Università Politecnica delle Marche
Facoltà di Ingegneria
Dipartimento di Ingegneria dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

Master Thesis

Un nuovo framework con Logic Tensor Networks per la risoluzione di problemi basati su grafi

A novel Logic Tensor Networks framework for addressing graph-based problems

Supervisor

Prof. Domenico Ursino

Candidate

Vito Scaraggi

Co-supervisors

Dr. Francesco Cauteruccio

Dr. Enrico Corradini

Academic Year 2023-2024

Nobody ever figures out what life is all about,
and it doesn't matter.
Explore the world.
Nearly everything is really interesting
if you go into it deeply enough.

Richard P. Feynman

Abstract

In the Artificial Intelligence research field, neurosymbolic approaches aim to combine neural networks and symbolic models to overcome their respective limitations and pave the way for more robust and explainable AI capable of reasoning. A notable example of this neurosymbolic integration is represented by the Logic Tensor Network (LTN), a framework leveraging Real Logic to build a model that maximizes the satisfiability of a set of logical facts in a knowledge base. With the goal of exploiting rule-based knowledge in graph-based tasks, this thesis proposes a framework called LTN-GCN, which merges LTNs and Graph Convolutional Neural Networks (GCNs) to address node and edge classification within graphs. The experimental results on a citation graph and a drug-drug interaction network show that LTN-GCN achieves performance levels comparable to other GCN models.

Keywords: NeuroSymbolic AI, Logic Tensor Network, Graph Convolutional Neural Network, Node Classification, Edge Classification, Graph-based task

Sommario esteso

Nel campo della ricerca sull'Intelligenza Artificiale, gli approcci neurosimbolici uniscono le reti neurali e i modelli simbolici per superare le rispettive limitazioni e favorire lo sviluppo di un'AI più robusta, *explainable* e capace di *reasoning*. Un esempio notevole di questa integrazione neurosimbolica è rappresentato da Logic Tensor Network (LTN), un framework che sfrutta la Real Logic per costruire modelli che massimizzano la soddisfacibilità di un insieme di fatti logici appartenenti a una *knowledge base*. Con l'obiettivo di impiegare la conoscenza *rule-based* per risolvere task basati su grafi, questa tesi propone un framework chiamato LTN-GCN, che combina LTN e Graph Convolutional Neural Networks (GCN) per risolvere problemi di classificazione di nodi o di archi all'interno di grafi. Nella metodologia presentata, le GCN sono utilizzate per definire la semantica o *grounding* di predicati e funzioni all'interno di formule logiche del primo ordine. In particolare, le reti convoluzionali a grafo impiegate sono composte da *layer* convoluzionali di tipo GCN o GraphSAGE. Inoltre, il framework include la formulazione di *knowledge base* specifiche per la risoluzione di problemi di classificazione binaria o multi-classe. Per valutare le prestazioni di LTN-GCN, sono stati condotti esperimenti su due dataset distinti, associati, rispettivamente, a task di classificazione di nodi e di archi. Il primo dataset è *ogbn-arxiv*, un *citation graph* di articoli scientifici di informatica, il cui task consiste nel predire la corretta categoria di appartenenza di un articolo. Utilizzando una *knowledge base* che esprime l'omofilia del grafo, LTN-GCN ottiene metriche di *accuracy* e *F1-score* paragonabili a quelle delle tradizionali GCN. Il secondo insieme di esperimenti utilizza una *drug-drug interaction network* ricavata dal dataset *ogbl-biokg*. Il task associato al dataset consiste nell'identificazione dell'insieme degli effetti collaterali relativi a una coppia di farmaci. Anche in questo caso, LTN-GCN raggiunge prestazioni paragonabili alle GCN tradizionali in termini di *accuracy*.

Parole chiave: NeuroSymbolic AI, Logic Tensor Network, Graph Convolutional Neural Network, Node Classification, Edge Classification, Graph-based task

Contents

List of Figures

List of Tables

Introduction

In the broad realm of Artificial Intelligence (AI), Deep Learning (DL) stands out for its importance and fundamental contributions to the entire field. DL, which is a subset of Machine Learning that uses multi-layered neural networks, offers a state-of-the-art solution for a wide range of different tasks, ranging from Computer Vision to Natural Language Processing. The outstanding results obtained in several domains and applications have brought new life to the AI sector, attracting increasing attention and investments from big tech companies. The AI race has lead to the development of ever-larger models, such as the much-discussed Large Language Models, which, however, are not without their drawbacks. In fact, the training of these models requires a lot of labeled data and is very energy-intensive. In addition, these networks are often described as "black boxes" as they lack interpretability, which is an indispensable requirement in the most critical use cases, e.g. those related to healthcare and human safety.

A promising shift in this trend is emerging through Neurosymbolic (NeSy) AI, which aims to integrate the current widespread deep neural network with symbolic methods dealing with the representation of knowledge and the manipulation of logic. The objective is to conceive hybrid models that inherit the advantages of neural networks, like the capability of extracting useful patterns from data and the efficient *learning by example* paradigm, and those of symbolic techniques that characterized the very first study in the AI field, like the possibility to incorporate domain knowledge and high explainability. NeSy AI has already produced relevant results in the computer vision area, where new methods have been proposed for semantic image interpretation, and in robot navigation, where "true reasoning" decision support systems have been prototyped.

One notable framework within this paradigm is the Logic Tensor Network (LTN), which realizes the neurosymbolic integration by encoding logical rules into the structure of a neural network. The theoretical foundations of LTNs rely on Real Logic, an extension of First Order Logic which defines a concrete semantics, called grounding, for symbols contained in logical formulas, i.e. variables, constants, operators, aggregators, predicates, and functions. Neural networks play a role in defining these groundings. Since Real Logic is fully differentiable, background propagation is supported. Specifically, the training seeks to maximally satisfy a set of custom logical rules, which can be defined based on the specific task.

Collocated in the context of NeSy AI, this work wants to contribute to the solution of graph-based tasks by exploiting the commonsense and domain knowledge available. In the-state-of-the-art, graph-related problems, such as node classification and link prediction, are mostly addressed with *ad-hoc* networks, called Graph Neural Networks (GNNs), which are designed to handle graph data structures. A particular type of GNN is that of Graph Con-

volutional Neural Network, characterized by the presence of a few generalized convolutional layers that compute the hidden representation of a central node from those of its neighbors. However, these methods do not integrate knowledge related to the task, which, instead, can contribute to the solution, and are based on a loss function independent from the task.

Our proposal consists of LTN-GCN, a framework using a Real Logic knowledge base to train an embedded Graph Convolutional Neural Network for addressing node classification and edge classification tasks. We design knowledge bases for training LTN-GCN on binary and multi-class classification and formalize groundings for logical symbols. The architecture includes a relatively simple Graph Convolutional Neural network, composed of GCN or GraphSAGE layers, to ground predicates and functions in the knowledge base, and a Stable Product Real Logic configuration to ground fuzzy operators. The implementation uses the Python modules *LTNTorch*, *torch*, and *torch geometric*.

To test LTN-GCN, we conducted some experiments on two different datasets. The first is *ogbn-arxiv*, a citation network of computer science papers where the task requires predicting the correct category of a paper; this is a multi-class single-label node classification. Using a knowledge base that "captures" the graph homophily, LTN-GCN obtained accuracy and F1-score values comparable to those of standalone graph neural networks.

The second set of experiments uses a drug-drug interaction network obtained from the *ogbl-biokg* dataset. The related prediction task is a multi-class multi-label edge classification, where the goal is to identify the set of side effects associated with a pair of drugs. Again, LTN-GCN obtained a performance comparable to the one of traditional graph neural networks and achieved consistent accuracy scores.

This thesis is organized as follows:

- Chapter 1: introduces Neurosymbolic AI, discussing its main concepts, applications, and the integration of symbolic and neural approaches. It covers the historical background of AI, contrasting Symbolic and Connectionist paradigms, and reviews Neurosymbolic AI architectures based on H. Kautz's taxonomy.

- Chapter 2: provides an overview of Logic Tensor Networks, starting with Real Logic and the concept of grounding. It then discusses the semantics of connectives and quantifiers, focusing on Stable Real Product Logic, and concludes by outlining LTN tasks, like learning, reasoning, and querying.

- Chapter 3: provides an overview of Graph Neural Networks, explaining their components and categorizing them by mechanisms like convolutional and attention-based models. It also discusses recent advancements in combining GNNs with Neurosymbolic AI and their applications across various fields.

- Chapter 4: introduces the LTN-GCN framework for graph-based tasks, focusing on knowledge base construction and the use of graph convolutional neural networks for grounding. It also covers the implementation details using the *LTNTorch* package.

- Chapter 5: evaluates the LTN-GCN framework for node classification using the *ogbn-arxiv* dataset, proposing a knowledge base with axioms for training. It concludes by detailing the experimental setup and presenting the results.

- Chapter 6: applies the LTN-GCN framework to edge classification using the *ogbl-biokg* biomedical knowledge graph. It details data preprocessing, multi-class multi-label classification with Real Logic, and discusses experimental results.

CHAPTER 1

Neurosymbolic AI

This chapter briefly introduces Neurosymbolic Artificial Intelligence by focusing on its main concepts and applications. In Section 1.1 we present a historical overview of AI research and explain the main differences between Symbolic and Connectionist paradigms, which are conceptually opposite in how they pursue human-like intelligence. Next, in Section 1.2, we present Neurosymbolic AI and describe motivations behind the recent research interest in merging neural and symbolic approaches, providing some examples of applications. In Section 1.3, we define several concepts and give a more clear context to neurosymbolic integration. Finally, in Section 1.4 we review neurosymbolic architectures outlined in H. Kautz's taxonomy.

1.1 Symbolic AI vs Connectionist AI

"Symbols vs Neuron" has always been a heated debate in the history of the Artificial Intelligence field. While, in the past, experts struggled in the attempt to declare a winner between symbolic and neural approaches, the current view believes that they are complementary, not competing.

Symbolic AI, also called rule-based or logic-based AI, is the subfield of Artificial Intelligence that deals with manipulating symbols, i.e. language items usually expressed in a kind of formal logic. These methods are typically employed in *knowledge representation and reasoning* tasks which were first addressed in the early years of AI studies. The birth of AI, conventionally dated in the summer of 1956 at the Dartmouth College workshop (Figure 1.1), coincides with the idea of a "computer program capable of thinking non-numerically" but in a human fashion, supported by deductive logic. This core concept, also referred to as Good Old Fashioned AI (GOFAI), initially brought to theorem-proving algorithms, game-playing programs and problem solvers; later, in the '80s, it led to expert systems. However, these algorithms proved effective only within clear boundaries ("microworlds") and had poor capability to tackle real-world challenges.

After several stagnant periods for AI (i.e. the first and the second "AI winter"), researchers came up with neural networks, which are the building blocks of Connectionist or Subsymbolic AI. The first work on artificial neurons was done by McCulloch and Pitts in 1943 and neural net prototypes were conceived in the '50s; however, neural networks did not meet great success until the '80s, when the backpropagation algorithm was re-discovered. This turning point shifted most of AI research effort from symbolic to connectionist approaches, overcoming the difficult problem of inserting all needed knowledge into a system, called "knowledge bottleneck". Nowadays, deep learning can boast of state-of-art performances in

disparate application fields, ranging from Natural Language Processing to Computer Vision.

Symbolic and Connectionist AI reveal both strengths and weaknesses. The former provides an easy way to integrate *commonsense* and *domain knowledge* directly into models, that are generally human-readable and explainable. A big difficulty resides in finding or producing high quality knowledge for these systems as well as reducing their computational complexity and "brittleness" to errors. The latter can efficiently extract useful patterns and rules from raw unstructured data and shows desirable robustness to partial or noisy information. The main drawbacks are that most algorithms are "data hungry" and unable to generalize well when a dataset shift occurs. Moreover, neural networks are essentially "black boxes" whose internal decision making process cannot be explained unless *post-hoc* techniques are adopted.

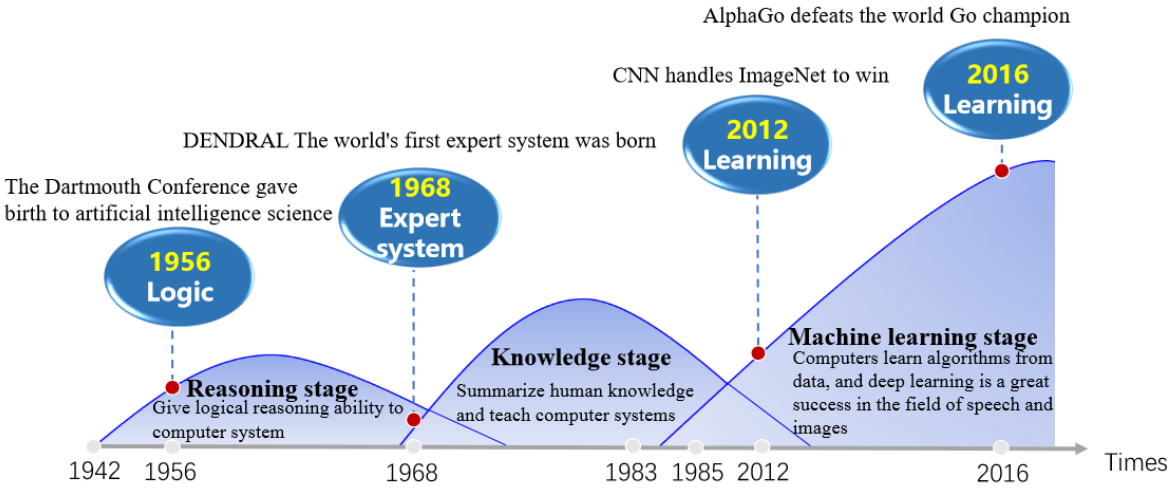


Figure 1.1: Brief history of AI (source: Yu *et al.* [2023])

Previous observations suggest that a hybrid approach can be developed to mitigate symbolic and connectionist shortcomings and exploit their advantages.

1.2 The third wave: neurosymbolic integration

Neurosymbolic (NeSy) AI is a research field in Artificial Intelligence that aims to achieve a "best of both worlds scenario" by pursuing the integration of neural and symbolic components (Figure 1.2). Research interest in this area is justified by two major reasons: the first is a better understanding of human thought process, specifically investigating how logic-based symbol manipulation can arise from perception (Hitzler *et al.* [2022]); the second is identifying necessary principles to make AI trustworthy for humans, especially in critical domains. In fact, there exist various contexts, including medical diagnosis and autonomous driving, where relying solely on perception can present limitations or yields unsatisfactory outcomes (Yu *et al.* [2023]).

NeSy AI proof of concept was initially proposed in the '80-'90s by researchers in AI and cognitive sciences but the formal definition of neuro-symbolic reasoning can be traced back to the IJCAI 2005 workshop dedicated to the topic. Although studies began decades ago, only recent advances in deep learning have renewed interest in the field, which has also become attractive for big industries such as IBM. The AAAI-2020 conference, attended by Francesca Rossi, Geoffrey Hinton, Yoshua Bengio and Yann Lecun, has pointed out the crucial importance of NeSy AI in shaping future research. It remarked on the need to include a sound reasoning layer in deep learning based methods in order to build rich

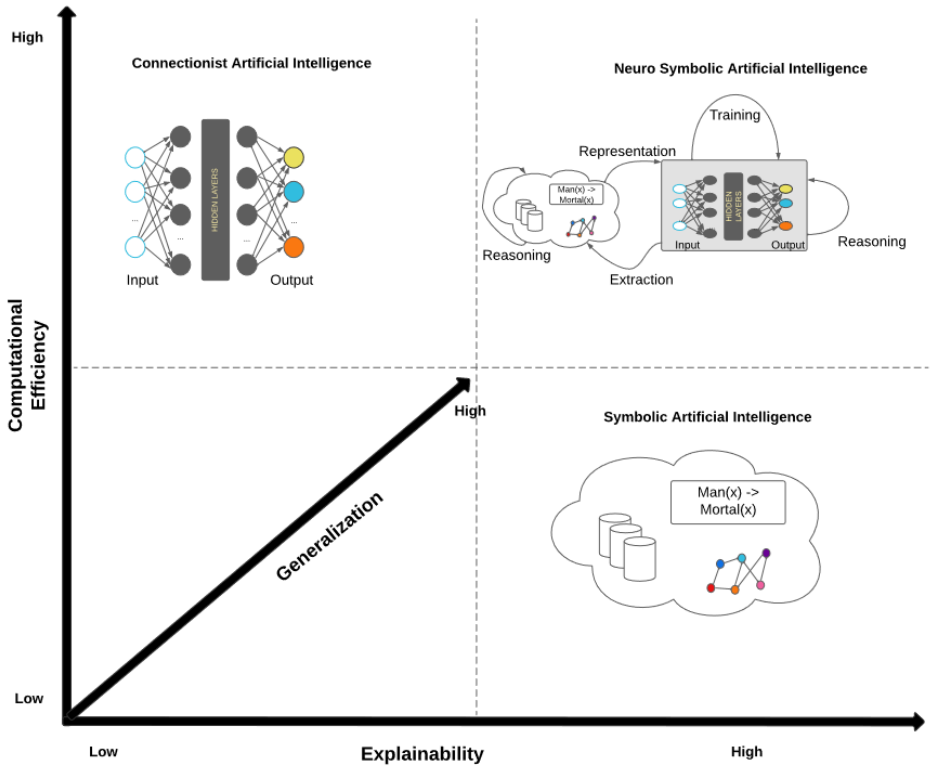
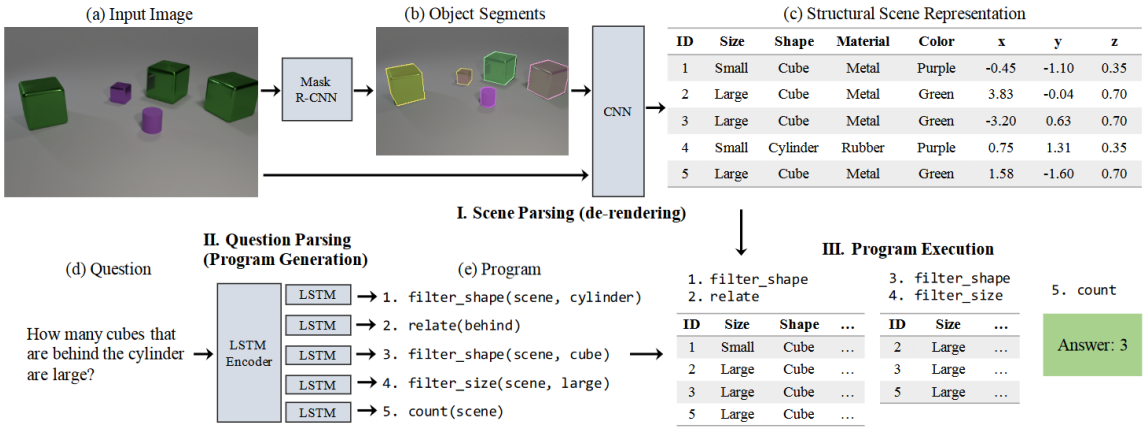


Figure 1.2: Neurosymbolic integration allows reaching high levels of computational efficiency, explainability and generalization at the same time (source: Bhuyan *et al.* [2024])

and robust AI systems (d'Avila Garcez and Lamb [2020]). Participants have converged on the idea that breakthroughs in AI are more likely to come from neurosymbolic integration rather than standalone neural networks: NeSy AI may lead to a transition from currently available Narrow AI, trained to perform a single specific task, to the ambitious General AI, theoretically capable of any human intellectual task. Another notable fact is that NeSy AI is tightly coupled to Explainable Artificial Intelligence (XAI) as embedding knowledge into systems is a fundamental step to make their decision process understandable by humans.

From a practical perspective, NeSy AI has been applied successfully in a large variety of different domains. In the Computer Vision area, NeSy approaches have shown the ability to enhance complex scene understanding by reasoning on abstract visual concepts. In (Donadello *et al.* [2017]), the authors leverage Logic Tensor Networks for Semantic Image Interpretation, i.e. the task of extracting structured semantic descriptions from images. In addition, NeSy AI is not limited to traditional Computer Vision tasks but it also has great potential in multimodal tasks. This is the case of Neurosymbolic Visual Question Answering (NS-VQA) (Figure 1.3), where deep learning and a symbolic program are combined to answer visual queries expressed in natural language.

