University of São Paulo
Institute of Mathematics, Statistics, and Computer Science
Bachelor of Computer Science

From Logic to Learning: A Framework for Converting Sentential Decision Diagrams into Differentiable Neural Networks

Matheus Sanches Jurgensen

Final Essay

mac 499 — Capstone Project

Advisor: Prof. Dr. Dênis Deratani Mauá
Co-advisor: Jonas Rodrigues Lima Gonçalves

São Paulo
2025

Acknowledgments

I would rather walk with a friend in the dark than walk alone in the light.

— Helen Keller

First and foremost, I would like to express my deepest gratitude to my advisor, Denis. It has been an absolute pleasure and a privilege to work alongside a true master in the fields of Artificial Intelligence and Probabilistic Reasoning. His guidance, deep insights, and academic rigor were indispensable to the realization of this work.

I am also immense grateful to my co-advisor, Jonas. A brilliant mind with the rare capability of explaining super complex concepts with remarkable didactics, he was always available to help navigate the technical challenges of this project.

I extend my sincere thanks to the University of São Paulo (USP), and specifically to the Institute of Mathematics and Statistics (IME). This institution provided me with the solid theoretical foundation necessary to write this work and opened countless academic and professional doors that have shaped my career.

Finally, I dedicate my heartfelt thanks to my family. Thank you for your unwavering support since the very beginning, and especially during my years at the university. Your complete faith in my capabilities and decisions gave me the strength to persevere and reach this milestone.

Resumo

Matheus Sanches Jurgensen. **From Logic to Learning: A Framework for Converting Sentential Decision Diagrams into Differentiable Neural Networks**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2025.

A integração do raciocínio simbólico com as capacidades de aprendizado de redes neurais representa um desafio de fronteira na Inteligência Artificial. Enquanto a Programação por Conjunto de Respostas (ASP) e os circuitos lógicos oferecem robustas capacidades dedutivas, eles tradicionalmente carecem da flexibilidade para lidar com incertezas e aprender parâmetros a partir de dados. Este trabalho propõe e implementa um framework de software abrangente para preencher essa lacuna, convertendo Diagramas de Decisão Sentencial (SDDs)—um subconjunto tratável e canônico de circuitos na Forma Normal de Negação (NNF)—em redes neurais diferenciáveis.

Introduzimos um pipeline metodológico que transforma SDDs simbólicos em Circuitos Aritméticos (ACs) compatíveis com ambientes de aprendizado profundo. Este processo envolve garantir a suavidade do circuito para assegurar uma Contagem Ponderada de Modelos (WMC) válida e mapear operações lógicas AND e OR para nós de produto e soma diferenciáveis, respectivamente. A arquitetura resultante trata parâmetros probabilísticos como pesos aprendíveis, permitindo a unificação da inferência probabilística e do aprendizado de parâmetros dentro de um único grafo computacional baseado em PyTorch.

O framework foi rigorosamente validado utilizando o estudo de caso da rede Bayesiana "Alarm". Resultados experimentais demonstram que o sistema realiza inferência probabilística de forma eficiente e recupera parâmetros probabilísticos latentes a partir de dados parcialmente observados com alta precisão, atingindo taxas de erro negligenciáveis quando treinado com conjuntos de dados suficientes. Além disso, a implementação adere a rigorosos padrões de engenharia de software, incluindo princípios SOLID e testes automatizados extensivos, garantindo uma base de código modular e extensível. Este trabalho estabelece uma fundação confiável para pesquisas futuras em IA Neuro-Simbólica, particularmente no domínio da lógica diferenciável e raciocínio probabilístico.

Palavras-chave: IA Neuro-Simbólica. Diagramas de Decisão Sentencial. Programação Diferenciável. Inferência Probabilística. Contagem Ponderada de Modelos.

Abstract

Matheus Sanches Jurgensen. **From Logic to Learning: A Framework for Converting Sentential Decision Diagrams into Differentiable Neural Networks**. Capstone Project Report (Bachelor). Institute of Mathematics, Statistics, and Computer Science, University of São Paulo, São Paulo, 2025.

The integration of symbolic reasoning with the learning capabilities of neural networks represents a frontier challenge in Artificial Intelligence. While Answer Set Programming (ASP) and logic circuits offer robust deductive capabilities, they traditionally lack the flexibility to handle uncertainty and learn parameters from data. This work proposes and implements a comprehensive software framework for bridging this gap by converting Sentential Decision Diagrams (SDDs)—a tractable, canonical subset of Negation Normal Form (NNF) circuits—into differentiable neural networks.

We introduce a methodological pipeline that transforms symbolic SDDs into Arithmetic Circuits (ACs) compatible with deep learning environments. This process involves enforcing circuit smoothness to ensure valid Weighted Model Counting (WMC) and mapping logical AND and OR operations to differentiable product and sum nodes, respectively. The resulting architecture treats probabilistic parameters as learnable weights, enabling the unification of probabilistic inference and parameter learning within a single PyTorch-based computational graph.

The framework was rigorously validated using the "Alarm" Bayesian network case study. Experimental results demonstrate that the system performs probabilistic inference efficiently and recovers latent probabilistic parameters from partially observed data with high precision, achieving negligible error rates when trained on sufficient datasets. Furthermore, the implementation adheres to strict software engineering standards, including SOLID principles and extensive automated testing, ensuring a modular and extensible codebase. This work establishes a reliable foundation for future research in Neuro-Symbolic AI, particularly in the domain of differentiable logic and probabilistic reasoning.

Keywords: Neuro-Symbolic AI. Sentential Decision Diagrams. Differentiable Programming. Probabilistic Inference. Weighted Model Counting.

List of Figures

List of Tables

List of Programs

Contents

Chapter 1

Introduction

In recent years, the field of **Artificial Intelligence (AI)** has seen a growing interest in bridging the gap between two distinct paradigms: **symbolic reasoning** and **deep learning**. This intersection, known as *Neuro-Symbolic Artificial Intelligence*, aims to combine the robustness and interpretability of logical systems with the generalization and learning capabilities of neural networks.

A key player in the symbolic realm is **Answer Set Programming (ASP)**, a declarative programming paradigm oriented towards complex search problems. While ASP is powerful for logical deduction, traditional implementations often lack the ability to handle uncertainty or learn parameters from data. To address this, projects like **dPASP** (Differentiable Probabilistic Answer Set Programming) have emerged (Geh *et al.*, 2023).

The goal of dPASP is to create an engine capable of interpreting probabilistic answer set programs and generating differentiable programs—such as PyTorch computation graphs. This allows for the execution of complex probabilistic operations and parameter learning in a highly efficient, gradient-based manner.

The transformation from a high-level probabilistic program to a differentiable network typically follows a multi-stage pipeline, as illustrated in Figure 1.1:

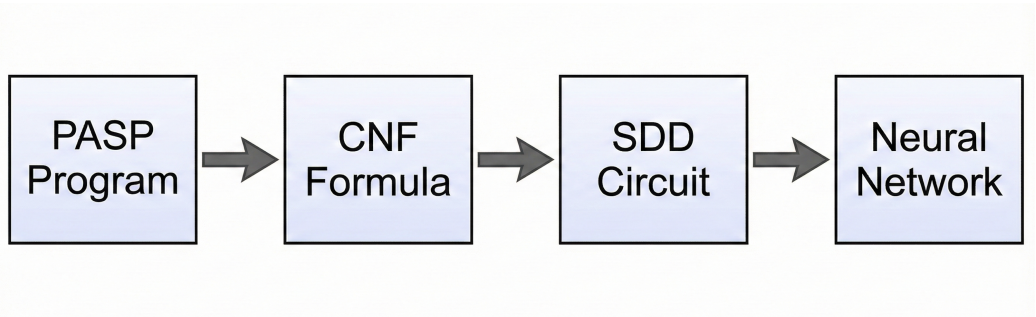

Figure 1.1: *The dPASP project pipeline: converting a PASP program into a Neural Network via CNF and SDD intermediate representations.*

1. The PASP program is converted into a **CNF** (Conjunctive Normal Form) expression.
2. The CNF is compiled into a tractable **Logical Circuit**.

3. Finally, this circuit is converted into a **Differentiable Program** (Neural Network).

This work focuses specifically on the final stage of this pipeline: the conversion of logical circuits into neural networks.

1.1 Goals

The primary objective of this work is to design and implement a comprehensive software framework capable of converting **Sentential Decision Diagrams (SDDs)**—a specific, highly structured type of logic circuit—into Neural Networks.

By mapping logical nodes to arithmetic operations within a neural architecture, we aim to integrate standard logical circuits with probabilistic weights. This integration unlocks two critical capabilities:

- **Probabilistic Inference:** Efficiently computing the probability of complex queries given evidence (Weighted Model Counting).
- **Optimization:** Utilizing the differentiability of the neural representation to learn probabilistic parameters from data using gradient descent.

Such a system has wide-ranging applications. For instance, in the gaming industry, it can be used to model non-player character (NPC) behaviors where actions are governed by logical rules but influenced by probabilistic uncertainties learned from player data.

Beyond the theoretical implementation, a central goal of this thesis is to ensure that the developed framework adheres to the highest standards of **Software Engineering**. The system is designed to be reliable, modular, and extensible, providing a robust foundation for future research in the neuro-symbolic domain.

