DEEP LEARNING WITH RELATIONAL LOGIC REPRESENTATIONS

GUSTAV ŠÍR

Department of Computer Science
Faculty of Electrical Engineering
Czech Technical University in Prague

A thesis submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy (Ph.D.)

Study Programme No. P2612 - Electrotechnics and Informatics
Branch No. 3902V035 - Artificial Intelligence and Biocybernetics

SUPERVISOR:
Prof. Ing. Filip Železný, Ph.D.

SUPERVISOR SPECIALIST:
Ing. Ondřej Kuželka, Ph.D.

December 2020

Gustav Šír: *Deep Learning with Relational Logic Representations,*
a thesis submitted for the degree of Doctor of Philosophy (Ph.D.),
Study Programme No. P2612-Electrotechnics and Informatics,
Branch No. 3902V035 - Artificial Intelligence and Biocybernetics.

SUPERVISORS:
Prof. Ing. Filip Železný, Ph.D.
Ing. Ondřej Kuželka, Ph.D.

LOCATION:
Prague, Czech Republic

TIME FRAME:
2013 – 2020

To my friends and family for constantly asking:
"when will you finally finish the school?!"
which provided the much needed encouragement.

ABSTRACT

In the recent years, we have seen tremendous resurgence of neural networks, applied with great success in highly diverse domains, ranging from speech recognition to game playing. The unprecedented progress of this new deep learning trend has even been seen as paving our way towards general artificial intelligence. However, the current deep learning models are still limited in many regards. Particularly, in this thesis we address the contemporary problem of learning neural networks from *relational* data and knowledge representations. While virtually all standard models are limited to data in the form of fixed-size tensors, the relational data are omnipresent in the interlinked structures of the Internet and relational databases. Likewise, in many domains a background knowledge in the form of relational logic rules or rich graph-based structures is often available, yet impossible or very difficult to exploit with the standard deep learning models.

To address this issue, we introduce a declarative deep relational learning framework called Lifted Relational Neural Networks (LRNNs). The main idea underlying the framework is to approach the neural networks through the *lifted modeling* paradigm, known otherwise from Statistical Relational Learning (SRL), where it is used to exploit *symmetries* in learning problems. Similarly to lifted graphical models from SRL, LRNNs are then represented as sets of *weighted relational logic* rules, used to describe the structure of a given learning setting.

As set out, we demonstrate that this paradigm allows for effective end-to-end neural learning with relational data and knowledge. The encoding through the weighted relational logic rules then provides flexible means for implementing a wide variety of novel modeling concepts incorporating various latent relational patterns. Notably, these also elegantly cover contemporary deep convolutional models, such as Graph Neural Networks, as a simple special case. We explain how to easily generalize these state-of-the-art models towards higher expressiveness, and also demonstrate the general LRNN framework on various practical learning scenarios and benchmarks, including computational efficiency.

Additionally, we introduce several enhancements to the framework. Firstly, we present an automated structure learning of the relational rules, composing the lifted models. Secondly, we introduce two principled optimization techniques used to scale up the integrative framework from both the logical and neural learning perspectives. Both the optimization methods are then effective also separately in the respective approaches to learning. Lastly, we demonstrate the framework on selected use cases in different domains.

Keywords: deep learning, relational data, logic, relational learning, neural-symbolic integration, inductive logic programming, statistical relational learning, weighted logic, symmetries, lifted inference, convolutional models, graph neural networks, logic representations, background knowledge

ABSTRAKT

V posledních letech jsme byli svědky dramatického oživení oblasti neuronových sítí, které se velmi úspěšně aplikují v nejrozmanitějších doménách, od rozpoznávání řeči po hraní her. Nebývalé úspěchy tohoto trendu tzv. "hlubokého učení" jsou dokonce často považovány za náznaky schopností obecné umělé inteligence. Současné modely hlubokého učení jsou však v mnoha ohledech stále velmi omezené. V této práci se konkrétně věnujeme aktuálnímu problému učení neuronových sítí z *relačních* dat a reprezentací znalostí. Zatímco prakticky všechny standardní modely jsou omezeny na data ve formě numerických tenzorů pevné velikosti, relační data jsou všudypřítomná, od vzájemně propojených struktur Internetu po relační databáze. Podobně je v mnoha doménách často dostupná znalost ve formě relačních logických pravidel či komplexních znalostních grafů, přesto je jejich využití se standardními modely hlubokého učení nemožné nebo velmi obtížné.

K řešení tohoto problému představujeme deklarativní systém pro hluboké relační učení s názvem "Lifted Relational Neural Networks" (LRNNs). Hlavní myšlenkou tohoto systému je přístup k neuronovým sítím prostřednictvím paradigmatu známého jako textit lifted modeling, které bylo představeno v oblasti Statistického Relačního Učení (SRL), kde se používá k efektivnímu kódování *symetrií* v problémech. Podobně jako "liftované" grafické modely známé z SRL jsou i LRNNs reprezetované jako sady vážených logických pravidel, sloužící k popisu struktury daného problému učení.

Jak bylo vytyčeno, v této práci demonstruujeme, že toto paradigma umožňuje efektivní přímé neurální učení nad relačními daty a znalostmi. Kódování prostřednictvím těchto vážených relačních pravidel pak poskytuje flexibilní prostředky pro implementaci široké škály nových konceptů prediktivního modelování, zahrnujících různé latentní relační vzory. Důležitým aspektem je i to, že tento přístup elegantně pokrývá moderní konvoluční neurální modely, například grafové neuronové sítě, jako jednoduchý speciální případ. V práci vysvětlujeme, jak snadno zobecnit tyto nejmodernější modely směrem k vyšší expresivitě, a také demonstrujeme navržený LRNN systém na různých srovnávacích testech kvality strojového učení, včetně testů výpočetní efektivity.

V práci poté představujeme i několik vylepšení toho přístupu. Nejprve představíme automatické učení struktury oněch relačních pravidel, reprezentujících celkovou učící architekturu. Poté uvedeme dvě principiální optimalizační techniky využité pro lepší škálování systému z perspektivy jak symbolického (logického), tak i neurálního přístupu k učení. Obě optimalizační metody jsou použitelné a účinné i samostatně v těchto příslušných přístupech k učení. Nakonec demonstruujeme systém na vybraných případech použití v různých doménách.

Klíčová slova: hluboké učení, relační data, logika, relační učení, neurálně-symbolická integrace, induktivní logické programování, statistické relační učení, vážená logika, symetrie, liftovaná inference, konvoluční modely, grafové neuronové sítě, logické reprezentace, postranní znalost

PUBLICATIONS

Some contents of this thesis have appeared previously in the following publications[1] :

[1] **Gustav Šourek**, Vojtěch Aschenbrenner, Filip Železný, Steven Schockaert, and Ondřej Kuželka. "Lifted relational neural networks: Efficient learning of latent relational structures." In: *Journal of Artificial Intelligence Research (JAIR)* 62 (2018), pp. 69–100.

[2] **Gustav Šourek**. "Deep learning with relational logic representations." In: *Proceedings of the 28th International Joint Conference on Artificial Intelligence*. AAAI Press. 2019.

[3] **Gustav Šourek**, Filip Železný, and Ondřej Kuželka. "Lossless Compression of Structured Convolutional Models via Lifting." In: *International Conference on Learning Representations*. ICLR OpenReview. 2021.

[4] **Gustav Šourek**, Martin Svatoš, Filip Železný, Steven Schockaert, and Ondřej Kuželka. "Stacked structure learning for lifted relational neural networks." In: *International Conference on Inductive Logic Programming*. Springer. 2017, 140–151, ⟵ **Best Paper Award**.

[5] **Gustav Šourek**, Suresh Manandhar, Filip Železný, Steven Schockaert, and Ondřej Kuželka. "Learning predictive categories using lifted relational neural networks." In: *International Conference on Inductive Logic Programming*. Springer. 2016, pp. 108–119.

[6] **Gustav Šourek**, Vojtěch Aschenbrenner, Filip Železný, and Ondřej Kuželka. "Lifted relational neural networks." In: *Proceedings of the Workshop on Cognitive Computation: Integrating Neural and Symbolic Approaches co-located with NeurIPS*. CEUR Workshop Proceedings. 2015.

[7] **Gustav Šourek**, Filip Železný, and Ondřej Kuželka. "Beyond Graph Neural Networks with Lifted Relational Neural Networks." In: *arXiv preprint arXiv:2007.06286* (2020).

[8] **Gustav Šourek**, Filip Železný, and Ondřej Kuželka. "Learning with Molecules beyond Graph Neural Networks." In: *Machine Learning for Molecules worshop at NeurIPS* (2020), paper 24.

[9] **Gustav Šourek**, Ondřej Kuželka, and Filip Železný. "Predicting top-k trends on twitter using graphlets and time features." In: *International Conference on Inductive Logic Programming 2013, Late Breaking Papers*. CEUR Workshop Proceedings. 2013.

[10] Martin Svatoš, **Gustav Šourek**, Filip Železný, Steven Schockaert, and Ondřej Kuželka. "Pruning hypothesis spaces using learned domain theories." In: *International Conference on Inductive Logic Programming*. Springer. 2017, pp. 152–168.

[11] Ondřej Hubáček, **Gustav Šourek**, and Filip Železný. "Lifted Relational Team Embeddings for Predictive Sports Analytics." In: *ILP Up-and-Coming / Short Papers*. 2018, pp. 84–91.

[12] Ondřej Hubáček, **Gustav Šourek**, and Filip Železný. "Learning to predict soccer results from relational data with gradient boosted trees." In: *Machine Learning* 108.1 (2019), pp. 29–47.

[13] Ondřej Hubáček, **Gustav Šourek**, and Filip Železný. "Deep learning from spatial relations for soccer pass prediction." In: *International Workshop on Machine Learning and Data Mining for Sports Analytics, ECML-PKDD*. Springer. 2018, pp. 159–166.

1 Please note that Gustav Šourek = Gustav Šír, the author of this thesis (I took my wife's surname recently).

"Never confuse education with intelligence,
you can have a PhD and still be an idiot."

— Richard P. Feynman

ACKNOWLEDGEMENTS

First and foremost, my deepest gratitude goes to my supervisor specialist Ondřej Kuželka for always answering my cries for help, proposing solutions to every problem at hand, and the immense patience he showed in his roles of a flawless adviser and an intellectual role model. Secondly, I am also deeply grateful to my supervisor Filip Železný for his steady support in all my endeavors since the very beginning of this very long PhD journey[2]. Thirdly, I would like to thank my friend and former classmate Vojtěch Aschenbrenner, who first started working on the deep relational learning ideas during his Master's 8 year ago (under supervision of Ondřej Kuželka), before opting for a different research subject.

My gratitude also belongs to my dear colleagues Ondřej Hubáček, Matěj Uhrín and Martin Svatoš, with whom I worked on various related projects. Also, I thank Steven Schockaert for help with some of the papers.

I am thankful to Suresh Manandhar from University of York for having me over for a research stay. Likewise, I am grateful to Ofer Lavi from IBM Research, Haifa/Tel-Aviv, and Dmitry Parashchenko from Google, Zürich, for their hospitality and the experience of also getting my hands dirty with some real projects.

Additionally, I would actually like to thank my high-school physics teacher Zdeněk Polák, who first instilled in me the passion for scientific thinking.

My warmest thanks then go to my family for their unconditional support, love, and patience. To my mother and father, whom I hold dearly in my heart. To my grandfather, who graduated from the same faculty. And to my grandmas, who sadly passed away during this PhD journey of mine.

Last but not least, I am profoundly grateful to the family I have started in the meantime. To my beloved wife, Věrka, for showing me what it really means to be a good person, and to my children, for reminding me what a true joy is.

2 I acknowledge support by the following grants provided by the Czech Science Foundation: 17-26999S (2017-2019), 20-29260S (2020-2022), by the Czech Technical University: SGS11/155/OHK3/3T/13 (2013), SGS14/079/OHK3/1T/13 (2014), SGS16/093/OHK3/1T/13 (2016), SGS17/189/OHK3/3T/13 (2017-2020), SGS20/178/OHK3/3T/13 (2020-2022), and by Cisco: 830130051C (2013-2015), 8301525C000 (2016).