NeSy integration has also been investigated about knowledge graph reasoning (Wang *et al.* [2019]), reinforcement learning (Garnelo *et al.* [2016]) and optimization (Sen *et al.* [2021]). NeSy applications in the medical industry are responsible for more accurate and personalized diagnoses, while contributions in robotics are supportive of robot navigation and decision making in complex environmental settings.



ID	Size	Shape	Material	Color	x	y	z
1	Small	Cube	Metal	Purple	-0.45	-1.10	0.35
2	Large	Cube	Metal	Green	3.83	-0.04	0.70
3	Large	Cube	Metal	Green	-3.20	0.63	0.70
4	Small	Cylinder	Rubber	Purple	0.75	1.31	0.35
5	Large	Cube	Metal	Green	1.58	-1.60	0.70

ID	Size	Shape	...
1	Small	Cube	...
2	Large	Cube	...
3	Large	Cube	...
5	Large	Cube	...

ID	Size	...
2	Large	...
3	Large	...
5	Large	...

Figure 1.3: Neurosymbolic Visual Question Answering is composed of three main components: a scene-parser produces structural scene representations from images in input; a question parser converts a visual query into a program; an executor runs the obtained program on the structural image representation to find an answer (source: Yi *et al.* [2019])

1.3 Neurosymbolic components

In this section, we define the fundamental elements of neurosymbolic systems and delve into concepts such as representation, grounding, learning, reasoning and logic.

Representation and grounding

"Localist" and "distributed" are terms frequently used to indicate how objects are represented in AI systems. Localist representation, common in symbolic AI, involves using isolated symbols to stand in for abstract ideas or concrete objects, while distributed representation, adopted in deep learning, employs a vector to capture different features of an object (Bhuyan *et al.* [2024]). NeSy AI leverages both representations.

Grounding is the process of transforming abstract symbols in items in a vector space. This operation is needed for generating data manageable by neural networks.

Learning and Reasoning

Two fundamental aspects of intelligence are learning and reasoning. Learning is the task of making generalizations when specific example data are provided; reasoning is the task of deriving new facts from available knowledge. It can be declined in various forms (deduction, induction and abduction) and be precise, i.e. deterministic, or approximate, i.e. characterized by uncertainty. Specifically, *commonsense reasoning* is a type of approximate reasoning that makes assumptions on general or experience knowledge. On the other hand, *combinatorial reasoning* integrates the capability to deal with mathematical problems.

NeSy systems differentiate in the way learning and reasoning are assembled. The "learning for reasoning" paradigm involves the use of neural networks to reduce the search space of symbolic systems, thereby fastening computation. Meaningful symbols are extracted for the next reasoning process. The dual "reasoning for learning" concept underlies methods that integrate prior knowledge into the training phase in the form of loss regularization terms or logic rules. Finally, in "learning - reasoning" the two components interact in a perfectly balanced setting, where each one serves the other.

Logic

The design of neurosymbolic frameworks depends on the choice of a specific logic language to manipulate knowledge. A viable option is Propositional Logic (PL), widely employed in NeSy architectures for its effectiveness and simplicity. A PL rule is made of simple connectives and logical variables, such as the following example:

$$A \wedge B \implies C \tag{1.3.1}$$

Here A is the proposition "Alice is Bob's parent", B is the proposition "Bob is Charlie's parent" and C is the proposition "Alice is Charlie's grandparent".

First Order Logic (FOL) introduces predicates, functions and quantifiers and results more expressive than PL and suitable for more complex problems. Returning to the example, we can easily reformulate it in FOL:

$$\forall x,y,z \quad Parent(x,y) \wedge Parent(y,z) \implies Grandparent(x,z) \tag{1.3.2}$$

where $Parent(x,y)$ is a predicate that holds if "x is parent of y" and $Grandparent(x,y)$ is a predicate that holds if "x is grandparent of y".

High-order logic (HOL) with quantification over predicates and functions has also been explored in the NeSy context; however, it raises doubts about computational complexity.

Since neural networks run approximate reasoning, in many NeSy scenarios forms of real fuzzy logic are necessary. Real Logic is a many-valued logic in which the truth degree of a formula can assume any value in the continuous interval $[0,1]$ and numerical implementations for connectives (AND, OR, NOT) and quantifiers are given. The Real Logic definition used in Logic Tensor Networks is presented in Chapter 2.

Another widespread formalism used to represent knowledge bases is the knowledge graph. KGs are directed and labeled graphs where nodes stand for semantic symbols and edges correspond to semantic relationships between nodes. Figure 1.4 shows a simple knowledge graph that implements the previous example.

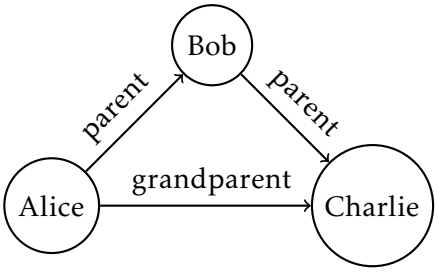


Figure 1.4: Example of Knowledge Graph

1.4 Neurosymbolic architectures

A full taxonomy of neurosymbolic architectures was provided by H. Kautz in Robert S. Englemore Memorial Lecture at AAAI 2020 (Kautz [2022]). He surveyed six possible designs, numbered in ascending order according to their maturity level in implementing the neurosymbolic integration.

Type 1 or Symbolic Neuro symbolic

Type 1 (Figure 1.5) is the Standard Operating Procedure (SOP) of current deep neural networks. Symbolic input is grounded into vectors passed to the network; output is converted

into a symbolic category, for instance, using a softmax function. Leaving out symbolic preprocessing and postprocessing layers, these systems are essentially connectionist.

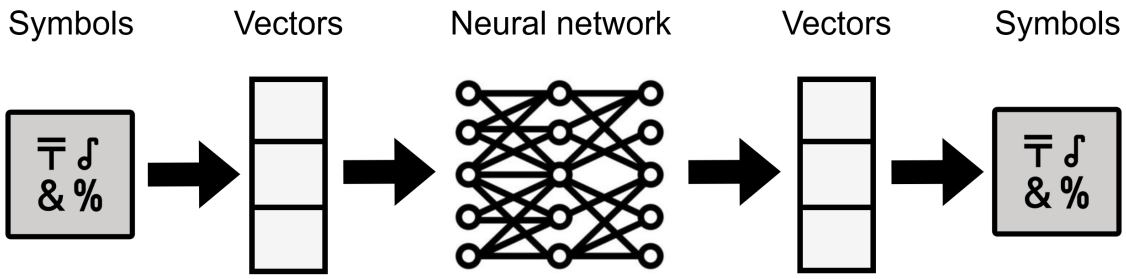


Figure 1.5: NeSy Type 1 diagram

Type 2 or Symbolic[Neuro]

In Type 2 (Figure 1.6), a pattern recognition subroutine is used as a heuristic function within a broader symbolic problem solver. DeepMinds's AlphaGo, the first computer program that managed to beat a world champion Go player, is a prototypical example of this design and embeds the heuristic search algorithm Monte Carlo Tree Search (MCTS). Type 2 is predominantly symbolic.

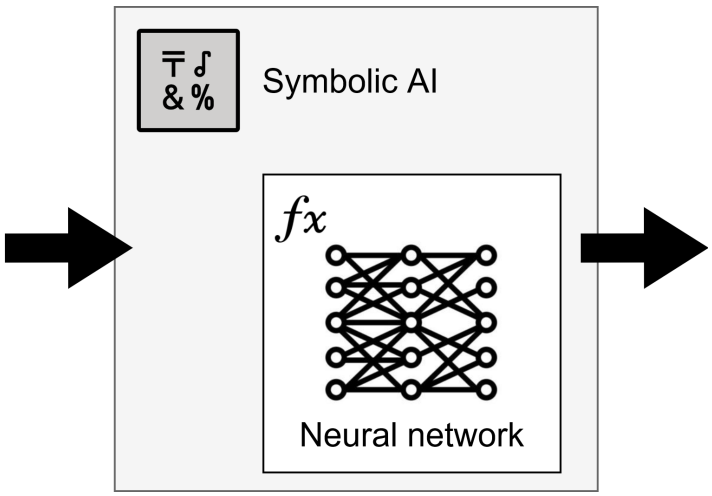


Figure 1.6: NeSy Type 2 diagram

Type 3 or Neuro|Symbolic

In Type 3 (Figure 1.7), a neural network converts non-symbolic input into a symbolic data structure, while a parallel symbolic system executes a complementary task. The interaction between neural and symbolic components is highly cooperative. Notable instances of Type 3 architectures are Concept Learner (Mao *et al.* [2019]), a hybrid pipeline for visual question answering, and DeepProbLog (Manhaeve *et al.* [2018]), which relies on neural networks to calculate the probabilities of probabilistic facts and performs reasoning through ProbLog inference engine.

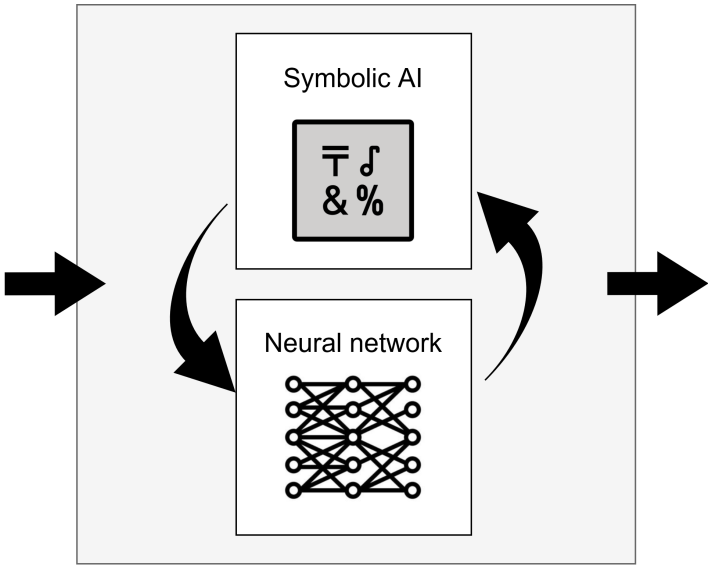


Figure 1.7: NeSy Type 3 diagram

Type 4 or Neuro:Symbolic→Neuro

Type 4 (Figure 1.8) structures compile symbolic knowledge into the training or the design of a neural network. An example is provided in (Lample and Charton [2019]): in this work, a sequence-to-sequence transformer-based model is trained to simplify mathematical expressions. Despite critics about effective reasoning capabilities, Kautz also classifies Graph Neural Networks (GNN) as Type 4. GNNs are more recently used for knowledge graph reasoning.

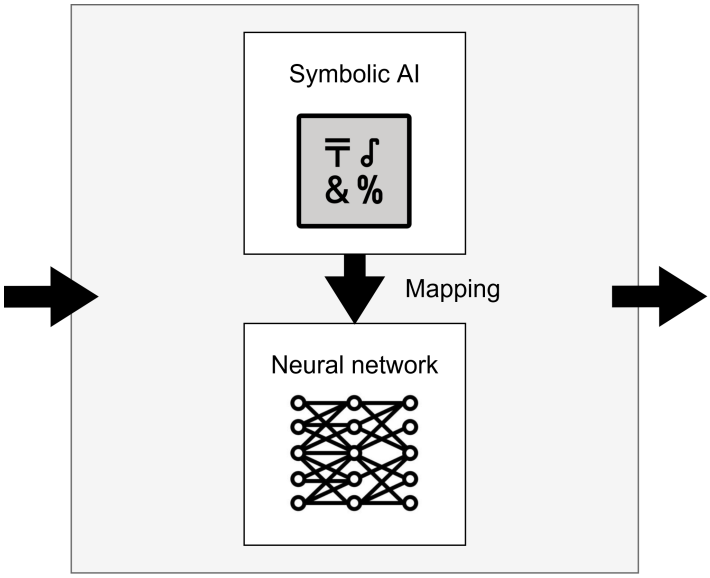


Figure 1.8: NeSy Type 4 diagram

Type 5 or Neuro_Symbolic

Type 5 (Figure 1.9) encodes symbolic rules into loss function and network weights. Logic Tensor Networks (LTNs) belong to this category. LTNs approximate FOL rules with fuzzy

logic: logic elements are grounded with real-valued tensors, enabling gradient-based learning and end-to-end training.

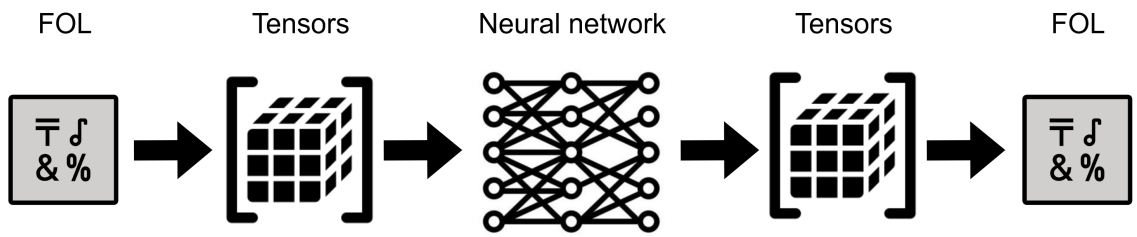


Figure 1.9: NeSy Type 5 diagram

Type 6 or Neuro[Symbolic]

Type 6 is referred to as a system capable of "true symbolic reasoning inside a neural engine". In Kautz's view, it represents the most promising trade-off between logic-based and neural-based AI, inspired by Daniel Kahneman's theory of "Thinking Fast and Slow". Kahneman states that brain activities are carried out by two distinct mechanisms: System 1, responsible for unconscious and intuitive aspects that require little and fast computations, and System 2, used for slow reasoning based on logic and planning. Nevertheless, implementations of type 6 do not exist yet.

CHAPTER 2

Logic Tensor Networks

This chapter provides an overview of Logic Tensor Networks (LTN). Section 2.1 introduces Real Logic, the foundational component of LTNs, and explains the concept of grounding. In Section 2.2, we explore the semantics of connectives and quantifiers, which are represented using fuzzy operators and aggregators. In particular, we present a range of possible configurations, then narrow our focus to Stable Real Product Logic, a popular choice in the literature due to its favorable properties for gradient-descent optimization. Finally, Section 2.3 outlines the various tasks that LTNs can perform, such as learning, reasoning and querying.

2.1 Real Logic

Real Logic is based on a First-Order Logic (FOL) language $\mathcal{L}$ defined by the signature $\mathcal{L}(\mathcal{C},\mathcal{X},\mathcal{F},\mathcal{P})$, which includes a set of constants $\mathcal{C}$, a set of logical variables $\mathcal{X}$, a set of functions $\mathcal{F}$ and a set of predicates $\mathcal{P}$. In $\mathcal{L}$, all elements are typed, i.e. they belong to a domain. Functions D, D_{in} and D_{out} allow us to represent domains:

$$D : \mathcal{X} \cup \mathcal{C} \to \mathcal{D} \tag{2.1.1}$$

$$D_{in} : \mathcal{F} \cup \mathcal{P} \to \mathcal{D}^* \tag{2.1.2}$$

$$D_{out} : \mathcal{F} \to \mathcal{D} \tag{2.1.3}$$

$D(x)$ and $D(c)$ (2.1.1) return the domain of a variable x and a constant c, respectively. D_{in} (2.1.2) returns the domains $\mathcal{D}^*$ of the arguments of a predicate or a function. D_{out} (2.1.3) returns the domain of function output. A predicate can be 0-ary, i.e. it can have no arguments.

We define a term as any combination of constants, variables and functions. In Real Logic, applying a predicate to a correct number of terms returns a real number in the continuous interval $[0,1]$, not in $\{true, false\}$. The truth degree denotes "how much a formula is true" and can assume infinite values.

A well-formed formula in Real Logic is defined recursively by the following statements:

1. $t_1 = t_2$ is an atomic formula, where t_1 and t_2 are terms and $D(t_1) = D(t_2)$;
2. $p(t)$ is an atomic formula, where p is a predicate and $D(t) = D_{in}(p)$;
3. if ϕ and ψ are formulas and $x_1, ..., x_n$ are n distinct variables then $\diamond\phi$, $\phi \circ \psi$ and $Qx_1...x_n\phi$ are all formulas. Here $\diamond$ is an item in the set of unary operators $\{\neg\}$, $\circ$ is an item in the set of binary operators $\{\wedge, \vee, \implies, \iff\}$ and Q is a quantifier $\in \{\exists, \forall\}$.

Unlike FOL, the semantics of Real Logic is *concrete* and domains are interpreted as tensors in the real field. Terms are interpreted as real-valued tensors, functions are interpreted as tensor operations and formulas are interpreted as functions mapping tensors to a real value $\in [0,1]$.

We use the expression *grounding* $\mathcal{G}$ to refer to a function that associates real-valued features to logic symbols in $\mathcal{L}$. Definition 2.1.1 determines how constants, variables, functions and predicates are grounded in Real Logic.

Definition 2.1.1 (Grounding)**.** A *grounding* $\mathcal{G}$ is a function defined on the signature $\mathcal{L}(\mathcal{C},\mathcal{X},\mathcal{F},\mathcal{P})$ that satisfies the following conditions:

- $\mathcal{G}(c) = d \in \mathcal{G}(D(c)), \quad \forall x \in \mathcal{C}$
  i.e. the grounding of a constant c is an item in the set $\mathcal{G}(D(c)) \subseteq \mathbb{R}^n$;
- $\mathcal{G}(x) = \langle d_1 ... d_k \rangle \in \bigtimes_{i=1}^{k} \mathcal{G}(D(x)), \quad \forall x \in \mathcal{X}, k \in \mathbb{N}_0^+$
  i.e. the grounding of a variable x is a sequence of length k of individuals in $\mathcal{G}(D(x)) \subseteq \mathbb{R}^n$;
- $\mathcal{G}(f) \in \mathcal{G}(D_{in}(f)) \to G(D_{out}(f)), \quad \forall f \in \mathcal{F}$
  i.e. the grounding of a function f is a function with parameters in $\mathcal{G}(D_{in}(f)) \subseteq \mathbb{R}^{n \cdot \alpha(f)}$ that returns values in $\mathcal{G}(D_{out}(f)) \subseteq \mathbb{R}^n$;
- $\mathcal{G}(p) \in \mathcal{G}(D_{in}(p)) \to [0,1], \quad \forall p \in \mathcal{P}$
  i.e. the grounding of a predicate p is a function with parameters in $\mathcal{G}(D_{in}(p)) \subseteq \mathbb{R}^{n \cdot \alpha(p)}$ that returns values in $[0,1]$.

Here $\alpha(\cdot)$ is the arity of a predicate or a function.

Definition 2.1.1 can be extended to all terms and formulas in $\mathcal{L}$ applying the following rules:

- $\mathcal{G}(t)$ is a tensor with dimension $(|\mathcal{G}(x_1)|,...,|\mathcal{G}(x_n)|)$, where $x_1,...,x_n$ are free variables in the term t;
- $\mathcal{G}(f(t_1,...,t_m)) = \mathcal{G}(f)(\mathcal{G}(t_1),...,\mathcal{G}(t_m))$ for terms $t_1,...,t_m$ and any function f;
- $\mathcal{G}(p(t_1,...,t_m)) = \mathcal{G}(p)(\mathcal{G}(t_1),...,\mathcal{G}(t_m))$ for terms $t_1,...,t_m$ and any predicate p.

In order to have a clearer understanding of the concept of grounding, consider the Example 2.1.1.

