Master of Science in Informatics at Grenoble
Master Informatique
Specialization DSAI

Neuro-Symbolic Techniques on Graphs

Guillaume Delplanque

June 14th, 2023

Research project performed at INRIA

Under the supervision of:
Nabil Layaïda
Pierre Genevès

Defended before a jury composed of:
Head of the jury
Emilie Devijver
Damien Graux

June 2023

Abstract

This report explores the field of neuro-symbolic artificial intelligence (AI), which aims to integrate the strengths of symbolic reasoning and neural network-based approaches. The motivation behind this integration arises from the unique capabilities and limitations of each approach. Neural networks excel in learning complex patterns from raw data but lack interpretability, while symbolic AI provides explicit knowledge representations but struggles with unstructured data. Effective integration requires innovative methodologies and model architectures that enable seamless communication between symbolic and neural components.

The report begins with an overview of deep learning, logical programming, and graphs, providing background for understanding neuro-symbolic methods. A taxonomy of neuro-symbolic approaches is presented, categorizing them based on their representation systems, integration methods, and learning algorithms. The report then focuses on the implementation of a neuro-symbolic system using Scallop, a logic program enhanced with a neural network.

The objective is to replicate and compare the results achieved by the Knowledge-Enhanced Neural Networks (KENN) framework on a node classification task. The implementation involves defining the neural network architecture, creating a symbolic program, establishing the connection between them, and incorporating weights into the training process. While KENN is specialized for classification tasks, Scallop offers a more versatile approach that can be applied to a wider range of tasks.

Acknowledgement

I would like to express my sincere gratitude to all the individuals who have contributed, directly or indirectly, to the completion of this work.

First and foremost, my supervisors, Pierre Genevès and Nabil Layaïda, for their guidance, valuable advice, and expertise. Their invaluable insights have helped me progress and successfully complete this project.

I am also deeply grateful to my colleagues, Pauline Attal, Luisa Werner, Sarah Chlyah, Fejza Amela and Chandan Sharma, for their support and collaboration throughout this project. Besides their valuable ideas, discussions, and constructive feedback, their camaraderie has greatly enriched my experience and made the internship enjoyable.

Résumé

Ce rapport explore le domaine de l'intelligence artificielle neuro-symbolique, qui vise à intégrer les forces du raisonnement symbolique et des approches basées sur les réseaux neuronaux. La motivation derrière cette intégration découle des capacités et des limites uniques de chaque approche. Les réseaux neuronaux excellent dans l'apprentissage de modèles complexes à partir de données brutes, mais manquent d'interprétabilité, tandis que l'IA symbolique fournit des représentations explicites de connaissances mais rencontre des difficultés avec les données

non structurées. Pour une intègration efficace, il est nécessaire d'utiliser des méthodologies innovantes et des architectures permettant une communication fluide entre les composants symboliques et neuronaux.

Le rapport commence par une vue d'ensemble de l'apprentissage profond, de la programmation logique et des graphes, offrant ainsi les bases nécessaires pour comprendre les méthodes neuro-symboliques. Une taxonomie des approches neuro-symboliques est présentée, les classant en fonction de leur système de représentation, de leur méthode d'intégration et de leur algorithme d'apprentissage. Le rapport se concentre ensuite sur la mise en œuvre d'un système neuro-symbolique utilisant Scallop, un programme logique amélioré par un réseau neuronal.

L'objectif est de reproduire et de comparer les résultats obtenus par le cadre des "Knowledge-Enhanced Neural Networks" (KENN) dans une tâche de classification des nœuds. La mise en œuvre implique la définition de l'architecture du réseau neuronal, la création d'un programme symbolic, l'établissement de la connexion entre les deux et l'incorporation de poids dans le processus d'apprentissage. Alors que KENN est spécialisé pour les tâches de classification, Scallop offre une approche plus polyvalente applicable à un éventail plus large de tâches.

Contents

— 1 —
Motivation

The field of artificial intelligence (AI) has witnessed remarkable advancements in two key approaches: symbolic AI, which utilizes logical rules and reasoning mechanisms to make decisions, and sub-symbolic AI, which encompasses neural network-based approaches and other techniques like reinforcement learning. These approaches bring unique strengths and limitations to the table, posing challenges when trying to reconcile their fundamental differences. In response, a promising paradigm known as neuro-symbolic AI has emerged, aiming to integrate the strengths of symbolic reasoning with the representation learning capabilities of neural networks. This introduction explores each approach individually, highlights the need for neuro-symbolic AI, and addresses the challenges involved in achieving this integration.

Neural networks, as a prominent example of sub-symbolic AI, have revolutionized AI by excelling in complex tasks such as image recognition and natural language processing. Their key advantage lies in their ability to learn complex patterns from raw data, eliminating the need for explicit feature engineering. However, neural networks also come with inherent limitations. One primary concern is their lack of explainability and interpretability. Deep learning models often function as black boxes, making it challenging to understand their decision-making processes. Furthermore, neural networks typically require vast amounts of labeled training data to achieve optimal performance, making them data-hungry and potentially impractical in domains where data is limited or costly to acquire.

Symbolic AI, on the other hand, emphasizes logical reasoning and rule-based systems. Symbolic models excel in capturing explicit knowledge representations and providing interpretable outputs. This interpretability makes them well-suited for domains where explainability is crucial, such as healthcare or legal applications. However, symbolic AI has its own limitations. These models heavily rely on human-engineered features and explicit rule creation, which can be labor-intensive and may struggle to handle complex patterns in unstructured data. This limits their applicability in domains requiring large-scale, data-driven decision-making.

To bridge the gap between symbolic and neural network-based AI, neuro-symbolic AI has emerged as a promising paradigm. By integrating symbolic reasoning with the representation learning capabilities of neural networks, neuro-symbolic AI aims to leverage the strengths of both approaches. This integration can enable interpretable models that learn complex patterns from raw data, thus offering the best of both worlds.

Integrating symbolic reasoning and neural network-based learning poses substantial challenges. How can we effectively combine the logical reasoning of symbolic AI with the representation learning capabilities of neural networks? Can we develop neuro-symbolic methods that retain the interpretability and explainability of symbolic systems while harnessing the

power of neural networks?

As the two mathematical worlds are far from each other and use different techniques, achieving effective integration requires novel methodologies and innovative model architectures that enable seamless communication between symbolic and neural components. Striking a balance between interpretability and representation power becomes crucial. Developing neuro-symbolic methods that provide explainable outputs while harnessing the representation capacity of neural networks remains an ongoing challenge.

The report will begin with a detailed overview of neuro-symbolic methods, after which it will present the specific contribution of the internship.

— 2 —
State-of-the-Art

2.1 Introduction

This section describes the state-of-the-art in neuro-symbolic integration whose aim is to bridge the gap between symbolic reasoning and neural networks. This report commences with a concise overview of three fundamental notions that form the foundation of neuro-symbolic methods: deep learning, logical programming, and graphs. This overview serves to establish the necessary background knowledge required to grasp the subsequent sections.

After establishing the foundational overview, a taxonomy of diverse neurosymbolic methods will be presented. This taxonomy categorizes these methods based on their distinct characteristics. It encompasses the employed representation formalism, the integration approach used to combine symbolic reasoning and neural networks, and the adopted learning algorithms. The taxonomy facilitates an understanding of the field's diversity and variations.

Key advancements made by these methods will be highlighted, accompanied by discussions on their applications in various domains, as well as an analysis of the challenges and limitations they encounter.

By thoroughly exploring the state-of-the-art in neuro-symbolic methods, this report aims to provide readers with a comprehensive understanding of the current research landscape in this rapidly evolving field. Furthermore, it sets the stage for the subsequent contribution section of this report, where novel insights and advancements will be presented. These contributions serve to further develop and apply neuro-symbolic methods.

2.2 Basics

2.2.1 Basics of Deep Learning

For complex tasks, such as computer vision, classical algorithms often struggle, while humans excel effortlessly. This disparity prompted the development of neural networks, initially inspired by the human brain. In recent years, significant advancements have transformed neural networks into one of the leading state-of-the-art machine learning methods. To gain a deeper understanding of neuro-symbolic approaches, it is crucial to first define neural networks. This section provides a comprehensive explanation of neural networks within the context of deep learning. Definitions of terms and network composition are presented, alongside discussions on their applications and distinctions from other machine learning methods.

Deep Learning

Deep learning [18] is a subcategory of machine learning that leverages neural networks to learn and make accurate predictions or decisions. Neural networks are composed of interconnected nodes, resembling the structure of neurons in the human brain. Through the use of extensive datasets, these networks can discern patterns and continually improve their predictions or decisions over time.

A formal neuron within a neural network can be defined as a mathematical function. This function takes input from other neurons, typically encompassing all neurons in the preceding layer, and produces an aggregation of these inputs multiplied by specific weights. Each neuron is associated with a set of n weights, which are subject to modification during the learning phase. Moreover, a neuron incorporates an "activation function" denoted as ϕ, which transforms the weighted sum of inputs into another single output value. To provide a visual representation, Figure 2.1 illustrates an example of a formal neuron from [17]. The term **activation** refers to the resulting output value, denoted as o_j, whereas **pre-activation** signifies the output of the weighted sum prior to the application of the activation function.

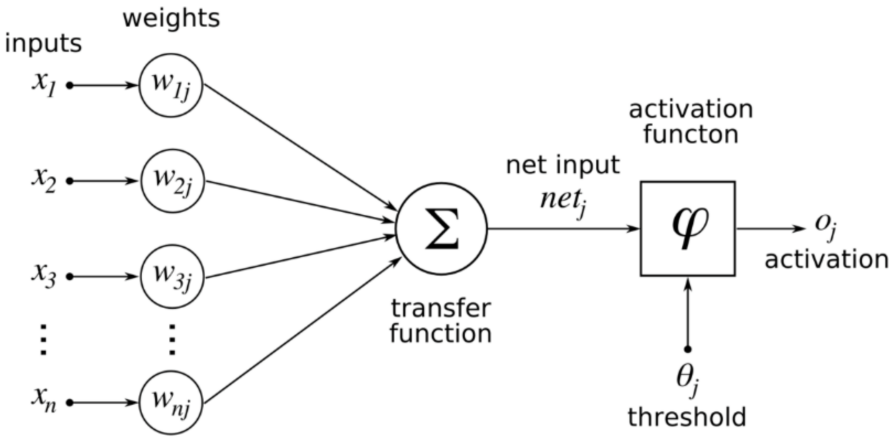

Figure 2.1: representation of a formal neuron

This report only focuses on supervised learning, which is a machine learning approach where a model is trained using labeled examples. The goal is to impart knowledge to the model though tangible examples. So we illustrate deep learning through an example of supervised learning: let's consider a labeled dataset comprising pairs of input (X) and corresponding output (y). The input (X) can take the form of a vector, matrix, sequence, or tensor. A deep learning neural network is essentially a function that learns the relationship between the input (X) and the desired output (y). When a new input (X) is provided to the function, it aims to generate an output (y') that closely matches the associated y. This can be represented as an optimization problem, for example with the square loss:

$$\texttt{argmin}_f \sum_{x \in X} (f(x) - y)^2$$

X representing any input vector from the dataset, and f representing the neural network function, so $f(x) = y'$. But how can a neural network "learn" to minimize this value?

To "learn" means to update the weights of the neural networks to make it the most accurate possible. To do that, a loss function and Stochastic Gradient Descent (SGD) method can be employed. The loss function quantifies the difference between the predicted output (y') and the desired output (y) (which is the optimization problem needed to be solved), allowing the neural network to adjust its weights through backpropagation.

The backpropagation is often realised with Stochastic Gradient Descent (SGD) [24]. SGD is an optimization algorithm widely used in deep learning to train neural networks. It involves updating the network's weights based on the computed gradients of the loss function. Here's a simplified explanation of how SGD works:

1. Initialization: The neural network's weights are randomly assigned, leading to initial predictions (y') that are far from the desired output (y).

2. Forward Pass: The network processes the inputs through its layers, generating predictions based on the current weights.

3. Loss Calculation: Using the predictions, the loss function is computed, quantifying the difference between the predicted outputs and the true labels.

4. Backpropagation: The gradients of the weights with respect to the loss are computed using the chain rule of calculus, indicating the direction and magnitude of weight updates needed to minimize the loss. To compute the gradients of the weights, the intermediate values and activation computed at each layer during forward pass step are needed, and so they need to be saved.

5. Weight Update: The weights are adjusted based on the computed gradients and a learning rate, which controls the step size in the direction opposite to the gradients.

6. Iteration: Steps 2 to 5 are repeated for multiple training examples until the network has processed all the examples in the dataset. Each complete pass through the dataset is called an epoch.

SGD is "stochastic" because it updates the weights based on individual training examples, rather than using the entire dataset at once. This randomness introduces noise into the weight updates, which can help the network avoid local optima and find better solutions. However, it also leads to more erratic learning compared to other optimization algorithms. Overall, SGD optimizes the neural network's weights by iteratively updating them based on the computed gradients. It is a fundamental part of training deep learning models, allowing them to learn from data and improve their predictions over time.

Furthermore, to address the issue of overfitting and improve the generalization performance of neural networks, **regularization** techniques are commonly employed. **Overfitting** occurs when a model becomes too complex and starts to fit the training data too closely, capturing noise and irrelevant patterns. As a result, the model may fail to generalize well to unseen data and exhibit poor performance on new inputs. Regularization serves as a countermeasure against overfitting.

Regularization involves adding an additional term, known as a regularization term, to the loss function during training. This term penalizes complex or large weights in the network, encouraging simpler weight configurations that are less prone to overfitting. One widely used regularization technique is L2 regularization, also referred to as weight decay. It adds a term

proportional to the squared magnitude of the weights to the original loss function. The strength of regularization is controlled by a parameter λ, which can be adjusted to balance the impact of regularization. It can be defined as a sum of all the weights w_i, to make them as small as possible:

$$\text{Regularization term } = \frac{\lambda}{2}\sum_i w_i^2$$

The overall optimization problem incorporates the regularization term by combining it with the original loss function which depends on the dataset values x and y. The complete loss function becomes:

$$\text{Loss } = \texttt{argmin}_f \sum_{x \in X} (f(x) - y)^2 + \frac{\lambda}{2}\sum_i w_i^2$$

This hyperparameter determines the trade-off between fitting the training data and keeping the weights small to prevent overfitting. By employing regularization techniques, neural networks can achieve improved generalization performance and more robust predictions.

Specialized Neural Network Architectures

Classical neural networks, also known as feedforward neural networks, struggle with capturing the unique properties of certain data types, as each layer is fully connected with the previous one. For example, when dealing with images, they can't effectively capture local patterns and spatial relationships. Similarly, when working with sequential data like speech or text, they fail to account for temporal dependencies and lose important contextual information. Modeling long sequences and complex dependencies is also challenging for classical neural networks.

Deep neural networks are designed with various architectures to address different tasks and data types. These specialized architectures, such as Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), and Generative Adversarial Networks (GANs), overcome the limitations of classical neural networks. These architectures are designed to capture specific features, temporal dependencies, and even generate new data samples.