1.2 Text Organization

The remainder of this text is organized as follows:

Chapter 2 – Preliminary Definitions: Establishes the theoretical foundations, defining Logic Circuits, Negation Normal Form (NNF), Sentential Decision Diagrams (SDDs), Weighted Model Counting (WMC), and the dPASP pipeline.

Chapter 3 – Methodological Approach: Details the process of preparing the circuit (including smoothing) and the logic devised for both probabilistic inference and parameter training.

Chapter 4 – Implementation Details: Describes the software architecture, detailing the specific packages (parser, converter, optimizer), the input file formats, and the software engineering principles applied.

Chapter 5 – Results: Presents a validation of the framework using the "Alarm" Bayesian network as a case study, analyzing the accuracy of inference and the efficiency of probability learning.

Chapter 6 – Conclusion: Summarizes the main contributions, discusses the technical challenges faced during development, and outlines potential avenues for future work.

Chapter 2

Preliminary Definitions

Before diving deep into the problem that we aim to resolve, it is important to establish some basic definitions that are the foundation of all the work that will be exposed here.

2.1 Logic Circuits

Logic circuits (also called *Boolean circuits*) are the main focus of the present work, so it is crucial to give a clear definition for them. A logic circuit is a finite directed acyclic graph (DAG) that represents a Boolean function. It is basically made of its leaf, internal, and output nodes, which can be better explained as follows:

Leaf nodes: Boolean variables (e.g., $x_1, x_2, \ldots, x_n$) and constant values (*True* and *False*).

Internal nodes: Logical operations such as **AND** ($\wedge$), **OR** ($\vee$), and **NOT** ($\neg$), which take one or more boolean inputs and produce a single boolean output.

Output node: The resulting Boolean function computed by the circuit.

Therefore, a Boolean circuit provides a structural way to efficiently compute a Boolean formula.

A significant subset of logic circuits relevant to this work consists of those represented in **Negation Normal Form (NNF)**. Adnan Darwiche formally defines NNF in his foundational paper *A Knowledge Compilation Map* as follows:

> "Let PS be a denumerable set of propositional variables. A sentence in NNF_{PS} is a rooted, directed acyclic graph (DAG) where each leaf node is labeled with *true*, *false*, X or $\neg X$, $X \in PS$; and each internal node is labeled with $\wedge$ or $\vee$ and can have arbitrarily many children. The size of a sentence Σ in NNF_{PS}, denoted $|\Sigma|$, is the number of its DAG edges. Its height is the maximum number of edges from the root to some leaf in the DAG."
>
> — (Darwiche, 2002)

Hence, an NNF is nothing more than a logic circuit with some restrictions in relation to its internal nodes. A visual example of an NNF representing the expression $(A \vee D) \wedge (\neg A \vee E)$

can be seen in Figure 2.1.

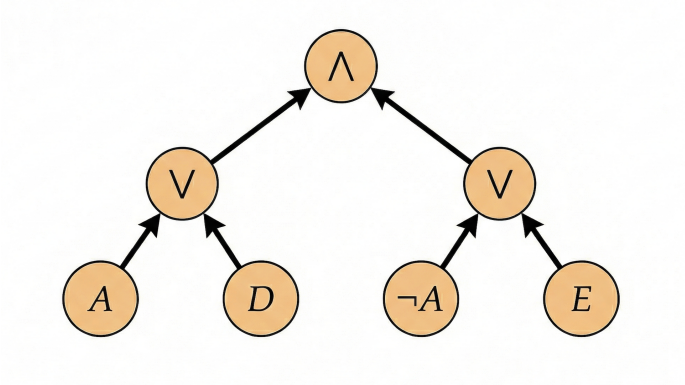

Figure 2.1: *Example of a Negation Normal Form (NNF) circuit for the expression* $(A \vee D) \wedge (\neg A \vee E)$.

Also, we can define some important properties of an NNF which will influence its usability and size.

2.2 Properties of NNF Circuits

When analyzing NNF circuits, we find that they can exhibit several important structural properties: **decomposability**, **determinism**, **smoothness**, **decision**, and **ordering**. These properties play a key role in determining both the succinctness of a circuit and its ability to perform certain logical operations efficiently (in polynomial time). It is interesting to note that generally these properties establish a trade-off between **succinctness** and computational **efficiency**: more compact circuits may support fewer polynomial-time operations.

Figure 2.2 illustrates a circuit that satisfies several of these key properties, which are defined below:

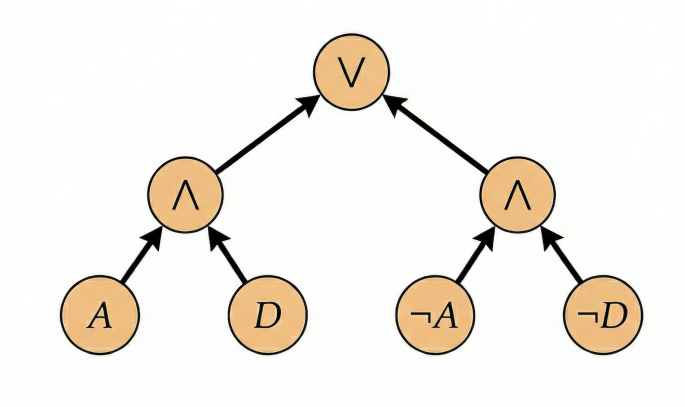

Figure 2.2: *An NNF circuit representing* $(A \wedge D) \vee (\neg A \wedge \neg D)$*, exhibiting smoothness, determinism, and decomposability.*

Decomposability: An NNF circuit is decomposable if, in every conjunction (AND node), the subcircuits connected by that node do not share any variables.

Determinism: An NNF circuit is deterministic if, in every disjunction (OR node), the subcircuits connected by that node are logically mutually exclusive: no assignment of variables can make more than one child true at the same time.

Smoothness: An NNF circuit is smooth if, in every disjunction (OR node), all subcircuits mention exactly the same set of variables.

Decision: A decision NNF satisfies this property if it is both decomposable and deterministic, and if every disjunction (OR node) represents an explicit decision on the value of a variable. In other words, each OR node can be seen as a "test" on a variable X, where one branch corresponds to X being true, and the other corresponds to X being false.

Ordering: An NNF circuit satisfies the ordering property if the variables appear according to a fixed global order along every path from the root to a leaf. This means that no matter which path is followed in the circuit, the variables are tested in the same sequence.

Starting from a general NNF circuit, as more of the properties defined above are enforced, the range of tractable operations increases; however, the succinctness of the representation tends to decrease. In other words, each additional constraint imposed on the structure of the circuit makes certain computations easier to perform, but often at the cost of a larger circuit size.

2.3 Sentential Decision Diagrams

In this work, only a specific subset of NNF circuits will be considered and used: the **Sentential Decision Diagrams (SDDs)**.

SDDs are a refined and well-structured subclass of NNFs designed to make reasoning tasks both efficient and reliable. As first introduced by Darwiche (2011), an SDD is a structured, decomposable, and deterministic type of NNF circuit that follows a predefined hierarchical organization of variables, known as a **variable tree (or vtree)**. This vtree defines how variables are grouped and combined, ensuring that the circuit maintains a clear logical organization.

The key idea behind SDDs is that they represent complex logical formulas through a sequence of structured decisions that respect the variable hierarchy defined by the vtree. Because of this organization, SDDs combine the compactness of NNFs with a high degree of computational efficiency. In other words, they are built in a way that allows many important reasoning operations—such as satisfiability checking, model counting, conditioning, and equivalence testing—to be performed in polynomial time.

Figure 2.3 presents an example of an SDD structure representing the expression $(A \wedge D) \vee (\neg D \wedge E)$.

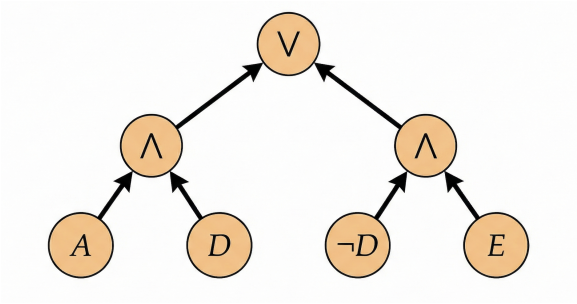

Figure 2.3: *Structure of a Sentential Decision Diagram (SDD) representing* $(A \wedge D) \vee (\neg D \wedge E)$.

2.4 Weighted Model Counting (WMC)

Weighted Model Counting (WMC) is the fundamental algebraic operation underpinning modern probabilistic inference in logical circuits. It generalizes the classical problem of model counting (#SAT)—which simply enumerates the number of satisfying assignments—by assigning a real-valued weight to each literal in the formula.

Formally, let ϕ be a propositional formula over a set of variables $\mathcal{V}$. Let $\mathcal{L}$ be the set of literals corresponding to these variables (i.e., $\{v, \neg v \mid v \in \mathcal{V}\}$). We define a weight function $w : \mathcal{L} \to \mathbb{R}_{\geq 0}$ that maps each literal to a non-negative real number.