CONTENTS

ACRONYMS

AI Artificial Intelligence

ML Machine Learning

FOL First-Order Logic

RML Relational Machine Learning

ILP Inductive Logic Programming

SRL Statistical Relational Learning

NSI Neural-Symbolic Integration

MLN(s) Markov Logic Network(s)

BLP(s) Bayesian Logic Program(s)

ANN(s), NN(s) Artificial Neural Networks

MLP(s) Multi-Layer Perceptron(s)

RNN(s) *Recursive* Neural Network(s)

CNN(s) Convolutional Neural Network(s)

G(C)NN(s) Graph (Convolutional) Neural Network(s)

GIN Graph Isomorphism Network(s)

KBE(s) Knowledge Base Embedding(s)

KBC Knowledge Base Completion

LTSM(s) Long-Short Term Memory Network(s)

KBANN Knowledge-Based Neural Networks

LRNN(s) Lifted Relational Neural Network(s)

WL Weisfeiler-Lehman (algorithm)

CSP Constraint Satisfaction Problem/Programming

SLD Selective Linear Definite (clause resolution)

SQL Structured Query Language

EM Expectation maximization

SotA State-of-the-Art

ADAM Adaptive Moment Estimation (optimizer)

(C/G)PU (Central/Graphics) Processing Unit

AUC-ROC/PR Area Under the Curve - Receiver Operating Characteristics / Precision Recall

Part I

INTRODUCTION

1

INTRODUCTION

All intelligent life forms instinctively *model* their surrounding environments in order to predict their possible futures. For it is only with such a model that one can actively navigate their paths through the environment towards (more) desirable ends.

Artificial intelligence (AI) then tries to understand and automate this interesting ability of living systems with the subfield of *machine learning* at the core. This data-driven approach is particularly useful in cases where the underlying principles of the environment, or system S, are unknown, or it is rather infeasible to manually derive a set of equations or otherwise explicit mathematical forms of S.[1] As opposed to that traditional scientific approach[2] , in machine learning we try to *model* complex systems S automatically, with much more generic mathematical forms, the representation of which constitutes the scope of this thesis. This automated approach is based on adapting the internal parameters or structure of the generic, flexible mathematical forms $\mathcal{M}$ to match the presented data samples $\mathcal{D}$ coming from S. This adaptation process, formally driven by minimization of some "loss" measure $L(S, M|\mathcal{D})$ capturing the discrepancy between the actual system S and its model M, as measured over some set of observed data $\mathcal{D}$, is then commonly referred to as *learning*.[3]

Typically, in *supervised* statistical machine learning, the data samples $(x_i, y_i) \in \mathcal{D}$ are considered to be randomly drawn from the respective sets, where $x_i \in \mathcal{X}$ is a description of a learning instance i, representing the system S's input, and $y_i \in \mathcal{Y}$ is the corresponding label, representing the output of S in scope. For the learning, we then formally assume an underlying joint probability distribution $P_{\mathcal{X}\mathcal{Y}}$ on $\mathcal{X} \times \mathcal{Y}$. The learning algorithm receives samples $\{(x_1, y_1), \ldots, (x_l, y_l)\}$ from $P_{\mathcal{X}\mathcal{Y}}$, and from these samples it estimates an approximation $\widetilde{P}_{\mathcal{X}\mathcal{Y}}$ of $P_{\mathcal{X}\mathcal{Y}}$, as captured by some discrepancy measure $L(y, \tilde{y}|x)$, or a function $f : \mathcal{X} \to \mathcal{Y}$ that well characterizes $P_{\mathcal{X}\mathcal{Y}}$, e.g. in that $f(x)$ approximates $\mathrm{argmax}_y P_{\mathcal{Y}|\mathcal{X}}(y|x)$. There are many various representations in which the latter two forms can be expressed, ranging from parametrized distributions, logical rules, and decision trees to neural networks.

Similarly, there are many ways of representing the learning problem with the respective $\mathcal{X}$ and $\mathcal{Y}$, varying across the wide range of domains where machine learning is being applied, ranging from computer vision to natural language processing. While the encoding of $\mathcal{Y}$ typically follows directly from the task definition, where $\mathcal{Y}$ commonly represents the set of discrete target classes or values of the continuous target signal, the optimal representation of the input states $\mathcal{X}$ has traditionally been a research subject on its own. Similarly to the representation of the hypothesis space $\mathcal{M}$, the representation choice for $\mathcal{X}$ is highly important, since it necessarily introduces an inductive bias into the learning process.

1 Naturally, such S include the living (learning) systems themselves which, despite arguably driven by the same laws of physics as their inanimate environments, do not (yet) succumb themselves easily to these mathematical forms within the standard realm of scientific reductionism [1].

2 This is not to say that machine learning falls outside of the scientific approach. On the contrary, it can be seen as an *automation* of the *scientific method* itself. It starts by collecting all the observable data $\mathcal{D} = \mathcal{D}_{\mathtt{train}} \cup \mathcal{D}_{\mathtt{test}}$ about the system S in scope. Consequently, one chooses a hypotheses space $\mathcal{M}$ in which to search for the best mathematical explanation of the observed phenomenon generating $\mathcal{D}_{\mathtt{train}}$. Importantly, predictions for left-out states of S, generating $\mathcal{D}_{\mathtt{test}}$, are then made, and the possible discrepancy between the predictions and test experiment data $\mathcal{D}_{\mathtt{test}}$ are consequently assessed to accept or refute the assumed hypothesis, more commonly referred to as a "model" $M \in \mathcal{M}$.

3 also known as "*training*" for the respective subpart $\mathcal{D}_{\mathtt{train}}$ which drives the adaptation process itself.

While researchers traditionally argued for various levels of abstraction[4] , and the corresponding amount of inductive bias, to be used in encoding of $\mathcal{X}$, the majority agreed upon its mathematical form. Here, similarly to the typical form studied in the, more traditional, system modeling for control theory [3], where the systems S in scope naturally compose of fixed-size input $\mathbf{x}$ and output $\mathbf{y}$ *vectors*, the common trait of machine learning techniques is that the sample descriptions $x_i \in \mathcal{X}$ are typically restricted to value-tuples $X = X_1 \times X_2 \times \ldots \times X_n$, where $n \in \mathbb{N}$ and X_j contains real numbers or categorical values. The sets X_j here are domains of n so-called *features* that are typically a-priori crafted by researchers in each specific domain, and used to commonly describe each considered sample $\mathbf{x}_i$ in the learning task. This process has been commonly referred to as *feature engineering*.

1.1 DEEP LEARNING

With the many different approaches to the input data $\mathcal{X}$ and model $\mathcal{M}$ representations being prevalent for different tasks in different domains, a great goal of machine learning became to render the whole process fully automated. Commonly, this is interpreted as involving as little inductive bias and human intervention as possible, which has been emphasized through the (recent) trend of *deep learning*, where generic architectures of large layered models are trained from "raw" data in various domains [4]. The core idea of deep learning is that the necessary higher-level abstractions of sample representations $\mathbf{x}$ are automatically induced in a hierarchical fashion by the layered, deep (neural) models, as opposed to the manual feature extraction by domain experts used in "shallow" models. Having such a generic, end-to-end architecture for variety of systems based on data representations with minimal preprocessing provides a considerable advantage, as has since been demonstrated with remarkable success in various tasks, ranging from speech recognition and computer vision to game playing [5].

Deep learning models generally compose of multiple layers of nonlinear processing units, with higher-level distributed representations $\mathbf{x}^k$ being successively extracted from the raw input data $\mathbf{x}$ in each layer k. These levels of abstraction are typically presented through hidden layers of large *neural networks*[5] – computational models biologically inspired by two types of cells in the primary visual cortex [6]. The concept of "deepness" then refers to the number of learnable transformations (layers) an input representation $\mathbf{x}$ undertakes before it reaches the output $\mathbf{y}$, where neural networks with more than two such layers are generally considered as "deep" by most researchers [7]. Since the resurgence of the neural networks started by the rapid success in decreasing accuracy on speech recognition tasks in 2010 [8], many variants of these cascading models have been rediscovered or newly introduced.

Perhaps the most successful variant have been Convolutional Neural Networks (CNNs) [9][6], with Neocognitron [10] as their first instance. Distinguishing features introduced in CNNs are the use of *shared weights* in convolutional layers, and the idea of *pooling*, first proposed in the Cresceptron architecture [11]. The shared weights induced by application of convolutional filters introduce *equivariance* w.r.t. the respective transformation of the filter, while incorporating the aggregation function (e.g. max or avg) on top via pooling extends it further into the transformation *invariance*. This technique has proved extremely useful across various tasks, particularly in computer vision, where it provides CNNs with robustness w.r.t. translation in images – an ability unprecedented in the previous machine learning models, enabling CNNs to achieve superior state-of-the-art (SotA) results without the, previously common, deformation-invariant feature extraction and preprocessing.

4 This has been the traditional subject of dispute between the sub-symbolic and symbolic streams of AI research, such as the most recent Montreal AI Debate between Yoshua Bengio and Gary Marcus [2]. The subject of this thesis then aims for the compromise zone between the two notional extremes.

5 but, interestingly, may also be thought of as complicated propositional formulae re-using many sub-formulae [4].

6 We put emphasis here on CNNs as they are discussed throughout the thesis more closely for their resemblance to the proposed methods.

PROBLEMS Although often presented as fully automatic and generic, successful training of deep neural models typically still involves the choice of architecture and related meta-parameters, which requires considerable expert knowledge or compute power. Moreover this knowledge has very limited interpretation w.r.t. the learning (data) domain, since neural networks operate as black-box models, with most confirmations on their learning being done empirically rather than analytically.

Despite the remarkable success of deep neural learning on various, mainly low-level perception focused, tasks, it is important to point out that the approaches typically still lack most of the basic capabilities deemed elementary to AI systems, such as integrating background knowledge, capturing relations and structure, reasoning with abstract concepts and logical inference [12]. Also, the black-box nature of classic neural networks renders their integration with other systems, possibly providing such capabilities, rather complicated [13].

1.2 RELATIONAL REPRESENTATIONS

Perhaps the main limitation of deep learning, which we target in this thesis, is the attribute-value-tuple $X = X_1 \times X_2 \times \ldots \times X_n$ representation of samples $x_i \in \mathcal{X}$, assumed to be identically and independently drawn (i.i.d.) from the P_{XY}. Despite its long tradition and adoption by majority of (statistical) machine learning approaches, this representation of $\mathcal{X}$ is a source of considerable limitations in various learning scenarios, where data samples x_i are naturally structured, and do not succumb themselves easily to the precanned form of numeric vectors or tensors $\mathbf{x}$.[7]

In many real world domains we are interested in modeling, the learning samples (x_i, y_i) are not independently drawn nor are they of a fixed size n, but rather exhibit internal external relational structure [14]. In practice, the real-world data are not stored as numeric tensors, but are present in interlinked structures of the internet, or spread across multiple tables of relational databases, where different samples consist of different types of entities, with each entity being characterized with a different set of attributes. This notion covers biological, social, or computer networks, and generally all domains exhibiting some topology, such as knowledge bases and graphs. Example of such data are abundant, including databases of organic molecules, social networks, protein-protein networks, gene regulatory networks, engineering designs etc., with tasks such as molecule toxicity modeling, social network analysis, protein function prediction and others.

Historically, the traditional approach towards learning from such structured data was to divide the problem into two steps of (i) data preprocessing, turning the structures into the requested feature vectors $\mathbf{x}$, and (ii) then applying the standard machinery of learning models based on this common representation. Such an approach is often referred to as *propositionalization* [15], which is a technique to generate a number of predefined relational features (e.g. subgraphs), and to let these features act as attributes for the standard attribute-value learners, such as the neural networks.

However, there is a fundamental issue with such an approach, since transforming the structures into numeric vectors (tensors) necessarily introduces bias, unrelated to the learning problem, by dropping out parts of the relational input information.[8] After all, if a structure, such as a graph, could be transformed into a fixed-size vector $\mathbf{x}$ without loss of information, it would trivially solve the graph isomorphism problem [17]. Since there is no definite way to turn generic structures into the standard attribute-value representation without the undesirable loss of information, learning representations $\mathbf{x}^k$ from relational data is a fundamental issue for standard neural networks. While there are neural architectures designed to learn from sequences (e.g. LSTMs) or rectangular grids (e.g. CNNs), it is difficult or impossible to use these classic neural architectures with data which have more complicated structure of $\mathcal{X}$, such as the graphs or, worse for the conventional neural

7 However we do note that it was the vectorized representation that stood behind much of the success of neural networks, enabling for the highly efficient parallel processing, allowing neural networks to scale to unprecedented dataset sizes, in turn surpassing accuracy of virtually every other machine learning model across the variety of domains.

8 Consequently, neural models utilizing the aforementioned scheme, such as CILP [16], are not, despite their popularity, truly relational learners, as they learn from the propositional, rather than relational, representation.

networks, if the input data are nodes in a large interconnected graph or if the samples x_i reside in a relational database.

