, pp. 369–376.

[LM17] J.-M. Lagniez and P. Marquis. "An Improved Decision-DNNF Compiler." In: *IJCAI*. Vol. 17. 2017.

[LMY21] Y. Lai, K. S. Meel, and R. H. Yap. "The power of literal equivalence in model counting". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 35. 5. 2021, pp. 3851–3859.

[LMY23] Y. Lai, K. S. Meel, and R. H. Yap. "Fast Converging Anytime Model Counting". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 37. 2023.

[LWK18] C. Louizos, M. Welling, and D. P. Kingma. "Learning sparse neural networks through L0 regularization". In: *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*. 2018.

[Man+18] R. Manhaeve, S. Dumancic, A. Kimmig, T. Demeester, and L. De Raedt. "Deepproblog: Neural Probabilistic Logic Programming". In: *advances in neural information processing systems* 31 (2018).

[Man+21a] R. Manhaeve, S. Dumančić, A. Kimmig, T. Demeester, and L. De Raedt. "Neural probabilistic logic programming in DeepProbLog". In: *Artificial Intelligence* 298 (2021), p. 103504.

[Man+21b] R. Manhaeve, G. Marra, T. Demeester, S. Dumančić, A. Kimmig, and L. De Raedt. "Neuro-symbolic AI= neural+ logical+ probabilistic AI". In: *Neuro-Symbolic Artificial Intelligence: The State of the Art*. IOS press, 2021, pp. 173–191.

[Mao+19] J. Mao, C. Gan, P. Kohli, J. B. Tenenbaum, and J. Wu. "The neuro-symbolic concept learner: Interpreting scenes, words, and sentences from natural supervision". In: *arXiv preprint arXiv:1904.12584* (2019).

[Mar+23] E. Marconato, S. Teso, A. Vergari, and A. Passerini. "Not all neuro-symbolic concepts are created equal: Analysis and mitigation of reasoning shortcuts". In: *Advances in Neural Information Processing Systems* 36 (2023), pp. 72507–72539.

[Mar+24] G. Marra, S. Dumančić, R. Manhaeve, and L. De Raedt. "From statistical relational to neurosymbolic artificial intelligence: A survey". In: *Artificial Intelligence* 328 (2024), p. 104062.

[MD19] G. Marcus and E. Davis. *Rebooting AI: Building artificial intelligence we can trust*. Vintage, 2019.

[MD25] J. Maene and L. De Raedt. "The Gradient of Algebraic Model Counting". In: *AAAI* (2025).

[MDD24] J. Maene, V. Derkinderen, and L. De Raedt. "On the Hardness of Probabilistic Neurosymbolic Learning". In: *arXiv preprint arXiv:2406.04472* (2024).

[MDM24] J. Maene, V. Derkinderen, and P. Z. D. Martires. "KLay: Accelerating Neurosymbolic AI". In: *arXiv preprint arXiv:2410.11415* (2024).

[Med+17] W. Medjroubi, U. P. Müller, M. Scharf, C. Matke, and D. Kleinhans. "Open Data in Power Grid Modelling: New Approaches towards Transparent Grid Models". In: *Energy Reports* 3 (2017).

[Mil19] T. Miller. "Explanation in artificial intelligence: Insights from the social sciences". In: *Artificial intelligence* 267 (2019), pp. 1–38.

[MMD21] R. Manhaeve, G. Marra, and L. De Raedt. "Approximate inference for neural probabilistic logic programming". In: *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning*. IJCAI Organization. 2021, pp. 475–486.

[Mui+10] C. Muise, S. McIlraith, J. C. Beck, and E. Hsu. "Fast d-DNNF Compilation with sharpSAT". In: *Workshops at the twenty-fourth AAAI conference on artificial intelligence*. 2010.