1. Convolutional Neural Networks (CNNs) [1]: used for image processing tasks, CNNs capture local patterns by applying filters. They extract meaningful information hierarchically, from simple features (e.g., edges, corners) to complex patterns (e.g., shapes, objects), leveraging the spatial structure of images.

2. Recurrent Neural Networks (RNNs) [14]: Designed for sequential data, RNNs capture temporal dependencies by utilizing feedback connections. They retain contextual information from previous steps and effectively model sequential relationships.

3. Generative Adversarial Networks (GANs) [6]: GANs are a specialized type of deep neural network used for generating new data samples. They consist of two components: a generator network and a discriminator network. The generator network learns to generate synthetic data that resembles the real data, while the discriminator network learns to distinguish between the real and generated data. GANs have been successful in various domains, including image generation, text generation, and data augmentation.

By choosing the appropriate architecture, practitioners can significantly improve the performance and accuracy of neural network models in various domains. These specialized architectures exploit the unique characteristics of different data types, allowing networks to capture and process relevant information effectively.

In conclusion, deep learning stands as a powerful technique that has brought about a revolutionary impact in the field of machine learning. Its capacity to acquire hierarchical representations of data has led to significant advancements across various domains.

However, deep learning also entails certain challenges and limitations that necessitate attention. Notably, it relies heavily on extensive data and resources, while suffering from a significant lack of explainability. Consequently, the integration of deep learning with another method that does not require extensive data and offers high explainability becomes desirable.

Therefore, the following section outlines the fundamental principles of logic programming.

2.2.2 Basics of Logic Programming

Logic programming is a widely used method in computer science that enables machines to reason and represent knowledge effectively. It utilizes logic as knowledge representation and employs inference to manipulate that knowledge.

One of the advantages of logic programming is its inherent understandability. The logical nature of representations makes it intuitive and easy to comprehend. Additionally, logic programming operates independently of external data, reducing the need for extensive data preprocessing. The diagram Figure 2.2 captures the essence of logic programming, visually representing its principles and methodology [3].

However, logic programming does have limitations. One of the main drawbacks is its lack of inherent learning capabilities. Unlike machine learning approaches, logic programming relies on predefined rules and logical deductions, limiting its ability to adapt to new information or dynamically adjust its behavior based on experience. It is also not well-suited for tasks that require human intuition or involve complex, real-world scenarios.

This section begins by presenting a series of essential definitions related to logical languages. It is followed by a explanation of logic programming. Additionally, a compilation of several logical languages utilized in neuro-symbolic applications is provided. Subsequently, the exploration shifts towards fuzzy logic to enable probabilistic programming.

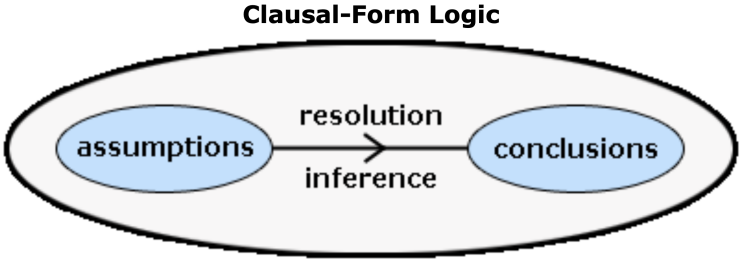

Figure 2.2: Clausal-Form Logic Resolution

Definitions for Logical Languages

A logical language, is a deliberately constructed language designed to embody formal logic. The primary objective of logical languages is to enable or enforce unambiguous statements. While most logical languages are based on predicate logic, they can be built upon any formal logic system.

Among the most well-known logical languages are Loglan, a language designed to eliminate syntactical ambiguity and minimize semantic ambiguity [34], and Prolog, which stands for "Programming in Logic", rooted in predicate logic. Prolog allows for the definition of relations and inference rules [5].

Some vocabulary is needed to understand deeper programming languages and probabilistic programming.

In this logic formula, $\forall x \forall y(\texttt{isFriend}(y,x) \rightarrow \texttt{isFriend}(y,x))$, which means for every couple of person x and y, if x is friend with y, so y is friend with x:

- $\texttt{isFriend}(x,y)$ is an **atom** (any expression of the form $q(t_1,...,t_n)$ is an **atom**)
- x and y are the **terms** of the atom
- **arity** is the number of terms of a predicate
- `isFriend` is a **predicate** (of arity 2)
- a **ground** or **grounded atom** is an expression with arity 0
- a **literal** is an atom or the negation of an atom (so $\texttt{isFriend}(x,y)$ or $\neg\,\texttt{isFriend}(x,y)$)
- a **rule** or a **clause** is an expression of the form h :- $b_1,...,b_n$ where h is an atom and b_i are literals, so for example: `alarm :- earthquake` , which means that if there is an earthquake, the alarm is activated. So, in a rule, :- is the logical implication ($\leftarrow$) and the comma (,) is a conjunction ($\wedge$). And so a rule, or clause, is composed of disjunctions of literals: $c = \bigvee_{i=1}^{k} l_i$. For example, the FOL clause $\neg\,\texttt{Smoker}(x) \vee \neg\,\texttt{Friends}(x,y) \vee \texttt{Smoker}(y)$ signifies that if x is a smoker and x is friends with y, then y is also a smoker. To **ground** these general clauses, specific examples replace the variables x and y, resulting in literals like $\neg\,\texttt{Smoker}(a) \vee \neg\,\texttt{Friends}(a,b) \vee \texttt{Smoker}(b)$. This is called **grounding**.
- a **fact** is a rule with n=0
- The terms t can take different forms:
  - **constant**
  - **variable**
  - **structured term** of the form $f(u_1,...,u_k)$, where f is a **functor** and u_i are terms
- the **domain type** can be constant, variable, functor and predicate. For example, *John* and *Paris* can be from the domain of *person* and *city* And for a functor $f : \mathscr{X} \rightarrow \mathscr{Y}$, the domains $D_{in} = \mathscr{X}$ and $D_{out} = \mathscr{Y}$ are the domain of the arguments and of the range of the functor.
- Besides, in an expression can be several operators, such as $\otimes$ (conjunction), $\oplus$ (disjunction), $\rightarrow$ (implication)...

Logical Languages

In the domain of neuro-symbolic applications, three prominent logical languages are widely used: **Prolog**, **Datalog** and **Real Logic**.

Prolog [5], developed in 1972 by Alain Colmerauer and Philippe Roussel, is a logic programming language rooted in Horn clause logic. It enables the definition of relations, facts, and rules, facilitating logical inference and backtracking. Prolog finds its utility in various fields such as natural language processing, expert systems, constraint solving, symbolic computation, and knowledge representation. It offers a broad spectrum of built-in operations and diverse data types, making it versatile for different tasks.

Datalog [4], a subset of Prolog, functions as a declarative programming language specifically tailored for database queries and deductive database programming. It distinguishes itself with a simpler and more intuitive syntax compared to Prolog, enhancing the querying experience for structured data. Datalog programs establish relationships between data items using rules, and the language encompasses a concise set of built-in operations and data types. Its logical rules make it particularly effective in reasoning about extensive volumes of structured data, finding application in data analytics, data mining, and knowledge graph construction.

Real Logic [11], also known as probabilistic logic programming, is a logical language that combines the principles of logic programming with probabilistic reasoning. It introduces probability distributions into the logical language, enabling the representation and manipulation of uncertain information. Real Logic extends the capabilities of traditional logic-based reasoning by incorporating probabilistic aspects, making it well-suited for applications involving uncertain data or probabilistic modeling.

Unlike Prolog and Datalog, which primarily focus on deterministic reasoning and structured data management, Real Logic seamlessly integrates probabilistic aspects into the logical framework. It introduces a new dimension to reasoning by allowing the expression of uncertain knowledge and facilitating probabilistic inference. Real Logic finds utility in various domains such as machine learning, decision support systems, statistical modeling, and data analysis. By combining the strengths of logic programming and probabilistic reasoning, Real Logic provides a versatile tool for tackling complex problems that require both logical deduction and probabilistic modeling.

In conclusion, Datalog and Prolog are prominent logical languages used in neuro-symbolic applications, with Datalog excelling in deductive database programming and providing a streamlined syntax for querying structured data, while Prolog serves as a versatile general-purpose programming language with a wider range of built-in operations. Additionally, Real Logic combines logic programming with probabilistic reasoning, enabling the handling of uncertain data and probabilistic modeling. These three languages offer distinct capabilities, expanding the domain of reasoning and addressing complex problems in their respective domains.

Logic Programming

Prolog and Datalog are used to create logic programs. A logic program is simply a finite set of rules (h :- $b_1, ..., b_n$, which is a fact when n=0). The goal of the logic program is to determine whether an atom q is true or false, within the underlying model of the program P. This atom is called query.

An example of a Prolog logic program is shown in Listing 2.1. It defines relationships between individuals in terms of being fathers and grandfathers. The program consists of facts and

rules that represent the knowledge base. In the example, we have facts stating the father-child relationships and rules defining the grandfather relationship based on the father and mother relationships. The program also includes a query asking if "John" is a grandfather of "Charles". The logic program is executed by the Prolog interpreter, which tries to find a proof or derivation for the query.

```
isFather("John","Paul").
isFather("Paul","Charles").
isGrandfather(X,Y) :- isFather(X,Z), isFather(Z,Y).
isGrandfather(X,Y) :- isFather(X,Z), isMother(Z,Y).

?- isGrandfather("John","Charles").
```

Listing 2.1: Prolog Code Example

To determine the truth value of a query (its semantics), a set of proofs S_q needs to be constructed, such that q can be derived from S_q. There are two methods to achieve this:

1. Forward method: In this method, we initially tag each input fact f with a proof set $S_f = f$ and then propagate the proofs by applying the rules until a solution for the query is found. The process involves matching the body conditions of the rules with the facts in the knowledge base to derive new facts and expand the proof sets.

2. Backward method: In this method, we start from the query and recursively find the facts or rules from which it can be derived. The process involves applying backward inference and searching for the facts that support the query until we reach the input facts or facts with no further dependencies.

These methods provide different approaches to reasoning in logic programs and can be chosen based on the specific requirements of the problem at hand. By employing logical inference mechanisms, Prolog and Datalog enable to explore and derive new information from the available knowledge in a systematic and logical manner.

Fuzzy Logic and Grounding/Interpretation for Giving Values to Facts

Prolog and Datalog use "boolean" logic, where facts, like `isFather("John","Paul")`, are categorized as either true or false. However, in practical scenarios, uncertainty arises, prompting the preference for assigning probabilities to facts. This necessitates the introduction of "fuzzy" logic, wherein each fact is assigned a truth value represented by a real number ranging from 0 to 1 (0 denoting false and 1 denoting true).

In boolean logic, operations like conjunction are straightforward. For instance, if fact a is false and fact b is true, their conjunction is evaluated as false. However, when dealing with probabilistic facts (e.g., a with probability 0.2 and b with probability 0.6), a new operation called a t-norm is introduced. A t-norm is an operator denoted as $t_{norm} : [0,1],[0,1] \rightarrow [0,1]$ that combines two real numbers into a single real number. It possesses properties similar to the boolean AND operator, including commutativity, associativity, monotonicity, continuity, and neutrality ($t_{norm}(1,x) = x$ and $t_{norm}(0,x) = 0$). In fuzzy logic [31], the conjunction is replaced with a t-norm. Different t-norms exist, such as the minimum function or the product.

For example, considering $a = 0.2$ and $b = 0.6$, selecting the product as the t-norm yields $t_{norm}(a,b) = 0.2 * 0.6 = 0.12$. Alternatively, choosing the minimum would result in 0.2. Likewise, t-conorms are employed to replace the disjunction (OR) operation, fuzzy implication is used in place of implication ($\rightarrow$), and fuzzy negation ($\neg$) substitutes negation in fuzzy logic. All these fuzzy operators are necessary to do reasoning about probability facts, what is called probabilistic reasoning.

Probabilistic Reasoning and Programming

Probabilistic reasoning is a crucial concept in artificial intelligence that complements the capabilities of logical programming languages such as Prolog and Datalog. While these languages excel in deterministic reasoning and structured data management, probabilistic reasoning introduces uncertainty and enables reasoning with probabilistic information. By assigning probabilities to different events or outcomes, probabilistic reasoning allows for the quantification and analysis of uncertainty.

Probabilistic programming plays a significant role within the scope of probabilistic reasoning by providing a dedicated framework for modeling and inference in probabilistic systems. It goes beyond traditional logical programming by allowing the representation of uncertain or probabilistic information through the use of probabilistic programs. These programs associate probabilities with different events or outcomes, enabling the expression of the inherent uncertainty in the system.

For instance, consider the example of a Prolog program with a probability extension in Listing 2.2. In this example, `isGrandfather("John","Charles")` is true only if `isFather("John","Paul")` and `isFather("John","Paul")` are both true. To compute the final query, a t-norm is applied to the probabilities.

```
0.9: isFather("John","Paul").
0.9: isFather("John","Paul").
isGrandfather(X,Y) :- isFather(X,Z), isFather(Z,Y).
isGrandfather(X,Y) :- isFather(X,Z), isMother(Z,Y).

?- isGrandfather("John","Charles").
```

Listing 2.2: Prolog Code Example

In summary, probabilistic programming extends the capabilities of logical programming languages by providing a dedicated framework for modeling and inference in probabilistic systems. It allows the representation of uncertain or probabilistic information and facilitates sophisticated reasoning about uncertainty. This integration of probabilistic reasoning and programming offers a versatile approach to tackle complex problems that involve uncertain or probabilistic phenomena.

In conclusion, logic programming offers several advantages in the field of artificial intelligence, including its understandability and ability to reason without the need for extensive data. The integration of fuzzy logic and probabilistic reasoning further enhances its capabilities for handling imprecise and uncertain information.

Moving forward, the next chapter will delve into the basics of graphs. Graphs are fundamental data structures that allow us to represent and analyze relationships between objects or

entities. They provide a powerful framework for modeling complex systems, like knowledge graphs, which can store a vast amount of data for learning algorithms.

2.2.3 Introduction to Graphs

Graphs are widely applicable in various domains, including social networks, programming code, molecular structures, and more. They offer a representation to solve diverse problems, such as classification tasks, predicting missing edges, and determining the shortest path between nodes.

Graphs: Definitions

A graph is a couple $G = (V, E)$ with:

- V a collection of **nodes**.
- $E \subseteq \big\{\{x, y\} \mid (x, y) \in V^2 \wedge x \neq y\big\}$ a collection of **edges** that establish connections between these nodes. An edge can be directed or undirected, representing relationships or connections between the nodes. If two nodes are connected by an edge, they are called **adjacent**.

Every graph G can be represented by an **adjacency matrix**. It is defined as:

$$a_{ij} = \begin{cases} 1 & \text{if } \big(v_i, v_j\big) \in E \\ 0 & \text{otherwise.} \end{cases}$$

Graphs can for example be used to store linked data, such as in a social network, where each person can be represented as a node, and edges can be established between individuals who are connected or have a relationship. This allows the modeling of social interactions and the analysis of various properties of the network, such as influence, communities, or information diffusion.