1.2.1 *Relational Learning*

Learning with data samples that are rather variously structured or being part of a bigger structure themselves (and thus interdependent) is generally covered by the area of *Relational Machine Learning* [18]. Relational machine learning has roots in earlier research on Inductive Logic Programming (ILP) [19], which has traditionally been the major opposition to the trend of statistical machine learning from the attribute-value vectors $X = X_1 \times X_2 \times \ldots \times X_n$. ILP typically focuses on learning logical rules from relational data, for which it uses the relational[9] (predicate) language of first-order logic (FOL) [20]. In ILP, FOL is used as a common formalism for both the input examples $x_i \in \mathcal{X}$ and hypotheses, i.e. the models $M \in \mathcal{M}$, but also *background knowledge* B on the given domain (i.e., system S), which can be elegantly incorporated to help in learning of more complex problems. Apart from the ability to learn from relational data and seamless integration with the background knowledge, the main advantage of ILP is the transparency to humans. This means that the resulting models M are typically interpretable, and a syntactic and semantic language bias can be selected prior to learning so as to restrict the form of hypothesis and their behavior.

While substantially more expressive in representation, ILP itself is not well suited for dealing with noise and uncertainty. The uncertainty in relational learning naturally arises from the data on many levels, from uncertainty about the attributes of an object and its type(s) to uncertainty on membership within a relationship and the overall numbers of objects and relations in scope. To tackle the issue, many methods arose to merge the expressiveness of mathematical logic, adopted from ILP, and probabilistic modeling under the notion of *Statistical Relational Learning* (SRL) [14], which covers learning of models from complex data that exhibit both uncertainty, within some probabilistic framework, and a rich relational structure, captured by the (subset of) FOL formalism. Particularly, SRL has extended ILP by techniques inspired in the non-logical learning world, such as kernel-based methods, graphical models, or by special operators for representations equivalent to certain strict subclasses of FOL (e.g. graphs, description logics, etc.).

PROBLEMS While there have been many interesting concepts proposed within the SRL community, the developed systems commonly lack the efficiency, robustness and deep representation learning abilities of neural networks. Moreover, apart from the increased expressiveness of FOL, they typically also inherit the extreme computational complexity of ILP, and they are only rarely selected by practitioners to target real life problems.

1.3 PROBLEM STATEMENT

Regardless of its difficulty, the problem to be solved can be put very simply - *integrate deep and relational learning* to enable neural networks learn directly from structured representations as complex as the expressiveness of relational logic. This includes learning directly from trees and arbitrary graphs to relational databases (hypergraphs) and rich knowledge bases. Additionally, it should allow neural networks to elegantly exploit symbolic background knowledge in the form of relational rules and logic theories, providing interpretable inference in turn. The integration should be tight, systematical, and should naturally cover both the worlds, i.e. the classic deep learning and classic ILP, as special cases.

9 We define relational and first-order logic more formally later in Section 3.1.

1.3.1 *Background*

Noticeably, given the principal differences of the two approaches to learning, such an integration may seem as a rather implausible patchwork. Indeed, historically the two encompassing streams of symbolic and sub-symbolic stances to AI have evolved in a largely separated manner, with each camp focusing on selected narrow problems of their own. Originally, researchers favored the discrete mathematical logic-based approaches[10] towards AI, targeting problems ranging from knowledge representation, rule learning and planning to automated theorem proving. Recently however, the field got completely dominated by the sub-symbolic techniques with distributed continuous representations, particularly the neural networks, originally targeting statistical machine learning problems such as classification, density estimation and generative learning.

However, as the rapid success of deep learning on large perceptual datasets slowly reaches beyond tangible limits, researchers start to explore increasingly ambitious goals for neural networks to strive for, with thoroughly tuned neural architectures continuously taking over new domains and tasks. Following upon this trend, we have seen an increasing number of neural architectures designed to model some of the structured mechanisms that were originally considered of a rather symbolic nature – from language modeling [22] and extracting semantics from images [7], to game playing [5] and even theorem proving [23].

While it might seem that large neural networks are on their way to solve every AI problem at hand, they still fundamentally lag behind the logic-based symbolic methods in terms of expressiveness and transparency (Section 1.1). Together, however, these two streams of competing approaches cover absolute majority of the AI landscape, and attempts for their integration in an efficient manner have become of great interest to researchers targeting highly general problems. The subject of this thesis then falls into this exciting area of AI research.

1.4 TOWARDS DEEP RELATIONAL LEARNING

It has been recently proposed by several authors that incorporating relational logic learning and reasoning capabilities into neural networks is crucial to achieve more powerful AI systems [2, 24, 25]. Indeed, we see a rising interest in enriching deep learning models with certain facets of symbolic AI, ranging from logical entailment [26], rule learning [27], and solving combinatorial problems [28–31], to proposing differentiable versions of the whole Turing machine [32, 33]. However, similarly to the Turing-completeness of recurrent neural networks, the expressiveness of these advanced neural architectures is not easily translatable into actual learning performance [34], and their optimization tends to be often prohibitively difficult [35].

There has also been a long stream of research in Neural-Symbolic Integration (NSI) [12, 36], traditionally focused on emulating logic reasoning within neural networks [37–40]. The efforts eventually evolved from propositional [37, 41] into full first order logic settings, mapping logic constructs and semantics into respective tensor spaces and optimization constraints [42–44]. Typically, the neural-symbolic works focused on providing various perspectives into the correspondence between symbolic and sub-symbolic representations and computing, targeting mostly theoretical expressiveness rather than practical learning applications, which is why they have been mostly marginalized by the mainstream deep learning research.

From the data representation perspective, there has been a continuous effort of applying neural network learning to increasingly complex relational data [45–49]. While the relational learning has been traditionally dominated by the approaches rooted in relational logic [19] and its probabilistic extensions [50–52], the neural networks offer highly efficient latent representation learning, which is beyond capabilities of the symbolic systems. Neural networks on the other hand have tradition-

10 Interestingly, even the very first proposal of a computational *neuron* (perceptron) by McCulloch and Pitts [21] was set to emulate *logic* gates.

ally been based on fixed tensor representations, which cannot explicitly capture the unbounded, dynamic and irregular nature of the relational data and knowledge.

To target this issue, a new approach applying the end-to-end learning paradigm to the raw relational structures, i.e. skipping the preprocessing (propositionalization) step of turning the data into fixed tensor representations, have emerged with dynamically structured neural models. The idea actually dates back to recursively transformed reduced descriptions [53], upon which auto-encoders with distributed representations (embeddings) and many other works have been building [54, 55]. However, this method has been limited to certain recursive structures only, and the preservation of information from deeper structures has been disputable [56].

Most recently, the principle has been generalized from recursive tree structures to arbitrary graphs in the form of Graph Neural Networks (GNNs) [57, 58]. GNN models can be viewed as a continuous, differentiable version of the famous Weisfeiler-Lehman (WL) label propagation algorithm used for graph isomorphism refutation checking [59]. In GNNs, however, instead of discrete labels, a continuous node representation (embedding) is being successively propagated into nodes' neighborhoods, and vice versa for the corresponding gradient updates[11].

These recursive [60] and Graph Neural Networks [57] introduced a highly successful paradigm shift in computation by moving away from fixed neural architectures to dynamically constructed computation graphs, directly following the structural bias presented by the relational input examples x_i. As opposed to the discussed recurrent [32] and "tensorization" [36] neural-symbolic approaches, this enabled to exploit the structural properties of the data more efficiently, as they are simply directly coded into the very structure of the model.

Most recently, this paradigm has become highly popular and achieved remarkable success in a wide range of tasks [58]. Nevertheless, there are still considerable limitations to this class of models, stemming from the limited expressiveness of the WL test, which is only based on the immediate neighborhood information gathered in each iteration [61, 62]. Consequently, information about more complex relational substructures and representations cannot be properly extracted.

1.4.1 *Our Approach*

In our approach, we strive for a very direct integration of deep and relational learning via minimal extensions of both. The core idea is to encode the structure and computation of neural networks in the language of relational logic.

To provide some intuition behind the idea, it is beneficial to realize that the original concept of deep learning was not proposed as bound to the neural networks, but rather universally to methods modeling hierarchical composition of useful concepts, that are then reused in different paths during the inference of target variables y from the input samples x [4]. Naturally, even this idea was not really new, as such hierarchical abstractions are rather very common to human thinking and logic reasoning. In fact, *logic* is the very science of deduction of useful concepts from simpler premises in a hierarchical fashion. While constructing a proof in logic, auxiliary lemmas are often created to reduce the complexity of the theory in scope in a similar fashion.

Thus, while the hierarchical levels of abstraction are typically presented by the hidden layers of neural networks, they may also be thought of as *"complicated propositional formulae re-using many sub-formulae"* (quotation by Y. Bengio [4]). This interconnection was also directly followed by, historically popular, Knowledge-Based Neural Networks (KBANN) [37], explicitly demonstrating the correspondence between the hierarchical structure of logic inference and classic feed-forward neural networks. However, such direct correspondence was insurmountably limited to the aforementioned *propositional* setting, and transferring the same principle to the *relational* setting has been a major challenge researchers have been struggling with.

11 The GNN models are further discussed in detail in Section 2.4 as they are highly relevant to the proposed approach.

Addressing the transfer of this direct neural-logic correspondence from the propositional to the relational setting is the main contribution of our work. The main underlying insight is that to address the relational expressiveness in neural networks, we need to step up from the classic neural networks to a generalization of the (graph) convolutional architectures, for which we take inspiration in a *lifted modeling* paradigm known from SRL. Consequently, in the same sense in which propositional logic is lifted to the relational logic, we propose to lift classic neural networks to a more expressive version we call "Lifted Relational Neural Networks" (LRNNs).

1.4.2 *Lifting*

Lifted modeling, also known as *templating*, has recently attracted significant attention in SRL [63, 64][12]. As opposed to standard machine learning methods, lifted models do not specify a particular model architecture, but rather a *template* from which the particular models are being derived as a part of the inference process itself, given the varying context of the relational input data and background knowledge.

For example, the most popular lifted model – a Markov Logic Network (MLN) [51] may express a general template that "friends of smokers tend to be smokers", which then constrains the probabilistic relationships in all sets of vertices corresponding to particular friends-smokers in a given social network[13]. Such lifted templates are typically encoded as weighted FOL formulas. These can be provided by domain experts, which offers a convenient way of guiding the learning process, although the formulas can also be learned from data, e.g. through the ILP methods [19] (Section 1.2.1). To make predictions, an MLN template is then combined with the particular set of facts about specific individuals to define a *ground* Markov network. In the example with smokers, these facts may include instances of people who smoke and of people who are in a friendship relation. This then allows to induce the smoking probabilities of people based on their social relationships, as if modeled by a regular Markov network, yet generalizing over diverse social network structures.

Importantly, this allows the underlying models to respect the inherent *symmetries* in the data, such as the equivalent meaning of the friendship relation across all the different people in the network, by tying the corresponding parameters.[14] This can significantly reduce the number of weights that have to be learned, and allow lifted models to convey a highly compressed representation of input information, since all the equivalent relational patterns are parameterized jointly by the single template, which in turn allows for greater generalization. Additionally, exploiting the symmetries of the ground models themselves can also significantly speedup the inference process[15].

The salient properties of our approach then stem from the utilized strategy of applying lifted modeling to neural networks. While this SRL technique may sound unfamiliar in the context of deep learning, we have already seen a great success of one simple incarnation of this concept in the popular CNNs [65], effectively following the very principle. As discussed in Section 1.1, the CNN filters can be seen as such parameter templates, capturing symmetries in the form of translation equivariance, while inducing weight-sharing in the respective (ground) neural networks, which can be seen as a simple instance of the equivariant reasoning emulated by the lifted models.

However, from the proposed fully relational perspective on lifted modeling, the CNN templates are rather narrowly tuned to the single specific transformation. As opposed to the mere position shifts over rectangular grids in CNNs, our proposed strategy of neural network-lifting enables to cope with the most diverse range of structural transformations, coverable by the expressiveness of

12 Although very similar principles are also being explored under different names in other fields, such as database engines and computer vision.

13 this concept is further explained in more detail in Section 3.4.2.

14 Note the similarity with the aforementioned CNNs (discussed in Section 1.1 and later in Section 2.2)

15 which we also successfully transferred into classic deep learning to significantly speed up (graph) convolutional neural models, as detailed in Chapter 9.

relational logic[16]. This can be seen along the lines of the recent GNN evolution, yet the expressiveness of our framework extends even further [17], as the parameterized patterns we are learning may compose of arbitrary relational structures, as opposed to the mere graph constituting nodes and edges parameterized with the GNNs.

While there is certainly no lack of similar ad-hoc solutions to tweak neural networks towards some facets of the sought after abilities of relational learning, we believe that framing the problem explicitly in relational *logic*, using the lifted modeling paradigm, yields a more general, highly expressive, and well founded learning formalism providing these abilities in a more direct fashion.