Example 2.1.1 (Grounding)**.** Suppose that we have a set of domains $\mathcal{D} = \{V, W\}$ and the following groundings:

$$\begin{aligned}
&\mathcal{G}(V) = \mathbb{R}^+ \\
&\mathcal{G}(W) = \mathbb{R}^- \\
&\mathcal{G}(x) = \langle v_1, v_2, v_3 \rangle \\
&\mathcal{G}(y) = \langle w_1, w_2 \rangle \\
&\mathcal{G}(f) = x, y \to x \cdot y \\
&\mathcal{G}(p) = x, y \to \sigma(x + y)
\end{aligned}$$

Then the term $f(x,y)$ and the formula $p(x, f(x,y))$ are grounded as:

$$\mathcal{G}(f(x,y)) = \begin{pmatrix} v_1 \cdot w_1 & v_1 \cdot w_2 \\ v_2 \cdot w_1 & v_2 \cdot w_2 \\ v_3 \cdot w_1 & v_3 \cdot w_2 \end{pmatrix}$$

$$\mathcal{G}(p(x,f(x,y))) = \begin{pmatrix} \sigma(v_1 + v_1 \cdot w_1) & \sigma(v_1 + v_1 \cdot w_2) \\ \sigma(v_2 + v_2 \cdot w_1) & \sigma(v_2 + v_2 \cdot w_2) \\ \sigma(v_3 + v_3 \cdot w_1) & \sigma(v_3 + v_3 \cdot w_2) \end{pmatrix}$$

Note that, given $\mathcal{G}(D_{out}(f)) = \mathbb{R}$, $\mathcal{G}(f(x,y))$ has dimension $|\mathcal{G}(x)| \times |\mathcal{G}(y)| \times 1$. The first two dimensions are used for indexing free variables x, y, the latter is called *feature dimension* as it depends on the output domain of f. With a grounding $\mathcal{G}(D_{out}(f)) = \mathbb{R}^m$, $\mathcal{G}(f(x,y))$ would have dimension $|\mathcal{G}(x)| \times |\mathcal{G}(y)| \times m$.

The grounding $\mathcal{G}(p(x,f(x,y)))$ has always dimension $|\mathcal{G}(x)| \times |\mathcal{G}(y)| \times 1$ since the predicate output domain is $[0,1]$. Tensors are "squeezed" for simplicity, i.e. dimensions of cardinality 1 are omitted. Figure 2.1 shows a graphical representation of Example 2.1.1 called *computational graph*.

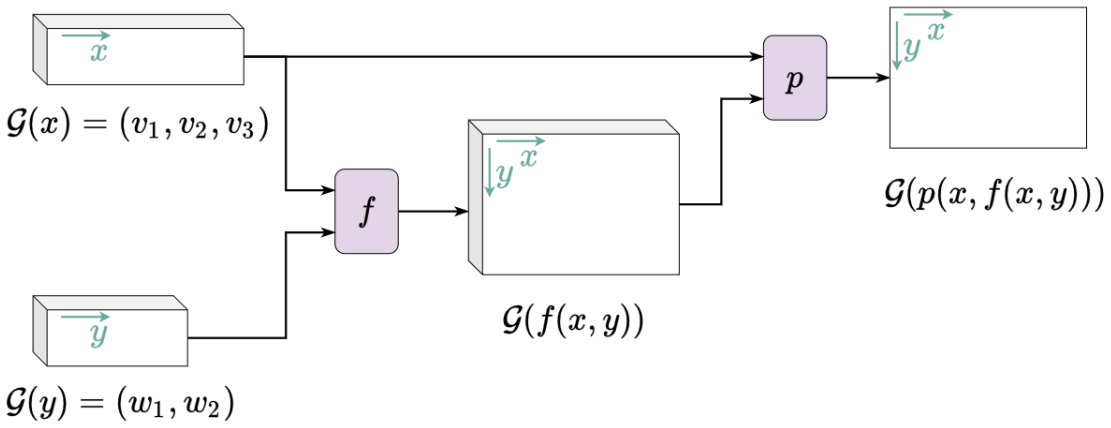


Figure 2.1: In this graphical convention, the width and length of boxes are associated with *indexing dimensions*, while depth represents optional *feature dimensions*. The final result is the evaluation of predicate p for every combination of individuals from $\mathcal{G}(x)$ and $\mathcal{G}(y)$ (source: Badreddine *et al.* [2022])

2.2 Connectives and Quantifiers

In Real Logic, the semantics of connectives and quantifiers is defined using fuzzy operators and aggregators. These functions should mimic the behavior of classical First-Order Logic (FOL) operators and yield the same truth tables when inputs are restricted to $\{0,1\}$.

Since fuzzy operators influence the learning process in LTNs, their derivative properties are crucial. In particular, LTN learning based on gradient-descent optimization might be affected by the following issues:

- *Vanishing gradient*: the gradient approaches zero in certain regions of the domain and corresponding weights are not updated;
- *Exploding gradient*: the gradient tends to diverge causing instability during learning;

- *Single-passing*: the gradient is non-zero for only one argument and learning is limited to one input at a time.

Next, we present various configurations of fuzzy operators and aggregators, each distinguished by different logical properties and derivative concerns.

2.2.1 Fuzzy operators

A fuzzy operator is a function $FuzzyOp(\cdot)$ that implements a logical connective. Formulas containing fuzzy operators can be grounded as follows. Let ϕ and ψ be two formulas, $\diamond$ a unary operator, $\circ$ a binary operator and $FuzzyOp$ a fuzzy operator then we can define the following groundings:

- $G(\diamond\phi)$ is a tensor obtained by applying $FuzzyOp(\diamond)$ element-wise to every individuals in ϕ;
- $G(\phi \circ \psi)$ is obtained by applying $FuzzyOp(\circ)$ element-wise to every combination of individuals from ϕ and ψ.

Connectives $\neg, \wedge, \vee$ and $\implies$ are implemented by *fuzzy negation* (N), *t-norm* (T), *t-conorm* (S) and *fuzzy implication* (I), respectively. In other works, t-norms and t-conorms are defined with two arguments, however, we opt for a more general notation based on m arguments, which can be simplified to the specific case of two arguments by setting $m = 2$. This alternative formulation is functional for introducing aggregators and *Stable Product Real Logic*.

Fuzzy negation

A fuzzy negation is a monotonically decreasing function $N : [0,1] \to [0,1]$ such that $N(0) = 1$, $N(1) = 0$. A fuzzy negation is said to be *strict* if it is continuous and strictly decreasing, and *strong* if $N(N(x)) = x, \forall x \in [0,1]$. The most common implementation of N in literature is the one reported in Equation 2.2.1:

$$N_s(x) = 1 - x \tag{2.2.1}$$

N_s is both *strict* and *strong*.

T-norm

Triangular norms or *T-norms* are the fuzzy operators used to represent conjunction ($\wedge$). T-norms are commutative, associative functions characterized by monotonicity (i.e. when all but one argument is fixed, the function increases monotonically as that argument increases) and neutrality for 1-valued arguments (i.e. arguments set to 1 are ignored). Common *T-norms* functions are (2.2.2):

$$\begin{aligned}
&\textit{Goedel t-norm} && T_G(x_1,...,x_m) = \min_{i=1...m} x_i \\
&\textit{Product (Goguen) t-norm} && T_{prod}(x_1,...,x_m) = \prod_{i=1}^{m} x_i \\
&\textit{Łukasiewicz t-norm} && T_{LK}(x_1,...,x_m) = \max\left(1 + \sum_{i=1}^{m}(x_i - 1), 0\right)
\end{aligned} \tag{2.2.2}$$

T-conorm

Triangular conorms or *T-conorms* are the fuzzy operators implementing disjunction ($\vee$). *T-conorms* are commutative, associative, monotonically increasing and neutral to 0-valued arguments. Popular implementations are (2.2.3):

$$\begin{aligned}
&\textit{Goedel t-conorm} && S_G(x_1,...,x_m) = \max_{i=1...m} x_i \\
&\textit{Product (Probabilistic Sum) t-conorm} && S_{prod}(x_1,...,x_m) = 1 - \prod_{i=1}^{m}(1-x_i) \\
&\textit{Łukasiewicz t-conorm} && S_{LK}(x_1,...,x_m) = \min\left(\sum_{i=1}^{m} x_i, 1\right)
\end{aligned} \tag{2.2.3}$$

Fuzzy implication

A fuzzy implication is a function $I : [0,1]^2 \to [0,1]$ such that $I(0,0) = I(0,1) = I(1,1) = 1$ and $I(1,0) = 0$. There exist two different classes of implication: a *strong implication* is based on the equivalence $x \implies y \equiv \neg x \vee y$, called *material implication*; a *residual implication* is defined using $x \implies y \equiv \sup\{z \in [0,1] \,|\, x \wedge z \leq y\}$. Common fuzzy implications are (2.2.4):

$$\begin{aligned}
&\textit{Łukasiewicz fuzzy implication} && I_{LK}(x,x') = \min(1-x+x',1) \\
&\textit{Kleene-Dienes S-implication} && I_{KD}(x,x') = \max(1-x,x') \\
&\textit{Reichenbach S-implication} && I_R(x,x') = 1-x+x\cdot x' \\
&\textit{Goedel R-implication} && I_G(x,x') = \begin{cases} 1 & x \leq x' \\ x' & \text{otherwise} \end{cases} \\
&\textit{Goguen R-implication} && I_{prod}(x,x') = \begin{cases} 1 & x \leq x' \\ \frac{x'}{x} & \text{otherwise} \end{cases}
\end{aligned} \tag{2.2.4}$$

The fuzzy operators described above show different behaviors. Table 2.1 lists several common logical properties and indicates whether they hold in *Goedel*, *Product* and *Łukasiewicz* configurations. As we can see, no configuration guarantees all properties.

	Goedel		*Product*		*Łukasiewicz*
	(T_G, S_G, N_S)		$(T_{prod}, S_{prod}, N_S)$		(T_{LK}, S_{LK}, N_S)
	I_{KD}	I_G	I_R	I_{prod}	I_{LK}
Commutativity of $\wedge, \vee$	✓	✓	✓	✓	✓
Associativity of $\wedge, \vee$	✓	✓	✓	✓	✓
Distributivity of $\wedge$ over $\vee$	✓	✓			
Distributivity of $\vee$ over $\wedge$	✓	✓			
Distributivity of $\implies$ over $\vee, \wedge$	✓	✓			
Double negation $\neg\neg\phi = \phi$	✓	✓	✓	✓	✓
Law of excluded middle					✓
Law of non contradiction					✓
De Morgan's laws	✓	✓	✓	✓	✓
Material Implication	✓		✓		✓
Contraposition	✓		✓		✓

Table 2.1: Logical properties for different configurations of fuzzy operators

Regarding learning issues, the *Goedel* configuration is prone to the single-passing problem, while the *Product* and *Łukasiewicz* implementations are susceptible to the vanishing gradient issue. In some extreme cases, the latter may also lead to the exploding gradient problem.

2.2.2 Aggregators

Quantifiers $\forall, \exists$ can be grounded using aggregators. An aggregator is a function $Agg(\cdot) : \cup_{n\in\mathbb{N}}[0,1]^n \to [0,1]$. Intuitively, the grounding $G(Qx_1...x_h\phi)$ is obtained by reducing dimensions associated with $x_1,...,x_h$ using the quantifier $Agg(Q)$. Figure 2.2 shows the computational graph for the grounding $\mathcal{G}(\forall y(\exists x(p(x,y))))$. The application of $\exists x$ reduces the grounding along the dimension x and produces a one-dimensional tensor indexed by the variable y. The second quantification $\forall y$ returns a truth degree as output.

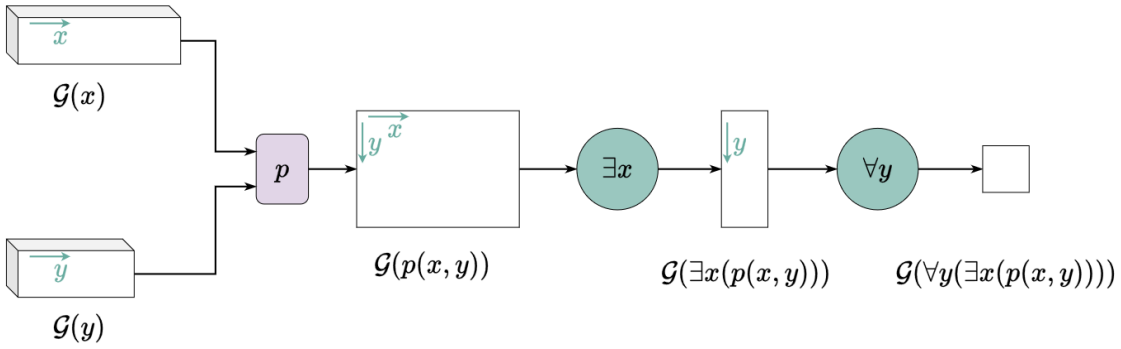


Figure 2.2: Grounding of $\forall y(\exists x(p(x,y)))$ (source: Badreddine *et al.* [2022])

The result of grounding depends on the sequence in which quantifiers are applied unless the aggregator operator is bi-simmetric (Badreddine *et al.* [2022]).

Existential and Universal Quantifier

By recalling some valid equivalences in classical first-order logic (2.2.5):

$$\begin{aligned} \exists x.\phi(x) &\equiv \bigwedge_{x\in X} \phi(x) \\ \forall x.\phi(x) &\equiv \bigvee_{x\in X} \phi(x) \end{aligned} \tag{2.2.5}$$

we could implement existential and universal quantifiers using the previously mentioned t-conorms and t-norms, respectively. In practice, other aggregators are used (2.2.6):

$$\begin{aligned} &\textit{Smoothed Existential Aggregator} && A_{pM}(x_1,...,x_n) = \left(\frac{1}{n}\sum_{i=1}^{n} x_i^p\right)^{\frac{1}{p}} \\ &\textit{Smoothed Universal Aggregator} && A_{pME}(x_1,...,x_n) = 1 - \left(\frac{1}{n}\sum_{i=1}^{n}(1-x_i)^p\right)^{\frac{1}{p}} \\ &\textit{Log-Product (Universal) Aggregator} && A_{\log T_p}(x_1,...,x_n) = \sum_{i=1}^{n}\log(x_i) \end{aligned} \tag{2.2.6}$$

For high positive values of p, A_{pM} converges to Godel (max) t-conorm while A_{pME} converges to Godel (min) t-norm. In particular, for $p = 1$, A_{pM} corresponds to the mean. The

hyperparameter p can be adjusted to control quantifiers' behavior. For A_{pM} an increase in p gives greater weight to true values, while, for A_{pME}, it gives greater weight to false values.

Example 2.2.1 shows a simple use of fuzzy operators and aggregators.

Example 2.2.1 (Fuzzy operators). Given $\mathcal{G}(\phi(x)) = (0.9, 0.7, 0.5)$ and $G(\phi(y)) = (0.8, 0.6, 0.4)$, we want to compute $\mathcal{G}(\forall y.\forall x.\phi(x) \wedge \phi(y))$. Suppose that $FuzzyOp(\wedge)$ is product t-norm T_{prod}, then:

$$\mathcal{G}(\phi(x) \wedge \phi(y)) = \begin{pmatrix} 0.72 & 0.54 & 0.36 \\ 0.56 & 0.42 & 0.28 \\ 0.4 & 0.3 & 0.2 \end{pmatrix}$$

This grounding results from applying $FuzzyOp(\wedge)$ (i.e. product) to every combination of individuals from $\mathcal{G}(x)$ and $\mathcal{G}(y)$. Rows and columns are indexed by variables x and y, respectively. Choosing Goedel t-norm (i.e. minimum) to ground existential quantification over x, we obtain:

$$\mathcal{G}(\forall x.\phi(x) \wedge \phi(y)) = (0.4, 0.3, 0.2)$$

Finally, using A_{pME} with $p = 1$ to ground universal quantification over y, it follows that:

$$\mathcal{G}(\forall y.\forall x.\phi(x) \wedge \phi(y)) = 1 - \frac{(1-0.4) + (1-0.3) + (1-0.2)}{3} = 0.3$$

Diagonal Quantifier

Diagonal quantification or $Diag(x_1, ..., x_h)$ quantifies over specific tuples such that the $i-th$ tuple contains only the $i-th$ individual of variables $x_1 ... x_h$. This quantifier works under the assumption that each variable has the same number of individuals. For example, the grounding of $Diag(x_1, x_2)(p(x_1, x_2))$, given $|G(x_1)| = |G(x_2|$, is a tensor of size $|G(x_1)|$ instead of $|G(x_1)| \times |G(x_2)|$, as we can see in Figure 2.3.

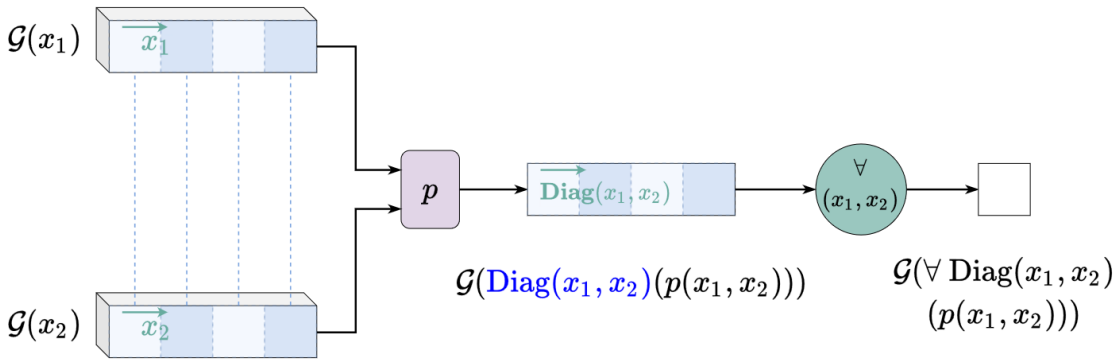


Figure 2.3: Grounding of diagonal quantifier (source: Badreddine *et al.* [2022])

Guarded Quantifier

Guarded quantification is used to quantify over a set of variables of a domain whose grounding is associated with some condition called *mask*. An example of this type is:

$$\forall y(\exists x : age(x) > age(y)(parent(x,y))) \tag{2.2.7}$$

The grounding of this formula corresponds to the universal quantification over y of values of x that satisfy the mask $age(x) > age(y)$ (Figure 2.4).

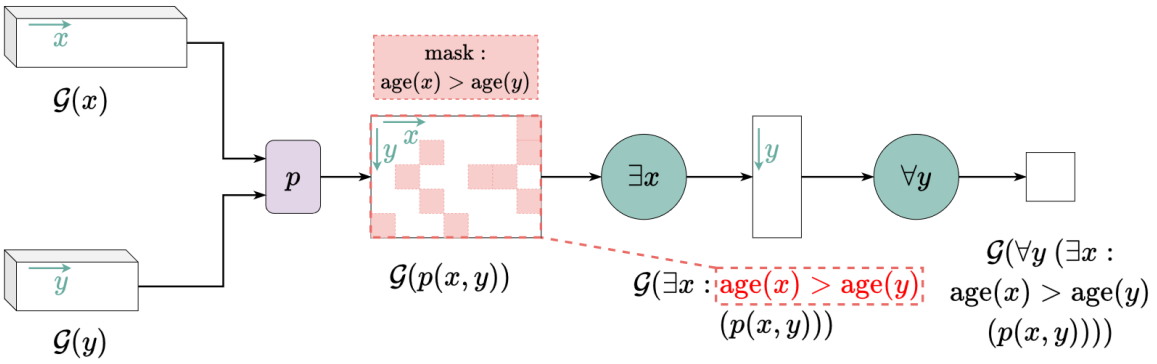


Figure 2.4: Grounding of guarded quantifier (source: Badreddine *et al.* [2022])

2.2.3 Stable Product Real Logic

Stable Product Real Logic is a fuzzy-logic semantics that has desirable properties for gradient-descent optimization. In the literature, this configuration is favored for ensuring improved performance and stable training. It includes strong negation (N_s) and adjustments of product t-norm (T_{prod}), product t-conorm (S_{prod}), Reichenbach S-implication (I_R) and smoothed aggregators (A_{pM}, A_{pME}):

$$\begin{aligned}
&N_s'(x) = N_s(x)\\
&T_{prod}'(x_1,...,x_n) = T_{prod}(\pi_0(x_1),...,\pi_0(x_n))\\
&S_{prod}'(x_1,...,x_n) = S_{prod}(\pi_1(x_1),...,\pi_1(x_n))\\
&I_R'(x,x') = I_R(\pi_0(x),\pi_1(x'))\\
&A_{pM}'(x_1,...,x_n) = A_{pM}(\pi_0(x_1),...,\pi_0(x_n))\\
&A_{pME}'(x_1,...,x_n) = A_{pME}(\pi_1(x_1),...,\pi_1(x_n))
\end{aligned} \tag{2.2.8}$$

Here (2.2.8) $\pi_0 : x \to (1-\epsilon)x + \epsilon$ and $\pi_1 x \to (1-\epsilon)x$ are functions used to avoid gradient-related problems in specific edge cases, and ϵ is an arbitrarily small number.