A **knowledge graph** (KG), also known as a semantic network, is a powerful representation of real-world entities and their relationships. It utilizes a graph structure to organize and connect diverse data, enabling efficient access, processing, and retrieval of information across various applications and industries. Knowledge graphs are typically stored in a dedicated graph database and visualized as a network of interconnected nodes, edges, and labels. Nodes represent objects, places, or people, while edges define the relationships and connections between them. Labels provide additional information or categorization. For example, DBPedia is a common knowledge graph extracted from Wikipedia, providing structured information about notable entities, such as people, places, and concepts. An example of simpler a knowledge graph is represented Figure 2.3 taken from [29].

Constructing knowledge graphs involves integrating datasets from multiple sources, which may have different structures. To structure the diverse data, schemas, identities, and context are employed. Schemas define the framework or structure of the knowledge graph, serving as a blueprint for organizing the information. Identities help categorize and classify nodes efficiently, while context provides the necessary understanding of the environment in which the knowledge resides.

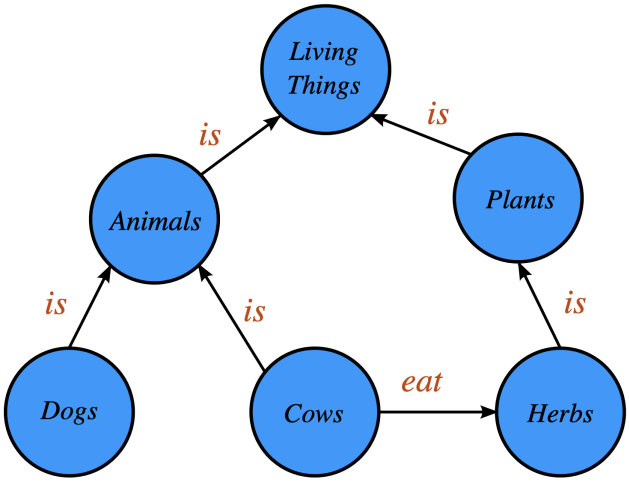

Figure 2.3: Knowledge Graph example

The applications of knowledge graphs span a wide range of industries, including data integration and management, search engines, and recommendation systems. In data integration and management, knowledge graphs facilitate the seamless integration of diverse datasets from various sources, creating a unified and comprehensive view of information. They greatly enhance search engine capabilities by providing structured and contextually relevant information in response to user queries. Knowledge graphs also play a crucial role in recommendation systems, leveraging the rich relationships and connections between items or entities to deliver personalized and targeted recommendations. Furthermore, knowledge graphs support data analytics by enabling effective data exploration, pattern identification, and generation of valuable insights using graph-based analytics techniques. Overall, knowledge graphs serve as a fundamental framework for representing and reasoning about knowledge in intelligent systems.

In the context of knowledge graphs, the Resource Description Framework (RDF) [21] holds significant importance. RDF enables the representation and interconnection of data, allowing for easy identification, disambiguation, and integration of information by AI systems. Acting as a foundational layer for knowledge graphs, RDF greatly enhances their ability to effectively represent, store, and process information, facilitating advanced knowledge discovery and utilization.

Embedding and Graph Neural Networks (GNN)

One approach frequently employed in the field of neuro-symbolic systems is the utilization of Graph Neural Networks (GNNs) [30]. GNNs aim to embed the nodes of a graph into a continuous vector space while preserving the similarity between nodes. This embedding allows us to define relationships between nodes using operations such as dot products in the embedding space.

In the context of graphs, **embedding** refers to the process of mapping nodes from a discrete graph structure into a continuous vector space, where each node is represented by a numerical vector. The embedding captures the essential characteristics of the node and encodes information about its local and global neighborhood relationships within the graph. By transforming nodes into a continuous vector space, we can perform various downstream tasks such as node

classification, link prediction, or graph generation, leveraging the rich representation of the embedded nodes.

GNNs can also be utilized for edge creation tasks. By randomly removing existing links in a graph, missing edges can be predicted by assigning scores to potential new connections. Furthermore, GNNs can envision a dynamic network that evolves over time and predict future connections, capturing the growth and evolution of the network.

Deep Learning with Graph Convolution Neural Networks (GCN)

When applying machine learning techniques to graphs, several considerations come into play. The vertex set, denoted as V, represents the nodes in the graph, while the adjacency matrix, denoted as A, captures the relationships between these nodes. Additionally, a matrix X containing node features is commonly utilized, where each row corresponds to a specific node, and the columns represent the associated features.

A naive approach involves directly using the adjacency matrix A as input for a deep neural network, along with the matrix X containing node features. However, this approach has limitations, such as a large number of parameters and sensitivity to node ordering, making it unsuitable for graphs of varying sizes.

To address the challenges associated with applying deep learning techniques to graphs, researchers have developed a specialized approach known as graph convolutional neural networks (GCNs) [33]. Unlike images, where neighboring pixels can be easily identified using concepts like locality or sliding windows, graphs lack such inherent structures. However, GCNs overcome this limitation by facilitating the exchange of messages between neighboring nodes.

The architecture of a GCN is specifically designed to accommodate the unique structure of the graph it operates on. It enables the efficient propagation of information throughout the network, capturing complex relationships and patterns within the graph.

During the training process, GCNs optimize their internal parameters using labeled data to effectively learn and adapt to the distinctive characteristics of the graph. This allows GCNs to proficiently propagate and transform information, enabling them to capture complex relationships and patterns within the graph.

By utilizing GCNs, various machine learning tasks tailored to specific problems can be performed on graphs. It is important to note that each node in a graph may have its own unique set of features, enabling the GCN to capture diverse information from different nodes. This flexibility allows for a comprehensive and accurate modeling of complex relationships within the graph.

Moreover, the learned graph embeddings can be leveraged for a wide range of downstream applications. For instance, in social network analysis, the node embeddings can be utilized to identify influential users or detect communities within the network. In the field of molecular chemistry, graph embeddings can aid in drug discovery by predicting the toxicity of molecules or suggesting potential drug candidates.

The integration of graph neural networks and embedding techniques has opened up new avenues for understanding and analyzing complex data represented as graphs. It enables the bridging of the gap between symbolic reasoning and neural network-based learning, leading to the development of neuro-symbolic systems that combine the strengths of both approaches.

In conclusion, graphs serve as a fundamental framework for representing relationships and connections in various domains. By employing graph neural networks and embedding tech-

niques, we can extract meaningful information, predict missing edges, and perform diverse machine learning tasks on graphs. This integration of neuro-symbolic approaches paves the way for advancements in fields such as social network analysis, bioinformatics, and knowledge representation.

2.3 Taxonomy

This section provides an overview of neuro-symbolic AI, which is a research field focused on combining deep learning and symbolic reasoning. The motivation behind this hybrid approach is the belief that neural and symbolic systems can complement each other and address their respective limitations. One key aspect of interest in many applications is interpretability, which refers to the ability of a model to be easily understood by humans, allowing them to comprehend the reasoning processes leading to model predictions.

Neuro-symbolic methods can be categorised in various ways. For the purpose of this report, we will categorize them into five parts based on related works [9, 2, 20]. It is important to mention that there are other existing methods, and some methods may belong to several categories. However, this taxonomy provides a broad understanding of the field and serves as a foundational framework for further exploration. In this section, we will first introduce all categories and then provide detailed explanations for each one.

1. **Logic as regularization:** This category involves using symbolic logic rules to structure neural networks by representing them as tensor embeddings. An example of this approach is Logic Tensor Networks (LTN) [2]. The main idea behind this line of research is to include logic as a regularizer during the optimization process or when learning the embeddings of a neural network.

2. **Logical specification of neural network architectures:** Here, the goal is to employ a logical language to specify the architecture of a neural network. This allows for the explicit representation of domain knowledge and constraints within the network's design.

3. **Systems that modify base neural network predictions:** This category includes methods like Knowledge-Enhanced Neural Networks (KENN) [7]. These systems aim to enhance the predictions of a base neural network by incorporating additional symbolic reasoning or knowledge representation techniques.

4. **Neurally guided search:** Systems falling under this category involve a symbolic engine querying a neural network during reasoning to, for example, estimate a utility function. An example is a symbolic search algorithm, such as tree search, where a neural network computes heuristics to expedite the search process. Applications of this approach can be found in game-playing AI, like DeepMind's AlphaGo, as well as many autonomous driving systems.

5. **Neural execution:** In this approach, neural networks and symbolic solvers collaborate to solve complementary tasks and frequently exchange information. Examples of this class include the neuro-symbolic Concept Learner and DeepProbLog [20]. DeepProbLog utilizes a logic program for reasoning while employing a neural network to estimate the probabilities of input facts within the logic program.

2.3.1 Logic as Regularization

Logic as regularization is a research direction in neuro-symbolic AI that aims to incorporate logic as a regularizer during the optimization or learning process of neural networks. By encoding logic into the weights of the network, the model follows the characteristics of the logic even when it is not explicitly present.

Several approaches have emerged that involve utilizing logical background knowledge as a regularizer during training. These approaches typically include logic-based penalty terms in the loss function to encourage adherence to logical rules and constraints. However, explicit logical knowledge is often lost as it is incorporated as a penalty term during training and not explicitly considered during prediction. Additionally, these regularization methods often rely on (fuzzy) first-order logic.

An example of logic as regularization is Logic Tensor Networks (LTN) [2]. LTN represents logical rules as tensor embeddings and incorporates them into the structure of the neural network. By integrating logic into the network architecture, LTN demonstrates the potential of logic as a regularization technique in neuro-symbolic AI using Real Logic.

In a conventional neural network (NN), the loss function is computed based on the difference between the predicted output and the ground truth labels, and the network weights are updated through backpropagation. In contrast, LTN introduces a "logical" term to the loss function, which evaluates how well the predictions satisfy the logical rules and constraints. This allows logical rules and constraints to be encoded directly into the weights of the neural network.

During training, LTN employs multiple neural networks to predict output tensors and uses logical checking to evaluate the extent to which the output tensors conform to the logical rules and constraints. The loss function incorporates both the difference between the predicted output and the ground truth labels and the logical term, which quantifies the compliance with the rules. The neural network weights are then updated using backpropagation.

During testing, only the predictions of the neural network are considered, as long as they satisfy the logical rules and constraints. By incorporating logical reasoning into the loss function, LTN can produce predictions that are not only accurate but also consistent with the logical rules and constraints.

LTN finds applications in various fields, such as healthcare and others, where logical reasoning plays a crucial role. However, there are limitations to consider. Scalability presents a challenge as handling large-scale datasets and complex logical rules can become computationally demanding. Additionally, ensuring strict adherence to logical rules and constraints in all cases may not be guaranteed, especially when dealing with noisy or incomplete data. Furthermore, the interpretability of LTN's models may be limited, as the complex interactions between the neural network and the embedded logic rules can make it challenging to fully understand and interpret the model's decisions and reasoning processes. These considerations highlight some of the practical considerations and trade-offs associated with the adoption of logic as a regularization technique in neuro-symbolic AI.

2.3.2 Logical Specification of Neural Network Architectures

The logical specification of neural network architectures involves utilizing a logical language to define the structure and components of a neural network. This approach allows for the integration of logic and neural reasoning, leading to enhanced capabilities and interpretability.

One way to achieve logical specification is through the use of extended logic programming languages and answer set programming. These languages provide a formal framework to set up the initial architecture and weight set of a recurrent neural network [8]. Additionally, first-order logic programs in the form of Horn clauses (conjunction of facts) can be used to define neural networks capable of solving Inductive Logic Programming tasks, starting from specific hypotheses covering the set of examples [12]. Inductive Logic Programming (ILP) refers to a subfield of machine learning that combines principles from logic programming and inductive reasoning to induce logical representations, rules, or programs from examples. ILP aims to learn general patterns and knowledge in the form of logical rules by observing specific instances, allowing for the application of logical reasoning to new cases.

Lifted relational neural networks, on the other hand, leverage a declarative framework that utilizes Datalog programs as a compact specification for a wide range of advanced neural architectures, with a specific focus on Graph Neural Networks (GNNs) and their generalizations [26].

In the context of logical specification of neural architectures, an interesting approach is presented in [23]. This work introduces a weighted Real Logic, which is used to specify neurons in a highly modular neural network. The neural network structure resembles a tree structure, where different neurons with various activation functions are utilized to implement different logic operators. This modular approach provides flexibility in combining logical operations within the network.

Logic-based neural networks demonstrate the potential of using logic to specify neural network architectures [15]. By defining classifiers as logical formulas, computational graphs can be constructed, combining sub-architectures based on the syntax of the logical formula. These approaches not only enhance interpretability but also enable the integration of reasoning capabilities, making them well-suited for handling structured or varying length data effectively.

2.3.3 Systems that Modify the Predictions of a Base Neural Network

Various systems have been developed to modify the predictions of a base neural network using logical rules. One such approach is the Deep Logic Models (DLM) architecture, which is designed to refine the predictions of a base neural network by incorporating logical formulas as constraints [22]. In DLM, two models collaborate: a neural network that predicts truth values of ground atoms based on input features, and an undirected graphical model representing the probability distribution characterized by logical constraints. The graphical model represents a set of variables as nodes and establishes connections (edges) between them to reflect their dependencies and interactions. Unlike directed graphical models, which encode causal relationships, the undirected nature of this model does not impose a specific directionality between variables. Instead, it captures the relationships between variables in a more flexible manner. The logical constraints serve as guidance for modifying the predictions of the neural network, ensuring adherence to the specified logical formulas. However, it's important to note that DLM primarily supports propositional connectives and fuzzy semantics. Other frameworks, as LTN, extend support to universal and existential quantifiers, as well as first-order logic [2].

Another approach for modifying neural network predictions using logical constraints is the Knowledge Enhanced Neural Networks (KENN) framework [7]. In KENN, a base neural

network initially generates predictions $y = f_{nn}(x|w)$, which are subsequently adjusted by a knowledge enhancer function based on a set of weighted constraints formulated as clauses.

A subsequent section of the report will explore the KENN framework in greater depth.

2.3.4 Neurally Guided Search

Another category of approaches involves the integration of neural components into search procedures for symbolic program induction techniques. While these methods are not strictly logic-based, they combine systematic symbolic search techniques with neural networks to guide and improve the search process. Neural networks are utilized to provide guidance and scoring during the search.

A prominent example of this category are tree search techniques. By training neural networks to provide guidance and scoring during the branch-and-bound search procedure, researchers have successfully combined symbolic search methods with neural components to improve program induction and search efficiency. These techniques leverage neural networks as heuristics to guide the search process and estimate the quality of different branches or candidate programs. For instance, Kalyan et al. [27] trained a neural network to predict branch scores, while Zhang et al. [32] used a neural network to select candidate programs based on input-output constraints. Additionally, Ellis et al. [10] employed a neural network to navigate a domain-specific language (DSL) efficiently. By integrating neural components into the search procedures, these approaches enhance the effectiveness of symbolic program induction and improve search efficiency.