1.5 CONTRIBUTIONS

Our main contribution is the introduction of Lifted Relational Neural Networks (LRNNs) – a framework that uses the lifted modeling paradigm (Section 1.4.2) for design of neural architectures with relational learning capabilities. Similarly to the outlined MLNs (Section 1.4.2), the template in LRNNs is represented as a set of weighted relational logic rules which, together with sets of relational data samples, defined by sets of weighted literals, define sets of standard neural networks. Consequently, different neural networks are created for different relational learning samples, yet all these networks share their weights, associated with the rules. These rules can be very generic, e.g. set to encode various convolutional architectures (Section 1.1) and their relational extensions, but they can also contain more specific background domain knowledge, as known from the ILP (Section 1.2.1). The contributions can be generally sorted into the following topics, referencing the respective chapters in the thesis.

DEEP RELATIONAL LEARNING (CHAPTER 5) While there have been other proposals from NSI and SRL on adapting neural networks for relational learning (Section 1.4), a salient property of the LRNNs introduced in Chapter 5, distinguishing it from these previous studies, is that, following the lifted modeling paradigm (Section 1.4.2), we dynamically construct a different ground network for each sample, enabling to flexibly exploit particular relational properties of the differently structured input representations. This can be seen along the lines of the recent evolution in deep learning with popular structured models such as Recursive Neural Tensor Networks [60] and Graph Neural Networks [58]. While these models follow a similar strategy to dynamically reflect the input example structure in the model computation structure, they do so in a much more restricted setting. Particularly, the structure of these models is set to follow the structure of each example *exactly*, lacking the additional expressiveness, flexibility and (possible) knowledge provided by the relational *template*. With the relational logic templates, the computation from the input data is not hardwired, but adaptively constructed in a hierarchical fashion, reminiscent of how theorems are being proved from simpler premises (Section 1.4.1).

LATENT RELATIONAL STRUCTURES (CHAPTER 5) The main advantage of the presented approach is that it can effectively learn representations of latent relational structures, i.e. relational patterns that abstract away from particular instances of objects and relationships (Section 5.2). Many approaches in machine learning rely on finding a latent representation of the objects of interest, e.g. by using probabilistic models, matrix factorization, or neural networks. Neural networks and matrix factorization have proven very effective for learning latent representations, but in their standard form, they cannot be used to find latent representations in the relational settings. Probabilistic models have been used to find latent relational representations, but the scalability of such methods is limited by the fact that they run expensive expectation maximization (EM) algorithms [66]. LRNNs allow to combine the modeling flexibility of the probabilistic models with the effective-

16 which, naturally, includes the basic CNN case, too, as demonstrated later in Section 7.2.2.
17 also, our LRNN framework was proposed earlier than most of the GNN works.

ness of neural network learning to efficiently explore different kinds of latent relational structures. While there have already been several works that combine propositional or first-order logic with neural networks [16, 39, 67], to the best of our knowledge, none of these methods is able to learn representations of latent *non-ground* relational structures[18].

ADVANCED NEURAL ARCHITECTURES (CHAPTER 7) While targeting integration of deep and relational learning, one of the core desired properties for an integrated system is to keep expressiveness of both the worlds as a special case. While much focus has been traditionally devoted to retain the expressiveness of the logic reasoning, considerably less attention was put on keeping the expressiveness of the neural models themselves. Indeed, majority of the integrative approaches are limited to basic fully-connected neural networks (e.g. [67]), or they are simply oblivious of the used neural architecture due to loose integration [68] (e.g. [69]). Consequently, the existing modern advances in deep learning architectures are out of scope of these integrated systems. In contrast to the classic efforts for emulation of complex logic reasoning within simple neural networks [70], the lifted modeling paradigm in LRNNs results in the ability to use *simple relational logic* programs to capture *advanced neural architectures* – in a tightly integrated and exact manner. Particularly, we demonstrate in Chapter 7 that a wide range of existing neural models, ranging from simple MLPs and CNNs to complex contemporary GNNs, can be elegantly and efficiently covered – not only from the perspective of expressiveness but, importantly, from the practical point of view. [19] As the GNN models are currently at the forefront of the deep learning research directed towards structured data, we devote an extra section to compare LRNNs against specialized state-of-the-art deep learning frameworks through the existing GNN models. We show how to easily encode the core GNN idea by specifying the underlying propagation rules in the (weighted) relational logic, extend it into some of the most contemporary GNN architectures and *beyond*.

STRUCTURE LEARNING (CHAPTER 6) Additionally, we also contribute a symbolic structure learning method for LRNNs, allowing to explore non-trivial latent relational learning structures in a fully automated manner (Section 6). Based on an automatic induction of a hierarchy of latent concepts, the method can be seen as inspired by the meta-interpretive learning idea [71] with predicate invention, which was previously studied in the context of crisp ILP, where it however faced severe limitations due to its complexity (Section 1.2.1), and has so far been applied only to small noiseless problems. With the introduced structure learning of LRNNs, we employ a neural variant of the predicate invention to successfully target real SRL benchmarks, as well as difficult artificial problems, adversarially designed against greedy optimization techniques used in classic deep learning, to prove the added value of the latent predicate invention.

SCALABILITY (CHAPTERS 9 AND 10) Apart from introducing the theoretical foundations for the integrative framework in Part iii, we devoted a substantial focus to the practical side of the problem. Virtually all of the methods exploiting the true expressiveness of relational logic struggle with scalability issues, which are fundamentally inherent to the representation formalism [72]. To (partially) address these issues in terms of both runtime and memory consumption, we contribute two principled optimization methods. Firstly, we address the (purely) symbolic part of the framework by introducing a technique incorporating learned domain theories for lossless pruning of the hypothesis search space (Section 10). While this technique significantly speeds up the structure learning of the LRNN templates, it can be directly used to scale up any classic ILP algorithm via pruning of hypotheses that are equivalent to those already considered. Secondly, we address the (purely) neural part by introducing a lossless compression method designed to scale up training of the resulting weight-sharing (dynamic) neural computation graphs (Section 9). Importantly, this

18 What we mean exactly by latent relational structures will be better explained in Section 5.2 where we present several types of latent structures which can be used in our framework.

19 The LRNN framework can be found at `https://github.com/GustikS/NeuraLogic`.

technique is again usable also outside of the LRNN framework, and can be directly used to scale up standard contemporary (convolutional) models, such as the GNNs, providing significant speedups over contemporary frameworks.

PRACTICAL APPLICATIONS (CHAPTERS 11 AND 12) Using the basic framework in Part iii, we demonstrate that with merely very simple templates, LRNNs are already able to achieve SotA results on classic SRL (and GNN) benchmarks, mainly in molecule classification. Within this domain, we also showcase the use of LRNNs as a practical differentiable programming language, enabling for elegant encoding of GNNs and their extensions, which we also demonstrate on a practical example of learning with molecular ring representations (Section 7.5.1).

In the applications of the framework in Part v, we then briefly illustrate some other use cases of the framework by designing various relational templates for specification of advanced learning constructs, such as learning with latent predictive categories of attributes, entities and relations for knowledge-base completion (Section 11.2). Lastly, we reach out to a completely different domain to show how LRNNs can also be successfully used for learning of relational team embeddings in the task of soccer match outcome prediction (Section 12). Some additional applications are then also mentioned in the appendix Section C.

1.6 THESIS ORGANIZATION

After this introductory Chapter 1, we continue with the background Part ii, where we introduce the preliminaries of deep learning (Chapter 2) and relational logic (Chapter 3) representations and learning paradigms. Additionally, we discuss the evolution of prior work on their integration in Chapter 4. In the subsequent Part iii, we introduce the LRNN framework itself (Chapter 5), its corresponding structure learning algorithm (Chapter 6), and its extended use as a differentiable (logic) programming language (Chapter 7). Subsequently in Part iv, we introduce two optimization techniques of lossless model compression (Chapter 9) and hypothesis space pruning (Chapter 10), improving scalability of the framework. Note that both these techniques are also usable on their own in the respective domains of standard deep learning and ILP, respectively. Finally in Part v, we showcase some applications of the framework, particularly learning of predictive categories for knowledge-base completion (Chapter 11) and relational team embeddings in soccer (Chapter 12), and conclude in Chapter 13. Additionally, we provide some technical and implementation details, and mentions of other applications in the appendix Part vii.

Part II

BACKGROUND

In this part we introduce the necessary background preliminaries of (i) deep learning (Chapter 2) and (ii) relational logic representations (Chapter 3), and also discuss evolution of prior work related to their proposed integration (Chapter 4).

2

DEEP LEARNING

Deep learning is a machine learning approach commonly characterized by the use of multi-layered *neural networks*. Similarly to other (supervised) machine learning models $\mathcal{M}$, a neural network is a mapping $X \underset{\mathcal{W}}{\rightarrow} Y$ from the input sample space (attribute-value) representations X to the output target labels Y, parameterized by $\mathcal{W}$. In the multi-layered networks, this mapping can be seen as a hierarchical composition of (nonlinear) activation functions which, following the pattern of the composition, can be conveniently represented as a *computation graph*.

A computation graph $G = (\mathcal{N}, \mathcal{E}, \mathcal{F})$, composed of nodes $\mathcal{N}$, edges $\mathcal{E}$ and the activation functions $\mathcal{F}$, is a general way to represent nested mathematical functions using the language of graph theory. The graphs are directed with the information flowing from the children nodes to parent nodes, where the children of a node $N \in \mathcal{N}$ are naturally defined as all those nodes M such that $(M, N) \in \mathcal{E}$, and analogically for the parents. The neural networks are then commonly conveyed by the means of differentiable, parameterized, data-flow computation graphs $G = (\mathcal{N}, \mathcal{E}, \mathcal{F}, \mathcal{W})$, associated also with a set of learnable parameters $\mathcal{W}$, commonly called *weights*. Here, the data flowing through the directed edges $e \in \mathcal{E}$ are being successively transformed by the differentiable activation functions $f \in \mathcal{F}$ associated with the nodes $N \in \mathcal{N}$, commonly referred to as "*neurons*". As discussed in the introduction, the data are then commonly represented as numeric vectors (or tensors) $\mathbf{x}^k$. The term neural "*layer*" k is then used to refer to a set of neurons $\{N \mid \mathsf{depth}_G(N) = k\}$ residing at the same depth k in G[1]. An input layer $k = 0$ is then commonly used to represent the feature values $\mathbf{x}$ of the input data samples $(\mathbf{x}_i, \mathbf{y}_i)$ themselves. An output layer $k = \mathsf{depth}(G)$ then corresponds to the target values $\mathbf{y}$. A "deep" neural network is a graph with multiple layers in between, i.e. with $\mathsf{depth}(G) > 3$.

By adapting the weights $w \in \mathcal{W}$, commonly associated with the edges $\mathcal{E} \rightarrow \mathcal{W}$, the model $M : X \underset{\mathcal{W}}{\rightarrow} Y$ can be trained to approximate some target function $t : X \rightarrow Y$, representing the original (deterministic) system S. This is done, as usual, via minimization of some given cost function $(\mathcal{W}; \mathcal{D}_{\mathtt{train}}, M) \rightarrow \mathbb{R}$ capturing the discrepancy between M and t over the set of training data samples $(x_i, t(x_i)) \in \mathcal{D}_{\mathtt{train}}$. Owing to the differentiability of the used activation functions $f \in \mathcal{F}$, the parameters $w \in \mathcal{W}$ of a graph G can be effectively adapted by gradient-descent routines, which is a distinguishing feature of all successful deep learning architectures.

DYNAMIC COMPUTATION GRAPHS In standard neural models, the structure of the computation graph G is static, and only the values $\mathbf{x_i} = (x_i^1, \ldots, x_i^m)$ forming input to the leave nodes $\{N_1, \ldots, N_m\} \subset \mathcal{N}$ are used to encode particularities of individual learning samples $\mathbf{x_i} \in \mathcal{D}$. The input nodes are then associated with identity functions $f^j(x_i) = x_i^j$. In contrast, many of the advanced relational neural models we assume in this thesis are based on dynamic computation graphs, mapping each x_i onto a *new* G_i to exploit particular structural properties of each input sample. Consequently, the leave nodes in these dynamic G_i's are associated with constant functions $f_i^j = x_i^j$, outputting the associated input sample values (if any). This enables to train neural models directly from structured data such as trees, graphs and databases.

1 This notion is somewhat complicated outside the common directed acyclic computation graphs, where the recurrent connections are normally ignored for the sake of the notion of model depth.

DIFFERENTIABLE PROGRAMMING Due to the increasingly complex nature of the computation graphs G and the operations $\mathcal{F}$ utilized in their nodes $\mathcal{N}$, the field has been recently also referred to as *differentiable programming*[2], reflecting the indirectness of the computation graph encoding. The term *neural architecture* is then often used to refer to common programming *patterns* used in creation of these programs, reflected also in the structure of the resulting computation graphs. In this chapter, we briefly review some of the most common and successful neural architectures used in deep learning [3], which are later referenced throughout the thesis.