2.3 LTN Tasks

Built on the top of Real Logic, LTNs are capable of performing a wide range of tasks. In LTN applications, knowledge can be expressed in different forms. For instance, it can be embedded in symbol groundings by setting limits on domain groundings or by fixing the grounding of specific symbols. When the grounding of a symbol is unknown, it can be determined by identifying a set of parameters, θ, that need to be learned. In such cases, we denote the grounding as $G(\cdot,\theta)$. A second way to represent knowledge is through formulas that apply

to known objects (*factual proposition*, e.g. $\mathcal{G}(two) = 2, is\_even(two)$) or all objects in the domain (*generalized proposition*, e.g. $\mathcal{G}(D(x)) = \mathbb{N}, \forall x.(is\_even(x) \iff eq(mod(x, two), zero))$).

The following paragraphs describe applications of LTN in *learning*, *reasoning* and *querying* tasks.

2.3.1 Learning

Before discussing how the learning process is carried out in LTN, we define *theory* in Real Logic.

Definition 2.3.1 (Theory)**.** A *theory* in Real Logic is a triple $\mathcal{T} = \langle \mathcal{K}, \mathcal{G}(\cdot, \theta), \Theta \rangle$, where $\mathcal{K}$ is a set of clauses $\mathcal{G}(\cdot, \theta)$ is a parametric grounding for signature $\mathcal{L}$ and Θ is the *hypothesis space*, i.e. the set of possible values of parameters.

$\mathcal{G}(\cdot, \theta)$ defines groundings for all symbols and operators in the signature $\mathcal{L}$. Specifically, in LTN functions with arity m are grounded to linear transformations:

$$\mathcal{G}(f)(v_1, ..., v_m) = M_f v + N_f \tag{2.3.1}$$

where M_f is a $n \times mn$ matrix, N_f a n-vector and $v = \langle v_1, ..., v_m \rangle$. Predicates with arity m are grounded to:

$$\mathcal{G}(P) = \sigma(u_p^T \tanh(v^T W_P^{[1:k]} v + V_P v + B_P)) \tag{2.3.2}$$

where $W_P^{[1:k]}$ is a tensor $\mathbb{R}^{mn \times mn \times k}$, V_P is a matrix $\mathbb{R}^{k \times mn}$, B_P is a vector in $\mathbb{R}^k$ and σ is a sigmoid function. This encoding enables a neural network to represent the grounding of a predicate.

The learning strategy consists of searching a grounding that maximizes the satisfiability of clauses in $\mathcal{K}$. This is formalized by the following definition:

Definition 2.3.2 (Learning)**.** Given a theory $\mathcal{T} = \langle \mathcal{K}, \mathcal{G}(\cdot, \theta), \Theta \rangle$ *learning* is the process of searching a set of parameters $\theta^* \in \Theta$ that maximizes the satisfiability of $\mathcal{T}$, such that:

$$\theta^* = \text{argmax}_{\theta \in \Theta} \text{SatAgg}_{\phi \in \mathcal{K}} \mathcal{G}_\theta(\phi) \tag{2.3.3}$$

Here SatAgg is an aggregator operator over formulas $\phi \in \mathcal{K}$.

The loss function can be defined as:

$$L = 1 - \text{SatAgg}_{\phi \in \mathcal{K}} \mathcal{G}_\theta(\phi) \tag{2.3.4}$$

By following the prior learning model, LTNs are capable of addressing various tasks, including *classification*, *regression*, *clustering* and *semi-supervised learning* as shown in (Badreddine *et al.* [2022]).

Example 2.3.1 illustrates the application of an LTN in a binary classification task.

Example 2.3.1 (Binary classification)**.** Let x be points in $[0,1]^2$. Positive examples x_+ are points within the circle of center $(0.5, 0.5)$ and radius 0.09, whereas negative examples x_- are the remaining ones. In this setting, we have the following groundings:

$$\begin{aligned}
&\mathcal{G}(x) \in [0,1]^{m \times 2} \\
&\mathcal{G}(x_+) = \langle d \in \mathcal{G}(x) | \quad \|d - (0.5, 0.5)\| < 0.09 \rangle \\
&\mathcal{G}(x_-) = \langle d \in \mathcal{G}(x) | \quad \|d - (0.5, 0.5)\| \geq 0.09 \rangle
\end{aligned}$$

The binary classification problem can be formulated using a knowledge base $\mathcal{K}$ con-

sisting of two facts:

$$\forall x_+ . P(x)$$
$$\forall x_- . \neg P(x)$$

where P is a predicate that holds if x is positive. An appropriate grounding for P is:

$$\mathcal{G}(P|\theta) : x \to \sigma(\text{MLP}_\theta(x))$$

where MLP is a multi-layer perceptron with a single output neuron and trainable parameters θ and σ is a sigmoid function. The optimizer uses the same loss function defined in Equation 2.3.4.

2.3.2 Reasoning

In a reasoning task, the goal is to check whether a closed formula ϕ is a logical consequence of a set of formulas $\mathcal{K}$; in symbols $\mathcal{K} \models \phi$. By definition, $\mathcal{K} \models \phi$ when, for every grounded theory $\langle \mathcal{K}, \mathcal{G}_\theta \rangle$, if all formulas in $\mathcal{K}$ are true then ϕ is true:

$$\forall \langle \mathcal{K}, \mathcal{G}_\theta \rangle \quad \text{SatAgg}(\mathcal{K}, \mathcal{G}_\theta) \geq q \quad \wedge \quad \mathcal{G}_\theta \geq q \quad q \in \left[\frac{1}{2}, 1\right] \tag{2.3.5}$$

Note that we consider a formula as true when it evaluates in $[q, 1]$.

The previous definition requires evaluating all possible grounded theories to establish whether ϕ is a logical consequence of $\mathcal{K}$. This fact contrasts with practical implementations of reasoning tasks. Therefore, LTNs use two derived methods. The first, called *querying after learning*, limits the grounded theories to search to only those that maximize satisfiability as in Definition 2.3.3. The second, called *proof by reputation*, involves searching for a counterexample to Definition 2.3.5.

2.3.3 Querying

Given a grounded theory, query answering involves checking whether a certain fact is true. We can distinguish at least two types of query. A *truth query* is any formula ϕ defined in $\mathcal{L}$ and the expected answer is its grounding $\mathcal{G}(\phi)$. The latter is a real value $[0, 1]$ if ϕ is a closed formula; otherwise, it is a tensor of order n equal to the number of free variables in ϕ.

On the other hand, a *value query* is any term t and the corresponding answer is the grounding $\mathcal{G}(t)$. As for truth queries, the dimension of the answer depends on the number of free variables in t. Both types of query admit a *generalized* variant, which applies to unseen objects in the domain of application.

CHAPTER 3

Graph Neural Networks

This chapter concisely describes Graph Neural Networks, which are powerful deep neural networks capable of handling graph-structured data. First, we introduce a common notation and explain the basic principles and components of Graph Neural Networks. We then categorize them based on their underlying mechanisms, such as convolutional, attention-based, and autoencoder-based. The models discussed serve as foundational elements for our proposed framework. Finally, we review recent advancements in integrating Graph Neural Networks and Nesy AI, highlighting applications across various domains.

3.1 Properties and Taxonomy

The term "Graph Neural Network" (GNN) refers to a broad class of neural networks specifically developed for processing graph-structured data, which arise in various fields such as chemistry (Gilmer *et al.* [2017]), traffic forecasting (Yu *et al.* [2018]), and recommender systems (Ying *et al.* [2018]). GNNs are essential for addressing challenges, unique to graph data, that do not occur with typical Euclidean structures, like images, text, or audio. This is because graphs can vary in size and exhibit complex topological structures. Before delving into GNN architectures, we briefly introduce the fundamental concepts of graphs and establish the notation used in the subsequent discussion.

A graph is a mathematical structure $G = (V, E)$, where V is a set of nodes with $|V| = n$, and $E \subseteq V \times V$ is a set of edges connecting nodes in V. The set of neighbors of a node v is denoted by $N(v)$ and the *degree* of a node v is the number $|N(v)|$. For a given node v, a k-th-hop neighbor is defined as a node that can be reached by traversing exactly k edges.

Graphs can be classified as directed or undirected, depending on whether the edges have a direction, and as weighted or unweighted, depending on whether the edges carry numerical weights. A common data structure for representing graphs is the adjacency matrix $A \in \mathbb{R}^{n \times n}$. For unweighted graphs, A is defined as:

$$A_{ij} = \begin{cases} 1 & (i,j) \in E \\ 0 & otherwise \end{cases} \tag{3.1.1}$$

For weighted graphs, we set $A_{ij} = w_{ij}$ for existing edges, where w_{ij} represents the edge weight. In a graph, attributes for nodes or edges can be provided as vectors. Node attributes are represented by a feature matrix $X \in \mathbb{R}^{n \times d}$, while edge attributes are represented by a feature matrix X_e.

The previous notation applies primarily to homogeneous graphs, which have a single type of nodes and edges. However, there are also heterogeneous graphs, such as knowledge graphs,

that include multiple types of nodes and edges. Another special category is spatial-temporal graph, an attributed graph where node attributes evolve over time.

Returning to GNNs, these are deep neural networks where the input consists of a representation of the graph structure, which may be provided as either an adjacency matrix or a list of edges, along with node attributes and, in some cases, edge attributes.

Depending on the task, the output of a GNN falls into one of the following categories:

- *Node-level*: a GNN produces a high-level representation of nodes that can be a class label, as in *node classification*, or a continuous value, as in *node regression*.
- *Edge-level*: an edge-related output is generated from the hidden representations of node pairs. This method is commonly employed in tasks such as *edge classification* and *link prediction*.
- *Graph-level*: the output relates to a whole graph, as in the *graph classification* task.

Processing all node features generated by a convolutional layer or similar mechanisms is computationally intensive. Consequently, GNNs often employ a down-sampling strategy to mitigate this cost. Pooling operations, such as *mean* or *max*, are commonly used to reduce the dimensionality of features and minimize the risk of overfitting. Readout operations, like SortPooling, are implemented to produce graph-level outputs from node representations. As other components in a GNN, these operations must remain permutation-invariant, ensuring they are not influenced by the ordering of nodes.

The training of a GNN can be undertaken in various configurations. In a semi-supervised learning setting, only a subset of nodes is labeled, while the remaining nodes are unlabeled. This approach is particularly relevant for node classification tasks. The most prevalent method employed in graph classification tasks is fully supervised learning, where each graph is associated with a label. GNNs based on the autoencoder mechanism learn graph embeddings in an end-to-end unsupervised manner.

Next, we delve into prominent GNN models relying on three different mechanisms: convolutional, attention-based and autoencoder-based.

3.1.1 Graph Convolutional Neural Network

Convolution-based Graph Neural Networks (ConvGNNs) extend the convolutional operation commonly used in Convolutional Neural Networks (CNNs) for image processing to the domain of graph-structured data. A graph-based variant of the convolutional mechanism is employed to generate a hidden representation by aggregating the features of a node together with those of its neighbors.

Figure 3.1 shows a side-by-side comparison of a 2D convolution over a graph representation of an image and a graph convolution.

An image can be viewed as a graph, where each pixel corresponds to a node, and neighbors are determined by the shape of the convolutional filter (in Figure 3.1, this is 3×3). Therefore, 2D convolution operates on a fixed-size set of nodes. In contrast, the graph convolution operates on a variable number of unordered neighboring nodes.

Multiple convolutional layers can be stacked to build networks that are deep at will. As the number of layers increases, node representations benefit from the influence of a greater number of neighbors, including nodes that are progressively more distant.

ConvGNNs can be categorized into two classes, i.e. *spectral-based* and *spatial-based*. Spectral-based methods adopt graph spectral theory and interpret convolutional operation as removing noises from graph signals (Wu *et al.* [2021]); spatial-based approaches

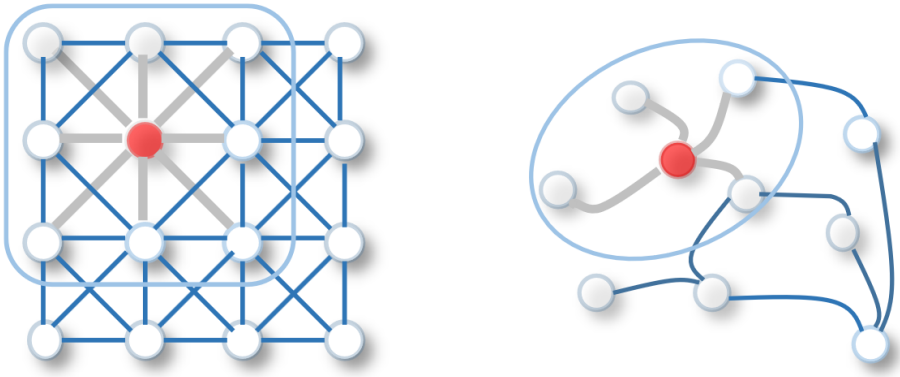


Figure 3.1: 2D convolution (left) vs Graph convolution (right). In both of them, the convolution operation takes the red node as the target and computes its hidden representation of it. Node neighbors are encircled by a light blue line (source: Wu *et al.* [2021])

conceptualize graph convolutions as a process of information propagation across the graph structure. In the following, we present two major examples of ConvGNNs.

The spectral-based graph convolutional network (GCN), introduced in (Kipf and Welling [2017]), adopts a convolutional layer that computes a hidden state of each target node by aggregating feature information from first-hop neighbors. GCN is built by stacking several convolutional layers to obtain a final representation of nodes, as shown in Figure 3.2.

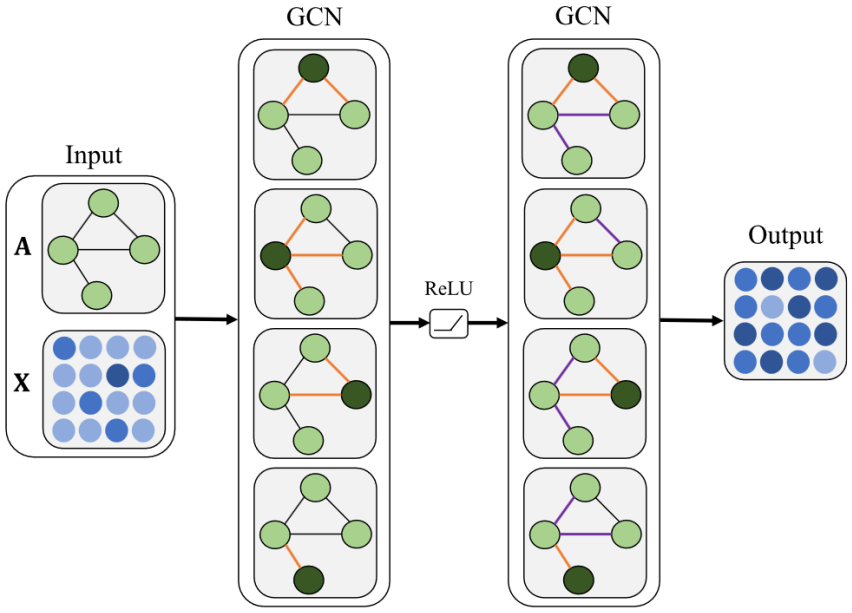


Figure 3.2: The architecture of a two-layer GCN model. The dark green denotes target nodes. The orange and the purple lines denote the information propagation for first-hop and second-hop neighbors (source: Xiao *et al.* [2021])

Each convolutional layer is described by the following formula:

$$H^{(l+1)} = \sigma(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}) \tag{3.1.2}$$

Here:

- $\tilde{A} = A + I_n$ is an adjacency matrix with self-loops;

- $\tilde{D}$ is a *degree matrix* such that $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$;
- $W^{(l)} \in \mathbb{R}^{h \times f}$ is a trainable weighted matrix, where h is the number of input channels and f denotes the dimension of the embedding feature;
- $H^{(l)} \in \mathbb{R}^{n \times h}$ is the matrix of hidden states for the l-th layer
- $H^{(l+1)} \in \mathbb{R}^{n \times f}$ is the matrix of hidden states for the $l+1$-th layer;
- σ is an activation function.

For the first layer, we set $H^{(0)} = X$. In typical implementations of Graph Convolutional Networks (GCNs), the matrix $\hat{A} = \tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$ is pre-computed and consistently used across each layer.

GraphSAGE (SAmple and aggreGatE, Hamilton *et al.* [2018a]) is a popular instance of a spatial-based graph convolutional network. This method defines a set of K aggregators $AGGREGATE_k$, which combine the information of the immediate neighborhood of a central node, and k learnable parameter matrices W^k. In the k-th step, the hidden state of a node v h_v^k is calculated as:

$$h_v^k = \sigma(W^k \cdot CONCAT(h_v^{k-1}, AGGREGATE_k(\{h_u^{k-1}, \forall u \in N(v)\}))) \tag{3.1.3}$$

where $CONCAT(\cdot)$ concatenates embeddings in input. In contrast to GCN, in GraphSAGE, the contributions of neighboring nodes differ across layers. Figure 3.3 illustrates the GraphSAGE model.

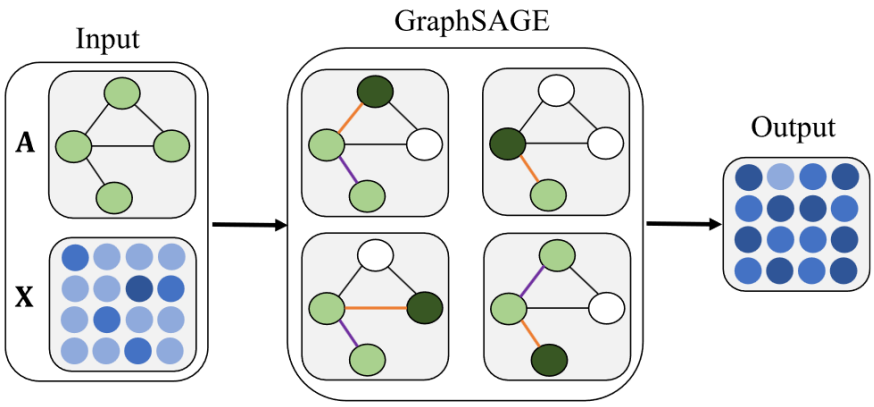


Figure 3.3: The architecture of a GraphSAGE model with sampling strategy. The maximum number of neighbors for hop is set to 1, while the maximum number of hops is 2. Orange and purple lines represent different aggregators (source: Xiao *et al.* [2021])

In their original paper, the authors introduce a sampling strategy designed to select a fixed-size set of neighbors for each node, effectively reducing the computational complexity of GraphSAGE. Additionally, they investigate the use of mean aggregation, LSTM aggregation, and max pooling aggregation methods to implement $AGGREGATE_k$.

Other significant works in the realm of ConvGNNs are *Message Passing Neural Network* (Gilmer *et al.* [2017]), which reformulates graph convolution as a message-passing mechanism that disseminates information through the edges of the graph, and *Cluster-GCN* (Chiang *et al.* [2019]), which operates convolution within subgraphs sampled using a graph clustering algorithm.

3.1.2 Graph Attention Networks

The Graph Attention Network (GAT) derives its name from the attention mechanism, which has been effectively used in a range of tasks within Natural Language Processing and Computer Vision. The main idea behind GATs is that different neighbors have variant contributions for the central node representation, depending on how much they are "relevant" to that.