The weight of a specific model (satisfying assignment) $\omega \models \phi$ is defined as the product of the weights of the literals that are true in that model:

$$\text{Weight}(\omega) = \prod_{l \in \omega} w(l) \tag{2.1}$$

The Weighted Model Count of the formula ϕ is then defined as the sum of the weights of all models that satisfy ϕ:

$$\text{WMC}(\phi) = \sum_{\omega \models \phi} \text{Weight}(\omega) = \sum_{\omega \models \phi} \prod_{l \in \omega} w(l) \tag{2.2}$$

In the context of Probabilistic Answer Set Programming and neural-symbolic integration, WMC provides the bridge between logic and probability. By assigning weights corresponding to probabilities—specifically, setting $w(v) = p$ and $w(\neg v) = 1 - p$ for a probabilistic fact v—the WMC of a formula yields exactly its probability $P(\phi)$ under the distribution defined by the independent facts (Chavira and Darwiche, 2008).

This reduction allows us to transform the problem of probabilistic inference into a graph traversal problem. Once the logical dependencies are compiled into a tractable circuit (like an SDD), computing the WMC becomes a linear-time operation in terms of the circuit size, enabling efficient forward passes in our neural pipeline.

2.5 The dPASP Pipeline

As outlined in the introduction, the transformation from a high-level probabilistic logic specification to a differentiable learning system follows a structured pipeline. This section details each stage of this pipeline—from the initial Probabilistic Answer Set Programming (PASP) source code to the final PyTorch-based neural network. To illustrate this process concretely, we will trace the transformation of a simple example program throughout this section.

2.5.1 Probabilistic Answer Set Programming (PASP)

Answer Set Programming (ASP) is a declarative programming paradigm rooted in the stable model semantics of logic programming (Lifschitz, 2008; Gelfond and Lifschitz, 1988). Unlike imperative languages that prescribe the control flow of computation, ASP enables the specification of problems via logical rules and constraints. It is particularly effective for knowledge representation and reasoning (KRR) and solving NP-hard combinatorial problems.

Formally, an ASP program Π consists of a finite set of rules of the form:

$$h \leftarrow b_1, \dots, b_n, \mathit{not}\ c_1, \dots, \mathit{not}\ c_m \tag{2.3}$$

where h, b_i, c_j are atoms (propositional symbols) and *not* denotes default negation (negation as failure). A rule states that the head h must be true if the body is satisfied—that is, if all b_i are true and no c_j can be proven true.

In a typical workflow, a first-order program containing variables is *grounded*—instantiated with all possible constants from the domain—resulting in a propositional program. The semantics of this ground program are given by its *answer sets* (or stable models), which are the minimal models satisfying the rules under the Gelfond-Lifschitz reduct. For a comprehensive treatment of ASP syntax and solving, we refer to Lifschitz (2019).

Probabilistic Answer Set Programming (PASP) extends this paradigm by introducing independent categorical random variables, often referred to as probabilistic facts. In this framework, the combination of a set of probabilistic facts and a standard ASP program defines a probability distribution over possible answer sets.

Formally, we define a PASP program $\mathcal{P}$ as a tuple $\langle \Pi, \mathcal{F}, P \rangle$, where Π is a set of logical rules, $\mathcal{F}$ is a set of probabilistic facts, and P is a probability distribution over $\mathcal{F}$.

Typically, each fact $f \in \mathcal{F}$ is associated with a probability p_f, such that $P(f = \text{true}) = p_f$. A *total choice* θ is a subset of $\mathcal{F}$ assumed to be true, while $\mathcal{F} \setminus \theta$ is assumed false. The probability of a specific total choice is given by the product of the independent probabilities:

$$P(\theta) = \prod_{f \in \theta} p_f \prod_{f \in \mathcal{F} \setminus \theta} (1 - p_f) \tag{2.4}$$

Crucially, in our framework, we assume the logical program Π is stratified or valid such that each total choice θ determines exactly one stable model. This assumption ensures a

functional relationship: while different total choices may lead to different stable models, a fixed total choice results in a single, unambiguous logical outcome.

In the context of our work—converting these programs into neural networks—we focus on two primary inference tasks:

1. **Model Counting (Inference):** Computing the probability of a query atom q. This is formally defined as the sum of the probabilities of all total choices θ such that the induced stable model entails q:

$$P(q) = \sum_{\theta \subseteq \mathcal{T}} P(\theta) \cdot \mathbb{I}(\Pi \cup \theta \models q) \tag{2.5}$$

2. **Parameter Learning:** Computing the gradient of the log-probability of a query with respect to the probabilistic parameters (weights), allowing the system to learn from observed data via gradient descent.

To illustrate, consider the following PASP program. It defines two independent probabilistic facts, p and q, and a deterministic rule stating r occurs if either p or q is present:

```
% Probabilistic facts
0.25::p.
0.33::q.

% Logical rules
r :- p.
r :- q.
```

Program 2.1: *Simple PASP example program*

Under the stable model semantics, the rules $r \leftarrow p$ and $r \leftarrow q$ define r completely in terms of p and q. Therefore, the semantics of this program is equivalent to the Boolean formula $r \leftrightarrow (p \vee q)$. In the probabilistic context, p and q represent independent choices. The set of total choices (or possible worlds) comprises $\emptyset$, $\{p\}$, $\{q\}$, and $\{p, q\}$.

In the dPASP pipeline, this high-level logic is the starting point for compilation.

2.5.2 Conjunctive Normal Form (CNF)

The first step in the compilation pipeline is the translation of the logic program's semantics into a standard propositional representation: *Conjunctive Normal Form* (CNF). A CNF formula is defined as a conjunction (AND) of clauses, where each clause is a disjunction (OR) of literals (variables or their negations).

For the class of tight logic programs typically used in these conversions, this translation is achieved via *Clark's Completion* (Clark, 1978). Clark's Completion transforms the sufficiency of ASP rules (where the body implies the head) into necessary and sufficient conditions (equivalence).

Consider the example from Program 2.1, where the rule set implies that r is true if p or q is true. Clark's Completion captures this by asserting $r \leftrightarrow (p \vee q)$. To facilitate circuit

compilation, this equivalence is rewritten into clauses:

$$(\neg p \vee r) \wedge (\neg q \vee r) \wedge (\neg r \vee p \vee q) \tag{2.6}$$

This formula rigorously enforces the program's dependencies: the first two clauses ensure that if either cause (p or q) is present, the effect (r) must hold; the third clause ensures that r cannot arise spontaneously without at least one of its causes (support).

2.5.3 Sentential Decision Diagram (SDD)

While CNF provides a universal representation for propositional logic, it is computationally intractable for probabilistic inference. Specifically, computing the probability of a query (Weighted Model Counting) on a raw CNF formula is a #P-complete problem (Valiant, 1979). To overcome this, the CNF is compiled into a *Sentential Decision Diagram* (SDD), a tractable circuit representation defined in Section 2.3.

For our running example, the CNF derived in Section 2.5.2 is compiled into the SDD structure shown in Figure 2.4. This directed acyclic graph (DAG) encodes the Boolean function $r \leftrightarrow (p \vee q)$ while guaranteeing the properties of determinism and decomposability required for efficient probability computation.

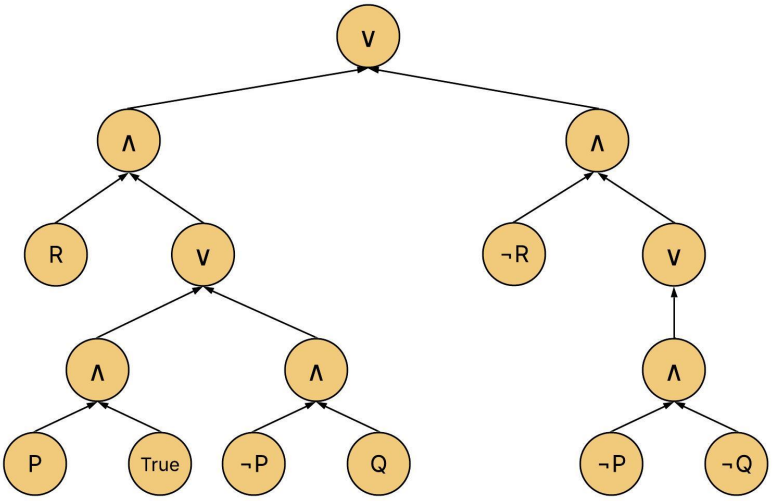

Figure 2.4: *Visual representation of the SDD circuit that structurally encodes the logic $r \leftrightarrow (p \vee q)$.*

This SDD represents the exact same Boolean function as the PASP program but in a form that is optimized for the next and final stage: conversion into a computational graph for inference and learning.

2.5.4 PyTorch and Neural Networks

The final stage of the pipeline converts the static SDD into a dynamic, differentiable model. Before detailing the implementation framework, it is crucial to understand the structural synergy that makes this conversion possible.

From a computational perspective, both NNF circuits (like SDDs) and neural networks share a similar architecture: they can be represented as directed acyclic graphs (DAGs) composed of interconnected computational nodes. In NNFs, each node represents a logical operation such as conjunction (AND) or disjunction (OR), while in neural networks, nodes perform arithmetic operations such as weighted summation and nonlinear activation. This similarity provides a natural mapping between the two structures, where Boolean logic can be reformulated as differentiable arithmetic operations suitable for gradient-based optimization.

The goal of these arithmetic operations is to preserve the circuit's flow of information while enabling continuous optimization in a neural network setting. Such a correspondence allows NNF-based logical models to be embedded into differentiable architectures, thus bridging symbolic reasoning and statistical learning (Garcez *et al.*, 2019). By translating SDD circuits into neural networks, it becomes possible to perform probabilistic inference and optimization through standard deep learning techniques, while maintaining the logical consistency of the original model.