[Mui+12] C. Muise, S. A. McIlraith, J. C. Beck, and E. I. Hsu. "D sharp: fast d-DNNF compilation with sharpSAT". In: *Advances in Artificial Intelligence: 25th Canadian Conference on Artificial Intelligence, Canadian AI 2012, Toronto, ON, Canada, May 28-30, 2012. Proceedings 25*. Springer. 2012, pp. 356–361.

[Nil86] N. J. Nilsson. "Probabilistic logic". In: *Artificial intelligence* 28.1 (1986), pp. 71–87.

[OD15] U. Oztok and A. Darwiche. "A Top-down Compiler for Sentential Decision Diagrams". In: *Twenty-Fourth International Joint Conference on Artificial Intelligence*. 2015.

[Pas+19] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. "Pytorch: An imperative style, high-performance deep learning library". In: *Advances in neural information processing systems* 32 (2019).

[Pog+19] M. V. Pogančić, A. Paulus, V. Musil, G. Martius, and M. Rolinek. "Differentiation of blackbox combinatorial solvers". In: *International Conference on Learning Representations*. 2019.

[QWL21] L. Qiao, W. Wang, and B. Lin. "Learning accurate and interpretable decision rule sets from neural networks". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 35. 5. 2021, pp. 4303–4311.

[Red+16] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. "You only look once: Unified, real-time object detection". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 779–788.

[RFB15] O. Ronneberger, P. Fischer, and T. Brox. "U-net: Convolutional networks for biomedical image segmentation". In: *International Conference on Medical image computing and computer-assisted intervention*. Springer. 2015, pp. 234–241.

[Rot96] D. Roth. "On the hardness of approximate reasoning". In: *Artificial intelligence* 82.1-2 (1996), pp. 273–302.

[Rud19] C. Rudin. "Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead". In: *Nature machine intelligence* 1.5 (2019), pp. 206–215.

[RVN12] J. Renkens, G. Van den Broeck, and S. Nijssen. "k-optimal: A novel approximate inference algorithm for ProbLog". In: *Machine learning* 89.3 (2012), pp. 215–231.

[San+04] T. Sang, F. Bacchus, P. Beame, H. A. Kautz, and T. Pitassi. "Combining Component Caching and Clause Learning for Effective Model Counting." In: *SAT* 4 (2004).

[Sar+22] M. K. Sarker, L. Zhou, A. Eberhart, and P. Hitzler. "Neuro-symbolic artificial intelligence: Current trends". In: *Ai Communications* 34.3 (2022), pp. 197–209.

[SBK05] T. Sang, P. Beame, and H. A. Kautz. "Performing Bayesian inference by weighted model counting". In: *AAAI*. Vol. 5. 2005, pp. 475–481.

[Scu09] M. Scutari. "Learning Bayesian networks with the bnlearn R package". In: *arXiv preprint arXiv:0908.3817* (2009).

[SG16] L. Serafini and A. d. Garcez. "Logic tensor networks: Deep learning and logical reasoning from data and knowledge". In: *arXiv preprint arXiv:1606.04422* (2016).

[SGM20] M. Soos, S. Gocht, and K. S. Meel. "Tinted, detached, and lazy CNF-XOR solving and its applications to counting and sampling". In: *International Conference on Computer Aided Verification*. Springer. 2020, pp. 463–484.

[Sha+19] S. Sharma, S. Roy, M. Soos, and K. S. Meel. "GANAK: A Scalable Probabilistic Exact Model Counter." In: *IJCAI*. Vol. 19. 2019.

[Shi+22] S. Shi, Y. Xie, Z. Wang, B. Ding, Y. Li, and M. Zhang. "Explainable Neural Rule Learning". In: *WWW 2022 - Proceedings of the ACM Web Conference 2022*. Association for Computing Machinery, Inc, 2022, pp. 3031–3041. ISBN: 9781450390965.

[Sil+16] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. "Mastering the game of Go with deep neural networks and tree search". In: *nature* 529.7587 (2016), pp. 484–489.

[SM19] M. Soos and K. S. Meel. "BIRD: Engineering an Efficient CNF-XOR SAT Solver and Its Applications to Approximate Model Counting". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33. 2019.