2.1 MULTI-LAYER PERCEPTRONS

A multi-layered perceptron (MLP) is the original and most common neural architecture. It encodes a directed feed-forward graph, where the interconnections between nodes in subsequent layers k and $k+1$ follow the "fully-connected" pattern where for all $N^k, N^{k+1} : (N^k, N^{k+1}) \in \mathcal{E}$, i.e. a complete bipartite graph. Moreover, each edge is associated with a unique weight as $\mathcal{E} \xrightarrow{1:1} \mathcal{W}$. Consequently, assuming the common vector form of the input data sample $\mathbf{x}$, the computation graph can be efficiently reduced to a linear series of full (dense) matrix W_k^{k+1} multiplications, each followed by an element-wise application of a non-linear function f^{k+1}, such as the common logistic "sigmoid" (σ), hyperbolic tangent (tanh) or rectified linear unit (ReLU).

The main idea behind MLPs is then in "representation learning" of the input data $\mathbf{x}$, often referred to as *embedding*, where one can think of outputs $\mathbf{x}^k$ of the individual layers k as transformed representations of the input $\mathbf{x}$, each extracting gradually more expressive information w.r.t. the output learning target $\mathbf{y}$.

2.2 CONVOLUTIONAL NETWORKS

A Convolutional Neural Network (CNN) is also a feed-forward architecture, yet not fully connected as the MLP. The interconnection patterns in one or more sub-parts of its computation graph G are characterized by utilizing the particular operations of "*convolution*" (filtering) and "*pooling*", as outlined in the introduction (Section 1.1). Given a vector input $\mathbf{x}$ of size n, the convolutional filter (kernel) will also be represented by a vector $\mathbf{c}$ of some size $c < n$. Scalar products of the filter and all the c-length subsequences of the input vector $\mathbf{x}$ are then successively calculated to produce $n - c + 1$ scalar values. The resulting values are commonly referred to as "feature-maps". The second operation is the pooling, which aggregates values from predefined spatial sub-regions of the input values (feature-maps) into a single output through application of some (non-parameterized) aggregation function, such as the commonly used mean (avg) or maximum (max). The layers of these operations can then be mixed together with the previously introduced layers from MLPs in various combinations.

The main idea behind the convolution operation is the application of the same $\mathbf{c}$-parameterized filter over different sub-regions of the input, inducing *equivariance* w.r.t. the filter $\mathbf{c}$ transformation. This enables to *abstract* away common patterns out of different sub-parts of the input representation. The main idea behind the pooling operation is then to enforce *invariance* w.r.t. translation in the input.

2 Note however that, despite being theoretically Turing-complete (e.g. recurrent neural networks), the learning models themselves are rarely as expressive in practice as standard programming languages used for their creation.

3 We introduce these architectures explicitly, despite being commonly known, as we further work with the introduced notions in detail, especially in Chapter 7.

2.3 RECURSIVE AND RECURRENT NETWORKS

A *Recursive* Neural Network (RNN)[4] is a neural architecture which differs significantly from the previous in that it is based on the *dynamic computation graphs* (Section 2), i.e. the exact form of the computation graph is not given in advance. Instead, the computation graph structure G_i directly follows the structure of each input example x_i, which takes the form of a k—regular *tree*. This enables to learn neural networks directly from differently-structured regular tree examples x_i, as opposed to the fixed-size tensors $\mathbf{x}$ (which can also be seen as graphs with completely regular grid topologies).

The leaf nodes N_j^0 in each input sample tree x_i can be associated with feature vector values (embeddings) $\mathbf{x}_i^j$. Every c leaf nodes $x^j, \dots, x^{j+c}$ with the same parent node N_j^1 in the respective computation tree G_i are consequently combined by a given $\mathbf{c}$-parameterized operation c, such as a $\mathbf{c}$-weighted dot-product, to compute the representation of the parent N_j^1. This combining operation c then continues recursively for all the interior nodes, until the representation for the root node $N^{k=d}$ is computed, which forms the output of the model for x_i. Similarly to the convolution in CNNs, the parameterized combining operation c over the children nodes remains the same over the whole tree [55][5].

The main idea behind recursive networks is that neural learning can be extended towards structured data by generating a dynamic computation graph for each individual example tree. The learning then exploits the convolution principle to discover the underlying *compositionality* of the learning representations in recursive structures.

Additionally, the basic form of the commonly known *Recurrent* Neural Networks [35] can then be seen as a "restriction" of the idea to sequential structures, i.e. linear chains of input nodes[6]. The computation graph G in the form of a linear chain is then successively unfolded along the input sequence to compute the hidden representation for each node N_i based on the previous node's N_{i-1} representation and the current node features $\mathbf{x_i}$ (current input). The main idea behind recurrent networks is that the hidden representation can store a form of *memory* or state of the computation.

2.4 GRAPH NEURAL NETWORKS

Graph Neural Networks (GNN) [73][7] can be seen as a further extension of the CNN principles to completely irregular graph structures $x_i = \{\mathcal{N}_i, \mathcal{E}_i\}$. For that purpose, they dynamically unfold each computational graph G_i from each input structure x_i, similarly to the recursive networks. However, a GNN is a multi-layered feed-forward neural architecture, where the structure of *each layer* k exactly follows the structure of the *whole* input graph x_i. Every node N_{x_i} in each input graph x_i can now be associated with a feature vector (embedding), forming the input layer representation h in the computation graph G_i as $h(N_{G_i})^{(0)} = \mathtt{features}(N_{x_i})$.[8]

For computation of the next layer $k+1$ representations of the nodes in G_i, each node N calculates its own value $h(N)$ by *aggregating* A ("pooling") the values of the nodes $M : (N, M) \in \mathcal{E}_i$

4 Note that the abbreviation is also used for the recurrent neural networks, in this thesis however, we use it solely to refer to *recursive* networks.

5 In some works, this architecture is further extended to use a set of different parameterizations, depending for instance on given types associated with the nodes, such as types of constituents in constituency-based parse trees.

6 We note that modern recurrent architectures use additional computation constructs to store the hidden state, such as the popular LSTM cells, which are more complex and do not directly follow from the input structure.

7 also known as "Graph Convolutional Networks", which slightly differ from the original GNN proposal [57], but share the general principles discussed.

8 Interestingly, however, this is not necessary in general, as the variance in the graph topologies of the individual examples can already provide enough discriminative information on its own.

adjacent in the input graph x_i ("message passing"), transformed by some parametric function C_{W_1} ("convolution"), which is being reused with the same parameterization W_1^k within each layer k as:

$$\tilde{h}(N)^{(k)} = A^{(k)}(\{C^{(k)}_{W_1^k}(h(M)^{(k-1)}) | M : (N, M) \in \mathcal{E}_i\}) \tag{1}$$

The $\tilde{h}^{(k)}(N)$ can be further *combined* through another C_{W_2} with the central node's N representation from the previous layer $k-1$ to obtain the final updated value $h^{(k)}(N)$ for layer k as:

$$h(N)^{(k)} = C^{(k)}_{W_2^k}(h(N)^{(k-1)}, \tilde{h}(N)^{(k)}) \tag{2}$$

Note that in contrast to recursive networks, a different parameterization is typically used at each layer. This general "aggregate and combine" [61] computation scheme covers a wide variety of the popular GNN models, which then reduces to the choice of particular aggregations A and transformations C_W. For instance in GraphSAGE [74], the operations are

$$\tilde{h}(N)^{(k)} = max\{ReLU(W \cdot h^{(k-1)}(M)) | M : (N, M) \in \mathcal{E}_i\}$$

and

$$h(N)^{(k)} = W_f \cdot [(h(N)^{(k-1)}, act^{(k)}(N)]$$

while in the popular Graph Convolutional Networks [75], these can be even merged into a single step as

$$h^{(k)}(N) = ReLU(W^k \cdot avg\{h^{(k-1)}(M) | M : (N, M) \in \mathcal{E}_i \cup \{N\}\})$$

and the same generic principle applies to many other GNN works [61, 76, 77].

GNNs can be directly utilized for both graph-level as well as node-level classification tasks. For output prediction on the level of individual nodes, we simply apply some activation function on top of its last layer representation, e.g. $query(N) = \sigma(h(N)^{(d)})$. For predictions on the level of the whole graph G, all the node representations need to be aggregated by some pooling operation such as $query(G) = \sigma(avg\{h^{(d)}(N) | N \in G\})$.

By following the same pattern at each layer k, the computation will produce increasingly more aggregated representations, since at layer k each node N effectively aggregates representations from its "k-hops" neighborhood. Intuitively, the GNN inference can thus be seen as a continuous version of the popular Weisfeiler-Lehman algorithm [59] for calculating graph fingerprints used for refutation checking in graph isomorphism testing.

A large number of different variants of the original GNNs [57] have been proposed, recently achieving state-of-the-art empirical performance in many tasks [58, 78]. In essence, each introduced GNN variant came up with a certain combination of common activation and aggregation functions, and/or proposed extending the architecture with additional connections [76] or layers borrowed from other neural architectures [79, 80], nevertheless they all share the same introduced idea of successive aggregation of node representations. For a general overview, we refer to [58, 78].

2.4.1 *Spectral GNNs*

Here we discussed "spatially" represented graphs and operations. However, some (older) GNN approaches represent the graphs and the convolution operation in spectral, Fourier-domain [58]. There the update operation is typically conveyed in the matrix form as

$$H^{(k)} = f(\hat{A} \times H^{(k-1)} \times W^{k-1})$$

where $\hat{A}$ is an altered[9] adjacency matrix of the graph, encoding the respective neighborhoods, $H^{(k)}$ contains the successive hidden node representations at layer k, and W^k are the learnable parameters at each layer. However we note that, not considering the specific normalizations and approximations used, these again follow the same "aggregate and combine" principles, and can be rewritten accordingly [61]. While theoretically substantiated in graph signal processing, spectral GNN models are generally inadvisable as they introduce substantial limitations in terms of efficiency, learning, generality, and flexibility [58], and we do not consider them further in this thesis.

2.4.2 *Knowledge Base Embeddings*

Knowledge Base Embeddings (KBEs) are a set of approaches designed for the task of knowledge base completion (KBC) [81], i.e. predicting existing (missing) edges in large knowledge graphs. Particularly, these methods approach the task through learning of a distributed representation (embedding) for the nodes. In multi-relational graphs, a representation of the edge (relation) can also be added, forming a commonly used triplet representation of $(\texttt{object}, \texttt{relation}, \texttt{subject})$. To predict the probability of a given edge in the knowledge graph, KBEs then choose one of a plethora of functions designed to *combine*[10] the three embeddings from the underlying triplet [81].

2.5 OTHER ARCHITECTURES

Since the modern resurgence of neural networks into deep learning in 2010 [8], many other neural architectures have been rediscovered or newly proposed, benefiting from the substantial increase in the parallel processing power of modern GPUs, and the ever increasing amount of available data. These include the idea of *auto-encoding*, with generative models such as Deep Belief Networks (DBN) [82] trained in a layer-by-layer manner with Restricted Boltzmann Machines (RBM) [83]. Apart from the auto-encoders, there are also more recent generative models such as Generative Adversarial Networks (GANs) [84]. Also the principle of recursion has been very successfully exploited with more advanced versions of the basic recurrent neural architectures (Section 2.3), such as the popular Long-Short-Term-Memory networks (LSTM) [85] and additional extensions to emulate memory in NNs, e.g. the Neural Turing Machines [32]. Driven by the recent success in providing state-of-the-art results for a number of tasks, a lot of attention is also dedicated into engineering various combinations and ensembles of the deep neural architectures to pursue further accuracy increases [7]. The most recently introduced ideas include the mechanism of *attention*, with the, now extremely popular, Transformer architecture [86], which is actually very closely related, as discussed in [87], to the previously described Graph Neural Networks (GNNs) (Section 2.4), which we cover extensively later in Chapter 7.

9 e.g. $\hat{A} = D^{-\frac{1}{2}}(A + I)D^{-\frac{1}{2}}$, where D is the diagonal node-degree matrix and I is an identity matrix, such as in the original Graph Convolutional Networks [75].

10 Note that there is no need for the "aggregate" operation in plain KBEs.

3

RELATIONAL LOGIC

Mathematical logic is the original language of the symbolic AI approaches, which was the dominant paradigm during the first decades in AI [88]. While there are other representation formalisms for structured data, knowledge and processes (e.g. UML, ERM, SQL, RDF, OWL etc.), specific to different application domains, mathematical logic still servers as the lingua franca for studying their expressiveness and relationships [89, 90]. Consequently, almost all the existing practical formalisms for symbolic representations can be seen as derived from FOL.

3.1 LANGUAGE REPRESENTATION

In this thesis we target *relational* logic, which limits the used FOL representation to contain *no function symbols* other than constants.[1] While this might seem limiting, the relational logic formalism itself already covers the widest range of existing practical learning domains with structured data sources, such as the graphs, networks, knowledge-bases, and relational databases [89].