To realize this conversion efficiently, we utilize **PyTorch**, a leading open-source deep learning library (Paszke *et al.*, 2019). PyTorch provides the necessary infrastructure to treat our logical circuits as optimized computational graphs.

PyTorch operates on a core data structure known as the `torch.Tensor`, a multi-dimensional array similar to NumPy's `ndarray`, but with the crucial capability of running on Graphics Processing Units (GPUs) for accelerated computation. The framework is built around the concept of dynamic computational graphs (or define-by-run), where the graph is built on the fly as operations are executed. This allows for immense flexibility in defining complex architectures.

The implementation leverages two key components:

1. **The Computational Graph (DAG):** We map the nodes of the SDD directly to `torch.nn.Module` components. The hierarchical structure of the SDD ensures that the resulting neural network is a Directed Acyclic Graph, where data flows from the probabilistic facts (leaves) to the query node (root).

2. **Automatic Differentiation (Autograd):** PyTorch's autograd engine allows us to compute gradients through the circuit. Since the network output represents the query probability $P(Q)$, the gradients computed via backpropagation correspond to the partial derivatives of the likelihood with respect to the probabilistic parameters.

The efficiency of this approach stems from PyTorch's underlying C++ and CUDA backend, which executes tensor operations (like the sums and products in our converted circuits) in parallel on hardware. Furthermore, PyTorch's **autograd** engine automatically records the graph of operations performed on tensors. By applying the chain rule for derivatives, autograd can traverse this graph backward to compute the gradients of the output (e.g., the query probability) with respect to the inputs (e.g., the atomic probabilities). This mechanism is what enables the use of efficient optimization algorithms, such as Stochastic Gradient Descent (SGD), to learn logical probabilities from data without the need for manual gradient derivation.

Chapter 3

Methodological Approach

To integrate logical circuits with probabilistic reasoning mechanisms for inference and optimization, it is essential to clearly define the transformation pipeline from a logic circuit to its corresponding neural representation. This section outlines the methodological steps required to prepare the circuit, enforce its formal properties (such as smoothness), and connect it to probabilistic inference and optimization frameworks. These steps form the conceptual foundation for the technical implementation presented later.

3.1 Preparing the Logical Circuit

Probabilistic reasoning based on logical circuits relies on circuit forms that enable efficient evaluation and differentiation. SDDs are particularly well suited for this task because of their decomposability and determinism properties (Darwiche, 2011; Choi *et al.*, 2020). However, smoothness is generally also required for probabilistic inference and neural parameterization.

3.1.1 Enforcing Circuit Smoothness

In the context of tractable circuits, smoothness ensures that all disjunctive (OR) nodes operate over the same set of variables. This property is crucial when performing probabilistic inference because it guarantees that the WMC is well-defined and complete over all variable assignments (Darwiche, 2003).

Intuitively, when two subcircuits under an OR node represent different conditions due to only partially overlapping variable sets, their combination might omit some variables, leading to inconsistencies or undercounted probability mass. Ensuring smoothness forces every subcircuit to "cover" the same variables, so that probability mass is distributed consistently across all models.

The standard way to enforce smoothness was originally described in Darwiche's seminal paper *On the Tractable Counting of Theory Models and its Application to Belief Revision and Truth Maintenance* (Darwiche, 1999). In that approach, whenever a node violates smoothness (i.e., its children do not mention the same set of variables), we augment

the missing variables with tautological subcircuits that leave the semantics unchanged. For example, if one child of an OR node does not include variable X, we expand it with the disjunction $(X \vee \neg X)$ so that all children mention X. This can be viewed as inserting "dummy" variable nodes that ensure structural uniformity without altering the truth function.

This smoothing technique effectively balances the variable scope across the circuit without any semantic change to it. It can be seen as padding each branch of the disjunction so that all operate over an identical feature (variable) space.

3.1.2 Constructing an Arithmetic Circuit

After enforcing smoothness, the logical circuit can be translated into an **Arithmetic Circuit** – a computational graph composed of arithmetic operations that is compatible with differentiable learning frameworks such as neural networks.

As demonstrated in Figure 3.1, the logical circuit is transformed into an AC by replacing logical operators with arithmetic ones. Crucially, this direct mapping preserves the probabilistic semantics **if and only if** the underlying logical circuit satisfies the properties of decomposability, determinism, and smoothness. Since SDDs enforce decomposability and determinism by definition, and smoothness is enforced in the previous step, this transformation yields a valid probabilistic model extended to continuous domains.

In particular:

- **AND nodes** are mapped to **product nodes** (×).
- **OR nodes** are mapped to **sum nodes** (+).

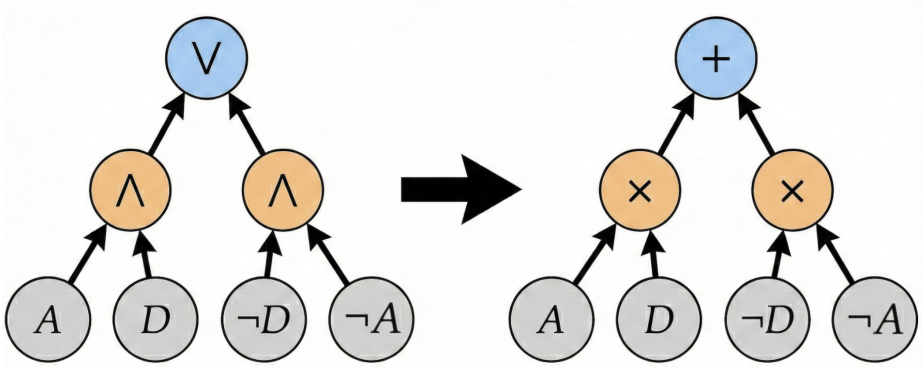

Figure 3.1: *Transformation of a Boolean Logic Circuit (left) into an Arithmetic Circuit (right).*

This translation relies on the structural guarantees of the circuit to perform valid Weighted Model Counting (WMC) (Darwiche, 2003). Specifically:

1. **Decomposability** ensures that the subcircuits of an AND node share no variables. This logical independence allows the probability of the conjunction to be computed as the product of the probabilities of its children: $P(A \wedge B) = P(A) \times P(B)$.
2. **Determinism** ensures that the subcircuits of an OR node are mutually exclusive (disjoint). This allows the probability of the disjunction to be computed as the sum of the probabilities of its children: $P(A \vee B) = P(A) + P(B)$.

3. **Smoothness** ensures that all children of an OR node cover the same set of variables, preventing the loss of probability mass for marginalized variables during summation.

When the circuit inputs represent probabilities instead of binary truth values, the resulting Arithmetic Circuit efficiently computes the WMC, as its operations align directly with the standard axioms of probability theory (Chavira and Darwiche, 2008).

To understand why these specific arithmetic operations are suitable, it is helpful to consider their behavior at the Boolean limits (inputs of 0 and 1). The intuition allows us to view the arithmetic operations as continuous generalizations of the logical ones:

1. A **product node** outputs 1 only when all of its inputs are 1, and 0 otherwise. This mirrors the logical AND operation (and corresponds to the probability of independent events).

2. A **sum node** outputs 0 only when all of its inputs are 0. This corresponds to the logical OR operation. Crucially, due to the **determinism** property of the SDD, at most one input to a sum node can be non-zero at any time, ensuring the sum represents a valid probability mass that never exceeds 1.

Thus, by leveraging the specific structural properties of smoothed SDDs, these arithmetic operations maintain logical consistency while allowing smooth differentiability – an essential requirement for integrating the circuit into neural or optimization-based systems.

3.1.3 Input Layer Structure

The leaves of the circuit correspond to the input variables, forming what can be viewed as the input layer of the network. For each Boolean variable X_i, we include two input nodes: one representing the variable itself X_i and another representing its negation $\neg X_i$. Consequently, for a set of N variables, the input layer comprises $2N$ nodes.

This duplication allows the circuit to explicitly represent both truth assignments and their complements, ensuring that subsequent arithmetic operations can evaluate any possible configuration of the Boolean inputs.

3.1.4 Deterministic and Probabilistic Inputs

For **deterministic variables** – those with fixed true or false values – the input nodes are sufficient to inject their Boolean assignments into the circuit. Each input node receives a real-valued signal, typically 1 for “true” and 0 for “false”.

However, when variables are **probabilistic**, their values are not fixed; instead, each variable X_i is associated with a probability $P(X_i)$ representing the likelihood that it is true. To correctly propagate this probabilistic information through the arithmetic circuit, we must integrate these probabilities into the computation.

Encoding Probabilities in the Circuit

The circuit can be viewed as a computation graph composed of sum and product nodes, analogous to those used in neural network architectures. This structure allows

probabilities to be integrated directly using arithmetic operations.

For each positive variable X_i, we replace its input node with a product node whose children are:

- The variable's Boolean input value (e.g., 1 if active, 0 otherwise), and
- A constant node representing its probability $P(X_i)$.

The constant node outputs a fixed scalar corresponding to the probability of the variable. Thus, the output of this small substructure is:

$$X_i \cdot P(X_i)$$

This ensures that whenever the variable is active, its influence in the circuit is weighted proportionally to its likelihood.