GAT architecture is formalized in (Veličković *et al.* [2018]). First, a linear transformer, with weight matrix $W \in \mathbb{R}^{f \times d}$, computes a hidden representation $H_i = WX_i$ for each node v_i. Then, an attentional function $ATTENTION : \mathbb{R}^f \times \mathbb{R}^f$ returns:

$$e_{ij} = ATTENTION(H_i, H_j) \tag{3.1.4}$$

Here, e_{ij} denotes the importance of the node v_j for the representation of the node v_i. The attention function is applied to a target node and its first-order neighbors. Then, e_{ij} is normalized with a softmax function obtaining:

$$a_{ij} = \frac{\exp(e_{ij})}{\sum_{v_k \in \mathcal{N}(i)} \exp(e_{ik})} \tag{3.1.5}$$

The normalized coefficients a_{ij}, calculated with a single layer feedforward network, determine a final representation for each target node v_i:

$$Z_i = \sigma\left(\sum_{v_j \in \mathcal{N}(i)} \alpha_{ij} H_j\right) \tag{3.1.6}$$

This approach can be extended by employing several attentional functions $ATTENTION_k$ and concatenating multiple output features with an appropriate operator. Figure 3.4 depicts a multi-head GAT model.

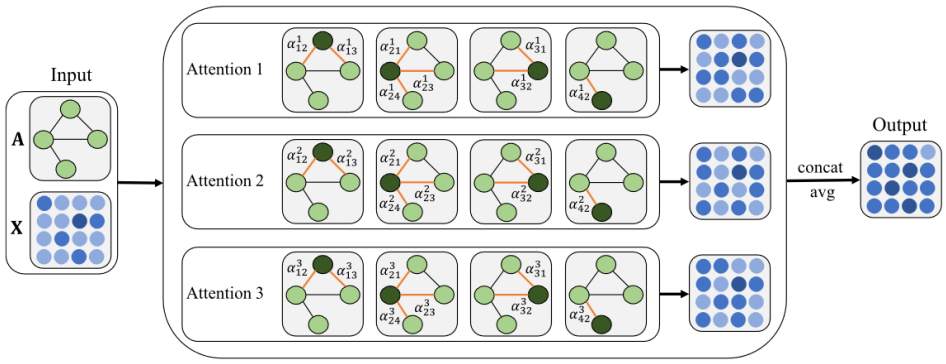


Figure 3.4: The architecture of a GAT model with $K = 3$ attention heads. Note that edges are weighted with normalized attention score a_{ij}. The aggregator *concat* is implemented with a simple mean operator (source: Xiao *et al.* [2021])

3.1.3 Graph Autoencoder

The Autoencoder mechanism is a common technique, especially adopted in semi-supervised and unsupervised tasks. Graph Autoencoders (GAEs) exploit this mechanism in combination with graph-structured data. In detail, a GAE is separated into two parts, namely:

- *Encoder*: a component that maps node features and structural information to embeddings in a low-dimensional *latent space*.
- *Decoder*: a component that decodes embeddings into vectors in the original space. The loss function designed for reconstruction aims to preserve important graph topological properties.

The Variational Graph Autoencoder (VGAE, Kingma and Welling [2022]) is an example of GNN leveraging the autoencoder mechanism to learn the distribution of data (Figure 3.5).

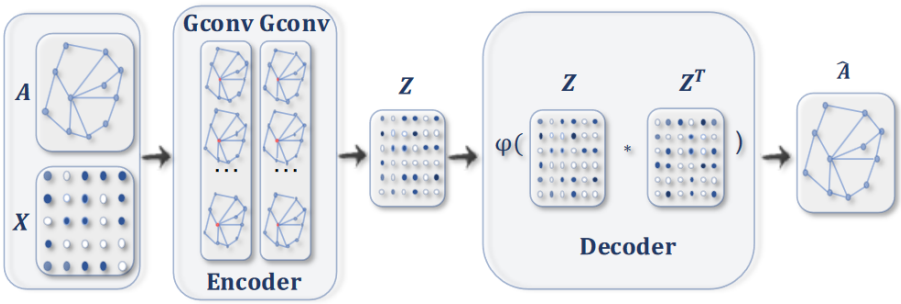


Figure 3.5: The architecture of Variational Graph Autoencoder (source: Wu *et al.* [2021])

The encoder consists of a two-layer GCN that generates network embeddings for each node. The decoder, in turn, utilizes an inner product operation on the latent variables, followed by an activation function ϕ, to yield the reconstructed adjacency matrix $\hat{A}$. The Variational Graph Autoencoder (VGAE) is trained to minimize the Kullback-Leibler divergence between $\hat{A}$ and the real adjacency matrix A.

3.2 Neurosymbolic AI and Graph Neural Networks

After thoroughly examining various GNNs and their underlying mechanisms, our attention now shifts to exploring contemporary research that integrates neurosymbolic methods with GNNs. We identified several studies that successfully merge NeSy AI and GNN approaches to address specific challenges.

A particularly noteworthy application of this integration lies within the life sciences, where data are frequently represented as graphs (e.g. molecular structures) and are complemented by extensive knowledge bases. For instance, DeepMind's AlphaFold, as described in (Wei [2019]), is designed to predict the three-dimensional structure of proteins based on their molecular descriptions.

Other research efforts have concentrated on software engineering, exemplified by Microsoft's *Deep Program Understanding* (Brockschmidt *et al.* [2019]). In this context, a variant of GNN known as the *Gated Graph Sequence Neural Network* is employed to analyze graph representations of programs, facilitating tasks such as error detection, variable name suggestion, and code completion.

Neurosymbolic frameworks incorporating GNNs have also been investigated for their potential in combinatorial optimization and constraint satisfaction problems. For example, the work in (Li *et al.* [2018]) uses a GCN as a heuristic search algorithm to solve instances of NP-complete problems, including the *Boolean Satisfiability Problem* (SAT) and the Maximal Clique problem. Additionally, a GNN-based solver is proposed in (Prates *et al.* [2018]) for a variant of the *Travelling Salesperson Problem* (TSP).

A successful combination of GNN and first-order logic is presented in Zhu *et al.* [2022], where the authors propose a Graph Neural Network Query Executor (GNN-QE) to answer complex logical queries on incomplete knowledge graphs.

Although numerous efforts have been made to integrate GNNs with NeSy AI, to the best of our knowledge, Logic Tensor Networks (LTNs) have not yet been combined with GNNs. In the next chapters, we propose a framework that integrates LTNs and GNNs to tackle node classification and link classification tasks and evaluate it on some benchmark datasets.

CHAPTER 4

LTN-GCN

This chapter presents an overview of LTN-GCN, the proposed framework developed for graph-based tasks. In Section 4.1, we outline the process of constructing a knowledge base and groundings for graph-related problems, with a particular emphasis on the use of graph convolutional neural network for grounding functions and predicates. Afterwards, we show the application of LTN-GCN in node classification tasks. In Section 4.2, we cover the package LTNTorch used in our implementation, and we delve into the architectural details of the graph convolutional neural network employed in LTN-GCN.

4.1 Framework description

Our framework, called LTN-GCN, uses a Logic Tensor Network and a convolution-based Graph Neural Network to address graph-based problems. For the training process, we define a Real Logic knowledge base composed of facts derived from the type of task considered within the application domain. These axioms include *commonsense* knowledge or capture relevant graph properties. We also provide a grounding for every symbol appearing in the knowledge base, including variables, predicates, functions, fuzzy operators, and quantifiers.

First, given a graph $G(V,E)$, we choose to ground nodes and edges with corresponding variables. This is consistent with Real Logic since variables consist of a sequence of individuals. Given the feature matrix X, nodes are grounded with their respective feature vector:

$$\mathcal{G}(v) = x_v, \qquad \forall v \in V \tag{4.1.1}$$

where $x_v \in \mathbb{R}^d$, i.e. node features are d-dimensional vectors.

We decide to ground an edge (u,v) by concatenating the groundings of nodes u and v. An alternative option consists of grounding edges with the associated feature vector from the edge attribute matrix X_e, if the latter is provided within the problem. In a complex knowledge base, additional variables beyond those representing nodes and edges may also be present; however, here, we provide a grounding only for nodes and edges.

As implied by the framework's name, a graph convolutional neural network is employed as the parametric grounding of functions and predicates. We formalize these groundings in the following equations:

$$\begin{aligned} \mathcal{G}(f) &: v \to \sigma(\text{genGCN}_{\theta_f}(v,A)) \\ \mathcal{G}(p) &: v \to \sigma(\text{genGCN}_{\theta_p}(v,A)) \end{aligned} \tag{4.1.2}$$

Here, A is the adjacency matrix, σ is a nonlinear function, and genGCN is a graph convolutional neural network with a set of learnable parameters θ. More details about genGCN are available in Section 4.2.

Functions and predicates can be grounded using a model that returns node-level or edge-level output. In Equation 4.1.2, the function f and the predicate p are grounded with models that return node-level output.

In addition, the output layer of genGCN varies according to whether it grounds a function or a predicate. For functions, the output is any vector in $\mathbb{R}^n$, where n is an arbitrary integer, hence, there is no constraint on the number of output channels of genGCN. For predicates, the output must be a real value in $[0,1]$; therefore the number of output channels of genGCN is limited to 1, and a sigmoid function is applied.

In LTN-GCN, a Logic Tensor Network trains a genGCN to maximize the satisfiability of facts contained in the knowledge base. The learning process uses the satisfiability loss presented in Equation 2.3.4:

$$L = 1 - \text{SatAgg}_{\phi \in \mathcal{K}} \mathcal{G}_\theta(\phi) \tag{4.1.3}$$

Here θ is the set of parameters of models used to ground predicates and functions symbols.

We have designed LTN-GCN to specifically address node and link classification. The construction of the knowledge base is largely influenced by the characteristics of the specific problem domain. Nevertheless, it is possible to define domain-independent facts. In particular, we can enforce that ground-truth labels match the predicted labels when the training is conducted in a supervised or semi-supervised manner. This observation allows us to define facts applicable to both node and edge classification. In what follows, we first present the formalization of a binary and multiclass node classification problem, and, then, the one for edge classification.

Similarly to (Badreddine *et al.* [2022]), we formalize a binary node classification problem in Real Logic as:

$$\begin{aligned} &\forall v_+, Class(v_+) \\ &\forall v_-, \neg Class(v_-) \end{aligned} \tag{4.1.4}$$

Here v_+ and v_- are positive and negative labeled nodes, respectively. $Class$ is a predicate that holds if the input node is positive. We ground $Class$ as:

$$\mathcal{G}(Class) : v \rightarrow sigmoid(\text{genGCN}_\theta(v, A)) \tag{4.1.5}$$

Note that the predicate $Class$ must return a fuzzy truth degree. Therefore, we apply the sigmoid function to the output of genGCN, which has a single output channel.

A multi-class single-label classification problem can be modeled with a different knowledge base. In this case, the goal is to predict the class of each node among C distinct classes. We identify the following axiom:

$$\forall Diag(v, y), Eq(Class(v), y) \tag{4.1.6}$$

Here v is a node and $y \in \mathbb{R}^C$ is a one-hot encoded node label, i.e. a vector so that $y_c = 1 \quad \wedge \quad y_i = 0, \forall i \neq c$, if node belongs to class c. The diagonal quantifier restricts the evaluation exclusively to pairs comprising a node and its corresponding label; otherwise, the predicate would be applied to any combinations of nodes and labels. Eq is a predicate that holds if labels are equal, and $Class$ is a function that returns softmax probabilities for each class. Eq and $Class$ are grounded as:

$$\begin{aligned} &\mathcal{G}(Class) : v \rightarrow softmax(\text{genGCN}_\theta(v,A)) \\ &\mathcal{G}(Eq) : x,y \rightarrow e^{-\alpha \sum_{i=1}^{c} |x_i - y_i|} \end{aligned} \tag{4.1.7}$$

genGCN returns a vector $\in \mathbb{R}^C$ to which a softmax function is applied. The grounding of the predicate Eq is an exponential function that depends on the Manhattan distance between vectors x and $y \in \mathbb{R}^C$, and a parameter α. The truth degree of predicate Eq is equal to 1 if x and y are identical and decreases as their Manhattan distance increases. Note that this is only one possible grounding of predicate Eq: it can be replaced with any function measuring the similarity of two vectors and returning a value in $[0,1]$.

A knowledge base and its corresponding groundings can be constructed in a similar way for link classification. For the binary case, i.e. link prediction, we use the following formulation:

$$\begin{aligned} &\forall \langle u,v \rangle_+, Class(\langle u,v \rangle_+) \\ &\forall \langle u,v \rangle_-, \neg Class(\langle u,v \rangle_-) \end{aligned} \tag{4.1.8}$$

Here $\langle u,v \rangle_+$ and $\langle u,v \rangle_-$ are positive-labeled and negative-labeled edges, respectively. $Class$ is a binary classifier grounded with:

$$\mathcal{G}(Class) : \langle u,v \rangle \rightarrow sigmoid(\text{MLP}_{\theta_m}(\langle \text{genGCN}_{\theta_g}(u,A), \text{genGCN}_{\theta_g}(v,A) \rangle)) \tag{4.1.9}$$

In contrast to node classification, the grounding of $Class$ includes an edge encoder implemented by a multi-layer perceptron (MLP). This component generates edge-level output from the concatenation of node embeddings. A $sigmoid$ function is applied to normalize the predicate output between 0 and 1.

Instead, a multi-class single-label link classification can be formulated as:

$$\forall Diag(\langle u,v \rangle, y), Eq(Class(\langle u,v \rangle), y) \tag{4.1.10}$$

where the predicate Eq and the function $Class$ are grounded with:

$$\begin{aligned} &\mathcal{G}(Class) : \langle u,v \rangle \rightarrow softmax(\text{MLP}_{\theta_m}(\langle \text{genGCN}_{\theta_g}(u,A), \text{genGCN}_{\theta_g}(v,A) \rangle) \\ &\mathcal{G}(Eq) : x,y \rightarrow e^{-\alpha \sum_{i=1}^{c} |x_i - y_i|} \end{aligned} \tag{4.1.11}$$

The grounding of the predicate Eq is identical to that used in node classification, while the grounding of the function $Class$ differs from the link prediction case only in the use of a $softmax$ function in place of a $sigmoid$ function.

Our formulation for multi-class single-label classification can be adapted to multi-class multi-label classification by replacing the $softmax$ function in $\mathcal{G}(Class)$ with a $sigmoid$ function.

The axioms described above can be considered as a "blueprint" for the knowledge base in node and edge classification tasks. The knowledge base can be arbitrarily augmented by introducing facts that apply to the specific use case and axioms which can be intended as constraints imposed by the problem. This approach results particularly useful in contexts where there already exists a knowledge base, which can be easily modeled in form of FOL rules.

Furthermore, LTN-GCN is a general framework that can be used in any classification task without putting effort in building an *ad-hoc* GCN model. In fact, it relies on a suite of pre-selected graph convolutional neural networks (in our case, GCN and GraphSAGE), that

are directly utilizable. Therefore, building a custom knowledge base assumes a greater importance than designing the GCN for the task resolution. By simply modifying the knowledge base, one can employ LTN-GCN to address a completely different classification problem or a variant of the original problem, where the knowledge base has changed over time. This makes our framework strongly generalizable.

The training with a LTN adds a reasoning layer that is absent in current Graph Neural Networks. Specifically, the loss function has a precise meaning as it represents the satisfiability of the knowledge base and depends on it. The loss value can be monitored during training, so that we can say if model predictions match the available knowledge about the task. This is a clear advantages over traditional graph neural network training, which uses a task-agnostic loss function.

Beside the loss function, we can also measure the satisfiability of other facts formulated in FOL. This is called *querying during learning*. Observing the truth degree of formulas during the training process is a significant step towards the explainability of the model, as it allows for monitoring of how well the model adheres to the given logical constraints, and offering a clearer understanding of the relationships between the input data and the predicted outcomes.

4.2 Implementation

LTN-GCN is built upon existing implementations of LTNs and GCNs. To realize LTNs, we use *LTNTorch*, a *PyTorch*-based library based on the results found in (Badreddine *et al.* [2022]). *LTNTorch* consists of two modules:

- *ltn.core*, which contains the definition of constants, variables, predicates, functions, connectives, and quantifiers;
- *ltn.fuzzy_ops*, which contains the definition of some of the most common fuzzy semantics (connective operators and aggregators).

A generic symbol in Real Logic is represented by an instance of *ltn.core.LTNObject*. This class defines the attributes *value*, which contains the grounding of the symbol, and *free_vars*, which contains a list of free variables appearing in the symbol. *LTNTorch* provides several *ad-hoc* classes for implementing non-logical and logical symbols; they are:

- *ltn.core.Constant*: it represents an LTN Constant, which can be pre-defined (fixed data point) or trainable (embedding). In the latter case, the value assigned to the constant is used for initialization and changes during learning.
- *ltn.core.Variable*: it represents an LTN Variable, denoted by a name and a sequence of individuals.
- *ltn.core.Predicate*: it represents an LTN Predicate, which can be grounded with a trainable PyTorch model or a Python function. It is suggested to define a predicate using a Python function only for simple and non-learnable mathematical operations. The output of an LTN predicate must always fall in the range $[0,1]$.
- *ltn.core.Function*: it represents an LTN Function and can be grounded as *ltn.core.Predicate*. Differently from LTN predicates, the output of an LTN function has no constraints.
- *ltn.core.Connective*: it wraps a grounded fuzzy operator from the module *ltn.fuzzy_ops*.
- *ltn.core.Quantifier*: it wraps a grounded fuzzy aggregator from the module *ltn.fuzzy_ops*.

The module *ltn.fuzzy_ops* contains various implementations of operators and aggregators based on fuzzy semantics. In our approach, we adopt a Stable Product Real Logic configuration (2.2.8) with $\epsilon = 10^{-4}$.

In *LTNtorch*, when a predicate (*ltn.core.Predicate*), function (*ltn.core.Function*), or connective (*ltn.core.Connective*) is invoked, the framework automatically performs the broadcasting of the inputs. This means that the evaluation involves all possible combinations of individuals associated with each variable. This behavior is disabled when using a diagonal quantifier with the function *ltn.core.diag*; it can be reactivated by the function *ltn.core.undiag*. It is also possible to define a guarded quantifier, specifying a boolean function to implement the mask.

The genGCN model described in LTN-GCN is implemented using *PyTorch* and *torch geometric*. The network architecture comprises stacked convolutional layers, which are constructed using one of two pre-selected methods, i.e. GCN and GraphSAGE. This choice was made to minimize the effort in the design phase of genGCN. The hyperparameters of this architecture are the number of hidden channels, the number of convolutional layers, and the output dimension.

The activation function is a *Leaky ReLU* defined below:

$$LeakyReLU(x) = \begin{cases} -\alpha x, & x < 0 \\ x, & x \geq 0 \end{cases} \tag{4.2.1}$$

Additionally, dropout and batch normalization are included to prevent overfitting and enhance the stability of the training process. Figure 4.1 shows the architecture of genGCN.

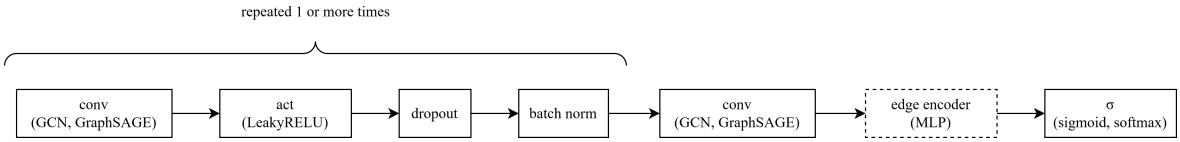


Figure 4.1: The architecture of genGCN

The block containing a convolution followed by an activation function, a dropout layer, and batch normalization, can be repeated several times. In practice, the number of convolutions is between 2 and 7. The dotted rectangle represents an edge encoder, implemented with a multi-layer perceptron composed of 2 dense layers, which is necessary only for generating edge-level outputs.

CHAPTER 5

Node classification with LTN-GCN

In this chapter, we test our framework LTN-GCN within a node classification task. In Section 5.1, we describe the ogbn-arxiv dataset, which is a citation graph containing labeled papers, then, we discuss homophily measures computed for this graph. In Section 5.2, we propose a knowledge base to train LTN-GCN for the given task. The two axioms found are specifically chosen for addressing a multi-class single-label classification problem on a homophilous graph. Finally, in Section 5.3, we detail the experimental setup and provide the results of our experiments.