3.1.1 *Syntax*

Syntax specifies the structure, or grammar, of the logic language, which is formed from *formulas*. In the relational logic used in this thesis, formulas are formed from a set of *constants*, a set of *variables*, a set of n-ary *predicates* for $n \in \mathbb{N}$, and the propositional *connectives* $\vee$, $\wedge$ and $\neg$ [20]. Constant symbols represent objects in the domain of interest (e.g. *alice*) and will be written in *lower-case*. Variables range over the objects in the domain and, to prevent confusion, will be written with a *capitalized* first letter (e.g. *Person*).

A *term* is a constant or a variable. An *atom* is an n-ary predicate symbol, for some $n \in \mathbb{N}$, applied to a tuple of n terms (e.g. $friends(X, bob)$). A *ground atom*, also called a *proposition*, is an atom which only has constants as arguments (e.g. $friends(alice, bob)$). A *literal* ϕ is an atom or the negation of an atom; a literal ϕ is called *positive* if it is an atom, and *negative* otherwise. A *clause* α is a universally quantified disjunction of literals $\forall x_1 ... \forall x_n : \phi_1 \vee ... \vee \phi_k$, such that $x_1, .., x_n$ are the only variables occurring in the literals $\phi_1, ..., \phi_k$. Further, we mostly omit explicit quantifiers, but any variables appearing in formulas are implicitly assumed to be universally quantified.

A clause containing exactly one positive literal is called a *definite clause*. A definite clause is sometimes also referred to as a *rule*, and a set of definite clauses is sometimes called a *logic program*. To help interpretability, a rule $h \vee \neg b_1 \vee \cdots \vee \neg b_k$ will usually be written as $h \leftarrow b_1 \wedge \cdots \wedge b_k$, as is common in the context of logic programming [91]. We refer to the literal h as the *head* of the rule and the conjunction $b_1 \wedge \cdots \wedge b_k$ as the *body*. A clause which consists of a single atom is also called a *fact*.

1 At some places, we will state explicitly that we consider only function-free theories when there is a risk of confusion.

3.1.2 *Semantics*

Semantics is an assignment of "meaning" to the, syntactically valid, logical sentences, which forms foundation for the logical entailment.

3.1.2.1 *Propositional*

The *Herbrand base* of a set of first-order formulas $\mathcal{P} = \{\alpha_1, \ldots, \alpha_m\}$ is the set of all ground atoms which can be constructed using the constants and predicates that appear in this set (while respecting the arity of each predicate). A *Herbrand interpretation* of $\mathcal{P}$, also called a *possible world* ω, is a mapping that assigns a truth value to each element from $\mathcal{P}$'s Herbrand base. This can also be seen simply as a set of *ground* atoms (those which are true). We say that a possible world ω *satisfies* a ground atom a, written $\omega \models a$, if $a \in \omega$. The satisfaction relation is then generalized to arbitrary ground formulas through the following cases:

- If a is a ground atom then $\omega \models a$ iff $a \in \omega$.
- If a is a disjunction $a_1 \vee ... \vee a_n$ then $\omega \models a$ iff there is an $i \in \{1, .., n\}$ such that $\omega \models a_i$.
- If a is a conjunction $a_1 \wedge ... \wedge a_n$ then $\omega \models a$ iff $\omega \models a_i$ for every $i \in \{1, .., n\}$.
- If a is a negation $\neg a_0$ then $\omega \models a$ iff $\omega \not\models a_0$.

A set of ground formulas is *satisfiable* if there exists at least one possible world in which all formulas from the set are true; such a possible world is called a *Herbrand model*. Further, given two (sets of) formulas $\mathcal{P}$ and $\mathcal{Q}$, we write $\mathcal{P} \models \mathcal{Q}$ if every model of $\mathcal{P}$ is also a model of $\mathcal{Q}$. Each set of definite clauses has a *unique* Herbrand model that is minimal w.r.t. the subset relation $\subset$, called its *least Herbrand model*.

The least Herbrand model of a finite set of ground definite clauses can be constructed in a finite number of steps using the *immediate-consequence operator* [92]. This immediate consequence operator is a mapping $T_p : \mathcal{I} \to \mathcal{I}$ from Herbrand interpretations to Herbrand interpretations, defined for a set of ground definite clauses $\mathcal{P}$ as $T_p(\omega) = \{h \mid (h \leftarrow b_1 \wedge \cdots \wedge b_k) \in \mathcal{P}, \{b_1, ..., b_k\} \subseteq \omega\}$.

3.1.2.2 *Relational*

Now consider a set of non-ground definite clauses $\mathcal{P}$. A *substitution* θ is a mapping from variables to terms. For a clause α, we write $\alpha\theta$ for the clause $\{\phi\theta \mid \phi \in \alpha\}$, where $\phi\theta$ is obtained by replacing each occurrence in ϕ of a variable v by the corresponding term $\theta(v)$. A *grounding substitution* is then a substitution in which each variable is mapped to a constant. Clearly, if θ is a grounding substitution, then for any literal ϕ it holds that $\phi\theta$ is ground. The *grounding* G of a clause α from $\mathcal{P}$ is the set of ground clauses $G(\alpha) = \{\alpha\theta_1, ..., \alpha\theta_n\}$ where $\theta_1, ..., \theta_n$ is the set of all possible grounding substitutions, each mapping the variables occurring in α to constants appearing in $\mathcal{P}$. Note that if α is already ground, its grounding is a singleton. The grounding of $\mathcal{P}$ is then given by $G(\mathcal{P}) = \bigcup_{\alpha \in \mathcal{P}} G(\alpha)$. The least Herbrand model of $\mathcal{P}$ is then defined as the least Herbrand model of $G(\mathcal{P})$.

3.2 LOGIC PROGRAMMING

Logic programming is a declarative programming paradigm for computation with logic programs $\mathcal{P} = \{\alpha_1, \ldots, \alpha_m\}$, which are used to encode data and knowledge about a given domain. Syntactically, the rules $h \leftarrow b_1 \wedge \cdots \wedge b_k$ in the program $\mathcal{P}$ are commonly written as

```
1 h :- b1 ,..., bk.
```

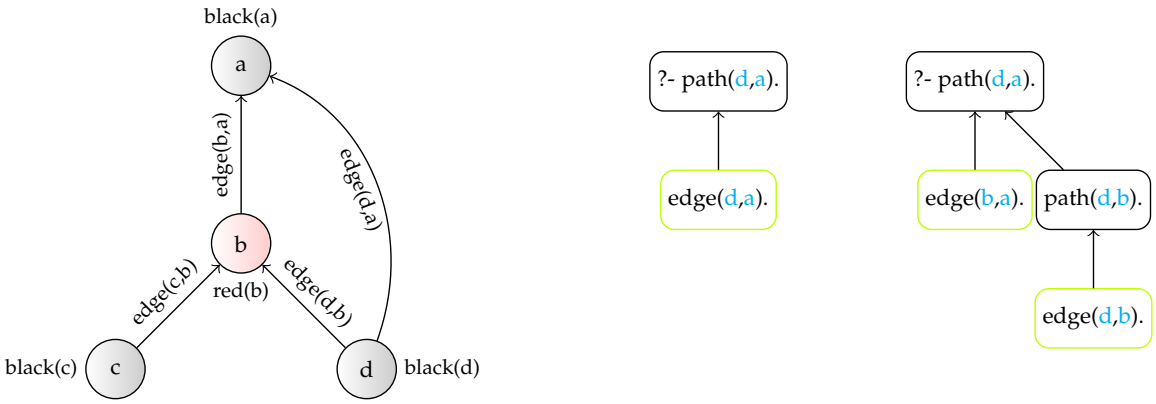

Figure 1: An example of a graph structure encoded in relational logic (left), with two possible proof trees of the query $\texttt{path}(\texttt{d}, \texttt{a})$ derived from it (right).

where each comma "," stands for conjunction, and ": –" replaces the logical implication, which now reads right-to-left. Recall that facts are definite clauses consisting of a single atom, i.e. rules with no body. Note that such (ground) facts may be conveniently used to represent structured data, such as, but *not limited* to, various graphs[2].

Example 1 *For graph structured data, we can simply define a binary predicate* $\texttt{edge}/2$ *with a set of atoms* $\texttt{edge}(X, Y)$ *for all adjacent nodes* X, Y *in the graph, while also retaining the orientation of each edge (given by the order of the terms). Additionally, we may also use other propositions to assign attributes to the nodes such as* $\texttt{red}(X)$ *etc.*[3]*. The rules then commonly express generic (conjunctive) patterns to be searched within the data (Section 3.3.1.1). An example of such encoding of graphs within logic is displayed in Figure 1 - left.*

The computation in logic programming is then generally carried out by the means of the logical entailment. This paradigm is particularly expressive with relational programs $\mathcal{P}$ containing (sets of) interconnected *non-ground* clauses, where the entailment needs to be resolved (recursively) by the means of substitution(s) (Section 3.1.2.2), enabling to compose general and reusable programming patterns to target structured data problems.

Example 2 *For example with the graph structured data (Example 1), we can define (recursive) patterns in* $\mathcal{P}$ *such as*

```
1 path(X,Y) :- edge(X,Y).
2 path(X,Y) :- edge(X,Z), path(Z,Y).
```

which then automatically binds to a (possibly) multitude of substructures in the graph(s) via different substitutions $\mathcal{P}\theta$ *for the variables* $\{X, Y, Z\}$ *upon execution of* $\mathcal{P}$ *(Figure 1).*

Particularly, to target the assumed relational logic setting (Section 3), we consider the language of *Datalog* [93] – a restricted *function-free* subset of Prolog [94]. Datalog is a domain specific language used in advanced deductive database engines [89]. In contrast with Prolog, Datalog is a truly *declarative* language, where the order of clauses and their literals does not influence execution, and it is also guaranteed to terminate. Separate efficient execution engines can then be used for computing the entailment [95].

While restricted, it is, importantly, still a relational language, and one can thus use *logic variables* in the clauses $\alpha \in \mathcal{P}$ to compose abstract relational patterns. We will further extensively use this

2 We later exploit the fact that relational logic is not limited to graphs while generalizing Graph Neural Networks in various ways in Section 7.3.1.

3 See Section 7.1.1.1 for further options stemming from such an encoding.

non-ground expressiveness in Part iii, while extending Datalog towards relational learning and differentiable programming[4].

3.2.1 *Semantics*

As mentioned, the Datalog paradigm allows for separation of the programs $\mathcal{P}$ from the underlying evaluation engine, which leads to different, but equivalent, semantics.

3.2.1.1 *Model-theory*

Here, the semantics of a Datalog program $\mathcal{P}$ is defined by the means of its unique *minimal model* ω. As outlined in Section 3.1.2, this minimal model can be constructed in a finite number of repeated applications of the immediate consequence operator T_p. The operator T_p then expands the current set of true atoms, i.e. the current Herbrand interpretation $\mathcal{I}$, with their immediate consequences as prescribed by the rules in $\mathcal{P}$. It is initially applied to an empty interpretation $\mathcal{I} = \varnothing$, iteratively adding the head atoms of each ground rule instance $\alpha\theta$, the body of which is satisfied by the current interpretation $\mathcal{I}_i$ as

1: $\mathcal{I}_1 = T_p(\varnothing)$

2: $\mathcal{I}_2 = T_p(T_p(\varnothing))$

3: $\mathcal{I}_3 = T_p(T_p(T_p(\varnothing)))$

$\dots$

n: $\mathcal{I}_n = T_p^n(\varnothing)$

The minimal model $\mathcal{I}_n = \omega$ of $\mathcal{P}$ then corresponds to the least *fixed-point* n of T_p, where no more facts are being added to $\mathcal{I}_{i=n}$. For instance, following up on the Example 2 (Figure 1), such $\mathcal{I}_{i=2}$ model will contain an atom $\mathtt{path}(.,.)$ for *all* the paths in the graph (with length 1 and 2).

This simple *bottom-up* method is called naive evaluation, but with some additional optimizations it is actually being used in practice. Likewise, we follow this approach, with some optimizations, it in the evaluation part of the proposed LRNN framework (Chapter 5) and the associated methods in Chapters 6 and 10.

3.2.1.2 *Proof-theory*

Similarly to querying a standard (non-deductive) database with SQL, in logic programming one may also provide a *query* atom q to drive the evaluation engine towards a logical *proof* of the specific target q. For instance, following up again on the Example 2, we can ask a query:

```
1 ?- path(d,a).
```

While this could be achieved by computing the minimal model ω of $\mathcal{P}$ in the bottom-up fashion (Section 3.2.1.1) and checking whether $q \subseteq \omega$, if all we need is to find *any* derivation of q from $\mathcal{P}$, that might be very inefficient. Consequently, we commonly employ a *top-down* "proving" strategy, which starts at the query atom q, and searches through the rules in $\mathcal{P}$ for a rule $h \leftarrow b_1, \dots, b_n$ for which there is some θ such that $h\theta = q$. This search then continues recursively for the (possible) body atoms $b_1\theta, \dots, b_n\theta$ of the rule that now need to be derived from $\mathcal{P}$. Ultimately, the atoms to be proved can be found directly as facts in $\mathcal{P}$, forming the leaves of the induced recursive proof-tree

4 This is a distinguishing feature from many other *procedural* differentiable programming languages, such as PyTorch or TensorFlow, which are effectively *propositional* in this sense (Section 8.2).

of q from $\mathcal{P}$, if successful. This procedure is visualized for the two possible derivations of $\mathtt{path}(d, a)$ in Figure 1 - right.