Handling Negated Variables

The situation for negated variables $\neg X_i$ is slightly different. The probability of a negation is the complement of the original probability, that is:

$$P(\neg X_i) = 1 - P(X_i)$$

Therefore, the node corresponding to $\neg X_i$ must multiply its Boolean input by the complement of its probability:

$$(1 - P(X_i)) \cdot \neg X_i$$

Since the circuit supports only addition and multiplication, this complement must be computed without explicit subtraction. To achieve this, we multiply the probability by -1 and then add the result to 1:

$$P(\neg X_i) = 1 + (-1 \times P(X_i))$$

Both operations are supported by the arithmetic circuit, since -1 can be represented as a constant node. The resulting subcircuit for $\neg X_i$ thus consists of:

1. A product node computing $(-1) \times P(X_i)$;
2. A sum node adding the result to 1;
3. A final product node multiplying this value by the Boolean input of $\neg X_i$:

$$\text{output}(\neg X_i) = \neg X_i \times (1 + (-1 \times P(X_i)))$$

Figure 3.2 illustrates the resulting computation graph. Note that the parameter θ corresponds to the probability $P(A)$. The graph explicitly calculates both the positive branch ($A \times \theta$) and the negative branch ($\neg A \times (1 - \theta)$) using only differentiable operations.

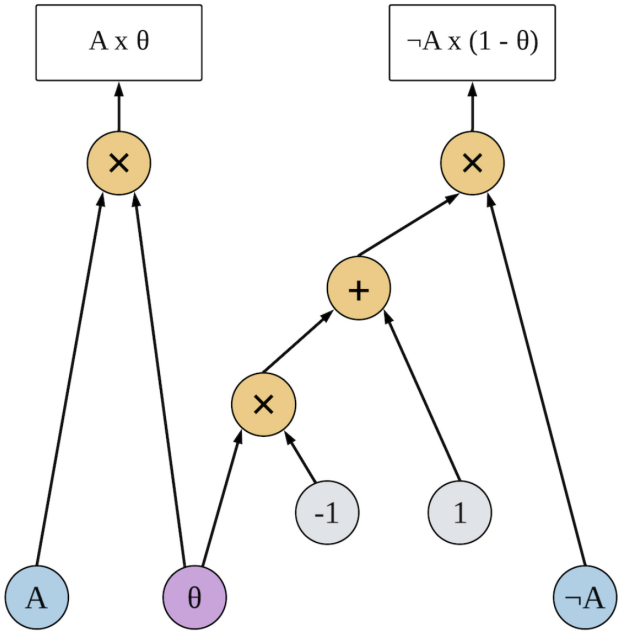

Figure 3.2: *Arithmetic circuit substructure for encoding probabilistic parameters (θ) and their complements.*

3.2 Inference Logic

Once the SDD has been transformed into a smooth Arithmetic Circuit and the input layer has been structured to handle probabilistic variables, the model is prepared for inference. The objective of inference is to compute the probability of a specific query (a partial or complete assignment of variables) by performing a single forward pass through the circuit. This computation is equivalent to the Weighted Model Counting (WMC) of the query.

The inference process is dictated by how the $2N$-sized input vector is populated. This vector consists of nodes for each literal X_i and its corresponding negation $\neg X_i$. The values assigned to these inputs depend on the nature of the query:

Assigned Variables (Evidence): If a variable X_i is part of the evidence, its value is fixed.

- If X_i is assigned **True**, its corresponding input node X_i receives a value of 1, while its negation node $\neg X_i$ receives a value of 0.
- If X_i is assigned **False**, its input node X_i receives a 0, and its negation node $\neg X_i$ receives a 1.

This "one-hot" encoding effectively selects the branch of the circuit corresponding to the evidence, zeroing out the probability mass from the contradictory path.

Marginalized Variables (Unobserved): If a variable X_i is not part of the query, its value is unknown, and it must be marginalized (or "summed out") from the computation. To achieve this, both the literal node X_i and its negation node $\neg X_i$ are assigned a value of 1.

This specific encoding is the key to performing marginalization. By setting both inputs to 1, the forward pass will sum the contributions of both the positive and negative branches of that variable.

Mathematically, if we consider a local substructure computing the probability contribution of X_i with weights $P(X_i)$ and $P(\neg X_i)$, the operation becomes:

$$X_i \cdot P(X_i) + \neg X_i \cdot P(\neg X_i)$$

When both inputs are set to 1, this simplifies to:

$$1 \cdot P(X_i) + 1 \cdot P(\neg X_i) = P(X_i) + P(\neg X_i) = 1$$

Because the circuit is *smooth*, this local summation guarantees that the variable is effectively integrated out of the joint distribution, preserving the probability mass for the remaining variables in the query.

After the $2N$ input vector is set according to these rules, a single forward pass is executed. The values propagate from the leaves, with sum nodes performing addition and product nodes performing multiplication. The final scalar value computed at the root node of the circuit represents the probability, or WMC, of the input query.

3.3 Training Logic

While the inference logic assumes that the probabilities $P(X_i)$ for probabilistic variables are known, the training logic provides a mechanism for learning these parameters from data. The objective of training is to adjust the circuit's parameters so that the probabilities it computes (its WMC) align with observations from a given dataset D.

This process reframes the probabilistic Arithmetic Circuit as a differentiable neural network. The core structure and the forward pass computation remain identical to those described in the inference logic. The fundamental change is the treatment of the probabilistic parameters.

Instead of being fixed constants, the probabilities associated with literal nodes are replaced by learnable parameters. For each probabilistic variable X_i, we introduce a corresponding unconstrained learnable parameter θ_i. To ensure that the output of this parameter can be interpreted as a valid probability (i.e., it remains in the interval $[0, 1]$), a sigmoid activation function is applied during the forward pass.

Thus, the probability $P(X_i)$ is no longer a static value but is computed as:

$$P(X_i) = \sigma(\theta_i) = \frac{1}{1 + e^{-\theta_i}}$$

This differentiable parameterization allows the entire circuit to be optimized using standard gradient-based methods. The training procedure is as follows:

1. **Forward Pass:** For a given data sample d from the dataset D, an input vector is constructed using the same rules as the inference logic (assigning 1/0 for observed variables and 1/1 for unobserved). The forward pass is executed, computing the probability of the sample $P(d \mid \theta)$ based on the current parameters θ.
2. **Loss Computation:** A loss function measures the discrepancy between the model's computed probabilities and the observed data. A standard choice for this probabilistic task is the Negative Log-Likelihood (NLL) of the dataset:

$$\mathcal{L}(\theta) = -\sum_{d \in D} \log P(d \mid \theta)$$

   Minimizing this NLL is equivalent to maximizing the likelihood that the model's parameters produced the observed data.
3. **Backward Pass (Optimization):** Because the entire network – from the sigmoid-activated parameters θ through the sum and product nodes – is fully differentiable, the gradient of the loss $\mathcal{L}$ with respect to each parameter θ_i ($\nabla_{\theta_i}\mathcal{L}$) can be computed via backpropagation.
4. **Parameter Update:** The gradients are used by an optimization algorithm, such as Stochastic Gradient Descent (SGD), to iteratively update the parameters θ. This update rule adjusts the parameters in the direction that minimizes the loss, thereby "learning" the probabilities that best explain the training data.

Chapter 4

Implementation Details

To realize the methodological approach described, a software framework was developed in Python, leveraging the PyTorch library for neural network components and optimization. The framework is architected into several distinct packages, each handling a specific part of the conversion and computation pipeline.

4.1 Expected Inputs

The system expects two files as input: an `.sdd` file defining the structure and a `.json` file defining the semantics.

4.1.1 The .sdd File

This file defines the logical structure of the Sentential Decision Diagram. It is a text file where nodes are defined bottom-up (children before parents). The syntax specifies the node type using a single letter:

F *id-of-false-sdd-node*: Defines the constant False node.

T *id-of-true-sdd-node*: Defines the constant True node.

L *id-of-literal-sdd-node id-of-vtree literal*: Defines a literal node, linking it to a vtree and a specific literal (e.g., a variable or its negation).

D *id-of-decomposition-sdd-node id-of-vtree number-of-elements {id-of-prime id-of-sub}**: Defines a decomposition node, which functions as an OR gate. This OR node has *number-of-elements* children. Each child is an implicit AND gate. The *{id-of-prime id-of-sub}** part specifies the two child node IDs (the prime and sub) for each of these implicit AND gates. Thus, the line defines a structure equivalent to:

$$(\text{prime}_1 \wedge \text{sub}_1) \vee (\text{prime}_2 \wedge \text{sub}_2) \vee \dots$$

The * indicates that all of these pairs are listed sequentially on the same line.

A simple example file is shown in Program 4.1. It defines a structure equivalent to $(X_1 \wedge X_2) \vee (\neg X_1 \wedge \neg X_2)$.

```
c Example .sdd file representing (X1 AND X2) OR (NOT X1 AND NOT X2)
sdd 7
F 0
T 1
L 2 1 1 c Literal node 2 for X1
L 3 1 -1 c Literal node 3 for NOT X1
L 4 2 2 c Literal node 4 for X2
L 5 2 -2 c Literal node 5 for NOT X2
c Root node (ID 6) is a decomposition (OR)
c It has 2 elements: (Node 2 AND Node 4) OR (Node 3 AND Node 5)
D 6 1 2 2 4 3 5
```

Program 4.1: *Example .sdd file representing an Equivalence (XNOR) structure*

4.1.2 The .json File

This file provides the semantic context and probabilistic information for the circuit. While it contains various pieces of metadata, the most critical sections for this framework are:

"atom_mapping": A dictionary that maps the internal numeric IDs used in the .sdd file (e.g., "1", "2") to human-readable variable names (e.g., "burglary", "earthquake").