2.3.5 Neural Execution

In the domain of neural execution, various approaches aim to capture program behavior using neural networks. One notable framework is DeepProbLog [20], which combines logical and probabilistic reasoning with neural networks for program execution. DeepProbLog utilizes ProbLog, a logic programming language that evaluates facts based on their probabilities, and incorporates neural networks to accurately estimate the probabilities, enabling a comprehensive integration of logical, probabilistic, and neural computations. DeepProbLog serves as a neural probabilistic logic programming language, extending the capabilities of ProbLog to support symbolic and subsymbolic representations, program induction, probabilistic (logic) programming, and (deep) learning from examples.

Another approach in neural execution involves utilizing neural networks to fill in incomplete Forth programs or learn programs through subsets of clauses.

Additionally, Scallop is another noteworthy framework in the field of neural execution, offering similar capabilities to DeepProbLog but with faster computation through an approximation of probabilities instead of exact computation [16] [19].

In conclusion, various approaches have been explored to integrate neural networks with symbolic reasoning techniques. These include logic as regularization, logical specification of neural network architectures, modifying predictions using logical constraints, neurally guided search, and neural execution. Each approach brings its own strengths and focuses on different aspects of combining logic and neural networks. While DeepProbLog demonstrates the integration of logical, probabilistic, and neural computations, other techniques primarily concentrate on specific aspects such as neural architecture specification, prediction modification,

guided search, or program execution using neural networks. The choice of approach depends on the specific requirements and objectives of the problem at hand, allowing researchers and practitioners to select the most suitable framework for their needs.

In subsequent sections, we will delve into a detailed exploration of two approaches of specific interest for our goals : KENN and Scallop.

2.4 Knowledge Enhanced Neural Networks (KENN)

In the previous section, an overview of the KENN method was mentioned. This section provides a detailed understanding of how KENN works.

2.4.1 Overview of KENN

KENN introduces a final layer called knowledge enhancer (KE) within the neural network architecture. The KE layer consists of weighted clauses that represent domain-specific knowledge and are utilized to refine the initial prediction by adapting it according to the constraints. Figure 2.4 shows a high level overview of KENN [7]. Features x are given as input to a neural network (NN) which predicts y. Then the Knowledge Enhancer modifies the prediction y to y' based on the logical knowledge $\mathscr{K}$.

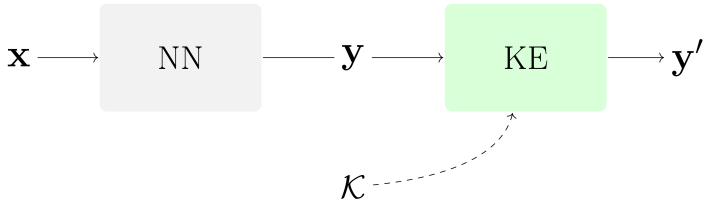

Figure 2.4: High-level overview of KENN

KENN primarily focuses on solving the problem of multi-class classification, which can also be applied to graphs. It achieves this by aligning predictions with the provided logic clauses. Additionally, KENN offers flexibility by allowing the system to learn the weights associated with these clauses in the KE layer. However, KENN has limited reasoning capabilities, which makes it less suitable for handling other types of tasks.

The following sections delve into the KENN method, providing a comprehensive explanation, evaluating its performance, and discussing its advantages and drawbacks.

2.4.2 Explanation of the Method

In KENN, the integration of prior knowledge, known as the "knowledge enhancer" (KE), involves incorporating First-Order Logic clauses representing the knowledge.

The Neural Network (NN) in KENN, used for multi-class classification, takes input features and produces a vector y containing elements for each class. Each class is treated as a ground atom, and the NN defines an interpretation $\mathscr{I}_{NN}$ where $\mathscr{I}_{NN}(A) = y_A$, A being an atom. Consequently, for a clause $(c = \bigvee_{i=1}^{k} l_i)$, the predicted truth value by the NN is denoted as $\perp(y_c)$.

To incorporate semantics, KENN employs fuzzy logic since classical logic cannot be directly applied to probabilities. The NN output vector y represents probabilities associated with each class. An interpretation function $\mathscr{I}$ maps terms, grounded atoms, and literals to real values between 0 and 1. The computation of clauses, involving disjunctions and negations of literals, uses a t-conorm function and Łukasiewicz negation ($\mathscr{I}(\neg x) = 1 - \mathscr{I}(x)$). Specifically, a t-conorm $\perp$: $[0,1] \times [0,1] \to [0,1]$ is defined by the following properties:

$$\begin{aligned} \perp(a,b) &= \perp(b,a) \\ \perp(a,b) &\leqslant \perp(c,d) \quad \texttt{if} \quad a \leqslant c \quad \texttt{and} \quad b \leqslant d \\ \perp(a,(\perp(b,c))) &= \perp(\perp(a,b),c) \\ \perp(a,0) &= a \quad \texttt{and} \quad \perp(a,1) = 1 \end{aligned}$$

However, the predicted values from the NN may conflict with the incorporated knowledge. To address this, the KE layer modifies the output y to y' in order to better satisfy the clauses while minimizing the difference $||y - y'||$. This modification is achieved using a t-conorm boost function $\delta : y \to y'$.

The t-conorm boost function increases the truth value of y for each clause c so that $y + \delta_c(y)$ has a higher truth value than y. The boost function is determined by weights w_c associated with each clause, denoting their respective influence on the final prediction. These weights are learned by the model using the training data. The boost function δ is defined as follows:

$$\begin{aligned} \delta &: [0,1]^n \to [0,1]^n \\ \forall n \in \mathbb{N}, \forall t \in [0,1]^n, 0 &\leqslant t_i + \delta(t)_i \leqslant 1 \\ \perp(t) &\leqslant \perp(t + \delta(t)) \end{aligned}$$

In KENN, the t-conorm boost function is applied to the pre-activation values just before the activation function σ of the neural network. By boosting the pre-activation values z to obtain $z' = \sigma(z + \delta(z))$, the output of the neural network is maintained within the desired range of 0 and 1, which is not the case if the boosting applied on the output directly.

To compute the boosts for each clause, the boost function $\delta_c(z)$ is determined by multiplying the learned weight w_c associated with the clause by the softmax function applied to the pre-activation values z. Mathematically, the boost $\delta_c(z)_i$ for each component i of z is calculated as:

$$\delta_c(z)_i = w_c \cdot \frac{e^{v_i}}{\sum_{j=1}^{n} e^{v_j}}$$

Where n represents the number of components in z. This computation ensures differentiability for efficient back-propagation, as both the neural network and the knowledge enhancer (KE) in KENN need to be differentiable. Figure 2.5 shows the detailed structure of the architecture of the Clause Enhancer for a clause c. First it selects the right literals, presents in the clause. Secondly, the t-conorm boost function is computed δ_s^w. Then, δ_c is obtained by putting a $+$ or $-$, if the literal contributes positively (resp. negatively).

Finally, as multiple clauses c with associated learned weights w_c contribute to the final prediction, the boosts δ_c are aggregated using a sum operation to simultaneously increase the truth value of all the clauses. This means that if a clause positively contributes to a class,

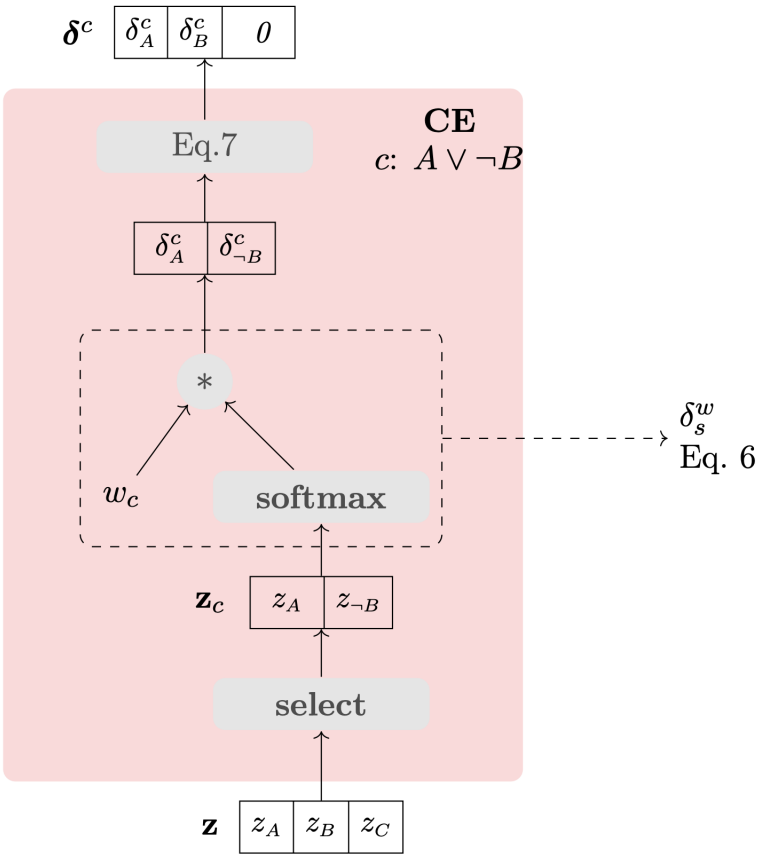

Figure 2.5: Clause Enhancer for $A \vee \neg B$

the prediction for that class is increased, while negative contributions decrease the prediction. Therefore, the boosted pre-activation values z' can be calculated as follows:

$$z' = \sigma(z + \sum_{c \in \mathscr{G}(\mathscr{K},\mathscr{C})} \delta_c)$$

The summation-based approach, while not intuitively representing a conjunction of clauses, offers faster learning, and the ability to handle potential inconsistencies between the final prediction and the logical rules (see section 8 of [7]).

The weights w_c associated with each clause are not hyperparameters but are learned by the model. Higher weights indicate a greater influence of the clause based on the training data, while smaller weights suggest less impact. To evaluate the effectiveness of the learned clauses, their performance with the dataset can be examined. The weights are assigned based on how frequently the clauses are satisfied. This evaluation involves calculating, for each node of each type, the fraction:

$$\frac{\texttt{number of neighbors with the same class}}{\texttt{total number of neighbors}}$$

Formally, the KENN model can be defined as follows:

$$KENN(x|\lambda, w) = \sigma(f'_{nn}(x|w) + \sum_c \lambda_c \cdot (\texttt{softmax}(\texttt{sign}(c) \odot f'_{nn}(x|w)) \odot \texttt{sign}(c)))$$

In this equation from [2], $f'_{nn}(x|w)$ represents the pre-activations of the base neural network $f_{nn}(\cdot|w)$, $\texttt{sign}(c)$ is a vector of the same dimension as y that contains 1, -1, and $-\infty$. The values in $\texttt{sign}(c)$ are set such that $\texttt{sign}(c)_i = 1$ (resp. $\texttt{sign}(c)_i = -1$) if the i-th atom occurs positively (resp. negatively) in c, or $-\infty$ otherwise. The operator $\odot$ denotes element-wise multiplication. The KENN model optimizes the weights λ of the clauses in the background knowledge and the base network parameters w by minimizing a standard loss function, such as cross-entropy, on a set of training data.

If the training data conflicts with a particular rule, the weight of that constraint will approach zero. This intuitive behavior implies a preference for latent knowledge inherent in the data over the knowledge specified by the constraints. In contrast, LTN represents training data and logical constraints uniformly using a formula and requires their joint satisfaction. Another distinction between KENN and LTN lies in the language employed: while LTN supports constraints expressed in full first-order logic, KENN restricts constraints to universally quantified clauses (conjunction of literals).

2.4.3 Evaluation of the KENN Model

This section presents an evaluation of the KENN model, focusing on its ability to classify nodes in a graph using both node features and edges. The performance of KENN is compared to other existing methods, and its strengths and limitations are analyzed.

For the evaluation, the Citeseer dataset is utilized, which represents a citation network of academic papers. The dataset includes six classes: AG (Algorithmic Graph Theory), AI (Artificial Intelligence), DB (Database), IR (Information Retrieval), ML (Machine Learning), and HCI (Human-Computer Interaction). Additionally, the dataset provides information about citations between papers, which is used as prior knowledge.

The original paper evaluated KENN using both inductive and transductive learning settings. However, this analysis primarily focuses on the inductive learning scenario, where the training and test sets are distinct graphs.

To assess the performance of KENN, a comparison is made with two other methods from the literature: SBR (Semantic Based Regularization) and RNM (Relational Neural Machines), in terms of accuracy. KENN demonstrates a slight improvement in accuracy compared to these methods. Evaluations are conducted using different proportions of training data to showcase the model's performance with limited training data. The results indicate that KENN, with the assistance of the KE layer, exhibits significant improvements, particularly in scenarios with scarce training data.

Furthermore, an important aspect is the interpretability of KENN's predictions. KENN appears capable of learning the weights associated with the logical clauses from the available data. This suggests that the model can effectively incorporate and utilize the knowledge encoded in the logical clauses.

Moreover, KENN demonstrates favorable characteristics in terms of inference time during testing, as it is quicker and scalable. This suggests that the method could be applied to large datasets effectively.

Plus, KENN is using a classical neural network to study graphs, but we can wonder if it could improve its performance using a graph neural network. This has been analysed in this paper [28].

In conclusion, the KENN method offers a powerful and unique approach to integrating neural and symbolic methods, particularly for node classification tasks. By incorporating First-Order Logic universally quantified clauses through the knowledge enhancer (KE) layer, KENN allows for the refinement of initial predictions based on domain-specific knowledge. The model has demonstrated its ability to learn the weights associated with these clauses, enabling effective utilization of logical knowledge.

However, it is important to note that KENN's flexibility is relatively limited compared to other methods because it only relies on logical disjunction. This constraint restricts the expressive power of KENN and its ability to handle certain types of logical reasoning. Future developments in the KENN model should aim to address this limitation and explore the inclusion of the existential quantifier to enhance its flexibility.

To tackle the flexibility problem of KENN, an alternative method called DeepProbLog will be introduced. DeepProbLog offers a solution by using a complete logical language, which provides more expressive capabilities compared to KENN. The DeepProbLog method aims to bridge the gap between neural and symbolic reasoning, offering a more comprehensive approach to neuro-symbolic AI.

2.5 Neural Execution Methods

Neuro-symbolic methods have emerged as a powerful approach to enhance prediction accuracy and interpretability in machine learning. These methods combine the strengths of neural networks and logical reasoning to tackle complex tasks. While approaches like KENN and LTN have successfully integrated logical reasoning into neural networks, a new class of methods known as neural execution methods, including Scallop and DeepProbLog, offer a distinct perspective. These methods leverage probabilistic logical languages and neural networks to estimate the probabilities of facts, providing a novel framework for neuro-symbolic AI.

In this section, the focus will be on exploring DeepProbLog, a pioneering neural execution method based on probabilistic logical languages. DeepProbLog combines logic programming with neural networks to estimate the probabilities of facts, providing a unique framework for neuro-symbolic reasoning. The key concepts and techniques of DeepProbLog will be examined, laying the foundation for understanding the advancements introduced by Scallop. Following the exploration of DeepProbLog, attention will shift to Scallop, an innovative extension of DeepProbLog that pushes the boundaries of neuro-symbolic AI. By building upon the principles of probabilistic logical languages and neural networks, Scallop represents a significant evolution in the field, offering enhanced capabilities for accurate and flexible reasoning. The advancements introduced by Scallop will be explored, providing valuable insights into the cutting-edge developments in neuro-symbolic AI.