5.1 Dataset

In our experiments related to node classification, we use the *ogbn-arxiv* dataset (link) from the *Open Graph Benchmark*. It is a directed graph representing a citation network between all Computer Science arxiv papers indexed by MAG (Microsoft Academic Graph). It contains 169,343 nodes and 1,166,243 directed edges. Each node corresponds to an arxiv paper and a directed edge denotes that one paper cites another one. Papers are described by a 128-dimensional feature vector, obtained by averaging the embeddings of words contained in both title and abstract. The word embeddings are computed by running a Word2Vec skip-gram model.

The task is a 40-class single-label classification problem that require to determine the subject area of a paper, e.g. cs.CV, cs.LG, cs.IT. Ground-truth labels are manually annotated by the paper's authors and arXiv moderators. The classes are highly unbalanced, as we can observe from the bar plot in Figure 5.1.

The *ogbn-arxiv* graph is homophilous, i.e. similar nodes tend to connect each other. This is to be expected since academic papers are more likely to cite other papers that address related topics. This phenomenon can be observed through the measurement of *edge homophily*:

$$h = \frac{|(u,v) \in E | c_u = c_v|}{|E|} \tag{5.1.1}$$

that is the proportion of edges between nodes of the same class. We found $h = 0.655$; however, *edge homophily* is sensitive to the number of classes C and it may be misleadingly large if classes are imbalanced. Therefore, we compute homophily through another measure, proposed in (Lim *et al.* [2021]):

$$\hat{h} = \frac{1}{C-1} \sum_{i=0}^{C-1} \left[h_i - \frac{|C_i|}{n} \right]_+ \tag{5.1.2}$$

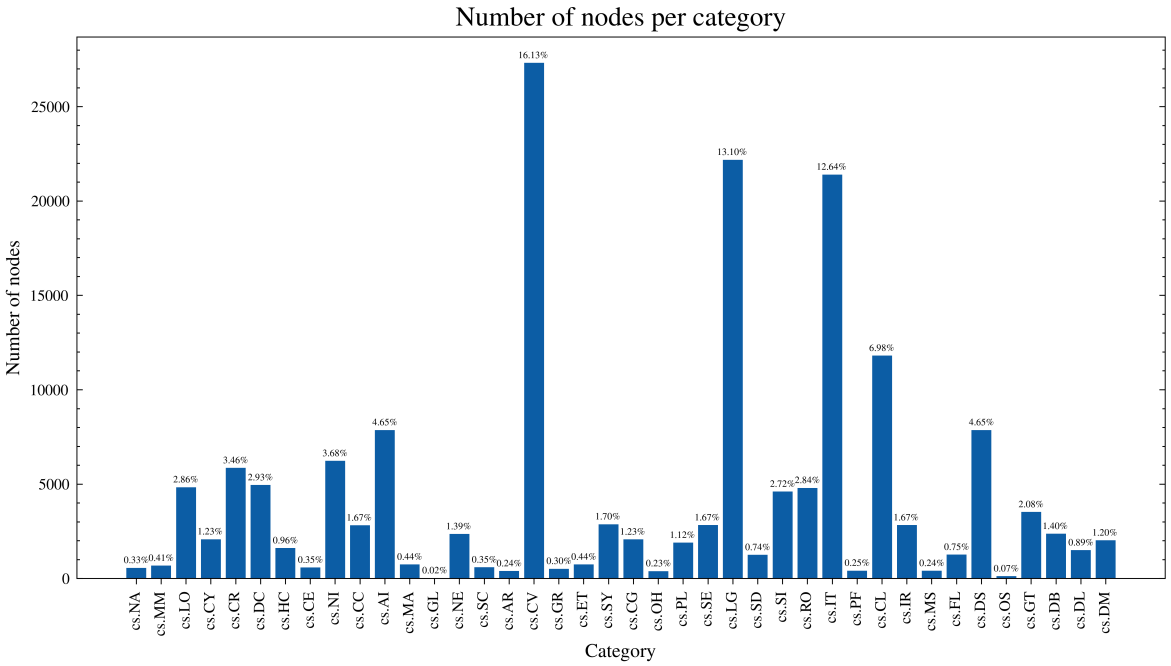


Figure 5.1: Bar plot of the number of nodes for each Computer Science arxiv category. The percentages displayed above bars denote the proportion of node belonging to a specific category.

where n is the total number of nodes, $|C_i|$ is the cardinality of class C_i, and h_i is the class-wise homophily metric:

$$h_i = \frac{\sum_{v \in C_i} d_v^{(c_v)}}{\sum_{v \in C_i} d_v} \tag{5.1.3}$$

Here d_v is the number of neighbors of node v, and $d_v^{(c_v)}$ is the number of neighbors with the same class of v. In the *ogbn-arxiv* graph, we found $\hat{h} = 0.42$. Class-wise homophily measures are shown in Figure 5.2.

Despite some classes showing a low homophily score, $\hat{h}$ is high. This observation about the graph homophily can be useful to define the knowledge base requested for LTN-GCN training.

We utilize the dataset split as provided by the original authors, which is organized temporally: the training set includes 90,941 papers published in 2017, the validation set contains 29,799 papers from 2018, and the test set contains 48,603 papers from 2019. Furthermore, we add reverse edges to the graph.

5.2 Knowledge Base

For the classification problem related to the *ogbn-arxiv* dataset, we model a set of axioms that is suitable for a multi-class single-label task and integrates our knowledge about the graph. It is composed of the following formulas:

$$\begin{aligned} \phi_1 \quad & \forall Diag(p,l) : Train(p) \quad Eq(Class(p),l) \\ \phi_2 \quad & \forall \langle p_1, p_2 \rangle \quad Sim(Cat(Class(p_1)), Cat(Class(p_2))) \end{aligned} \tag{5.2.1}$$

where p is a symbol representing papers, $\langle p_1, p_2 \rangle$ denotes direct and reverse citation edges, and l represents the class label assigned to a paper.

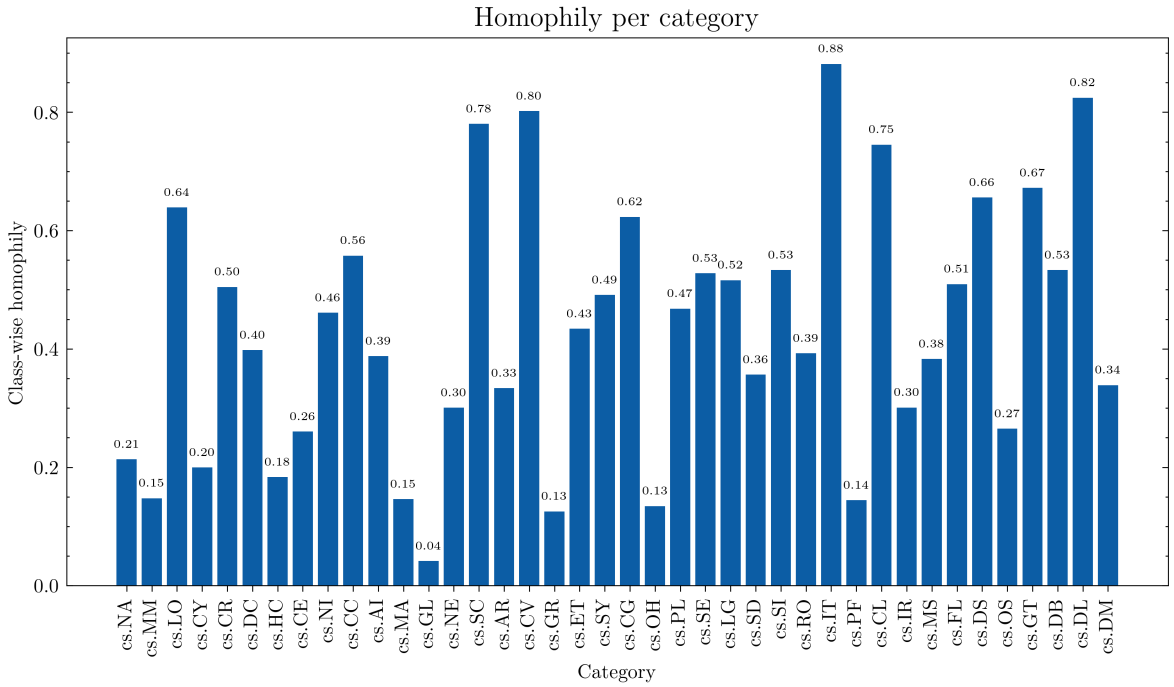


Figure 5.2: Bar plot of homophily for each Computer Science arxiv category. Classes exhibit different homophily scores.

ϕ_1 is a variation of the formula illustrated in Equation 4.1.6 for multi-class node classification. It includes an additional guarded quantifier instantiated by the mask $Train(p)$, where $Train$ is a predicate indicating whether or not a paper belongs to the train set. In fact, we do not use validation and test labels during the training process. $Class$ is a function returning the predicted label of a paper.

ϕ_2 encapsulates our insights about the graph's homophily, asserting that papers, among which one cites the other, tend to belong to similar categories. The function Cat returns the category object associated with a predicted label, while the predicate Sim holds true when two categories are similar. In ϕ_2, we do not impose equality of categories between citing papers, as this is not always observed within the graph and would be an overly restrictive condition.

Next, we discuss the grounding of symbols shown in Equation 5.2.1 according to the Real Logic definition. Domains are grounded as follows:

$$\begin{aligned} \mathcal{G}(papers) &= \mathbb{R}^{128} \\ \mathcal{G}(label) &= [0,1]^{40} \\ \mathcal{G}(categories) &= \mathbb{R}^{128} \end{aligned} \tag{5.2.2}$$

Specifically, papers are represented by a feature vector corresponding to the 128-dimension embedding provided in the dataset, labels are one-hot encoding, and categories are constants represented by a feature vector obtained by averaging embedding of papers belonging to the same category. We ground categories with:

$$\mathcal{G}(c_i) = \frac{1}{|C_i|} \sum_{p \in C_i\}} x_p \qquad \forall i = 1...40 \tag{5.2.3}$$

where x_p is the embedding of paper p.

The grounding of variables, predicates, and functions is the following:

$$
\begin{aligned}
&\mathcal{G}(p) \in \mathbb{R}^{n\times 128}\\
&\mathcal{G}(\langle p_1, p_2\rangle) \in \mathbb{R}^{m\times 2\times 128}\\
&\mathcal{G}(l) \in [0,1]^{n\times 40}\\
&\mathcal{G}(Eq) : l', l'' \to e^{-\sum_{i=1}^{c} |l_i' - l_i''|} \qquad \alpha = 1.0\\
&\mathcal{G}(Train) : p \to x \in \{0,1\}\\
&\mathcal{G}(Class) : p \to softmax(\text{genGCN}_\theta(p, A))\\
&\mathcal{G}(Sim) : c', c'' \to \frac{1}{2}\left(\frac{c' \cdot c''}{|c'| \cdot |c''|} + 1\right)\\
&\mathcal{G}(Cat) : l \to c \in \{c_1, ..., c_{40}\}
\end{aligned}
\tag{5.2.4}
$$

The variables p and l are sequences of n individuals, where n is the number of nodes, while variable $\langle p_1, p_2\rangle$ is a sequence of m individuals, where m is the number of edges. The grounding of Eq and $Class$ are derived from Equation 4.1.6. $\mathcal{G}(Sim)$ quantifies the similarity between two category embeddings by employing a metric that depends on the cosine similarity, while $\mathcal{G}(Cat)$ returns the category object corresponding to a given label.

The aggregators $\forall$, figuring in the knowledge base, and $SatAgg$, used for computing loss, are both grounded with A_{pME}, with p set to 2.

5.3 Experiments

5.3.1 Setup

We conduct our experiments with *ogbn-arxiv* dataset on a GPU NVIDIA RTX A2000 12GB. Three different models have been tested: a standalone convolutional neural network corresponding to our genGCN, a LTN-GCN trained to maximally satisfy the formula ϕ_1, denoted as LTN-GCN$_1$, and a LTN-GCN trained to maximally satisfy both formulas ϕ_1 and ϕ_2, denoted as LTN-GCN$_{12}$.

The standalone genGCN was trained with the cross entropy loss (Equation 5.3.1) averaged over mini-batches.

$$
Cross - Entropy = \sum_{i=1}^{C} y_i \log(p_i) \tag{5.3.1}
$$

Here y is the ground-truth one-hot encoded label and p is the predicted probability vector.

For each of the three models, we instantiate two variants, i.e., one including a genGCN with GCN convolutional layers, and another including a genGCN with graphSAGE convolutional layers. In both cases, genGCN presents $k = 3$ convolution layers and $h = 256$ hidden channels.

We train our models for 500 epochs with ADAM optimizer and a learning rate of $5 \cdot 10^{-3}$. To shorten training times, we adopt an early stop strategy that interrupts the training after 30 epochs without any improvement in validation loss. The training was repeated multiple times, employing different mini-batch sampling techniques, namely:

- *Neighbor Sampling*: introduced in (Hamilton *et al.* [2018b]), this method samples a fixed number of neighbors from random seed nodes. The sampling process may involve multiple iterations, where in each iteration a fixed number of neighbors is sampled from the nodes selected in the preceding iteration. We use 20% of training nodes selected randomly as seed nodes and select 8 first-hop neighbors, 4 second-hop neighbors and

2-third hop neighbors. Every batch contains 64 seed nodes. In addition, the sampling process returns the subgraph induced by the sampled node, i.e. it also includes edges between neighbors.

- *Cluster Sampling*: presented in (Chiang *et al.* [2019]), this sampling technique splits the graph in non-overlapping subgraphs. Every batch contains exactly a subgraph. We set the number of subgraphs to 2.
- *GraphSAINT Random Walk Sampling*: proposed in (Zeng *et al.* [2020]), it picks a root node, starts a random walk, and adds all visited nodes to the obtained subgraph. Similar to *Neighbor Sampling*, we set the number of seed nodes in each batch to 64. The length of the random walk is set to 128, and the number of batches to create is set to 512.

The mini-batch sampling was adopted to mitigate the computational overhead associated with full-batch training. Training with LTN using full batch was not feasible on our machine due to memory constraints. Nevertheless, this does not pose a limitation, as the use of mini-batch training allows our approach to scale effectively to large graphs.

To compare the performances of the models, we compute accuracy and macro F1-score, which equals the arithmetic mean of all per-class F1 scores. The formulas of accuracy and macro F1-score are illustrated in Equation 5.3.2.

$$\begin{aligned} Accuracy &= \frac{1}{n}\sum_{i=1}^{n} acc(i), \qquad acc(i) = \begin{cases} 1, & \text{if} \quad \operatorname{argmax} y_i = \operatorname{argmax} p_i \\ 0, & \text{otherwise} \end{cases} \\ macroF1 &= \frac{1}{C}\sum_{i=1}^{C} F1_i, \qquad F1_i = 2 \cdot \frac{Precision_i \cdot Recall_i}{Precision_i + Recall_i} \end{aligned} \tag{5.3.2}$$

Here y_i is the ground-truth one-hot encoded label of the i-th node, while p_i is the predicted probability vector of the i-th node.

5.3.2 Results

Table 5.1 shows the evaluation metrics for experiments where our models were trained with *Neighbor Sampling*, Table 5.2 shows experiments using *Cluster Sampling*, and Table 5.3 shows experiments using *GraphSaint Random Walk Sampling*. The metrics refer to models showing the best validation accuracy during training.

By analyzing the results in Table 5.1, GCN exhibits the highest train accuracy (0.664), train macro F1-score (0.308), validation macro F1-score (0.296), and test macro F1-score (0.286). Other models surpass GCN only in two cases: LTN-GCN$_1$ (GCN) achieves the best validation accuracy (0.678), while LTN-GCN$_{12}$ (GCN) achieves the best test accuracy (0.676). LTN-GCN$_1$ (GCN) outperforms LTN-GCN$_1$ (GraphSAGE) by +0.080 in test accuracy, and by +0.135 in test macro F1-score. Similarly, LTN-GCN$_{12}$ (GCN) achieves better metrics than those of LTN-GCN$_{12}$ (GraphSAGE) (+0.081 test accuracy, +0.080 test macro F1-score). The comparison between LTN-GCN$_1$ (GCN) and LTN-GCN$_{12}$ (GCN) indicates that the inclusion of ϕ_2 in the knowledge base does not consistently enhance performances; indeed, it may decrease the macro F1-score. In contrast, LTN-GCN$_{12}$ (GraphSAGE) shows performances that are often slightly better than those of LTN-GCN$_1$ (GraphSAGE).

In the results listed in Table 5.2, GCN achieve slightly better results than other models across all evaluated metrics. In this case, LTN-GCN based on GraphSAGE performs slightly better than LTN-GCN based on GCN. The presence of ϕ_2 in the knowledge base seems to be beneficial for model performance. In fact, LTN-GCN$_{12}$ (GCN) metrics are superior to those

Model	Accuracy			macro F1		
	Train	Val	Test	Train	Val	Test
GCN	**0.664**	0.672	0.671	**0.308**	**0.296**	**0.286**
GraphSAGE	0.604	0.616	0.614	0.248	0.225	0.218
LTN-GCN$_1$ (GCN)	<u>0.662</u>	**0.678**	<u>0.674</u>	<u>0.301</u>	<u>0.291</u>	<u>0.283</u>
LTN-GCN$_1$ (GraphSAGE)	0.585	0.599	0.594	0.200	0.186	0.148
LTN-GCN$_{12}$ (GCN)	0.659	<u>0.677</u>	**0.676**	0.291	0.282	0.274
LTN-GCN$_{12}$ (GraphSAGE)	0.587	0.593	0.595	0.225	0.203	0.194

Table 5.1: Accuracy and F1-score for models trained with *Neighbor Sampling* in train, validation and test set. For each column, the bold value represents the best score, while the underlined one coincides with the second-best score.

Model	Accuracy			macro F1		
	Train	Val	Test	Train	Val	Test
GCN	**0.708**	**0.697**	**0.690**	**0.389**	**0.356**	**0.343**
GraphSAGE	<u>0.680</u>	<u>0.687</u>	<u>0.685</u>	<u>0.336</u>	<u>0.319</u>	<u>0.311</u>
LTN-GCN$_1$ (GCN)	0.656	0.670	0.662	0.282	0.267	0.259
LTN-GCN$_1$ (GraphSAGE)	0.676	0.675	0.666	0.309	0.288	0.281
LTN-GCN$_{12}$ (GCN)	0.670	0.674	0.671	0.304	0.285	0.277
LTN-GCN$_{12}$ (GraphSAGE)	0.673	0.682	0.678	0.310	0.293	0.284

Table 5.2: Accuracy and F1-score for models trained with *Cluster Sampling* in train, validation and test set. For each column, the bold value represents the best score, while the underlined one coincides with the second-best score.

Model	Accuracy			macro F1		
	Train	Val	Test	Train	Val	Test
GCN	**0.714**	**0.715**	**0.704**	**0.421**	**0.396**	**0.381**
GraphSAGE	0.690	0.675	0.671	<u>0.384</u>	0.345	0.332
LTN-GCN$_1$ (GCN)	0.680	0.686	0.685	0.353	0.332	0.321
LTN-GCN$_1$ (GraphSAGE)	0.689	0.680	0.675	0.376	<u>0.348</u>	0.337
LTN-GCN$_{12}$ (GCN)	0.677	0.681	0.684	0.319	0.301	0.293
LTN-GCN$_{12}$ (GraphSAGE)	<u>0.701</u>	<u>0.689</u>	<u>0.685</u>	0.378	0.348	<u>0.338</u>

Table 5.3: Accuracy and F1-score for models trained with *GraphSAINT Random Walk Sampling* in train, validation and test set. For each column, the bold value represents the best score, while the underlined one coincides with the second-best score.

obtained with LTN-GCN$_1$ (GCN) (+0.009 test accuracy, +0.018 test macro F1-score), and LTN-GCN$_{12}$ (GraphSAGE) metrics are superior to those obtained with LTN-GCN$_1$ (GraphSAGE) (+0.012 test accuracy, +0.003 test macro F1-score).