"prob": An object containing the details of the probabilistic variables.

"pvars": A list of all variable IDs that are considered probabilistic.

"pfacts": A list of pairs, where each pair `[id, probability]` assigns a specific probability (e.g., `[1, 0.1]`) to a probabilistic variable, mapping it to the corresponding atom ID. This section is used when probabilities are fixed and not learned.

Program 4.2 illustrates a standard configuration file.

```
{
   "atom_mapping": {
      "1": "burglary",
      "2": "earthquake",
      "3": "alarm"
   },
   "metadata": {
      "num_atoms": 3,
      "num_pfacts": 3
   },
   "prob": {
      "pfacts": [
         [ 1, 0.1 ],
         [ 2, 0.2 ],
         [ 3, 0.7 ]
```

```
16          ],
17          "pvars": [ 1, 2, 3 ]
18      }
19  }
```

Program 4.2: *Example .json semantic configuration file*

4.2 Package Structure and Core Components

The framework was developed with modularity in mind, ensuring a clear separation of concerns. Figure 4.1 provides an overview of the repository's organization, highlighting how the source code is divided into distinct packages for parsing, network conversion, parameter optimization, and querying.

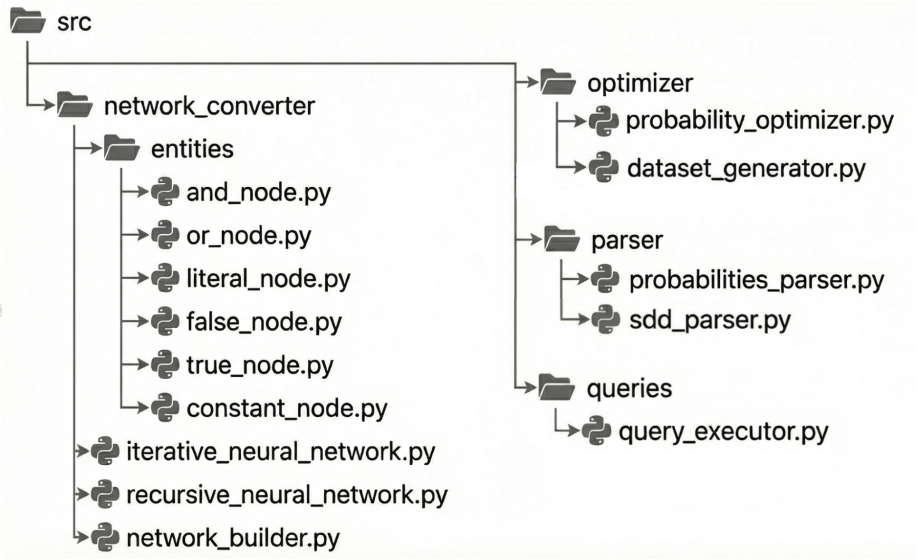

Figure 4.1: *Directory structure of the framework's source code.*

4.2.1 Parser

This package is responsible for ingesting the input files. It contains two main classes:

- **SDDParser:** Reads and parses the .sdd file, constructing an in-memory graph representation of the logical circuit.
- **ProbabilitiesParser:** Reads and parses the .json file. Its primary role is to create a dictionary mapping each probabilistic variable to its given probability. This parser is used in scenarios where probabilities are explicitly provided, rather than learned from a dataset.

4.2.2 Network Converter

This is the central package responsible for the SDD-to-neural-network translation.

Entities

It first defines the core components of the neural circuit, including PyTorch modules for each node type: `LiteralNodeModule`, `ConstantNode`, `TrueNode`, `FalseNode`, `BaseOrNode`, and `BaseAndNode`.

NetworkBuilder

This class performs the recursive, depth-first conversion of the SDD graph into a neural network. It utilizes a `node_cache` dictionary, mapping SDD node IDs to their neural network module counterparts to ensure each node is created only once (memoization). During this traversal, it applies logical optimizations (e.g., an AND node with a `FalseNode` child simplifies to a `FalseNode`) and enforces smoothness by inserting tautological subcircuits.

A key function is wiring the probabilities. The logic for a negated literal $\neg X_i$ is built to compute $P(\neg X_i)$ using only sum and product nodes, as shown in Program 4.3.

```
1    def _create_negated_literal_node(self, node_id, literal_node,
         probability_node):
2        # Implements 1 + (-1 * P(Xi))
3        and_node = AndNode([probability_node, ConstantNode(-1.0)])
4        or_node = OrNode([and_node, ConstantNode(1.0)])
5        # Final result is (1 - P(Xi)) * literal_input
6        return AndNode([or_node, literal_node], node_id=node_id)
```

Program 4.3: *Dynamic creation of negation nodes using arithmetic operations*

Forward Pass Implementations

The package provides two different, but functionally equivalent, mechanisms for executing the network's forward pass:

IterativeNN: Implements a bottom-up, iterative computation. In its constructor, it performs a single topological sort of the network graph (via a post-order traversal) and stores this in an `execution_order` list. The forward pass then becomes a simple, efficient loop over this pre-computed list, storing intermediate results in a dictionary.

RecursiveNN: Implements a top-down, recursive computation. Its `forward` method initiates a recursive call from the root node. To ensure (like the `NetworkBuilder`) that each node is computed only once per pass, it passes a `memoization_cache` dictionary down through the recursive calls.

Program 4.4 demonstrates the topological sort logic used by the `IterativeNN`.

```
1    def _topological_sort(self):
2        """
3        Performs a topological sort of the nodes in the network.
4        Returns a list of nodes in the order they should be executed (
             leaves first).
5        """
6        sorted_nodes = []
```

```
7       visited = set()
8
9       def visit(node):
10          if node in visited:
11              return
12          visited.add(node)
13          # Recursively visit children first for post-order traversal
14          if hasattr(node, 'children_nodes'):
15              for child in node.children_nodes:
16                  visit(child)
17          # Add the node to the list after all its children have been
                added
18          sorted_nodes.append(node)
19
20      visit(self.root)
21      return sorted_nodes
```

Program 4.4: *Topological sort for linearizing the execution graph*

4.2.3 Queries

This package handles probabilistic inference. The **QueryExecutor** class computes conditional probabilistic queries. A user provides a list of query variables (Q) and evidence variables (E). The class then computes the conditional probability $P(Q \mid E)$ by leveraging the formula:

$$P(Q \mid E) = \frac{P(Q, E)}{P(E)}$$

It performs two separate forward passes – one for the joint probability of query and evidence $P(Q, E)$ and one for the probability of the evidence $P(E)$ – and returns the result of their division.

The core of its logic lies in constructing the two input tensors, which it does by initializing both to all 1s (for marginalization) and then setting specific indices to 0 based on the query and evidence.

4.2.4 Optimizer

This package implements the training logic. The **ProbabilityOptimizer** class is responsible for probabilistic optimization. Given a list of variables whose probabilities are to be learned and a dataset, the class dynamically injects learnable parameters into the network. It does not add new nodes; instead, it replaces the static probability value inside a `ConstantNode` with a `torch.nn.Parameter`.

It then executes the training loop, using Stochastic Gradient Descent (SGD) for optimization and the Negative Log-Likelihood (NLL) as the loss function. The entire process relies on PyTorch's automatic differentiation (`autograd`) engine, which automatically

computes the gradients of the loss with respect to the learnable parameters, eliminating the need for manual gradient derivation.

Program 4.5 illustrates this parameter injection logic.

```
1    def _add_learnable_parameters_to_network(self, literals):
2        """
3        Adds learnable parameters to the network for a given list of
             literals.
4        """
5        for literal in literals:
6            # ...
7            prob_node = self.literal_to_prob_node[literal]
8            # Initialize logit to 0.0 (which corresponds to a probability of
                 0.5)
9            learnable_parameter = nn.Parameter(torch.tensor(0.0))
10           # Replace the node's static value with the learnable parameter
11           prob_node.set_constant(learnable_parameter)
```

Program 4.5: *Injecting learnable parameters into the network*

4.3 Software Engineering Details

The framework was developed following an object-oriented design, with a clear separation of concerns among the different classes and packages.

Inheritance: Class inheritance is used to promote code reuse and maintain a logical hierarchy. This is most evident in the node classes, where `BaseANDNode` and `BaseORNode` define abstract classes, which are then implemented by specific recursive (`RecursiveANDNode`, `RecursiveORNode`) and iterative (`IterativeANDNode`, `IterativeORNode`) versions.

Testing: The reliability of the framework is ensured by a comprehensive test suite. Using Pytest, 149 automated tests were implemented, including both fine-grained unit tests for individual classes and methods, as well as broader integration tests that validate the correctness of the entire pipeline.

Documentation: All classes and public methods are meticulously documented with docstrings, clearly specifying their purpose, input parameters (and their types), and return values. This focus on documentation enhances code readability and maintainability.

Chapter 5

Results

This section evaluates the success of the project by mapping the outcomes to its primary research goals. The framework was evaluated on its ability to perform accurate inference, learn probabilistic parameters, and provide a high-quality, reusable software architecture.