2.5.1 DeepProbLog (DPL)

DeepProbLog (DPL) combines logical reasoning and deep learning to answer questions and estimate probabilities. It employs a neural probabilistic logic programming language, which integrates logical reasoning techniques with neural predicates. This section presents the components and steps involved in DPL and discuss its key experiments to demonstrate its capabilities.

Components and Capabilities

DPL supports both symbolic and subsymbolic representations and inference. It enables program induction, probabilistic programming, and reasoning with neural networks. The framework consists of two essential components: neural facts and neural arithmetic circuits (ACs).

Neural facts represent the probabilistic facts derived from neural networks. For example, in the case of estimating the probability that an image represents the digit "2", DPL uses a neural network trained to predict the probabilities of different digits. The neural fact for the predicate "addition(X,Y,Z)" can be defined as follows:

$\texttt{addition}(I_X, I_Y, N_Z)$:- $\texttt{digit}(I_X, N_X), \texttt{digit}(I_Y, N_Y), N_Z$ is $N_X + N_Y$

This neural fact implies that adding images X and Y results in image Z, where X represents the digit N_X, Y represents the digit N_Y, and Z represents the sum of N_X and N_Y.

DPL also incorporates probabilistic logic programs known as Annotated Disjunctions (ADs). The semantics of the program and the neural fact define the logical relationships and probabilities between the predicates.

ADs are probabilistic logic constructs used to represent uncertain knowledge and capture probabilities associated with logical relationships between predicates. ADs allow for the specification of multiple possible outcomes or alternatives with associated probabilities. They are defined in the form: $p :: p_1; p_2; ...; p_n$. Here, p represents a logical predicate, and $p_1, p_2, ..., p_n$ are alternative predicates or conditions that can occur with respective probabilities. The probabilities $p_1, p_2, ..., p_n$ must sum up to 1.

For example, consider the AD:

$$\texttt{coin\_flip} :: \texttt{heads}; \texttt{tails}$$

which represents the outcome of flipping a coin. The predicate `coin_flip` can be either heads or tails, each with a probability of 0.5.

Prediction Process

The prediction process in DPL involves the following steps:

1. **Grounding Step:** The goal of this step is to estimate a query q from a set of ground rules and facts by identifying the relevant ones. DPL employs backward reasoning, starting from the query and tracing back the rules that imply it. This is in contrast to forward reasoning used in other frameworks like Scallop.

2. **Probabilistic Facts:** Neural facts need to be transformed into probabilistic facts to enable reasoning with probabilities. For example, consider a DeepProbLog (also a ProbLog) program with the following probabilistic facts:

$$0.2 :: \texttt{earthquake}.$$
$$0.1 :: \texttt{burglary}.$$
$$0.5 :: \texttt{at\_home(Mary)}.$$
$$0.4 :: \texttt{at\_home(john)}$$

3. **Program Rewriting:** After obtaining the probabilistic facts, the next step is to rewrite the logic program in order to compute the truth value of the query based on the truth values of the ground facts. For example, consider the ProbLog program with the following rules:

$$\texttt{alarm} \leftarrow \texttt{earthquake}.$$
$$\texttt{alarm} \leftarrow \texttt{burglary}.$$
$$\texttt{calls(X)} \leftarrow \texttt{alarm}, \texttt{at\_home(X)}.$$

   Using the ground fact `at_home(Mary)`, the program can be rewritten as:

$$\texttt{alarm} \leftarrow \texttt{earthquake}.$$
$$\texttt{alarm} \leftarrow \texttt{burglary}.$$
$$\texttt{calls(Mary)} \leftarrow \texttt{alarm}, \texttt{at\_home(Mary)}.$$

4. **Knowledge Compilation:** The logic formula obtained from program rewriting is transformed into a more efficient form for computing the weighted model counting (WMC). WMC is a technique used to calculate the probability of a query being true given a probabilistic logic program. In DPL, this transformation is achieved by utilizing Sentential Decision Diagrams (SDDs) for knowledge compilation. SDDs are a type of data structure that compactly represents the logic formula and enable efficient WMC computations.

5. **Arithmetic Circuit (AC) and WMC:** The SDD is further transformed into an arithmetic circuit (AC). An example is presented Figure 2.9. In the AC, the values corresponding to ground facts (or their negation) are assigned to the terminal nodes, and operators are used at the internal nodes. This allows the computation of the desired result by performing addition and multiplication, replacing the AND and OR operations, respectively. The AC provides an efficient framework for performing Weighted Model Counting (WMC) calculations.

Example: Arithmetic Circuit (AC) and Weighted Model Counting (WMC)
To illustrate AC and WMC calculation in DPL, let's consider the following example:
Given the ProbLog program with the following probabilistic facts and rules Figure 2.6:

```
0.2::earthquake.
0.1::burglary.
0.5::at_home(mary).
0.4::at_home(john).
alarm :- earthquake.
alarm :- burglary.
calls(X):-alarm,at_home(X).
```

Figure 2.6: The ProbLog program

We can rewrite the program with the ground fact `at_home(mary)` as in Figure 2.7:
Next, we perform knowledge compilation using SDDs, resulting in an AC representation for the query `calls(mary)`.

```
0.2::earthquake.
0.1::burglary.
0.5::at_home(mary).

alarm :- earthquake.
alarm :- burglary.
calls(mary):-alarm,at_home(mary).
```

Figure 2.7: The relevant ground program

The resulting formula of the previous step is:

$$\texttt{calls(mary)} \leftrightarrow \texttt{at\_home(mary)} \wedge (\texttt{burglary} \vee \texttt{earthquake})$$

The relevant WMC is given by Figure 2.8:

Models of $\texttt{calls(mary)} \leftrightarrow \texttt{at\_home(mary)} \wedge (\texttt{burglary} \vee \texttt{earthquake})$	w
{}	0.36
{at_home(mary)}	0.36
{earthquake}	0.09
{earthquake, at_home(mary),calls(mary)}	**0.09**
{burglary}	0.04
{burglary, at_home(mary),calls(mary)}	**0.04**
{burglary, earthquake}	0.01
{burglary, earthquake, at_home(mary),calls(mary)}	**0.01**
$\sum_{\texttt{calls(mary)} \in \text{model}}$	**0.14**

Figure 2.8: The weighted count of the models where $\texttt{calls(mary)}$ is true.

Therefore, the WMC for the query $\texttt{calls(mary)}$ is 0.14. This indicates the probability of the query being true based on the provided probabilistic logic program. However, it is not calculated as shown here, but the AC is used instead.

The AC for the query $\texttt{calls(mary)}$ can be constructed using the relevant ground program and its associated probabilities. The AC allows for efficient computation of the query's result by combining the probabilities and logical operations.

Here's an example of the AC construction based on the given ground program Figure 2.9:

This AC represents the logical formula of the query $\texttt{calls(mary)}$, incorporating the associated probabilities of the ground program.

These steps collectively enable DPL to perform predictions by combining logical reasoning with deep learning, leveraging the probabilistic facts, program rewriting, and efficient knowledge compilation techniques using SDDs and ACs.

Learning

Learning in DPL encompasses training the neural network's parameters, a process that can be challenging due to the absence of intermediate results for end-to-end supervision. To address this challenge, DPL employs a gradient-based approach that leverages the underlying structure of the logic program.

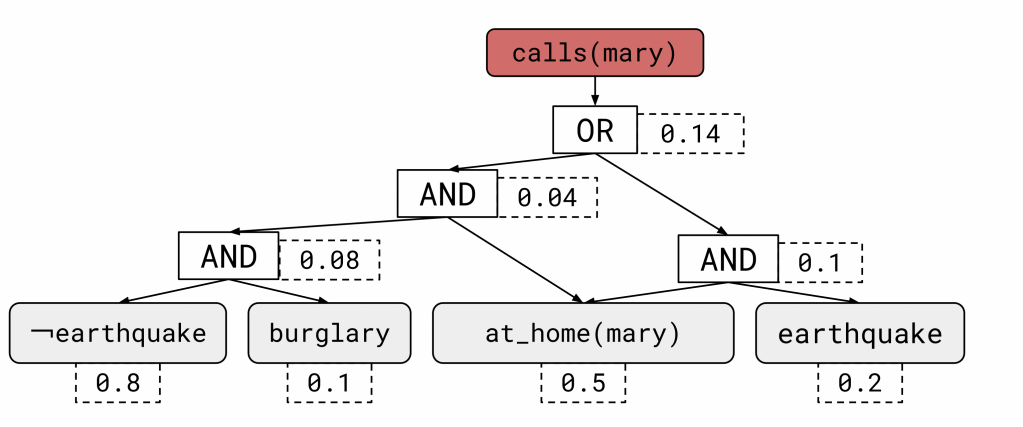

Figure 2.9: the AC for the query `calls(mary)`

In DPL, the network's parameters are updated by calculating gradients based on the logic program's structure. This is achieved through retropropagation, a widely used technique in neural network training. By calculating the gradients of the loss function with respect to the network's parameters, DPL iteratively updates the parameters, continually improving the network's performance.

What distinguishes DPL is its integration of the logic program's structure into the learning process. By leveraging the logical relationships and dependencies encoded in the program, DPL enhances the effectiveness of learning.

By combining gradient-based learning with the utilization of the logic program's structure, DPL overcomes the challenge of limited intermediate results and enables the neural network to learn and optimize its parameters.

Experiments

To demonstrate its capabilities, DPL has been evaluated through various experiments:

1. **1: Addition:** This experiment focuses on training a model to perform addition using pairs of images labeled with their respective sums. The DeepProbLog program includes a clause for addition and a neural AD for classifying MNIST images. The results demonstrate that DeepProbLog achieves higher accuracy and faster convergence than the baseline, showcasing the combination of logical reasoning and deep learning in solving arithmetic tasks.

2. **7: Word Algebra Problems (WAPs):** In this experiment, natural language sentences describing mathematical problems (WAPs) are used as input. The problems involve three numbers, and the model needs to solve them by permuting numbers, applying operations, and potentially swapping intermediate results. DeepProbLog achieves a good accuracy in solving these problems, demonstrating its capability in program induction and solving complex problems through multi-step reasoning.

3. **8: Coin Classification and Comparison:** This experiment involves training two neural networks using distant supervision to classify coins in images. The networks are tasked with recognizing and distinguishing two different coins and classifying each coin

as heads or tails. The experiment investigates whether the networks can recover the latent structure imposed by the logic program. The results show that the networks can agree on the coin's sides (heads or tails) but may exhibit inversions. Additionally, the networks may vary in their classification of the coins. With additional labeled examples, DeepProbLog converges more reliably on the desired solution. This experiment showcases the integration of probabilistic programming and deep learning in solving multi-step tasks.

4. **10: Successor:** In this experiment, embeddings are directly represented in DeepProbLog programs using MNIST images. The objective is to induce an order relation in the embedding space by learning embeddings that capture the successor relationship between images. The model utilizes a clustering objective and implements a translation-based successor relationship. The results demonstrate that DeepProbLog can learn structured embedding spaces and perform soft unification. The embeddings align with the image labels, indicating successful learning of the successor relationship. This experiment highlights DeepProbLog's capability in learning embeddings and performing natural language reasoning.

Limitations and Perspective

DPL incorporates several optimizations to improve efficiency. One such optimization is performing the grounding step only once, reducing the time it typically consumes. However, the current version of DPL has certain limitations. It may not be well-suited for handling embeddings, but ongoing research endeavors to address this limitation through the exploration of tensor-based representations.

Additionally, DPL relies on exact probabilities instead of approximations. While this ensures accuracy, it can lead to increased computation time as the program length grows. To tackle this challenge, the Scallop framework comes into play. Scallop, with its ability to approximate probabilities, surpasses DPL in various applications, delivering significantly faster performance. The next section will introduce the Scallop framework.

2.5.2 Scallop

Description of Scallop

Scallop is a neuro-symbolic framework inspired by DeepProbLog which combines logic and probability to make better predictions. However, unlike DPL, Scallop focuses on efficiently approximating probabilities, leading to improved performance. It introduces a parameter, denoted as k, that controls the reasoning granularity. By computing only the k most probable proofs, Scallop reduces computational cost while maintaining reasonable accuracy.

Scallop utilizes Datalog plus a probabilistic extension to enable probabilistic reasoning.

One of the key features of Scallop is its utilization of a probabilistic database. This database associates each candidate output of the model with a probability, enabling the representation of probabilistic data. It supports the representation of graphs in applications such as abstract syntax trees, scene graphs, and canonical symbolic representations of images. By utilizing a relational database, Scallop enables the representation of multi-modal data.

Scallop provides an interface for machine learning programmers to integrate logical domain knowledge into their models. The program is specified in a declarative logic programming

language that extends Datalog, making it expressive enough for programmers to specify the knowledge. The rule-based nature of Datalog simplifies the writing, debugging, and verification processes. Furthermore, Scallop leverages techniques from program synthesis and inductive logic programming, enhancing the flexibility and usability of the framework. Scallop also offers the capability for users to select the best heuristic for their calculations and to choose the desired fuzzy operators, like $*$ for the t-norm or $+$ for the t-conorm. This flexibility allows users to tailor the framework to their specific needs and domain requirements.

Experiment of the sum of two images

To illustrate the functionality of Scallop, consider an example shown in Figure 2.10, where a neural network classifier provides probabilities for 2 images to be each figure between 0 and 9. The goal of Scallop is to determine the probability of the sum to be x, for x between 0 and 18.