This top-down, backward rule-chaining approach is then commonly used in Prolog and theorem provers. Likewise, it was also used in some earlier versions [96, 97] of the proposed LRNN framework, too. We note that in the supervised learning setting, we do evaluate LRNN programs w.r.t. a target query atom. However, since the LRNN semantics[5] requires evaluation of *all* possible derivations of each such query, we ultimately found it more efficient to employ (an optimized version of) the bottom-up approach from Section 3.2.1.1.

3.3 RELATIONAL LEARNING

Relational machine learning (RML) is a subfield of machine learning targeting learning from samples that are variously structured or being part of a bigger structure themselves [18]. For that, RML commonly relies on the relational logic language as the representation formalism for both the data and models[6], which is rooted in the earlier research on Inductive Logic Programming [19].

3.3.1 *Inductive Logic Programming*

As briefly outlined in the introduction Section 1.2.1, Inductive Logic Programming (ILP) uses (up to first-order) logic representations to target logical concept learning. The goal of ILP learning is to find a hypothesis (a set of rules) $M \in \mathcal{M}$ that *covers* all positive examples $x \in \mathcal{X}^+$ but no negative example $x \in \mathcal{X}^-$, where the notion of "covering" is realized through some form of the logical entailment $\models$ relation. Additionally, one can also take into account a given background theory B, typically also in the form of a set of (definite) clauses. Procedurally, this is typically implemented via some form of *search* through the structured[7] space of all possible hypotheses $\mathcal{M}$.

More formally, an ILP learning from a set of positive $\mathcal{X}^+$ and negative $\mathcal{X}^-$ examples $\mathcal{X}$ (represented also in FOL) can be defined as follows [19]:

Definition 1 *Given a language describing hypotheses $\mathcal{L}_h$, instances $\mathcal{L}_i$, a (optional) background theory B, and the "covering" relation between $\mathcal{L}_h$ and $\mathcal{L}_i$, find a hypothesis $M \in \mathcal{L}_h$ such that $\forall x \in \mathcal{X}^+ : \mathtt{covers}(B, M, x) = \mathtt{true}$ and $\forall x \in \mathcal{X}^- : \mathtt{covers}(B, M, x) = \mathtt{false}$.*

There are different ways to represent learning problems in ILP. While all commonly use definite clause logic as the hypothesis language $\mathcal{L}_h$, they typically differ in the notion of an example w.r.t. the covering relation. This results into different learning settings [102], such as learning from entailment:

Definition 2 *Let M and B be clausal theories and x be a clause. Then we say that M covers x (w.r.t. B) if and only if $M \wedge B \models x$.*

and learning from interpretations:

Definition 3 *Let M be a clausal theory and x a Herbrand interpretation. Then we say that M covers x under interpretations if and only if $x \models M$.*

Note that, apart from inclusion of the background theory B, which is optional, the two settings differ in that the examples in the latter learning from interpretations setting must be fully specified,

5 which will be defined in Section 5.1.1

6 While there are relational learning frameworks based outside of the logical formalism, such as relational gaussian processes [98] or probabilistic relational models [99], the most well-known relational learning systems are based on logic, such as Markov logic networks [51], Bayesian logic programs [100], or Progol [101].

7 commonly structured into a hierarchical lattice w.r.t the subsumption $\preceq_\theta$ relation (Section 3.3.1.1).

i.e. the truth value of every ground atom must be known. However, in most real-life problems the examples indeed are fully given, and so learning from interpretations is typically a fully sufficient learning setting, which we also mostly consider in this thesis[8].

A NOTE ON COMPLEXITY As discussed, ILP is a declarative, interpretable and highly expressive learning paradigm. However, these characteristics come with a major drawback of extreme computational complexity. For instance, the introduced general ILP learning problem is undecidable in general, and the complexity of most of the less general (decidable) versions still falls into the NP-complete (or worse) category. To (partially) mitigate the issue in practice, the common strategy is to dispense with the generality of the used FOL language or completeness (correctness) of hypothesis evaluation, typically by selecting a bias restricting the hypotheses to have certain limited form (as we do in Chapter 6) or by replacing the logical entailment with θ-subsumption.

3.3.1.1 θ-*Subsumption*

Generally, subsumption is a binary ("is-a") relation between concepts used for categorization of object classes from general to specific. In logic, we then commonly use θ-subsumption [103] to categorize clauses into such taxonomies [104]. For that, we often treat the clauses α as sets of their corresponding literals $\{\phi_1, \ldots, \phi_k\}$. We can then define θ-subsumption as follows.

Definition 4 *If α and β are clauses, we say that α* θ-subsumes *β (denoted $\alpha \preceq_\theta \beta$) if and only if there is a substitution θ such that $\alpha\theta \subseteq \beta$.*

As mentioned in the ILP Section 3.3.1, θ-subsumption is commonly used as a replacement for the generally undecidable[9] entailment $\models$ relation. Note that if $\alpha \preceq_\theta \beta$ holds for clauses α and β then $\alpha \models \beta$, but the converse does not hold in general[10].

This notion is also particularly useful in relational *pattern matching* [72], where the pattern[11] α is being searched within the structure β, where the latter is commonly ground, as in the ILP setting of learning from interpretations (Definition 3). Specifically for the graph-structured data (Example 1), the θ-subsumption relation $\alpha\theta \subseteq \beta$ between a non-ground clause (pattern) α and a ground clause β directly corresponds to subgraph homomorphism matching used in graph pattern mining.

Example 3 *Assume the following non-ground pattern (clause) α*

$$\alpha = \mathtt{edge}(X, Y),\ \mathtt{red}(Y),\ \mathtt{edge}(Y, Z)$$

and a ground structure (clause) β, representing the attributed (colored) graph from Example 1

$$\beta = \mathtt{black}(a), \mathtt{red}(b), \mathtt{black}(c), \mathtt{black}(d), \mathtt{edge}(b, a), \mathtt{edge}(c, b), \mathtt{edge}(d, b), \mathtt{edge}(d, a)$$

Then α θ-subsumes β, because for $\theta = \{X/c, Y/b, Z/a\}$ it is true that $\alpha\theta \subseteq \beta$.

A NOTE ON COMPLEXITY Deciding θ-subsumption between two clauses is an NP-complete problem. It is closely related to constraint satisfaction problems (CSP) with finite domains and tabular constraints [105], conjunctive query containment [106] and homomorphism of relational structures. The formulation of θ-subsumption as a constraint satisfaction problem has been exploited in the ILP literature for the development of fast θ-subsumption algorithms [107, 108]. CSP solvers can also be used to check whether two clauses are isomorphic, by using the primal CSP encoding described in [107] together with an *alldifferent* constraint [109] over CSP variables representing logical variables. In practice, this approach to isomorphism checking can be further optimized by pre-computing a

8 particularly in Chapter 10

9 Note however that deciding θ-subsumption is still a difficult, particularly an NP-complete, problem.

10 However, it does hold for non-selfresolving function-free clauses.

11 here we actually use De-Morgan's laws to convert the disjunctive clause into conjunctive pattern while flipping the signs of all the literals (see Section 10.2 for further details).

directed hypergraph variant of Weisfeiler-Lehman (WL) coloring [59] (where terms play the role of hyper-vertices and literals the role of directed hyper-edges) and by enriching the respective clauses by unary literals with predicates representing the labels obtained by the Weisfeiler-Lehman procedure, which helps the CSP solver to reduce its search space.

3.4 STATISTICAL RELATIONAL LEARNING

It can be noted that the described ILP setting is an idealized problem which focuses only on the *search* aspect of learning, and does not take into account the generalization performance of the learned programs on unobserved data [72]. Also, such ILP systems do not cope well with noise, or the possibly inherent uncertainty, in a given learning domain. While there have been modifications proposed to address these issues, such as setting a tolerance for misclassification rate of the learned theories on training data, or to restrict the theories to some syntactically limited class or by their length [101], the means to cope with these principled issues are rather limited in plain ILP.

To tackle these issues in a more principled way, many methods arose to merge the expressiveness of logic and statistical machine learning under the area of *Statistical Relational Learning* (SRL) [64, 110]. SRL methods commonly approach the task by extending the ILP learning setting in two ways. Firstly, they provide numerical annotations W to the clauses, typically encoding some probabilistic information. Secondly, they relax the covering relation into a probabilistic (soft) setting as follows [110]:

Definition 5 *Given an example* x*, a hypothesis* M*, and (optionally) a background theory* B*, the probabilistic* $\mathrm{covers}(B, M, x)$ *relation returns a numeric value, typically reflecting the likelihood* $P(x|M, B)$ *of the example* x *given* M *and* B .

The learning task can then be seen as an optimization search for a hypothesis (model) M maximizing the likelihood of the training data, or a similar objective function, e.g. employing some form of regularization. Different choices of the probabilistic covers relation then lead to different SRL approaches [110], akin to the aforementioned learning settings in ILP (Section 3.3.1).

Following this view, the statistical ML and ILP can be elegantly unified within the SRL framework, since both can be seen as special cases of the same setting. Particularly, using the deterministic corner case of the probabilistic covering relation yields classic ILP, while restricting the logic language to (uniformly structured) propositional clauses, and providing only coverable[12] examples (i.e. $\forall x \in X : P(x) > 0$), yields classic statistical ML. The SRL task can then be commonly divided into 2 subproblems:

1. *Parameter learning* – where only the parameters W of the theory (model) are being learned from data (akin to classic statistical ML, as we do in Chapter 5)
2. *Structure learning* – where also the form of the logical theory M itself is a subject to learning (akin to classic ILP + ML, as we do in Chapter 6)

There is a great variety of SRL methods [14], typically based either on imposing a probabilistic interpretation on logical inference (programming), such as Stochastic Logic Programs [111] and ProbLog [52], or utilizing representations defining probabilistic (graphical) models using a relational (logic) language, with Markov Logic Networks (MLNs) [51] and Bayesian logic programs (BLPs) [50] as the main representatives. We will now detail these two main streams of SRL approaches a bit further to provide background for our proposed LRNN framework (Chapter 5) which borrows ideas from both the paradigms.

12 Note that in SRL we can now also distinguish 2 different notions for a negative example – one that yields a small enough value (akin to statistical ML), and one that is not covered at all (akin to ILP), often also referred to as a "counter-example" [110].

3.4.1 *Probabilistic Logic Programming*

This SRL paradigm can be seen as extending logic programming (Section 3.2) with probabilistic choices [110]. This notion covers works such as Stochastic Logic Programs [111], Prism [112], and Independent Choice Logic (ICL) [113]. Here we detail a bit further the, arguably most popular, example of this paradigm called ProbLog [52], which can be seen as a probabilistic extension of the logic programming language (Section 3.2).

In Problog, facts $\mathtt{F}$ for *probabilistic* predicates may be annotated with a probability value, indicating the degree of belief that any ground instance $\mathtt{F}\theta$ of $\mathtt{F}$ is true. It is further assumed that the $\mathtt{F}\theta$ are marginally independent. The probabilistic facts are then augmented with a logic program defining further predicates. Let us demonstrate the setting on an example (adapted from [52]).

Example 4 *Let us again follow up on the Example 1, displayed in Figure 1, by annotating the facts, corresponding to the edges in the graph, with probabilities as follows:*

$$0.9 :: \mathtt{edge}(c,b). \qquad 0.7 :: \mathtt{edge}(b,a). \qquad 0.6 :: \mathtt{edge}(d,b). \qquad 0.9 :: \mathtt{edge}(d,a).$$

Additionally, we assume the (simplified) definition of the $\mathtt{path}(X,Y)$ *relation from Example 2*

Problog then defines a probability distribution on (ground) proofs, i.e. executions of the logic program (Section 3.2.1.2), simply as the product of the probabilities of the (ground) clauses (here, facts) used in the proof. For instance, for the (only) proof of the goal query $\mathtt{path}(c,a)$*, using the (marginally independent) facts* $\mathtt{edge}(c,b)$ *and* $\mathtt{edge}(b,a)$*, the probability calculation for* $P(\mathtt{path}(c,a))$ *will yield* 0.9×0.7*.*

Note that while the calculation was simple in the case where there is but a single ground proof derivation of a query, the situation is much harder when there are multiple proofs, such as for the derivation of $P(\mathtt{path}(d,a))$ (Figure 1 - right), which consists of the edge $\mathtt{edge}(d,a) \rightsquigarrow 0.9$ itself as well as the path $\mathtt{edge}(d,b) \wedge \mathtt{edge}(b,a) \rightsquigarrow 0.6 \times 0.7 = 0.42$. The complexity follows from the fact that the different proofs are not commonly mutually exclusive, and their probabilities thus cannot be combined easily (e.g. summing the $0.9 + 0.42$ obviously does not yield a probability value). The resulting problem can be reduced to computing the probability of a disjunctive normal form boolean formula $P(\mathtt{edge}(d,a) \vee (\mathtt{edge}(d,b) \wedge \mathtt{edge}(b,a)))$, which is, however, a known "disjoint-sum" NP-hard problem. Additional restrictions thus have been proposed to enforce mutual exclusiveness of the ground proofs and, consequently, improve the inference efficiency [112].