In the results from Table 5.3, GCN continues to achieve solid performance, although it is comparable with other models. The second-best model, considering accuracy, is LTN-GCN$_{12}$ (GraphSAGE) (-0.013 train accuracy, -0.026 validation accuracy, -0.019 test accuracy). Differently from experiments using *Neighbor Sampling*, LTN-GCN models based on GCN do not show relevant improvements compared to those based on GraphSAGE. LTN-GCN$_{12}$ (GraphSAGE) slightly improves compared to LTN-GCN$_1$ (GraphSAGE), while LTN-GCN$_{12}$ (GCN) does not outperform LTN-GCN$_1$ (GCN).

On average, LTN-GCN achieves performances comparable to standalone convolutional neural networks (GCN and GraphSAGE) in terms of accuracy and macro F1-score. Although GCN shows the best metrics in most cases, LTN-GCN models frequently rank as the second-best. The results confirm the applicability of our framework to node classification tasks. The similarity between the metrics obtained with standalone GCN and those obtained with LTN-GCN is relevant, given that the knowledge base used is quite simple. Moreover, this task has been used just to test the capabilities of the framework. We believe that LTN-GCN holds significant potential when combined with complete knowledge base and applied to more complex tasks.

Finally, we describe the training process by commenting the plot of loss function. Figures 5.3 and 5.4 show the train and the validation loss for Experiments 16 and 18, respectively. In Experiment 16, the loss corresponds to the satisfiability of the formula ϕ_1, while in Experiment 18, the loss corresponds to the satisfiability of formulas ϕ_1 and ϕ_2. The validation loss is computed using a version of ϕ_1 with a guarded quantifier that selects only nodes in the validation set.

In both experiments, we observe that the validation is greater than the training loss. This discrepancy may arise from the model because it is optimized specifically to maximize the satisfiability of the formula ϕ_1 for nodes present only in the training set, producing an effect of overfitting. Consequently, the satisfiability of ϕ_1 is lower for nodes in the validation set.

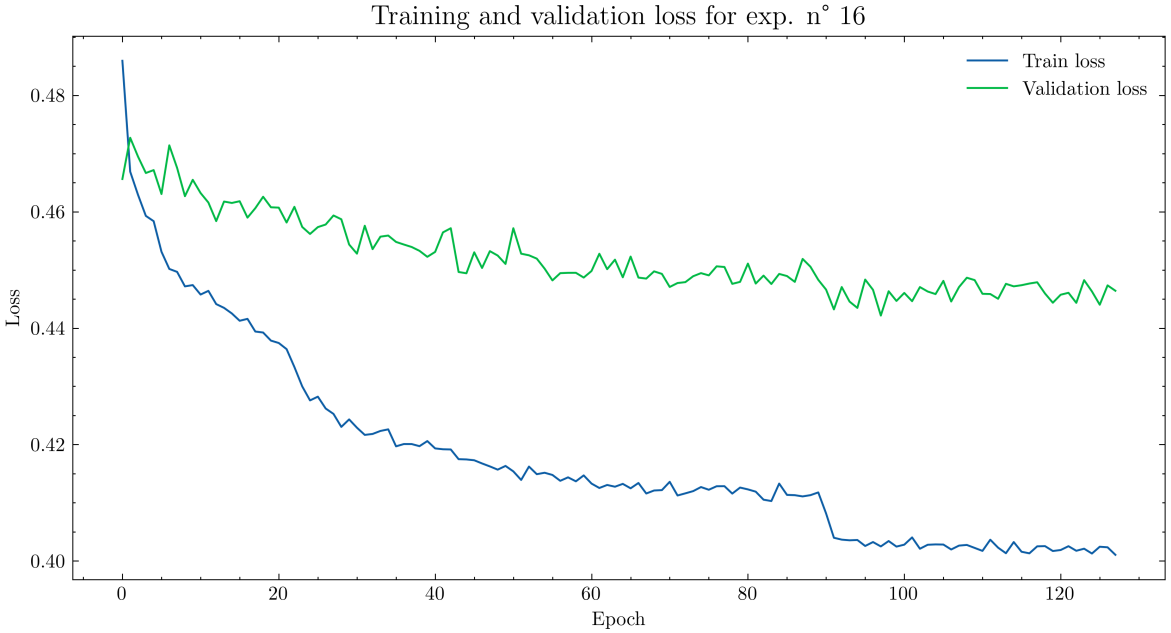


Figure 5.3: Train and validation loss function for Experiment 16

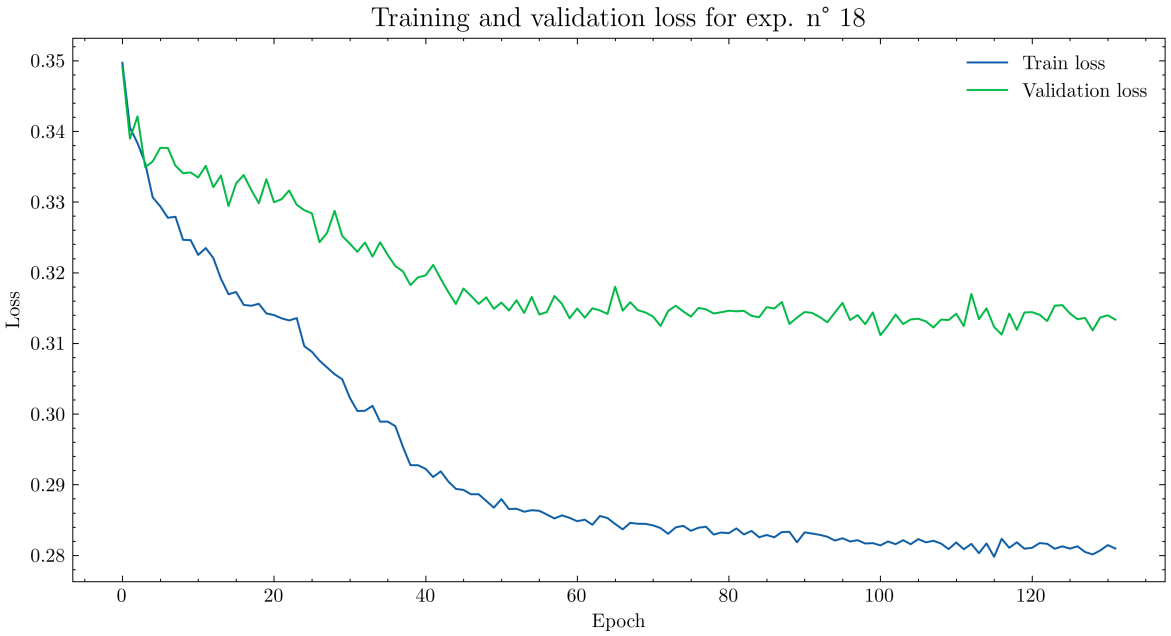


Figure 5.4: Train and validation loss function for Experiment 18

CHAPTER 6

Link classification with LTN-GCN

In this chapter, we apply the framework LTN-GCN to edge classification. In Section 6.1, we introduce the ogbl-biokg dataset, representing a biomedical knowledge graph provided with many types of entities and relationships. Then, we describe the pre-processing steps to construct the drug-drug interaction graph used in our example. In Section 6.2, the Real Logic knowledge base is specifically designed for tackling a multi-class multi-label classification problem. Finally, in Section 6.3, we discuss the results achieved within our experiments.

6.1 Dataset

The *ogbl-biokg* dataset (link) from the *Open Graph Benchmark*, is a Knowledge Graph provided with 5 types of entities (i.e. diseases, proteins, drugs, side effects, protein functions) and 51 directed relationships connecting two types of entities, including drug-drug interaction, protein-protein interaction as well drug-protein, drug-side effect, function-function relationships. All relationships correspond to directed edges, and those connecting the same type of entities are represented by bidirectional edges. Nodes, corresponding to entities, are featureless, i.e. they are not provided with a feature vector.

The dataset is created by collecting data from various biomedical repositories with different confidence levels. The resulting knowledge base is incomplete and may contain noise information.

We do not use the original *ogbl-biokg* curated by the authors for our purposes; instead, we apply the following pre-processing steps. Starting from the knowledge graph, we build an undirected graph, where nodes are entities of type "drug" and a single edge exists between any pair of nodes that are connected in the original graph. Essentially, we keep only nodes representing drugs and the relationships between them. Afterwards, for each pair of nodes, we replace the edges connecting them with a single edge labeled with an encoding vector $\in R^C$, where C denotes the number of drug-drug interaction classes. The encoding has a 1 in the $i-th$ position if the two nodes are connected by a relationship of the $i-th$ type in the original knowledge graph, and a 0 otherwise.

The obtained graph presents 10,533 nodes and 75,570 undirected edges. The related task is a 38-class multi-label link classification problem, where the goal is to predict multiple types of drug-drug interaction for each pair of drugs. Drug-drug interaction classes represent potential side effects that arise when one drug affects the activity of another, e.g. "immune_system_disease", "irritable_bowel_syndrome", "viral_infectious_disease".

In our experiments, we adopt a random split of 80%-10%-10%, producing a training

set with 60,456 edges, a validation set with 7,557 edges, and a test set with 7,557 edges. Figure 6.1 shows the distribution of drug-drug interaction types over the training set. Such a distribution is unbalanced since some classes contain much more samples than others.

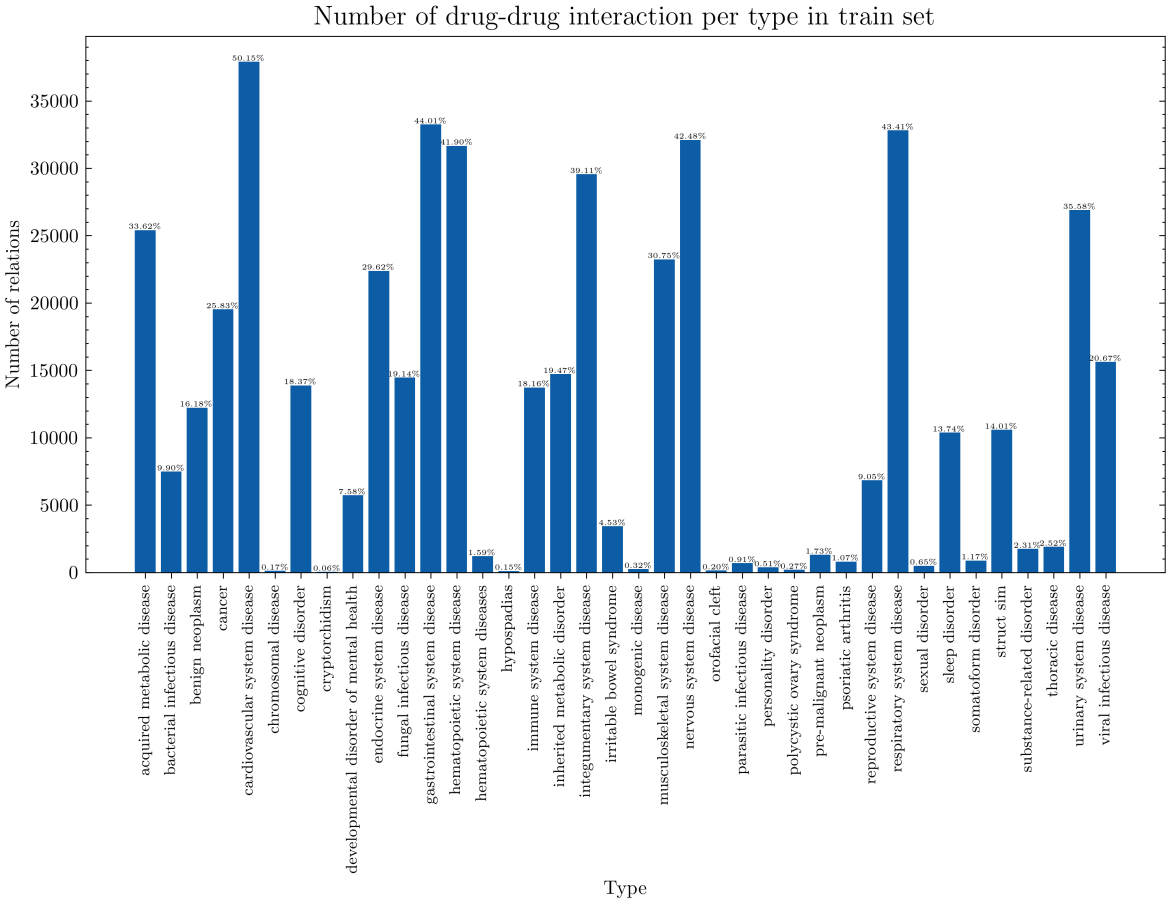


Figure 6.1: Bar plot of the number of relationships for each drug-drug interaction type. The percentages displayed above bars denote the proportion of edges labeled with a specific drug-drug interaction type.

Similarly to the node classification example, the main idea here is to infer meaningful insights from the dataset and leverage them to construct a robust knowledge base for training the LTN-GCN. In this case, we investigate whether drug-drug interaction types are correlated, specifically whether the presence of one interaction between two drugs suggests the existence of another. To this end, association rules of the form $t_1 \implies t_2$, where t_1 (i.e. the antecedent) and t_2 (i.e. the consequent) are types of drug-drug interaction, are extracted using the Apriori Algorithm. The itemsets used for this computation correspond to the sets of ground-truth classes assigned to each edge in the training partition. In addition, for searched rules, we set the minimum support to 0.01 and the minimum confidence to 0.85. Given an association rule $t_1 \implies t_2$, we recall that support and confidence are defined by the formulas:

$$\begin{aligned} Support &= \frac{|\{i \in I | \{t_1, t_2\} \subseteq i\}|}{|I|} \\ Confidence &= \frac{|\{i \in I | \{t_1, t_2\} \subseteq i\}|}{|\{i \in I | t_1 \in i\}|} \end{aligned} \tag{6.1.1}$$

where i is an itemset and I is the set of itemsets. By using appropriate thresholds for support and confidence, we derive only the association rules that consistently apply to the

training set. With these settings, the resulting association rules are 24 (Table 6.1).

Antecedent	Consequent	Support	Confidence
irritable bowel syndrome	gastrointestinal system disease	0.051	0.894
pre-malignant neoplasm	cardiovascular system disease	0.019	0.889
psoriatic arthritis	cardiovascular system disease	0.012	0.885
thoracic disease	gastrointestinal system disease	0.028	0.884
psoriatic arthritis	gastrointestinal system disease	0.012	0.874
irritable bowel syndrome	cardiovascular system disease	0.049	0.871
thoracic disease	cardiovascular system disease	0.027	0.87
psoriatic arthritis	nervous system disease	0.012	0.87
psoriatic arthritis	respiratory system disease	0.012	0.866
somatoform disorder	nervous system disease	0.013	0.866
pre-malignant neoplasm	respiratory system disease	0.019	0.864
pre-malignant neoplasm	integumentary system disease	0.019	0.863
bacterial infectious disease	cardiovascular system disease	0.107	0.863
somatoform disorder	cardiovascular system disease	0.013	0.862
irritable bowel syndrome	nervous system disease	0.049	0.861
reproductive system disease	cardiovascular system disease	0.097	0.86
benign neoplasm	cardiovascular system disease	0.174	0.859
pre-malignant neoplasm	nervous system disease	0.019	0.858
thoracic disease	nervous system disease	0.027	0.857
reproductive system disease	gastrointestinal system disease	0.097	0.855
irritable bowel syndrome	respiratory system disease	0.048	0.854
psoriatic arthritis	musculoskeletal system disease	0.011	0.854
inherited metabolic disorder	cardiovascular system disease	0.208	0.853
developmental disorder of mental health	cardiovascular system disease	0.081	0.852

Table 6.1: Association rules mined from training set edges using min_support = 0.01 and min_confidence=0.85, sorted in decreasing order by confidence

These association rules can be beneficial for the multi-class multi-label classification task if integrated into an *ad-hoc* knowledge base.

6.2 Knowledge Base

The knowledge base modeled for this task is composed of the following axioms:

$$\begin{aligned} \phi_1 \quad & \forall Diag(\langle d_1, d_2\rangle, l) : Train(\langle d_1, d_2\rangle) \quad Eq(Class(\langle d_1, d_2\rangle), l) \\ \phi_{2k} \quad & \forall \langle d_1, d_2\rangle \quad Antecedent_k(Class(\langle d_1, d_2\rangle)) \implies Consequent_k(Class(\langle d_1, d_2\rangle)) \end{aligned} \tag{6.2.1}$$

for $k = 1..K$, where $K = 24$ is the number of association rules identified.

In Equation 6.2.1, $\langle d_1, d_2\rangle$ denotes a drug-drug interaction edge, and l corresponds to the edge label. Note that the knowledge base counts $K + 1$ axioms, i.e. ϕ_1 and K additional axioms ϕ_{2k}, one for each association rule.

The formula ϕ_1 is similar to Equation 4.1.10. The guarded quantifier using the predicate $Train(\langle d_1, d_2\rangle)$ is added to restrict the evaluation of the formula to the train set. The predicate $Train$ holds only if the edge $\langle d_1, d_2\rangle$ belongs to the train partition. The predicate Eq holds if two edge labels are equal, while the function $Class$ returns the predicted label for a given edge.

The axioms ϕ_{2k} incorporate our knowledge from the mining of association rules. These formulas state that if the predicted label contains a class figuring in the antecedent of an association rule, then it must also contain the class figuring in the consequent. This fact is not restrictive because the selected association rules are all characterized by a high confidence,

hence it is true in most cases. In these axioms, $Antecedent_k$ and $Consequent_k$ are predicates that hold if the predicted label contains the class in the antecedent or the consequent of the k–th associate rule, respectively.

In what follows, we provide a Real Logic grounding for symbols introduced in Equation 6.2.1. First, the domains are grounded as:

$$\begin{aligned} \mathcal{G}(drug) &= \mathbb{R}^{128} \\ \mathcal{G}(label) &= [0,1]^{38} \end{aligned} \tag{6.2.2}$$

As drugs are not given any node-level information in the dataset, we choose to ground them with 128-dimension embeddings that are to be learned during training. Labels correspond to the encoding built in the pre-processing phase described at the beginning of this chapter.

Equation 6.2.3 illustrates the grounding of variables, predicates, and functions used in the knowledge base:

$$\begin{aligned} &\mathcal{G}(d) \in \mathbb{R}^{n\times 128} \\ &\mathcal{G}(\langle d_1, d_2\rangle) \in \mathbb{R}^{m\times 2\times 128} \\ &\mathcal{G}(l) \in [0,1]^{m\times 38} \\ &\mathcal{G}(Eq) : l', l'' \to e^{-\sum_{i=1}^{c} |l_i' - l_i''|} \qquad \alpha = 0.1 \\ &\mathcal{G}(Train) : \langle d_1, d_2\rangle \to x \in \{0,1\} \\ &\mathcal{G}(Class) : \langle d_1, d_2\rangle \to sigmoid(\mathrm{MLP}_{\theta_m}(\langle \mathrm{genGCN}_{\theta_g}(d_1, A), \mathrm{genGCN}_{\theta_g}(d_2, A)\rangle) \\ &\mathcal{G}(Antecedent_k) : l \to l_{a_k} \\ &\mathcal{G}(Consequent_k) : l \to l_{c_k} \end{aligned} \tag{6.2.3}$$

The variables d and l are sequences of n individuals, where n is the number of drugs (i.e. nodes). Instead the variable $\langle d_1, d_2\rangle$ is a sequence of m individuals, where m is the number of drug-drug interactions (i.e. edges). The groundings of Eq and $Class$ are derived from Equation 4.1.11. The activation function used in $\mathcal{G}(Class)$ is a $sigmoid$ since the classification task is a multi-label. Given a predicted label grounding l, the predicate $Antecedent_k$ returns the confidence score $\in [0,1]$ predicted for the class figuring in the antecedent of the k-th association rule, denoted as l_{a_k}. Similarly, the predicate $Consequent_k$ returns the confidence score predicted for the class figuring in the consequent of the k-th association rule, denoted as l_{c_k}.

The universal aggregator ($\forall$) and the aggregator $SatAgg$ are grounded with A_{pME} ($p = 2$). The implication ($\Longrightarrow$) is grounded with the Reichenbach fuzzy operator, following the Stable Product Real Logic configuration.