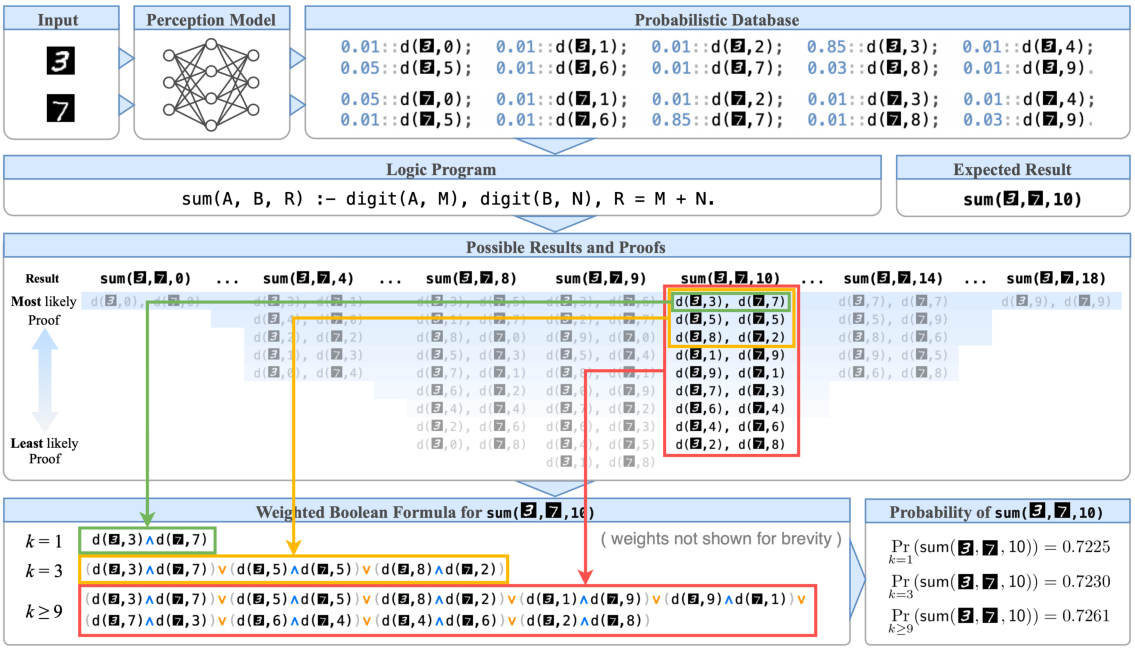

Figure 2.10: Illustration of Scallop on the sum of two images

The neural network classifier provides 10 probabilities for each image. As there are two images, it leads to 100 possible situations, as shown in the "possible results and proofs" table in Figure 2.10. Then the algorithm has to compute the probability for the sum to be 0, then 1... In the following we detail the example of how Scallop compute the probability that the sum is 10.

In DeepProbLog (or in Scallop with a really high granularity, here for $k > 9$), computing the correct probability for the sum to be 10 requires adding the probability of each possibility, resulting in a probability of 0.7261 in this case. This is how a disjunction is computed. However, with Scallop and a lower granularity, it is sufficient to select the highest k probabilities as the result. For $k = 1$, it yields a probability of 0.7225. This transformation reduces the computational complexity from linear to constant time. Formally, it is the replacement of logical operations $\bigoplus$ and $\bigotimes$ with $\bigoplus^{(k)}$ and $\bigotimes^{(k)}$, respectively.

While this reduction in complexity is beneficial, it is essential to ensure that the accuracy remains acceptable. The approximation error bound can be calculated. Noting $\texttt{Pr}(q)$ the success probability of a query q, S_q the set of proofs leading to q, and $\tilde{S_q}$ the approximated set of proofs, it leads to the following expression of the error bound [16]:

$$|\texttt{Pr}(S_q) - \texttt{Pr}(\tilde{S_q})| \leqslant \sum_{F \in S_q \setminus \tilde{S_q}} \texttt{Pr}(F)$$

However, it's worth noting that even with Scallop, computing the probability of all possibilities is necessary, even if they are not used in the disjunction calculation, which can be challenging in certain situations. For instance, even with $k = 1$, the algorithm must compute the probability of the 100 of possibilities. For example, if the first image has probability of 0.01 to be 2, and second image has probability 0.01 to be 8, Scallop has to compute the probability of the conjunction anyway. Moreover, this example of calculations is for the sum to be 10, but it must be done for all possible results, making the algorithm's complexity grows exponentially while the logical program becomes more complexe.

Moving on to learning, the neural network is trained to predict probabilities using a process similar to DeepProbLog. This training process incorporates Sentential Decision Diagrams (SDD) and Arithmetic Circuits (AC) to establish an end-to-end differentiable structure. Provenance, which is the fact to save information on how a computation is done, is used during the forward step by saving weights and predicted values, which allows for retropropagation during the learning process. In the learning phase of a classical neural networks, the loss is computed, and the provenance is used directly with the derivation of the weights to update the weights of the network. But for Scallop, as their is no loss at the output of the neural network, the provenance has to be modified during the logical program, the loss computed at the end, and then backpropagation is done on the whole architecture to optimize the neural network. So globally Aanotations are defined and propagated from the inputs to the outputs of relational algebra queries, which can include recursion, negation, and aggregation. However, all these calculations are handled by Scallop and are not seen by the user.

Experiment of the Visual Question Answering (VQA)

Another example is the visual question answering (VQA) described Figure 2.11, the task is to answer a query, here "Identify the tall animal on the left", based on an image. This is a too complexe scenario for classical methods. Let's go through each step of the algorithm to illustrate how Scallop works.

1. First, the natural language query is transformed into a programmatic query that Scallop can understand. Currently, this transformation is done manually, but in the future, it could be automated. The query for this example is:

   $$\texttt{target}(O) \texttt{ :- } \texttt{name}(O, \texttt{animal}), \texttt{left}(O, O'), \texttt{attr}(O, \texttt{tall}).$$

   This query defines the target object (O) as an animal on the left with the attribute of being tall.

2. An image shape recognition algorithm is used to identify multiple objects in the image. In this example, only three objects (o2, o5, and o12) are presented. For each object,

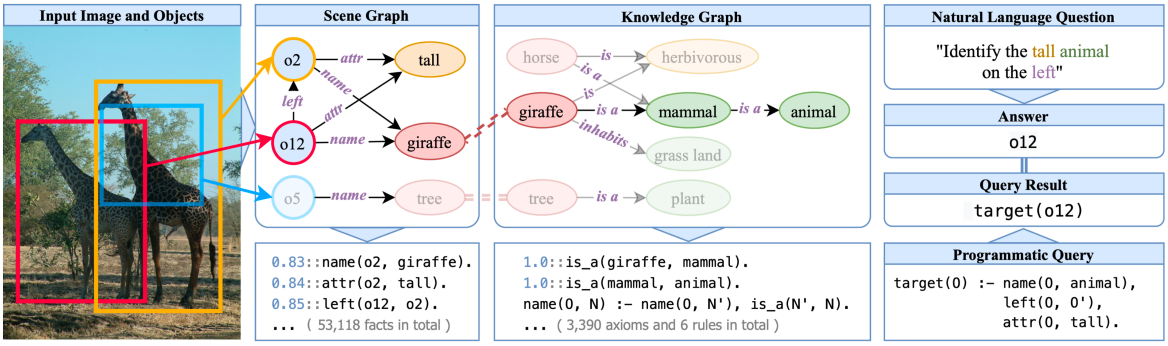

Figure 2.11: Visual Question Answering example

the algorithm computes probabilities for various properties: the object being an animal, being tall, and being on the left of another object.

3. To determine if an object is an animal, a neural network classifier is employed. The classifier takes the object as input and produces the probability of it being a specific type of animal (e.g., horse, giraffe, etc.). The probability of an object (e.g., o12) being an animal is computed by summing the probabilities for all animal types classified by the neural network (e.g., $\texttt{p}(o12, \texttt{horse}) + \texttt{p}(o12, \texttt{giraffe}) + ...$). Scallop "knows" these are animals because it can explore a knowledge graph, partially represented in Figure 2.11.

4. Additionally, the probabilities of an object being tall and being on the left need to be computed. These probabilities are calculated based on the object's characteristics and spatial relationships.

Overall, this VQA example involves estimating a large number of facts and computing numerous axioms. For instance, in this small example, there are more than ten objects to consider, probably more than a hundred animals in the knowledge graph, adding calculation of probabilities of their tallness and left position, and it require estimating 53,118 facts and computing 3,390 axioms. Even this small example is already computationally intensive for Scallop, and it would be too lengthy for deepProbLog.

Evaluation

Thiq evaluation compares Scallop performance against DeepProbLog. The evaluation explored the use of granularity (parameter k) for different question complexities, categorized into C2 to C6. Higher values of k (reasoning granularity) indicate more complex and longer reasoning. Three MLP classifiers were used in the evaluation: one for determining object names, one for object attributes, and one for object relations.

- Overall, the accuracy of Scallop was comparable to DeepProbLog, but with significantly faster runtime. Increasing the value of k led to improved accuracy.

- The choice of k was found to be task-dependent. Larger values of k were more suitable for training to converge faster, while smaller values were sufficient for prediction, reducing calculation time.

Limitations and Perspective

Scalability is an important aspect to consider in the effectiveness of a system like Scallop. While it performs well with a knowledge base of 250,000 triplets, taking approximately 2 seconds per query, further optimization would be required for larger knowledge bases like ConceptNet (34 million) or Wikidata (94 million). This scalability challenge indicates the need for ongoing development to ensure efficient performance across different scales of knowledge bases.

Addressing natural language questions is another area of consideration for Scallop. Currently, Scallop operates on programmatic queries, but handling natural language questions would require a separate model. This model could employ techniques such as program synthesis, semantic parsing, or inductive logic to convert natural language queries into their programmatic form.

Looking ahead, there are several avenues for further development and improvement of Scallop. One such direction involves exploring expressive extensions to make Scallop capable of tackling more complex neuro-symbolic applications. This extension would involve incorporating additional functionalities and capabilities to handle intricate scenarios and reasoning tasks effectively.

Optimizing the end-to-end pipeline is another focus for the future. By refining and enhancing the performance of each component in the pipeline, Scallop can deliver improved overall efficiency. This optimization effort ensures that the system operates smoothly and maximizes its potential impact.

Lastly, Scallop aims to apply its capabilities to real-world and safety-critical domains. For instance, it could be utilized in the field of autonomous vehicles to specify soft temporal constraints. In the medical field, Scallop's ability to explain diseases and medical conditions could prove beneficial. These practical applications demonstrate Scallop's potential to contribute to critical areas where interpretability and reliability are essential.

2.6 Conclusion

DeepProbLog and Scallop are two methods that combine neural networks and logical reasoning for improved prediction accuracy and interpretability in machine learning. DeepProbLog is a neural probabilistic logic programming language that uses neural networks and logical rules to estimate probabilities and answer queries. Scallop is a framework inspired by DeepProbLog that approximates probabilities with improved efficiency. It introduces a parameter to control reasoning granularity and reduce computational cost while maintaining accuracy. Overall, they offer accurate and flexible reasoning capabilities.

In the domain of neural network architectures, KENN stands out for its incorporation of weighted clauses to enhance prediction refinement. Its primary focus lies in tackling multi-class classification tasks. On the other hand, Scallop introduces a more versatile framework by harnessing a Datalog logic program, enabling it to handle a diverse range of tasks. Given these fundamental disparities, an intriguing question arises: Can Scallop effectively address the same problem as KENN, so multi-class classification on graphs, delivering comparable speed and accuracy? This will be explored further in the next section.

— 3 —
Contribution

3.1 Introduction

In this section, we present the implementation of a neuro-symbolic system using Scallop, a logic program enhanced with a neural network. The goal is to replicate the experiment conducted with Knowledge-Enhanced Neural Networks (KENN) on Citeseer [13] and investigate whether the Scallop program's implementation can achieve similar results.

The experiment is a 6-class classification problem of the nodes of Citeseer.

Our main objective is to explore whether is it possible to implement KENN example as a special case of Scallop and achieve comparable accuracy and runtime performance.

To accomplish this, we will follow the steps outlined below:

1. Define the neural network architecture suitable for the problem.
2. Define the symbolic program using the clause "$\forall x \forall y \neg \mathtt{T}(x) \vee \neg \mathtt{Cite}(x,y) \vee \mathtt{T}(y)$" (similar to KENN).
3. Establish the connection between the neural network output and the symbolic program input to create an end-to-end differentiable structure.
4. Incorporate weights into the clauses during the training process.

By implementing these steps and evaluating the results, we aim to determine if Scallop can effectively replicate the performance of KENN in this specific example. Additionally, the possibility of using more complex clauses to improve the accuracy further will be explored.

In the following subsections, we will delve into the details of each step, starting with the definition of the neural network architecture suitable for the problem.

3.2 Defining the Neural Network

The first step is to define a neural network adapted to the problem and ensure that it performs well on its own, without the symbolic program.

To begin, the data are imported using PyTorch, a machine learning framework in Python. The input data consists of a matrix containing all the nodes of the graph, and each node is composed of 3703 features. As each node represent a paper, the features are represented by a list of 3703 words, by the value 1 if the word is present in the paper, and 0 if it is not.

Next, the architecture of the neural network is defined. For this problem, a neural network with a single hidden layer containing 32 neurons is chosen. The output of the neural network is a tensor of size $1663 * 6$, representing the probability for all the nodes (1663) to belong to each of the 6 classes. For each node, the corresponding output is called the "pre-class", as this prediction will be modified by the symbolic layer. For example,

$$[[0.5, 0.1, 0.1, 0.1, 0.1, 0.1], [0.05, 0.7, 0.05, 0.1, 0.05, 0.05], [0.4, 0.1, 0.2, 0.1, 0.1, 0.1], ...]$$

indicates the probability distribution for each class for the corresponding nodes. [0.5,0.1,0.1,0.1,0.1,0.1] is the probability distribution of the first node for each class. A representation of the neural network architecture is showed Figure 3.1.

Before training the neural network, it is important to choose the best hyperparameters. After several runs, it appeared that a learning rate (lr) of 0.01 and a weight decay of $1e^{-8}$ yielded good results. Weight decay is a regularization technique commonly used during the training of neural networks. It helps prevent overfitting. The chosen loss function is cross entropy, which is suitable for multi-class classification problems.

It is worth noting that we are able to train the neural network separately because we have intermediate results between the neural network and the Scallop program, which is not the case in all experiments.

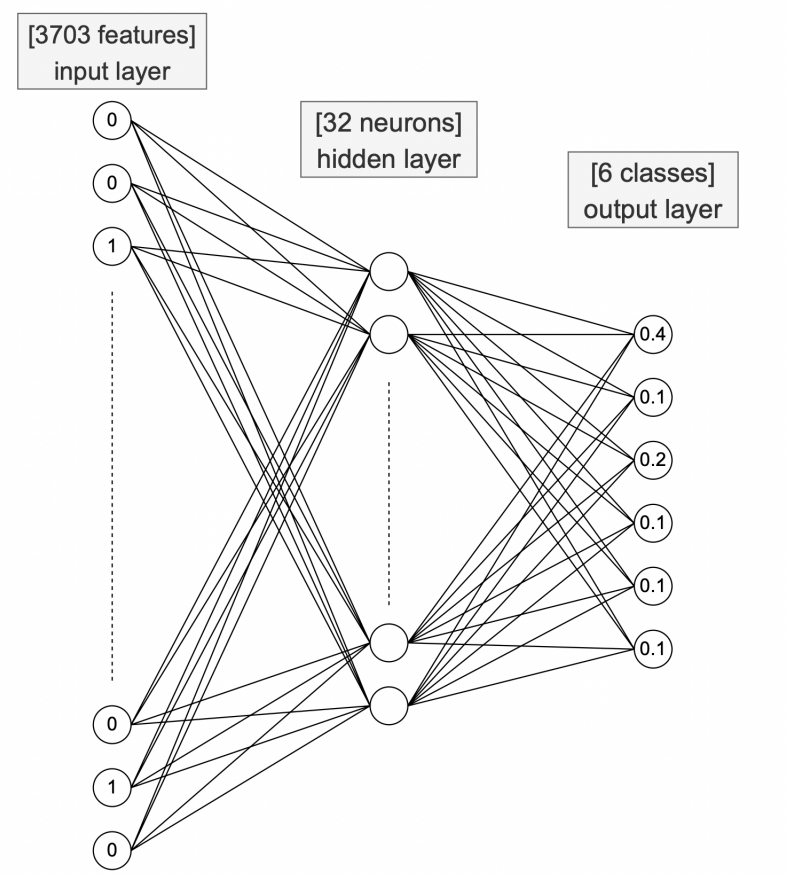

Figure 3.1: Neural network for Citeseer

3.3 Defining the Symbolic Layer

To implement the symbolic program, we start by defining the clauses, which will be the same as the one used in KENN: "$\forall x \forall y \neg \mathtt{T}(x) \vee \neg \mathtt{Cite}(x,y) \vee \mathtt{T}(y)$".