5.1 Case Study: The Alarm Problem

To provide concrete results, the well-known "Alarm" Bayesian network was used as a case study. The problem is defined by a set of logical rules and probabilistic facts, represented in the following `.pasp` program. As shown in Program 5.1, the program defines both the probabilistic priors and the logical implications.

```
% ground probabilistic facts
0.1::burglary.
0.2::earthquake.
0.7::hears_alarm(john).

% ground rules
alarm :- burglary.
alarm :- earthquake.
calls(john) :- alarm, hears_alarm(john).
```

Program 5.1: *The Alarm problem definition in .pasp format*

This program defines five variables (1: burglary, 2: earthquake, 3: hears_alarm(john), 4: alarm, 5: calls(john)) where the first three are probabilistic and the latter two are logical. This program was compiled into the `alarm_balanced.sdd` and `alarm.json` files, which served as the inputs for the framework. They can be found in the "Appendix" section.

5.2 Probabilistic Inference

Goal: Convert SDDs to neural networks for efficient probabilistic inference.

This goal was successfully achieved. The `NetworkBuilder` class correctly parsed the SDD and JSON files, converting the 19-node SDD into a PyTorch neural network module. The `QueryExecutor` was then used to perform conditional probability queries, demonstrating the model's inference capabilities.

The inference process is highly efficient. By leveraging the compiled Arithmetic Circuit, each query requires only two forward passes (one for the numerator $P(Q, E)$ and one for the denominator $P(E)$, regardless of the query's complexity).

Table 5.1 presents a sample of queries performed on the alarm network and their results:

Query	Probability	Time (ms)
P(calls(john) | burglary)	0.700	0.448
P(burglary | calls(john))	0.357	0.399
P(earthquake | calls(john))	0.714	0.386
P(burglary | calls(john), earthquake)	0.100	0.422
P(alarm | burglary, earthquake)	1.000	0.398
P(alarm)	0.280	0.395
P(burglary, earthquake, calls(john))	0.014	0.384

Table 5.1: *Inference results on the Alarm Bayesian Network*

These results confirm that the framework can successfully convert a logical-probabilistic problem into a neural network and perform interesting inference tasks efficiently.

5.3 Probabilistic Learning

Goal: Utilize deep learning frameworks (PyTorch) for optimization.

This goal was achieved and rigorously verified through a comprehensive grid-search experiment. To assess the consistency and validity of the learning mechanism, we executed a set of experiments varying both the dataset size (`num_samples`) and the training duration (`num_epochs`).

5.3.1 Dataset Generation

A dataset was generated using the `AlarmDatasetGenerator` class. This generation process is crucial because it simulates the semi-supervised nature of many real-world problems: we often observe the consequences (logical variables) but rarely observe the underlying causes (probabilistic variables).

First, the generator samples the probabilistic "root" variables (burglary, earthquake, hears_alarm) according to their true probabilities. Then, it deterministically computes the logical "child" variables (alarm, calls) based on the rules, ensuring logical consistency. Program 5.2 shows how these logical implications are enforced during generation.

```
    def _add_logical_variables_to_map(self,
        literal_to_instantiated_value_map):
       # ... (omitted for brevity)
       if burglary or earthquake:
          alarm = 1
       if alarm and hears_alarm:
          calls = 1
       literal_to_instantiated_value_map[4] = alarm
       literal_to_instantiated_value_map[5] = calls
```

Program 5.2: *Enforcing logical consistency during dataset generation*

Crucially, the dataset is constructed to be partially observable. In 80% of the samples, the probabilistic variables (indices 1, 2, 3) are hidden (marginalized out). In the input tensor, this is represented by setting both the literal and its negation to 1. The model is thus forced to learn the latent probabilities of burglary and earthquake based solely on the evidence provided by alarm and calls.

5.3.2 Experimental Methodology

We defined the following grid of hyperparameters:

- **Number of Samples:** [1,000, 5,000, 10,000, 50,000, 100,000]
- **Number of Epochs:** [1,000, 2,000, 5,000, 10,000, 20,000]

Every combination of samples and epochs was executed 10 times to account for stochastic variations in dataset generation and random weight initialization. For each run, we trained the alarm network from scratch using a learning rate of 0.01. We measured:

1. **Average Mean Absolute Error (MAE):** The deviation of learned probabilities from the true values (0.1, 0.2, 0.7).
2. **Standard Error of MAE:** To measure the stability of the training.
3. **Average Training Time:** To assess computational cost.

5.3.3 Experimental Results

The results in Table 5.2 demonstrate a clear correlation: increasing the dataset size significantly reduces the error, while increasing epochs yields diminishing returns after a certain point. It is also evident that larger sample sizes produce smaller standard errors, reflecting reduced variability as more data are incorporated. Overall, the framework demonstrates strong robustness, with the low standard errors indicating stable and consistent convergence.

To further illustrate the precision of the framework, we analyzed the predicted values for the two worst and two best performing combinations. Table 5.3 displays the learned probabilities for the three probabilistic variables. The values are reported as intervals of *Average Prediction* $\pm$ *Standard Error* over 10 runs.

Num Samples	Num Epochs	Avg. MAE	Std. Error	Avg. Loss	Avg. Time (ms)
1,000	1,000	0.0179	0.0054	0.9141	1,010
5,000	2,000	0.0045	0.0023	0.9009	4,735
5,000	5,000	0.0058	0.0020	0.8994	11,506
10,000	5,000	0.0063	0.0016	0.9000	19,321
10,000	20,000	0.0045	0.0019	0.8984	80,335
50,000	10,000	0.0017	0.0005	0.8970	78,622
50,000	20,000	0.0017	0.0006	0.8987	139,622
100,000	10,000	0.0013	0.0004	0.8989	107,696
100,000	20,000	0.0017	0.0006	0.8994	245,248

Note: A subset of the grid search results is shown for brevity.

Table 5.2: *Grid search results: Impact of dataset size and training duration*

Samples / Epochs	Burglary (True: 0.1)	Earthquake (True: 0.2)	Hears Alarm (True: 0.7)	Avg. Time (ms)
Worst Configurations				
1,000 / 1,000	0.114 ± 0.019	0.197 ± 0.019	0.697 ± 0.020	1,010
10,000 / 5,000	0.105 ± 0.005	0.195 ± 0.003	0.697 ± 0.006	19,321
Best Configurations				
50,000 / 10,000	0.099 ± 0.001	0.199 ± 0.001	0.700 ± 0.002	78,622
100,000 / 10,000	0.100 ± 0.001	0.199 ± 0.001	0.699 ± 0.001	107,696

Table 5.3: *Detailed predictions for worst and best hyperparameter configurations*

These detailed predictions reveal that even in the worst-case scenarios, the model learns the correct direction of the probabilities, though with higher variance. However, with sufficient data (50,000 samples), the model converges to the true parameters with remarkable precision and negligible variance, effectively recovering the underlying probabilistic facts hidden within the data. Consequently, the optimal choice of hyperparameters depends on the required level of precision, given the clear trade-off between accuracy and computational cost.

5.4 Framework and Software Engineering

Goal: Develop a reusable, extensible, and reliable software framework.

This goal was a central focus of the implementation and was achieved through careful architectural design, adherence to software engineering principles, and comprehensive testing.

Reliability: The framework's robustness is ensured by a suite of 149 automated tests implemented in Pytest. This suite includes unit tests for individual classes and methods, as well as integration tests that validate entire workflows, such as parsing the `alarm.sdd` file, computing inferences, and checking for correct exception handling. External-facing classes, like `QueryExecutor`, include robust validation to provide clear, readable error messages for invalid inputs.

Modularity and Reusability: The codebase is highly modular, consisting of 26 distinct classes, each with a specific, well-defined purpose (e.g., `SDDParser`, `NetworkBuilder`, `ProbabilityOptimizer`). This separation of concerns prevents monolithic, unmanageable scripts and promotes code reuse.

Extensibility and Design Principles: The framework was designed following SOLID principles.

- **Single Responsibility Principle (SRP):** Each class has one primary responsibility. `SDDParser` only parses SDDs, `NetworkBuilder` only converts the graph, and `QueryExecutor` only runs queries.
- **Open/Closed Principle (OCP):** The framework is open to extension but closed for modification. This is best demonstrated by the use of `BaseANDNode` and `BaseORNode` superclasses. The `NetworkBuilder` is coded against these base classes. This allows new implementations (e.g., a future `CUDABasedANDNode`) to be added to the system without requiring any modification to the core `NetworkBuilder` logic.
- **Dependency Inversion Principle (DIP):** High-level modules do not depend on low-level modules; both depend on abstractions. This is implemented via dependency injection in the `NetworkBuilder`'s constructor, which receives the classes (`or_node_class` and `and_node_class`) to use. This inverts the control flow, allowing the `NetworkBuilder` to be configured with any concrete node implementation (like `IterativeORNode` or `RecursiveORNode`) that adheres to the base abstraction, rather than depending directly on a specific implementation.

Efficiency: Computational efficiency was a key consideration. The `NetworkBuilder` uses memoization to ensure that each unique SDD node is converted to a neural module only once. Similarly, the `RecursiveNN` implementation uses a `memoization_cache` during its forward pass to avoid redundant computations on shared sub-circuits, effectively implementing dynamic programming.

Maintainability: All public classes and methods are fully documented with docstrings, detailing their purpose, parameters, and return values. This, combined with the modular design and lack of duplicated code, results in a framework that is highly maintainable and can be expanded by other developers.