[Sto+24] M. C. Stoian, A. Tatomir, T. Lukasiewicz, and E. Giunchiglia. "PiShield: A PyTorch Package for Learning with Requirements". In: *arXiv preprint arXiv:2402.18285* (2024).

[Thu06] M. Thurley. "sharpSAT-counting Models with Advanced Component Caching and Implicit BCP". In: *SAT* 4121 (2006).

[Val79] L. G. Valiant. "The complexity of computing the permanent". In: *Theoretical computer science* 8.2 (1979), pp. 189–201.

[VAV22] E. Van Krieken, E. Acar, and F. Van Harmelen. "Analyzing differentiable fuzzy logic operators". In: *Artificial Intelligence* 302 (2022), p. 103602.

[VDS23] V. Verreet, L. De Smet, and E. Sansone. "EXPLAIN, AGREE and LEARN: A Recipe for Scalable Neural-Symbolic Learning". In: *ICML Workshop: Knowledge and Logical Reasoning in the Era of Data-driven Learning, Location: Honolulu, Hawaii, USA*. 2023.

[Ver+24] V. Verreet, L. De Smet, L. De Raedt, and E. Sansone. "EXPLAIN, AGREE, LEARN: Scaling Learning for Neural Probabilistic Logic". In: *arXiv preprint arXiv:2408.08133* (2024).

[Vla+14] J. Vlasselaer, J. Renkens, G. Van den Broeck, and L. De Raedt. "Compiling Probabilistic Logic Programs into Sentential Decision Diagrams". In: *Proceedings Workshop on Probabilistic Logic Programming (PLP)*. 2014.

[Vla+16] J. Vlasselaer, A. Kimmig, A. Dries, W. Meert, and L. De Raedt. "Knowledge Compilation and Weighted Model Counting for Inference in Probabilistic Logic Programs." In: *AAAI Workshop: Beyond NP*. Vol. 101. 2016.

[VMM23] A. Vermeulen, R. Manhaeve, and G. Marra. "An Experimental Overview of Neural-Symbolic Systems". In: *International Conference on Inductive Logic Programming*. Springer. 2023, pp. 124–138.

[Von+21] L. Von Rueden, S. Mayer, K. Beckh, B. Georgiev, S. Giesselbach, R. Heese, B. Kirsch, J. Pfrommer, A. Pick, R. Ramamurthy, et al. "Informed machine learning–a taxonomy and survey of integrating prior knowledge into learning systems". In: *IEEE Transactions on Knowledge and Data Engineering* 35.1 (2021), pp. 614–633.

[VT19] F. Van Harmelen and A. Ten Teije. "A boxology of design patterns for hybrid learning and reasoning systems". In: *Journal of Web Engineering* 18.1-3 (2019), pp. 97–123.

[WFV24] N. P. Walter, J. Fischer, and J. Vreeken. "Finding interpretable class-specific patterns through efficient neural search". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 38. 8. 2024, pp. 9062–9070.

[Wie16] B. Wiegmans. *Gridkit: European And North-American Extracts*. Mar. 2016. DOI: 10.5281/ZENODO.47317.

[Xu+18] J. Xu, Z. Zhang, T. Friedman, Y. Liang, and G. Broeck. "A semantic loss function for deep learning with symbolic knowledge". In: *International conference on machine learning*. PMLR. 2018, pp. 5502–5511.

[XWV24] S. Xu, N. P. Walter, and J. Vreeken. "Neuro-Symbolic Rule Lists". In: *arXiv preprint arXiv:2411.06428* (2024).

[YMH18] Y. Yang, I. G. Morillo, and T. M. Hospedales. "Deep neural decision trees". In: *ICML Workshop on Human Interpretability in Machine Learning. arXiv preprint arXiv:1806.06988* (2018).

[Zha+23] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola. *Dive into deep learning*. Cambridge University Press, 2023.