Since there is no explicit documentation on how to implement a symbolic program for Scallop, we had to rely on the available examples to make educated guesses. Based on this, we have formulated the basic Scallop symbolic program as shown in Listing 3.1.

```
ctx = scallopy.ScallopContext(provenance="topkproofs",k=k)

ctx.add_relation("node_preclass", (int,int))
ctx.add_facts("node_preclass", node_preclass_matrix)

ctx.add_relation("cite", (int, int))
ctx.add_facts("cite", edge_matrix)

ctx.add_rule("node_class(a,c) = cite(b,a) and node_preclass
   (b,c)")

# Run the scallop program
ctx.run()

# Get the results from relation node_class
for prob, tup in ctx.relation("node_class"):
  print(prob, tup)
```

Listing 3.1: basic Scallop program

Here is an explanation of the steps involved in the symbolic program:

1. We create a Scallop context, specifying the provenance as "`topkproofs`" and the value of k. The provenance determines the type of output Scallop provides, and "`topkproofs`" is chosen here for individual testing without direct integration with the neural network. The value of k is set to 1, which has been determined after several tests and doesn't seem to significantly affect the results for this case.

2. We add the necessary relations to the Scallop context. In this case, we define two relations: `node_preclass` and `cite`. The `node_preclass` relation represents the connection between a node and its class assigned firstly by the neural network, while the `cite` relation represents the connection between two nodes. We also provide the corresponding "facts" for these relations: the node matrix (`node_preclass_matrix`) and the edge matrix (`edge_matrix`). These probabilistic facts must be in a specific format: a tensor of tuples representing a relation associated with a probability. For example, a fact $(0.5,(0,3))$ means the probability for node 0 to be of class 3 is 0.5. The edge matrix needs to be in the same format, so $(1,(2,80))$ means there is a link between nodes 2 and 80 with a probability of 1.

3. We define a rule using the clause mentioned earlier: "`node_class`(b,c) = `cite`(a,b) `and node_preclass`(a,c)". This rule specifies how the `node_class` relation is derived based on the `cite` and `node_preclass` relations.

4. We run the Scallop program by invoking the `run()` method on the Scallop context. The program will then compute all the possibilities using the relations and facts defined before: for each element (a,b) in `edge_matrix` and for each element (b,c) in `node_preclass_matrix`, the program will store a new relation `node_class`(a,c) with the probability associated.

5. Finally, we retrieve the results from the `node_class` relation using a loop. Each result consists of a probability value and a tuple representing the connection between a node and a class. These results are then displayed.

An example of the program's output is presented in Listing 3.2. The probability for node zero to be of class zero is 0.176...

```
0.17689722776412964 (0, 0)
0.003220014739781618 (0, 1)
0.2866590619087219 (0, 2)
0.8141879439353943 (0, 3)
2.0996298189857043e-05 (0, 4)
7.903708319645375e-05 (0, 5)
0.18351562321186066 (1, 0)
0.9995589852333069 (1, 1)
0.893634021282196 (1, 2)
1.4324333278636914e-05 (1, 3)
0.00016254637739621103 (1, 4)
0.13098755478858948 (1, 5)
5.53224963368848e-05 (2, 0)
...
```

Listing 3.2: example output of the symbolic program in Listing 3.1

3.4 Linking the Output of the NN to the Input of the Symbolic Program for an End-to-End Differentiable Structure

Next, we build a framework that modifies the matrices so that the output of the neural network can be used as input for the symbolic program. As observed earlier, the output of the neural network does not have the same format as the input of the symbolic program. Therefore, we need another function to map them accordingly.

Here's an explanation of the process:

1. We take the output of the neural network, which is a tensor of size $1663 * 6$, as there are 1663 nodes in the training set.

2. For each element in the tensor (a total of 9978 elements), we create a probabilistic relational tuple, such as (0.05, (1,3)), and add it to a Python list.

3. We then transform the list into a PyTorch tensor, which serves as the input for the symbolic program.

By incorporating this process along with the symoblic program, we establish an end-to-end differentiable structure that enables training of the neural network weights.

However, we encounter a problem in the second phase. During this phase, the output tensor of the neural network is copied into another format. And in the tensor, each value has a corresponding provenance (information about its origin). We need to update this provenance to reflect the operations of the change of the format. This is crucial for the neural network to learn properly during the backpropagation phase.

Unfortunately, the Scallop framework doesn't have automatic support for updating the provenance. As a result, we have to manually modify the data format, and the provenance remains unchanged. To address this issue, we could potentially modify the Scallop algorithm itself to include this functionality. However, since we couldn't find specific documentation on this aspect of Scallop, we have decided to explore alternative approaches. The documentation of Scallop can be found here: [25].

Fortunately, we came across an example of an end-to-end differentiable structure for handwritten digit addition. We attempted to align our case with this example:

- Input of the symbolic part: any number of fact tensors, one for each defined relation. They all need to be of the same size.

- Output: Tensor prediction of the relation desired. It is the same size as the tensor inputs.

Since we cannot provide two matrices of different lengths to the symbolic program, we devised a solution. We create two matrices of the same size and modify the Scallop symbolic program accordingly. The idea is to build two node feature matrices, such that the i_{th} element of the first matrix is linked in the graph with the i_{th} element of the second matrix.

Here's the algorithm:

1. Create two Python lists.

2. For each edge:

    - Add the features of the first node to the first matrix.
    - Add the features of the second node to the second matrix.

3. Transform the two lists into PyTorch tensors.

Now, these two tensors can be fed into the neural network. The output, which is two tensors of size 1663 * 6, is then used as input for the modified symbolic program shown in Listing 3.3:

```
self.scl_ctx = scallopy.ScallopContext(provenance="
    difftopkproofs", k=1)
self.scl_ctx.add_relation("node_class_1", int,
    input_mapping=list(range(6)))
```

```
self.scl_ctx.add_relation("node_class_2", int,
   input_mapping=list(range(6)))
self.scl_ctx.add_rule("node_class(a) = node_class_1(a) and
   node_class_2(b)")

# The ‘classification‘ logical reasoning module
self.classif = self.scl_ctx.forward_function("node_class",
   output_mapping=[(i,) for i in range(6)])

\dots

return self.classif(node_class_1 = node_classif_1,
   node_class_2 = node_classif_2)
```

Listing 3.3: new symbolic program

In this modified Scallop symbolic program, the provenance is set to "difftopkproofs". Specifying "diff" enables training end-to-end the neural network with the symbolic program. The facts are passed as input to the function `self.classif`. Since this is a differentiable structure, the format of the input differs from the previous version. Moreover, it is needed to define the format of the relation lists: `input_mapping` = `list`(`range`(6)) indicates the node can have 6 different classes, and it is the same for the output, defined in the `forward_function` second argument.

The rule in the modified program, "`node_class`(a) = `node_class_1`(a) `and` `node_class_2`(b)", specifies that node a should be of the same class as node b. This modification essentially alters the prediction matrix of node a.

Let's now revisit the experiment:

1. Import Citeseer and construct the training and testing sets randomly, ensuring they have the same size (1663 for training and 1664 for testing, though the number of edges may vary depending on the degree of connectivity between nodes).

2. Execute a trained neural network alongside this method and KENN on the example.

3. Assess the improvement achieved by combining the neural network with these two methods compared to the neural network alone.

While the algorithm is functional, there are a few issues to address:

1. First, the modification of matrices to fit the symbolic program introduces inefficiency and may limit the possibilities for other experiments. It should be possible for Scallop to only take as input the raw data and to do the necessary modifications automatically

2. Then, the training of the neural network is not uniform: some nodes will go through the neural network more times than others. For example with 50 epochs, each node should contribute to the training 50 times (once each epoch). But in this case, if a node has for example 10 edges, it will go through the learning process 10 times, and a node with only 1 edge will go through the learning process of the network only one time. So all nodes do not contribute to the training the same way. This decreases the quality of the training.

3.5 Integrating Weights

As Scallop does not provide a mechanism to train weights of clauses, we implemented a separate neural network for this purpose. We created a second, very simple neural network with one input neuron (with a value of 1) and six output neurons (representing the weights) represented Figure 3.2. In this example, the value 1 is always set as input, and the weights of the neural network (which are exactly the weights of the clauses) are 0.4, 0.2... So to analyse the evolution of clause weights during training, it suffices to print the weights of this neural network.

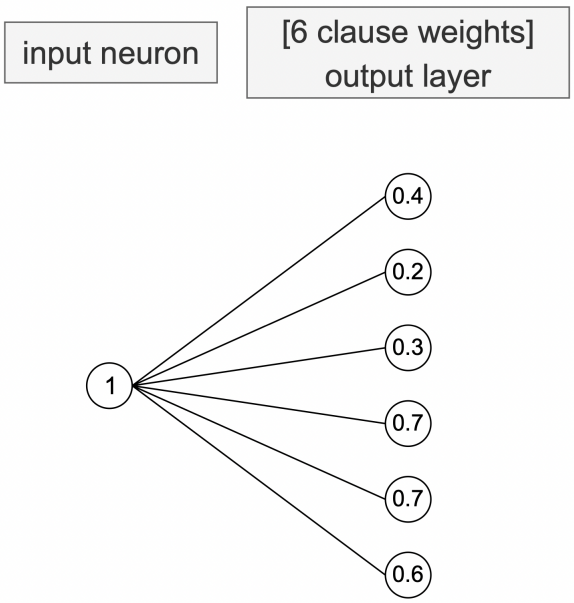

Figure 3.2: Neural network for clause weights

This additional neural network is trained simultaneously with the original one, allowing us to compute learned weights and to add them to each clause. The code to integrate the weights into the clauses is showed in Listing 3.4.

```
...
self.scl_ctx.add_relation("weight", int, input_mapping=list
   (range(6)))
self.scl_ctx.add_rule("node_class(a) = node_class_1(a) and
   node_class_2(b) and weight(a)")
...
return self.classif(node_class_1 = node_classif_1,
   node_class_2 = node_classif_2, weight = weight_matrix) #
    nb_nodes x 6 tensors
```

Listing 3.4: Scallop program part with definition and integration of weights

But to add the weights this way, it is needed to forward pass the weight neural network each time we go through the other neural network, so the three matrices have the same size nb_nodes $*$ 6.

Figure 3.3 represents the entire architecture of the algorithm for the prediction case. At the left, the neural network for node classification takes as input the two node feature matrices, in two different instances, which leads to 2 prediction matrices. In the same time, clause weights are generated, and these 3 same size matrices are put into the logical reasoning program. The program uses the clauses defined to provide a new node classification matrix. For training, the same architecture is used, but a loss function is computed with the final node classification and is backpropagated to modify the weights of the two neural networks.

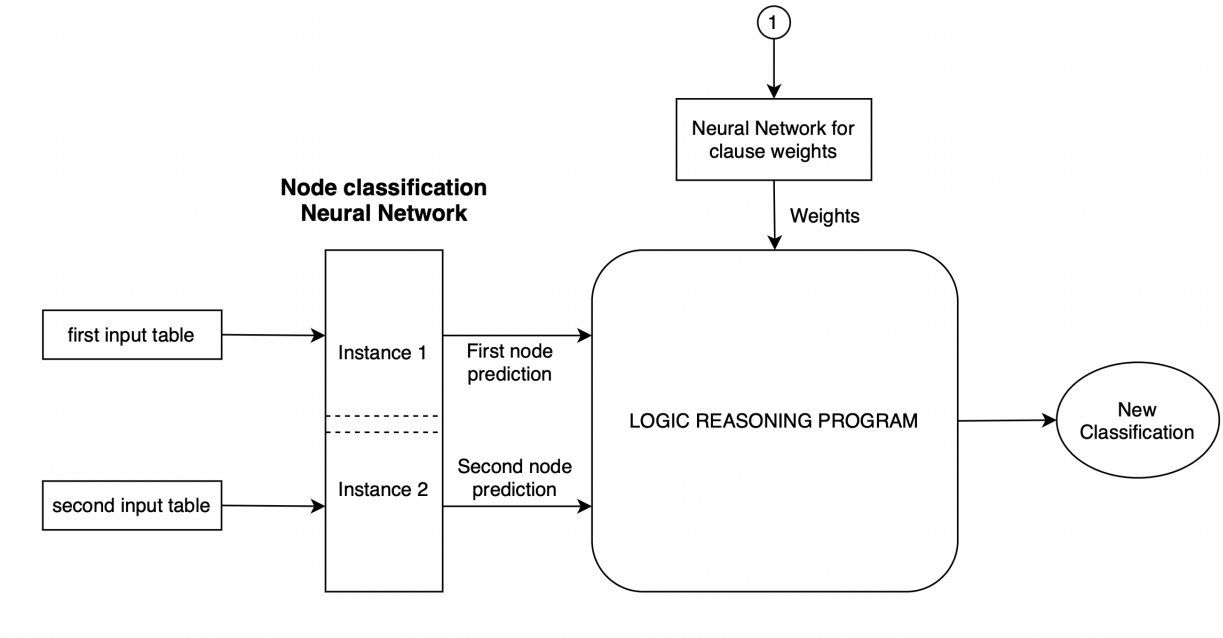

Figure 3.3: Illustration of the architecture for prediction

Now that the differentiable structure has been implemented, it is time to evaluate its performance for two granularities, $k = 1$ and $k = 5$, and compare it with the performance of a classical neural network to assess its effectiveness. The Citeseer dataset is imported and randomly divided into training and validation sets, allowing both methods to be tested on the same dataset partition. The results are presented in Table 3.1. The experiment was conducted 20 times, and the reported accuracy represents the average of these trials, accompanied by the corresponding variability.

Table 3.1: Average accuracy improvement with the differentiable Scallop program

	NN alone	**Scallop (k=1)**	Improvement (k=1)	**Scallop (k=5)**	improvement (k=5)
Accuracy	0.732	0.725	**-0.007**	0.725	**-0.007**
Variability	0.013	0.017		0.016	

Upon evaluating the Scallop program, it is observed that it learns to classify nodes in a manner similar to a classical neural network. However, its overall performance does not improve,

neither for $k = 1$ nor $k = 5$. This can be attributed to a specific rule: "node_class(a) = node_class_1(a) and node_class_2(b) and weight(a)". The calculation of this rule causes the final probability tensor represented by node_class(a) to have values that lie exactly in the middle between the probability tensors of node_class_1(a) and node_class_2(b). However, the original intention was to make slight modifications to node_class_1(a). Consequently, the impact of this rule extends beyond its intended target and significantly alters the predicted probabilities for all classes, thereby affecting the accuracy of the predictions.