Chapter 6

Conclusion

This work has successfully demonstrated a novel methodology for bridging the gap between symbolic logic and deep learning. By converting Sentential Decision Diagrams (SDDs) into differentiable neural networks, we have developed a system that not only preserves the rigorous semantics of logical circuits but also enables the seamless integration of probabilistic parameters and gradient-based optimization.

6.1 Contributions

The primary contributions of this work are threefold, directly addressing the goals outlined at the outset:

Differentiable Logic Framework: We established a robust pipeline for converting static symbolic circuits into dynamic PyTorch modules. This allows logical circuits to be treated as a computation graph, enabling standard backpropagation to optimize probabilistic weights associated with logical atoms.

Unified Inference and Learning: The system unifies probabilistic inference (via Weighted Model Counting) and parameter learning into a single architecture. As demonstrated by the experimental results on the "Alarm" network, the framework allows for efficient queries and accurate recovery of latent probabilities from partially observed data.

Software Engineering Excellence: Beyond the theoretical model, we delivered a production-grade software library. Built on SOLID principles, with extensive testing and modular design, the framework is designed to be a reliable foundation for future research in Neuro-Symbolic AI.

6.2 Challenges Faced

The development of this system was not without significant challenges. The intersection of logic circuits and neural networks is a highly theoretical domain with a steep learning curve. One of the most critical technical hurdles was the realization that standard

SDDs are not inherently suitable for the specific inference logic implemented in this work. Specifically, we discovered that for the probabilistic inference logic to be mathematically valid, the circuit must satisfy the property of **smoothness**. This required the implementation of a graph transformation step within the `NetworkBuilder` to enforce smoothness by injecting tautological subcircuits.

6.3 Next Steps

While this work provides a solid foundation, there are several promising avenues for future research and extension:

Complex Probabilistic Primitives: The current system handles binary probabilistic variables. A natural extension is to support Annotated Disjunctions, where a single logical event can have multiple mutually exclusive outcomes (e.g., a die roll), each with an associated probability summing to one.

Non-Stratified Programs: The current framework assumes acyclic dependencies. Extending the logic to handle Non-Stratified Programs – logic programs containing cycles or recursion involving negation – would significantly broaden the scope of problems the system can represent.

Architectural Optimization: To further improve computational efficiency, particularly for very large circuits, we can explore advanced neural architectures. Approaches like **KLAY** (Yu *et al.*, 2024) propose layering nodes of the same type (Sum/Product) to maximize parallel execution on GPUs. Integrating such architectural optimizations into our `NetworkBuilder` would be a valuable step toward scaling the system to industrial-sized problems.

Appendix A

Alarm Problem Input Files

This appendix contains the complete source files used for the “Alarm” case study presented in Chapter 5. These files serve as the inputs for the framework, defining the probabilistic rules, the compiled logical circuit, and the semantic metadata.

A.1 alarm.pasp

The following file defines the probabilistic logic program using the PASP syntax. It declares the probabilistic facts (priors) and the logical rules.

```
% ground probabilistic facts
0.1::burglary.
0.2::earthquake.
0.7::hears_alarm(john).

% ground rules
alarm :- burglary.
alarm :- earthquake.
calls(john) :- alarm, hears_alarm(john).
```

Program A.1: *alarm.pasp source file*

A.2 alarm.sdd

The following is the Sentential Decision Diagram (SDD) compiled from the logic rules. It represents the logical structure of the problem in the standard text-based SDD format.

```
sdd 19
L 2 0 -1
L 3 2 2
L 4 0 1
T 5
D 1 1 2 2 3 4 5
```

```
L 7 4 -3
L 9 6 4
L 10 8 -5
L 11 6 -4
F 12
D 8 7 2 9 10 11 12
L 13 4 3
L 15 8 5
D 14 7 2 9 15 11 12
D 6 5 2 7 8 13 14
L 17 2 -2
D 16 1 2 2 17 4 12
D 18 7 2 11 10 9 12
D 0 3 2 1 6 16 18
```

Program A.2: *alarm.sdd compiled circuit*

A.3 alarm.json

The JSON file provides the semantic mapping between the numeric IDs used in the SDD and the human-readable variable names. It also defines the initial probabilities and the structure of the logic rules for the parser.

```
{
   "atom_mapping": {
      "1": "burglary",
      "2": "earthquake",
      "3": "hears_alarm(john)",
      "4": "alarm",
      "5": "calls(john)"
   },
   "rules": {
      "normal": [
         {
            "head": 4,
            "body": {
               "pos": [ 1 ],
               "neg": []
            },
            "text": "alarm#4:-burglary#1."
         },
         {
            "head": 4,
            "body": {
               "pos": [ 2 ],
               "neg": []
            },
            "text": "alarm#4:-earthquake#2."
         },
         {
            "head": 5,
            "body": {
```

```
                  "pos": [ 4, 3 ],
                  "neg": []
               },
               "text": "calls(john)#5:-alarm#4,hears_alarm(john)#3."
            }
         ],
         "disjunctive": [],
         "choice": []
      },
      "head_rules": {
         "4": [
            { "pos_body": [ 1 ], "neg_body": [] },
            { "pos_body": [ 2 ], "neg_body": [] }
         ],
         "5": [
            { "pos_body": [ 4, 3 ], "neg_body": [] }
         ]
      },
      "metadata": {
         "num_atoms": 5,
         "num_rules": 3,
         "num_pfacts": 3,
         "num_ads": 0,
         "num_drules": 0,
         "num_crules": 0
      },
      "prob": {
         "pfacts": [
            [ 1, 0.1 ],
            [ 2, 0.2 ],
            [ 3, 0.7 ]
         ],
         "ads": [],
         "pvars": [ 1, 2, 3 ]
      },
      "exactly_one_constraints": [],
      "loops": [],
      "loop_formulas": []
   }
```

Program A.3: *alarm.json metadata and configuration*

References

[Chavira and Darwiche 2008] Mark Chavira and Adnan Darwiche. "On probabilistic inference by weighted model counting". *Artificial Intelligence* 172.6–7 (2008), pp. 772–799 (cit. on pp. 8, 15).

[Choi *et al.* 2020] YooJung Choi, Antonio Vergari, and Guy Van den Broeck. "Probabilistic circuits: a unifying framework for tractable probabilistic models". *Frontiers in Artificial Intelligence* 3.11 (2020) (cit. on p. 13).

[Clark 1978] Keith L. Clark. "Negation as failure". *Logic and Data Bases* (1978), pp. 293–322 (cit. on p. 10).

[Darwiche 1999] Adnan Darwiche. "On the tractable counting of theory models and its application to belief revision and truth maintenance". *Journal of Applied Non-Classical Logics* 9.1 (1999), pp. 37–60 (cit. on p. 13).

[Darwiche 2002] Adnan Darwiche. "A knowledge compilation map". *Journal of Artificial Intelligence Research* 17 (2002), pp. 229–264. doi: 10.1613/jair.989 (cit. on p. 5).

[Darwiche 2003] Adnan Darwiche. "A differential approach to inference in bayesian networks". *Journal of the ACM* 50.3 (2003), pp. 280–305 (cit. on pp. 13, 14).

[Darwiche 2011] Adnan Darwiche. "SDD: a new canonical representation of propositional knowledge bases". In: *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI)*. 2011, pp. 819–826 (cit. on pp. 7, 13).

[Garcez *et al.* 2019] Artur S. d'Avila Garcez, Tarek R. Besold, and Luis C. Lamb. "Neural-symbolic learning and reasoning: contributions and challenges". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33. 01. 2019, pp. 9489–9496 (cit. on p. 12).

[Geh *et al.* 2023] Renato L. Geh, Jonas Gonçalves, Igor C. Silveira, Denis D. Mauá, and Fabio G. Cozman. "dPASP: a comprehensive differentiable probabilistic answer set programming environment for neurosymbolic learning and reasoning". *arXiv preprint arXiv:2308.02944* (2023). url: https://arxiv.org/abs/2308.02944 (cit. on p. 1).

[Gelfond and Lifschitz 1988] Michael Gelfond and Vladimir Lifschitz. "The stable model semantics for logic programming". In: *Proceedings of the 5th International Conference on Logic Programming*. 1988, pp. 1070–1080 (cit. on p. 9).

[Lifschitz 2008] Vladimir Lifschitz. "What is answer set programming?" *Proceedings of the AAAI Conference on Artificial Intelligence* 23.1 (2008), pp. 1594–1597 (cit. on p. 9).

[Lifschitz 2019] Vladimir Lifschitz. *Answer Set Programming*. Berlin, Heidelberg: Springer-Verlag, 2019 (cit. on p. 9).

[Paszke *et al.* 2019] Adam Paszke *et al.* "Pytorch: an imperative style, high-performance deep learning library". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035 (cit. on p. 12).

[Valiant 1979] Leslie G. Valiant. "The complexity of computing the permanent". *Theoretical Computer Science* 8.2 (1979), pp. 189–201 (cit. on p. 11).

[Yu *et al.* 2024] Honghua Yu, Zichao Yang, Ji Liu, and Haifeng Wang. "KLAY: accelerating arithmetic circuits for neurosymbolic AI". *arXiv preprint arXiv:2406.04276* (2024). url: https://arxiv.org/abs/2406.04276 (cit. on p. 34).