6.3 Experiments

6.3.1 Setup

The experiments with the *ogbl-biokg* dataset are carried out on a GPU NVIDIA RTX A2000 12GB, as in the node classification case. We test a graph convolutional neural network implemented as genGCN and trained with binary-cross entropy loss (Equation 6.3.1), and a LTN-GCN trained to maximally satisfy the formula ϕ_1, denoted as LTN-GCN$_1$.

$$Binary \quad Cross-Entropy = \frac{1}{C}\sum_{i=1}^{C} y_i \log(p_i) + (1-y_i)\log(1-p_i) \tag{6.3.1}$$

Here y is the ground-truth one-hot encoded label and p is the predicted probability vector.

We do not manage to train a LTN-GCN based on the formulas ϕ_{2k} illustrated in Equation 6.2.1, due to memory limitations.

For each of the two models, we build two implementations using different types of convolutional layers. The first genGCN includes a GCN convolution, while the second includes a GraphSAGE layer. The number k of layers is set to 3 in both configurations, and the hidden channels are fixed to 256. Since the given task is link classification, a Multi-Layer Perceptron, used as an edge encoder, is appended to our genGCN implementation. The MLP is obtained by stacking 3 dense layers.

We train our models for 500 epochs with ADAM optimizer and a learning rate of $5 \cdot 10^{-3}$, and run each training $n = 5$ times. The full-batch mode is adopted, i.e. models are trained on a single batch containing the whole graph, instead of recurring to mini-batch sampling strategies. Even in this case, an early stop strategy reduces training time, by using 30 patience epochs.

The evaluation metrics used here are three multi-label accuracy scores that compare the predicted label with the ground-truth label employing different criteria. We first apply function t (Equation 6.3.2) to each element of the predicted vector p, obtaining p'.

$$t(x) = \begin{cases} 1 & if\, x \geq 0.5 \\ 0 & otherwise \end{cases} \tag{6.3.2}$$

Then, we compute the following metrics:

$$\begin{aligned} Accuracy_{hamming} &= \frac{1}{m \cdot C} \sum_{i=1}^{m} \sum_{j=1}^{C} acc_1(y_{ij}, p'_{ij}) \\ Accuracy_{exact} &= \frac{1}{m} \sum_{i=1}^{m} acc_2(y_i, p'_i) \\ Accuracy_{belong} &= \frac{1}{m} \sum_{i=1}^{m} acc_3(y_i, p'_i) \end{aligned} \tag{6.3.3}$$

where y_i is the ground-truth label of the i-th edge and p'_i is the predicted label of the i-th edge after applying the function t element-wise. The additional subscript j denotes the j-th element in vectors y_i and p'_i. C is the number of classes, while m is the number of edges. In Equation 6.3.3, we introduce the following functions:

$$\begin{aligned} acc_1(y_{ij}, p'_{ij}) &= \begin{cases} 1 & \text{if} \quad y_{ij} = p'_{ij} \\ 0 & \text{otherwise} \end{cases} \\ acc_2(y_i, p'_i) &= \begin{cases} 1 & \text{if} \quad y_i = p'_i \\ 0 & \text{otherwise} \end{cases} \\ acc_3(y_i, p'_i) &= \begin{cases} 1 & \text{if} \quad \{j | y_{ij} = 1\} \subseteq \{j | p'_{ij} = 1\} \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{6.3.4}$$

Intuitively, $Accuracy_{hamming}$ measures the fraction of correctly predicted classes within labels over the total number of classes within labels (i.e. $m \cdot C$); $Accuracy_{exact}$ measures the fraction of predicted labels that exactly match the ground truth labels, in each position of the vector; $Accuracy_{belong}$ computes the fraction of predicted labels so that the set of predicted classes fully belongs to the corresponding set of classes in the ground-truth label.

6.3.2 Results

Table 6.2 collects the evaluation metrics measured in our experiments. These are obtained by averaging the scores of the model with the best validation $Accuracy_{exact}$ in each run.

Model	$Accuracy_{hamming}$			$Accuracy_{exact}$			$Accuracy_{belong}$		
	Train	Val	Test	Train	Val	Test	Train	Val	Test
GCN	<u>0.856</u>	<u>0.856</u>	<u>0.857</u>	<u>0.165</u>	<u>0.157</u>	<u>0.165</u>	**0.463**	**0.459**	**0.466**
GraphSAGE	**0.878**	**0.878**	**0.878**	**0.172**	**0.163**	**0.171**	0.387	0.380	0.387
LTN-GCN$_1$ (GCN)	0.849	0.850	0.850	0.144	0.138	0.143	0.363	0.359	0.366
LTN-GCN$_1$ (GraphSAGE)	0.847	0.847	0.848	0.137	0.123	0.127	<u>0.435</u>	<u>0.427</u>	<u>0.433</u>

Table 6.2: $Accuracy_{hamming}$, $Accuracy_{exact}$, and $Accuracy_{belong}$ in training, validation, and test set. For each column, the bold value represents the best score, while the underlined one coincides with the second-best score.

In our experiment, LTN-GCN$_1$ models achieve performances comparable to those of the standalone graph neural networks. Limiting the scope to the test metrics, we observe that LTN-GCN$_1$ (GCN) and LTN-GCN$_1$ (GraphSAGE) are slightly worse than GraphSAGE in $Accuracy_{hamming}$ (-0.028 and -0.030, respectively) and in $Accuracy_{exact}$ (-0.028 and -0.034, respectively). Moreover, LTN-GCN$_1$ (GraphSAGE) obtains an $Accuracy_{belong}$ comparable with that of GCN (-0.033), while the difference in $Accuracy_{belong}$ between LTN-GCN$_1$ (GraphSAGE) and GCN is bigger (-0.100). Similarly, the accuracy scores related to the train and the validation sets are comparable across all models. Note that $Accuracy_{exact}$ tends to assume low values since it requires the exact match between labels. On the other hand, $Accuracy_{hamming}$ and $Accuracy_{belong}$ are less restrictive metrics and are characterized by higher scores.

The results achieved with LTN-GCN do not deviate from those achieved with traditional graph neural networks and confirm that our framework can be successfully applied to link classification tasks. It is worth pointing out that the knowledge base used to train LTN-GCN$_1$ includes the logical formula ϕ_1, and the facts derived from association rules are not included. Consequently, we expect that augmenting the knowledge base with more facts could enhance the accuracy of LTN-GCN. The *ogbl-biokg* use case has been instrumental in evaluating the capabilities of LTN-GCN and verifying its applicability to other link classification tasks.

Conclusion

In this study, we explored the combination of Logic Tensor Networks with Graph Neural Networks for addressing graph-based problems. The primary contribution of our work is the development of a novel framework, called LTN-GCN, which involves training Graph Convolutional Neural Network in order to maximize the satisfiability of a knowledge base composed of First Order Logic (FOL) rules.

After introducing the necessary theoretical background, we focused on formalizing the proposed framework. The novel idea of LTN-GCN is grounding predicates and function symbols with Graph Convolutional Neural Network models. Building on this concept, we defined a knowledge base and related groundings for two typical graph-related problems, namely node classification and edge classification. We worked out various Real Logic formulations for binary and multi-class classification tasks and discussed different potential knowledge bases. These are to be intended as templates applicable to any node or edge classification task and can be extended with additional task-specific facts.

Next, we covered the implementation details of LTN-GCN. The embedded Graph Convolutional Neural Network was modeled with a simple architecture, named genGCN, that enables to use either a GCN or a GraphSAGE convolutional layer. These models were realized by means of *Pytorch* and *torch geometric*. The LTN component was implemented with the module *LTNTorch*, which provides built-in fuzzy operators within the Stable Product Real Logic configuration.

To validate our approach, we selected two classification tasks using benchmark datasets. In the first task, we trained a LTN-GCN to classify the category of papers in the *ogbn-arxiv* citation graph. The knowledge base for LTN training included a FOL fact that formulated the multi-class single-label node classification problem in Real Logic, as well as a rule reflecting the graph homophily. The latter states that pairs of papers, where one cites the other, are likely to belong to a similar category. In terms of accuracy and F1-score, LTN-GCN performed comparable to a standalone graph neural network trained with cross-entropy loss, regardless of the type of convolutional layer used.

In the second round of experiments, we applied LTN-GCN to a drug-drug interaction network, training it to classify edges with the correct set of labels denoting side effects caused by drug interactions. For this task, we formulated an FOL fact for the multi-class multi-label edge classification problem in Real Logic and derived additional axioms using association rules between side effects. However, these rules were not implemented due to computational limitations. As with the previous task, LTN-GCN achieved performance similar to that of traditional GCNs trained with binary cross-entropy loss.

To summarize, our framework can be successfully applied to node and edge classification

tasks. The major outcomes are the formalization of LTN-GCN and its utilization in two illustrative problems. Nevertheless, our work raises several open questions and suggests directions for future research.

The next step in this work would be to investigate the effectiveness of LTN-GCN when applied to more sophisticated knowledge bases. In the current study, the knowledge bases used were relatively simple, and the precise impact of task-specific facts on model performance was not thoroughly evaluated. Incorporating more facts that impose constraints on the given task could be beneficial to the training of LTN-GCN, potentially offering a distinct advantage over standard graph neural networks, by leveraging richer domain-specific information.

On the other hand, increasing the complexity of the knowledge base introduces new challenges, particularly in terms of computational cost. As the number of axioms grows, the training process becomes more resource-intensive. The overhead does not depend only on the size of the knowledge base, but it is also influenced by the complexity of the formulas, such as those involving aggregators or nested logic. Future work should focus on improving the efficiency of LTN-GCN training process from a computational perspective, especially when adopting a broad knowledge base.

Furthermore, the flexibility of the LTN-GCN framework could be further explored by integrating other graph neural network models, beyond GCN and GraphSAGE. Other architectures, such as GAT or GAE, could be studied within LTN-GCN, potentially offering better alignment with the logic-based structure of LTN and leading to improved performance.

While LTN-GCN was presented as a framework for addressing node and edge classification problems, it could be extended to other graph-related tasks, such as graph classification and graph clustering. By constructing new knowledge bases and corresponding groundings tailored to these tasks, it would be possible to demonstrate its versatility.

In conclusion, LTN-GCN holds the potential for solving graph-based problems where a well-defined knowledge base is available. Applying LTN-GCN to real-world datasets, such as social networks, molecular interaction networks, or financial graphs, would provide valuable insights into its practical utility. Moreover, this framework pursues the interpretability of the models used in these applications by establishing a human-readable set of axioms for the training process.

Bibliography

Badreddine, S., d'Avila Garcez, A., Serafini, L. and Spranger, M. (2022), «Logic Tensor Networks», *Artificial Intelligence*, vol. 303, p. 103 649, URL http://dx.doi.org/10.1016/j.artint.2021.103649. (Cited at pages v, 13, 16, 17, 18, 19, 29 e 31)

Bhuyan, B. P., Ramdane-Cherif, A., Tomar, R. and Singh, T. P. (2024), «Neuro-symbolic artificial intelligence: a survey», *Neural Computing and Applications*, vol. 36 (21), p. 12 809–12 844, URL https://doi.org/10.1007/s00521-024-09960-z. (Cited at pages v, 5 e 6)

Brockschmidt, M., Allamanis, M., Gaunt, A. L. and Polozov, O. (2019), «Generative Code Modeling with Graphs», URL https://arxiv.org/abs/1805.08490. (Cited at page 26)

Chiang, W.-L., Liu, X., Si, S., Li, Y., Bengio, S. and Hsieh, C.-J. (2019), «Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks», in «Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery Data Mining», KDD '19, ACM, URL http://dx.doi.org/10.1145/3292500.3330925. (Cited at pages 24 e 37)

d'Avila Garcez, A. S. and Lamb, L. C. (2020), «Neurosymbolic AI: The 3rd Wave», *CoRR*, vol. abs/2012.05876, URL https://arxiv.org/abs/2012.05876. (Cited at page 5)

Donadello, I., Serafini, L. and d'Avila Garcez, A. (2017), «Logic Tensor Networks for Semantic Image Interpretation», URL https://arxiv.org/abs/1705.08968. (Cited at page 5)

Garnelo, M., Arulkumaran, K. and Shanahan, M. (2016), «Towards Deep Symbolic Reinforcement Learning», *CoRR*, vol. abs/1609.05518, URL http://arxiv.org/abs/1609.05518. (Cited at page 5)

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O. and Dahl, G. E. (2017), «Neural Message Passing for Quantum Chemistry», URL https://arxiv.org/abs/1704.01212. (Cited at pages 21 e 24)

Hamilton, W. L., Ying, R. and Leskovec, J. (2018a), «Inductive Representation Learning on Large Graphs», URL https://arxiv.org/abs/1706.02216. (Cited at page 24)

Hamilton, W. L., Ying, R. and Leskovec, J. (2018b), «Inductive Representation Learning on Large Graphs», URL https://arxiv.org/abs/1706.02216. (Cited at page 36)

Hitzler, P., Eberhart, A., Ebrahimi, M., Sarker, M. K. and Zhou, L. (2022), «Neuro-symbolic approaches in artificial intelligence», *National Science Review*, vol. 9 (6), p. nwac035, URL https://doi.org/10.1093/nsr/nwac035. (Cited at page 4)

Kautz, H. A. (2022), «The third AI summer: AAAI Robert S. Engelmore Memorial Lecture», *AI Magazine*, vol. 43 (1), p. 105–125, URL https://onlinelibrary.wiley.com/doi/abs/10.1002/aaai.12036. (Cited at page 7)

Kingma, D. P. and Welling, M. (2022), «Auto-Encoding Variational Bayes», URL https://arxiv.org/abs/1312.6114. (Cited at page 26)

Kipf, T. N. and Welling, M. (2017), «Semi-Supervised Classification with Graph Convolutional Networks», URL https://arxiv.org/abs/1609.02907. (Cited at page 23)

Lample, G. and Charton, F. (2019), «Deep Learning for Symbolic Mathematics», URL https://arxiv.org/abs/1912.01412. (Cited at page 9)

Li, Z., Chen, Q. and Koltun, V. (2018), «Combinatorial Optimization with Graph Convolutional Networks and Guided Tree Search», URL https://arxiv.org/abs/1810.10659. (Cited at page 26)

Lim, D., Li, X., Hohne, F. and Lim, S.-N. (2021), «New Benchmarks for Learning on Non-Homophilous Graphs», URL https://arxiv.org/abs/2104.01404. (Cited at page 33)

Manhaeve, R., Dumančić, S., Kimmig, A., Demeester, T. and Raedt, L. D. (2018), «DeepProbLog: Neural Probabilistic Logic Programming», URL https://arxiv.org/abs/1805.10872. (Cited at page 8)

Mao, J., Gan, C., Kohli, P., Tenenbaum, J. B. and Wu, J. (2019), «The Neuro-Symbolic Concept Learner: Interpreting Scenes, Words, and Sentences From Natural Supervision», URL https://arxiv.org/abs/1904.12584. (Cited at page 8)

Prates, M. O. R., Avelar, P. H. C., Lemos, H., Lamb, L. and Vardi, M. (2018), «Learning to Solve NP-Complete Problems - A Graph Neural Network for Decision TSP», URL https://arxiv.org/abs/1809.02721. (Cited at page 26)

Sen, P., de Carvalho, B. W. S. R., Riegel, R. and Gray, A. (2021), «Neuro-Symbolic Inductive Logic Programming with Logical Neural Networks», URL https://arxiv.org/abs/2112.03324. (Cited at page 5)

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P. and Bengio, Y. (2018), «Graph Attention Networks», URL https://arxiv.org/abs/1710.10903. (Cited at page 25)

Wang, P., Dou, D., Wu, F., de Silva, N. and Jin, L. (2019), «Logic Rules Powered Knowledge Graph Embedding», URL https://arxiv.org/abs/1903.03772. (Cited at page 5)

Wei, G.-W. (2019), «Protein structure prediction beyond AlphaFold», *Nature Machine Intelligence*, vol. 1 (8), p. 336–337. (Cited at page 26)

Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C. and Yu, P. S. (2021), «A Comprehensive Survey on Graph Neural Networks», *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32 (1), p. 4–24, URL http://dx.doi.org/10.1109/TNNLS.2020.2978386. (Cited at pages v, vi, 22, 23 e 26)

Xiao, S., Wang, S., Dai, Y. and Guo, W. (2021), «Graph neural networks in node classification: survey and evaluation», vol. 33 (1), p. 4. (Cited at pages v, vi, 23, 24 e 25)

Yi, K., Wu, J., Gan, C., Torralba, A., Kohli, P. and Tenenbaum, J. B. (2019), «Neural-Symbolic VQA: Disentangling Reasoning from Vision and Language Understanding», URL https://arxiv.org/abs/1810.02338. (Cited at pages v e 6)

Ying, R., He, R., Chen, K., Eksombatchai, P., Hamilton, W. L. and Leskovec, J. (2018), «Graph Convolutional Neural Networks for Web-Scale Recommender Systems», in «Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery Data Mining», ACM, URL http://dx.doi.org/10.1145/3219819.3219890. (Cited at page 21)

Yu, B., Yin, H. and Zhu, Z. (2018), «Spatio-Temporal Graph Convolutional Networks: A Deep Learning Framework for Traffic Forecasting», in «Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence», p. 3634–3640, International Joint Conferences on Artificial Intelligence Organization, URL http://dx.doi.org/10.24963/ijcai.2018/505. (Cited at page 21)

Yu, D., Yang, B., Liu, D., Wang, H. and Pan, S. (2023), «A Survey on Neural-symbolic Learning Systems», URL https://arxiv.org/abs/2111.08164. (Cited at pages v e 4)

Zeng, H., Zhou, H., Srivastava, A., Kannan, R. and Prasanna, V. (2020), «GraphSAINT: Graph Sampling Based Inductive Learning Method», URL https://arxiv.org/abs/1907.04931. (Cited at page 37)

Zhu, Z., Galkin, M., Zhang, Z. and Tang, J. (2022), «Neural-Symbolic Models for Logical Queries on Knowledge Graphs», URL https://arxiv.org/abs/2205.10128. (Cited at page 27)

Sitography

- "A Gentle Introduction to Graph Neural Networks" – https://distill.pub/2021/gnn-intro/
- "A tour of PyG's data loaders. By Grant Uy and Huijian Cai as part of... | by Grant Uy | Stanford CS224W GraphML Tutorials | Medium" – https://medium.com/stanford-cs224w/a-tour-of-pygs-data-loaders-9f2384e48f8f
- "Link Prediction on Heterogeneous Graphs with PyG | by PyTorch Geometric | Medium" – https://medium.com/@pytorch_geometric/link-prediction-on-heterogeneous-graphs-with-pyg-6d5c29677c70
- "logictensornetworks/LTNtorch: PyTorch implementation of Logic Tensor Networks, a Neural-Symbolic framework." – https://github.com/logictensornetworks/LTNtorch
- "LTNtorch's documentation — LTNtorch 0.9 documentation" – https://tommasocarraro.github.io/LTNtorch/index.html
- "Open Graph Benchmark | A collection of benchmark datasets, data-loaders and evaluators for graph machine learning in PyTorch." – https://ogb.stanford.edu/ https://pytorch-geometric.readthedocs.io/en/latest/index.html
- "PyTorch documentation — PyTorch 2.3 documentation" – https://pytorch.org/docs/2.3/

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Prof. Domenico Ursino, for his guidance and availability throughout the production and correction of this thesis. His commitment during this intense period of study has been remarkable.

I am also very grateful to my co-supervisors, Dr. Francesco Cauteruccio and Dr. Enrico Corradini, for their valuable contributions and suggestions to this research. With their help, I was able to overcome some difficulties related to the novelty of the field of application.

Special thanks go to Dr. Lucia Migliorelli, for her sincere dedication and passionate coaching activity during the time I was "taking my first steps" towards a career in research.

I can only thank my supportive parents Antonio and Daniela, my precious sister Emma, my loving grandparents, and all my relatives (near and far) for their constant encouragement and strong belief in me.

Finally, I would like to thank my flatmates Alessandro, Alessio and Marco, my fellow students, with a special mention to Christopher and Luca, and all friends from my hometown for the formative experiences and the great time we have shared over the last two years.