And it is possible, in a similar manner as in KENN, to interpret weight clauses evolution. They are bigger when more weights respect the rules, and smaller when it is the opposite.

3.6 Using New Clauses

In the previous section, we observed that by using the same clauses as in KENN, we were able to train the architecture in a similar way, but not to improve the initial prediction.

However, Scallop offers greater flexibility in clause definition through its program-based approach. This opens up the possibility of constructing a more complex Scallop program to improve accuracy. In this section, we explore this idea.

We aimed to recreate the structure used previously but with more complex clauses. The new clauses can be summarized as follows: "If node A has n neighbors, and the majority $(n/2)$ of its neighbors are of class C, then node A is more likely to be of class C."

Below is the code implementation of the modified Scallop program 3.5:

```
ctx = scallopy.ScallopContext(provenance="topkproofs",k=1)

ctx.add_relation("node_preclass", (int,int))
ctx.add_facts("node_preclass", preclass_matrix)

ctx.add_relation("edge", (int, int))
ctx.add_facts("edge", edge_matrix)

ctx.add_rule("neighbor_count(node,num_neighbors) :-
   num_neighbors = count (n: edge(node,n))")
ctx.add_rule("majority_neighbor_count(node,num_majority) :-
    neighbor_count(node,num_neighbors) and num_majority ==
   (num_neighbors+1)/2")
ctx.add_rule("neighbor_count_of_class(node,c,num_class_c)
   :- num_class_c = count (n: edge(node,n) and
   node_preclass(n,c))")
ctx.add_rule("majority_class_among_neighbors(node,c) :-
   neighbor_count_of_class(node,c,num_class_c) and
   majority_neighbor_count(node,num_majority) and
   num_class_c>=num_majority")

ctx.add_relation("weight", (int,int))
ctx.add_facts("weight", weight_matrix)
```

```
ctx.add_rule("node_new_class(node,c) :- node_preclass(node,
   c) and majority_class_among_neighbors(node,c) and weight
   (0,c)")

# Run the scallop program
ctx.run()

for prob, tup in ctx.relation("node_new_class"):
  print(prob, tup)
```

Listing 3.5: new clauses Scallop program

Explanation of the code:

- First, the number of neighbors is computed.
- Then, the majority count of neighbors is determined.
- Next, the number of neighbors of class C is computed.
- Finally, the class of node n is assigned as c if c is more than half of the number of its neighbors.

The program has been implemented and performance issues were encountered: it took too much time to modify the 1663 training node predictions, so it was impossible to have results. Despite Scallop only looking for top-1 provenance (with $k = 1$), the way we defined the "majority" in the program caused Scallop to attempt computations for every value between $n/2$ and n, which was unnecessary for finding the result. To address this, we may explore potential updates to Scallop or refine the logic program itself (a consideration for future work).

3.7 Suggestions to Improve Runtime

To explore further improvements, an ad-hoc implementation of the Scallop program was attempted with the goal of drastically reducing computation time. The manual implementation followed a different approach compared to the previous steps. Here are the key aspects of this manual implementation:

- **Non-end-to-end training:** Instead of training the entire architecture end-to-end, the process was split into two stages. First, a neural network was trained independently. Then, a symbolic program modified the prediction, simulating the logic rules.
- **Manual computation of weights:** The weights associated with the rules were not learned by the system, but computed manually. For each rule, the number of edges that satisfied the rule was divided by the total number of edges that did not satisfy the rule. This computation was based on statistics obtained from the training set. For example, if there were 70 edges between two AI papers and 30 edges between an AI paper and any other paper, the weight for the clause for AI would be 70%.

This program simulates the previous symbolic program for the case $k = 1$. But the clauses has been implemented directly, replacing disjunction and conjunction with addition and multiplication operations between tensors. The following steps were performed for each node:

1. Compute the number of neighbors (n) for the current node.

2. For each class c:

   a) Select the top half of the nodes ($\texttt{maj} = \texttt{int}(n/2+1)$) that have the highest probability of being of class c.

   b) Multiply the probabilities of each of these nodes being of class c ($tot_c = p_{1,c} * p_{2,c} * \ldots * p_{\texttt{maj},c}$). This operation simulates the logical conjunction in the original Scallop program.

   c) Multiply the probability of the node being of class c (its preclass) by tot_c. This simulates the conjunction "a node is of class c if its preclass is c and the majority of its neighbors are of class c". And multiply this by the weight of the clause c

The results of this ad-hoc approach are presented in Table 3.2. By manually updating the probabilities of all nodes based on their neighbors, without using a Scallop program, the results are clearly better (+0.033). However, it is important to note that this manual implementation may not be easily usable for other examples or adaptable to different values of k. It lacks the flexibility and generalizability of the Scallop framework. Besides, the computation time was significantly reduced. this only takes 1 minute for the neural network training, and one additional minute for the computation of the results, instead of more than 3 hours with Scallop. These times have been obtained with a laptop with one 10-core CPU, one 32-core GPU and 64GO of RAM.

This accuracy improvement is almost the same as for KENN, but it is difficult to compare them, as the KENN method has not been realised in the same conditions as here, which is a work for later. Besides, KENN is implemented with "basic" clauses. It would be interesting to implement KENN with more complex clauses, if it is feasible.

Table 3.2: Experiment Results 2

Experiment	**Accuracy**	**Improvement from Neural Network Alone**	**Variability**
Neural Network Alone	0.732	0.0000	0.013
KENN	0.714*	**+0.034***	
Differentiable Scallop Algorithm	0.725	-0.007	0.017
HAD-OC Method	0.765	**+0.033**	0.010

*Taken from KENN paper with other experiment condition.

3.8 Conclusion

This work aimed to evaluate the Scallop system's capability to achieve comparable performance to a specific architecture for a particular node classification task. The primary focus was to test the Scallop model's effectiveness and explore its potential as an alternative approach.

The goal was to address the limitations of the KENN framework, which combines a classical neural network with a logic program, by investigating the scalability and adaptability of the Scallop system. While the KENN framework achieved good accuracy, it lacked flexibility and adaptability to different datasets and problem domains.

By testing the Scallop system, we sought to determine its ability to deliver similar performance while offering enhanced flexibility. The Scallop system provides a program-based approach, allowing the definition of intricate clauses that capture domain-specific knowledge.

The evaluation of the Scallop program on a benchmark dataset demonstrated promising results. Despite not explicitly focusing on improving explainability, the Scallop system showed potential in enhancing model interpretability. By incorporating logical clauses and leveraging the reasoning capabilities of the Scallop system, it was possible to gain insights into the decision-making process.

Furthermore, the manual implementation of the program, tailored specifically to the node classification task, addressed the runtime concerns associated with the original Scallop implementation. This manual implementation achieved comparable accuracy to the KENN framework while significantly reducing computation time of Scallop.

In conclusion, this work explored the potential of the Scallop system for node classification tasks. While the specific modifications necessary to achieve performance with the KENN framework were not implemented in this internship, the findings suggest that with appropriate adaptations and optimizations, the Scallop system could potentially achieve comparable performance. The primary advantage of the Scallop system lies in its flexibility and adaptability, enabling the incorporation of domain-specific knowledge through logical clauses.

— 4 —
Perspective

In light of the findings and limitations identified in this work, there are several perspectives to consider for further development and improvement of the Scallop system:

- **Improving the efficiency of Scallop:** One of the main challenges encountered was the extensive runtime for complex clauses. To enhance Scallop's efficiency, it is crucial to focus on reducing computation time, particularly when dealing with complex clauses. This could involve exploring algorithmic improvements and parallel processing techniques to expedite the reasoning process. Another avenue to explore is integrating deep learning directly into the symbolic system. By incorporating techniques such as [27] presented in the "**neurally guided search**" part, where a machine learning process determines the relevance of computing specific probabilities. With that, it would be possible to significantly reduce the number of calculations and improve efficiency.

- **Automating provenance updates:** Currently, modifying the matrices requires manual updates to the Scallop program. To enhance usability, it would be valuable to develop tools or mechanisms that can automatically update the provenance when the user modifies the matrices. This would simplify the process and enable users to apply Scallop to a wider range of tasks without requiring extensive manual adjustments.

- **Weighted rules implementation:** By enabling the learning of these weights within the Scallop architecture, users would not need to implement an additional neural network specifically for weight computation. This would simplify the overall framework and improve its efficiency.

- **Comparative analysis with more complex KENN clauses:** While this work compared the Scallop system to the KENN framework with "basic" clauses, it would be interesting to investigate the performance of KENN when incorporating more complex clauses. This would provide a more comprehensive understanding of the strengths and limitations of both approaches and potentially identify areas where one approach outperforms the other.

It is important to note that the conclusions drawn in this work are specific to the tested node classification task and dataset. Further research and experimentation would be necessary to assess the scalability and applicability of the Scallop system across diverse problem domains and larger-scale applications.

Addressing these perspectives will contribute to further advancing and refining the Scallop system, transforming it into a robust and efficient framework for node classification tasks.

Continued research and development efforts in these areas will strengthen the field of explainable AI and enhance the interpretability, scalability and adaptability of symbolic reasoning approaches in machine learning.

References

References

[1] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. Understanding of a convolutional neural network. In *2017 International Conference on Engineering and Technology (ICET)*, pages 1–6. Ieee, 2017.

[2] Samy Badreddine, Artur d'Avila Garcez, Luciano Serafini, and Michael Spranger. Logic tensor network. *Artificial Intelligence*, 303, 2021.

[3] Simran Cashyap, Max Sheremet, and Charence Wong. Topics in ai - logic programming - group 11. http://www.doc.ic.ac.uk/ cclw05/topics1/index.html, 2006.

[4] Stefano Ceri, Georg Gottlob, Letizia Tanca, et al. What you always wanted to know about datalog(and never dared to ask). *IEEE transactions on knowledge and data engineering*, 1(1):146–166, 1989.

[5] Alain Colmerauer and Philippe Roussel. The birth of prolog. In *History of programming languages—II*, pages 331–367. Association for Computing Machinery New York, NY, United States, 1996.

[6] Antonia Creswell, Tom White, Vincent Dumoulin, Kai Arulkumaran, Biswa Sengupta, and Anil A. Bharath. Generative adversarial networks: An overview. *IEEE Signal Processing Magazine*, 35(1):53–65, 2018.

[7] Alessandro Daniele and Luciano Serafini. Neural networks enhancement with logical knowledge, 2021.

[8] Artur d'Avila Garcez, Dov M. Gabbay, and Krysia B. Broda. Neural-symbolic learning system: Foundations and applications. *Springer-Verlag, Berlin, Heidelberg*, 2002.

[9] Lauren Nicole DeLong, Ramon Fernández Mir, Matthew Whyte, Zonglin Ji, and Jacques D. Fleuriot. Neurosymbolic ai for reasoning on graph structures: A survey, 2023.

[10] Kevin Ellis, Lucas Morales, Mathias Sablé-Meyer, Armando Solar-Lezama, and Josh Tenenbaum. Learning libraries of subroutines for neurally–guided bayesian program induction. *Advances in Neural Information Processing Systems*, 31, 2018.

[11] Ronald Fagin, Ryan Riegel, and Alexander Gray. Foundations of reasoning with uncertainty via real-valued logics, 2022.

[12] Manoel V. M. França, Gerson Zaverucha, and Artur S. d'Avila Garcez. Fast relational learning using bottom clause propositionalization with artificial neural networks. *Machine Learning*, 94, 2014.

[13] C Lee Giles, Kurt D Bollacker, and Steve Lawrence. Citeseer: An automatic citation indexing system. In *Proceedings of the third ACM conference on Digital libraries*, pages 89–98, 1998.

[14] Alex Graves. Generating sequences with recurrent neural networks. *CoRR*, abs/1308.0850, 2013.

[15] Kaoru Hirota and Witold Pedrycz. Logic-based neural networks. *Information sciences*, 71(1-2):99–130, 1993.

[16] Jiani Huang, Ziyang Li, Binghong Chen, Karan Samel, Mayur Naik, Le Song, and Xujie Si. *Scallop: From Probabilistic Deductive Databases to Scalable Differentiable Reasoning*, volume 34. Curran Associates, Inc., 2021.

[17] i2tutorials. Explain activation function in neural network and its types. https://www.i2tutorials.com/explain-activation-function-in-neural-network-and-its-types/, 2019.

[18] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[19] Ziyang Li, Jiani Huang, and Mayur Naik. Scallop: A language for neurosymbolic programming, 2023.

[20] Robin Manhaeve, Sebastijan Dumančić, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. Neural probabilistic logic programming in deepproblog. *Artificial Intelligence*, 298, 2021.

[21] Frank Manola, Eric Miller, Brian McBride, et al. Rdf primer. *W3C recommendation*, 10(1-107):6, 2004.

[22] Giuseppe Marra, Francesco Giannini, Michelangelo Diligenti, and Marco Gori. Integrating learning and reasoning with deep logic models. *CoRR*, abs/1901.04195, 2019.

[23] Ryan Riegel, Alexander Gray, Francois Luus, Naweed Khan, Ndivhuwo Makondo, Ismail Yunus Akhalwaya, Haifeng Qian, Ronald Fagin, Francisco Barahona, Udit Sharma, Shajith Ikbal, Hima Karanam, Sumit Neelam, Ankita Likhyani, and Santosh Srivastava. Logical neural networks. *CoRR*, abs/2006.13155, 2020.

[24] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2017.

[25] University of Pennsylvania Scallop Development Team. Scallop - tutorial. https://scallop-lang.github.io/tutorial.html, 2022.

[26] Gustav Sourek, Vojtech Aschenbrenner, Filip Zelezny, Steven Schockaert, and Ondrej Kuzelka. Lifted relational neural networks: Efficient learning of latent relational structures. *Journal of Artificial Intelligence Research*, 62:69–100, 2018.

[27] Ashwin J. Vijayakumar, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. Neural-guided deductive search for real-time program synthesis from examples. *CoRR*, abs/1804.01186, 2018.

[28] Luisa Werner, Nabil Layaïda, Pierre Genevès, and Sarah Chlyah. Knowledge enhanced graph neural networks, 2023.

[29] Wikipedia. Knowledge graph. https://en.wikipedia.org/wiki/Knowledge_graph, 2023.

[30] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1):4–24, 2021.

[31] Lotfi A Zadeh. Fuzzy logic. In *Granular, Fuzzy, and Soft Computing*, pages 19–49. Springer, 2023.

[32] Lisa Zhang, Gregory Rosenblatt, Ethan Fetaya, Renjie Liao, William E. Byrd, Matthew Might, Raquel Urtasun, and Richard S. Zemel. Neural guided constraint logic programming for program synthesis. *CoRR*, abs/1809.02840, 2018.

[33] Si Zhang, Hanghang Tong, Jiejun Xu, and Ross Maciejewski. Graph convolutional networks: a comprehensive review. *Computational Social Networks*, 6(1), 2019.

[34] Arnold M Zwicky. Loglan: A logical language, 1969.