The discussed distribution at the level of individual facts F can be generalized to the possible world semantics (Section 3.1.2), specifying a probability distribution on interpretations, following from the "distribution semantics" introduced in [112].

3.4.2 *Lifted Graphical Models*

The other main stream of SRL approaches are arguably Lifted Graphical Models [63], representing functions assigning probabilities to Herbrand interpretations (Section 3.1.2), with popular instances such as Markov Logic Networks (MLNs) [51] or Bayesian Logic Programs (BLPs) [50].

The lifted models lift representation of probabilistic models the same way that propositional logic is being lifted by FOL, so that rather than reasoning about particular instances, i.e. random variables (e.g., $\mathtt{pixelAt}(2,3)$), we can reason about whole groups of instances, also called parametrized random variables (e.g., $\forall X : \mathtt{pixelAt}(2,X)$). As compared to standard graphical models, this allows for greater generalization, by tying parameters[13] of varying number of objects, and also speedup of inference, by exploiting symmetries of the ground model. Depending on the use of underlying probabilistic model, lifted models can seen as either directed (e.g., Bayesian logic) or undirected (e.g., Markov logic), which can both be albeit generalized by the notion of factor graphs [63].

13 Note the principal similarity of lifted (graphical) models with Convolutional Neural Networks (Section 1.1 and 2.2).

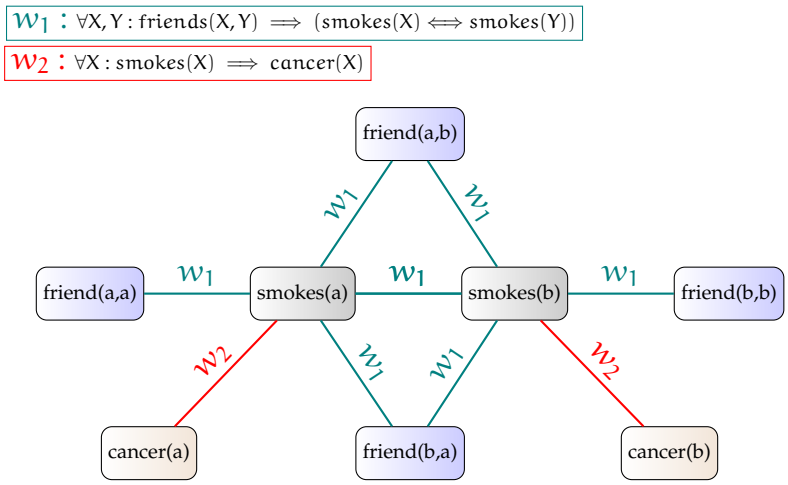

Figure 2: A simple ground MLN from Example 5 unfolded for two constants $\{a, b\}$ (adapted from [51]).

The distinguishing feature of particular lifted models is the choice of representation of random variables and their grouping via shared parametrization, for which object oriented (ERM), structured query (SQL), and FOL languages exist. Here we will again focus on the logic-based formalisms, which are also by far the most prominent, with MLNs as the most important example[14].

An MLN is given by a set of constants, a set of predicate symbols, and a set of FOL clauses ("features") associated with weights, commonly also referred to as "soft constraints" over the set of possible worlds Ω. That is if a possible world ω does not satisfy a logical clause, it becomes less probable, with the associated weights reflecting how strong each constraint is. Let us consider the hackneyed example of a "friend-smokers" MLN (adapted from [51]), encoding the probabilistic relationships between smoking and cancer, as well as the social influence of friends on the smoking habit.

Example 5 *Let us have the following weighted clauses*

$$w_1 : \forall X, Y : \text{friends}(X, Y) \implies (\text{smokes}(X) \iff \text{smokes}(Y))$$
$$w_2 : \quad \forall X : \text{smokes}(X) \quad \implies \text{cancer}(X)$$

Here the clauses encode (i) the intuition that having a friend-smoker increases one's probability of smoking, and (ii) that smokers are more likely to have cancer.

Similarly to Problog, these clauses then provide means for assigning probabilities to possible worlds (Section 3.1.2). Particularly the probability of a possible world ω w.r.t. an MLN is given by

$$P(\Omega = \omega) = \frac{1}{Z} \exp\Big(\sum_i w_i \cdot n_{F_i}(\omega) \Big)$$

where $n_{F_i}(\omega)$ is the number of groundings (substitutions) of F_i present in ω, w_i is the associated weight of the clause F_i, and Z is a normalization constant, also known as partition function, summing the expression over all the possible worlds as $\sum_{\omega \in \Omega} \exp\Big(\sum_i w_i \cdot n_{F_i}(\omega) \Big)$ to form a proper probability distribution over the possible worlds Ω.

An MLN can be also viewed as a *template* for constructing ordinary Markov networks. Together with a Herbrand domain (here a set of constants), an MLN corresponds to an ordinary Markov network, where the nodes (random variables) correspond to all the possible atoms in the respective

14 While there are other systems besides MLNs, such as Bayesian logic programs [50], Probabilistic relational models [99], and relational dependency networks [114], most of these can be emulated within the framework of Markov logic. Similarly, there is also an interesting correspondence of MLNs to the introduced Problog (Section 3.4.1) language, further detailed in [115].

Herbrand base formed from the given predicates and constants (Section 3.1.2.2). Further, for every ground instance $F_i\theta$ of a clause F_i in an MLN there will be an edge between any pair of atoms $\phi_1\theta$, $\phi_2\theta$ that occurs in $F_i\theta$. The random variables (nodes) are then binary, reflecting simply whether the respective atom is present in the Herbrand interpretation or not. For demonstration, a Markov network obtained by grounding the aforementioned friend-smokers MLN for the constants (representing people) $\{a, b\}$ is visualized in Figure 2. Note that for different (Herbrand) domains, different ground Markov networks would be produced from the same MLN (knowledge-based) template.

The correspondence with ordinary Markov networks then makes it possible to use standard inference methods such as Gibbs sampling, and the weights of Markov logic networks can then be learned from data via maximization of some objective function, such as a weighted pseudo-likelihood [51].

4

PRIOR WORK

This background chapter provides a high-level (historical) survey into the main related research areas, with a particular focus on the crossover works between relational learning and neural networks. As the combination of these two categories seems attractive from many viewpoints (Section 1.4), relevant works have arisen from different backgrounds and communities. Particularly, it has been the Statistical Relational Learning (SRL) community, the community of Neural-Symbolic Integration (NSI), and recently also the "main-stream" deep learning (DL) community. Although aspects of the particular methods often overlap across the communities, in this chapter we attempted to sort the works accordingly, in order to provide an insight into the evolution of the diverse body of the prior work combining relational logic representations with neural networks.

We notify in advance that in this chapter we only discuss works that were published *before* the proposed LRNN framework (2015), and a further discussion of the more recent[1], and also more closely related, works will be presented in more detail in Chapter 8, after introduction of the LRNN framework itself, where we can better explain the differences from our approach.

4.1 STATISTICAL RELATIONAL LEARNING

For their explicit logical approach to relational inference, the SRL methods are highly relevant to our proposal. These include the discussed probabilistic logic programming methods (Section 3.4.1), sharing the weighted definite clause representation formalism and the means of performing logic inference, and the lifted (graphical) models, sharing additionally the strategy of using the weighted logic theories as a template for unfolding standard "ground" machine learning models.

As already introduced in Section 3.4.1, Problog [52] is a probabilistic extension of Prolog [94], specifying a probability distribution over all possible non-probabilistic subprograms of the given Problog program. Building on top of the logic programming formalism (Section 3.2), which it also extends with numerical parameters, the LRNN framework is syntactically very similar to Problog. The main difference is in the explicit probabilistic interpretation of the numerical parameters in a Problog program, as opposed to the mere (neural) weights used in LRNNs.

A closely related [115] approach to Problog is that of MLNs [51]. As introduced in Section 3.4.2, an MLN is a first-order knowledge base with a (scalar) weight attached to each formula (clause), and can be viewed as a template for constructing and parameterizing ground Markov network corresponding to its Herbrand interpretation(s). As opposed to classic Markov Networks, this strategy enables MLNs to learn from arbitrarily structured data and to flexibly and modularly incorporate varying domain knowledge [51]. Moreover, from the plain computation point of view, MLNs can also be used as a compact language to specify very large Markov networks by efficiently capturing the (possible) distribution symmetries.

In a similar fashion, Bayesian Logic Programs (BLPs) can be seen as a generalization (lifting) of Bayesian networks [116], designed to overcome limitations of their propositional nature. For that purpose they also use the language of definite clause logic, and establish a mapping between ground atoms and random variables, for which they again define probability distributions over

1 This separation can also be largely understood as before and after the modern "deep learning era".

the FOL interpretations, and between the immediate consequence operator [92] in logic and the dependency relation in the directed graphical model [117]. For the directed nature of the used definite clauses (rules) and the underlying statistical model (Bayes networks), BLPs can be seen as even more closely related to the LRNNs.

Very similarly, Dependency Networks [118] which, in contrast to the Markov or Bayesian networks, can be seen as an approximate representation of the underlying joint distribution with a set of independently learned conditional probability distributions, were also lifted into the relational setting [119].

4.1.1 *Distinctions*

The lifted modeling paradigm from SRL served as the main inspiration for the proposed LRNN framework. The implied characteristics, together with the associated benefits as compared to the underlying ground models, are shared by the LRNN framework, too. The major distinction is that we utilize neural, rather than Markov or Bayesian, networks as the underlying ground model specification. Consequently, LRNNs are *not a probabilistic model*, which can be seen as the most major limitation of LRNNs. Nevertheless, this choice also alleviates the computational complexity associated with normalization of probability values in the SRL models, enabling us to scale to much larger problems.[2] Also, none of these existing lifted graphical models is particularly well suited for learning parameters of latent relational structures, similarly to how latent features are learned in deep learning models, as they typically need to employ expensive expectation maximization algorithms for the purpose.

4.2 NEURAL NETWORKS FOR RELATIONAL DATA

Historically, the most common way to tackle learning from structured, relational data has been propositionalization [15], which proved reasonably effective in practice [72]. Propositionalization is a framework based on aggregating relational features (e.g. subgraphs) out of relational data into standard attribute-value vectors. A number of practical works utilized this preprocessing scheme with classic neural networks, which was sometimes presented as an "integrated" learning system [16, 120, 121].

A more deeply integrated approach towards structured data representations with NNs was based on learning *embeddings*, i.e. distributed fixed-size numerical (vector) representations of the symbolic structures. This idea dates back to Linear Relational Embeddings [122] (due to Paccanaro and Hinton), with follow-up papers proposing different schemas to improve learning [123] and extend the embedding of symbols to their compositions [22]. This particular work then became widely popular as "word2vec" [22]. Extensions to embeddings of structures, such as trees, then date back to recursive auto-associative memories (RAAMs) [124], followed by [124–126], where varying structures were transformed into vectors by training special auto-encoder neural networks [127]. Similar line of work arose also from the idea of reduced descriptions [53] (also due to Hinton), with some more recent representatives [128–131]. The representation of a relation in these works is typically learned as a correlation or similarity measure, parametrized e.g. by a tensor [55], between couples of entities, each of which is already encoded by a vector. This approach later became popular as Recursive Neural Networks (Section 2.3).

In contrast to the logic-based approaches, an issue with standard neural networks is that to bind (substitute) an abstract variable to some particular (ground) entity, and by those means reason about its relations to other entities, the network is required to emulate some sort of memory. A breakthrough in these attempts were Long-Short Term Memory networks (LSTM) [85], which can learn very long causal dependencies through adaptive memory gates, emulated by a recursive

2 The reasoning here is completely analogical to the differences between classic neural networks and graphical models.

circuit of neurons, and truncating the gradient where it does not harm (Section 2.5). This gave rise to an interesting line of work tackling the issue of solving relational, and related combinatorial, problems by introducing differentiable external memory addressing and operations. This paradigm was made famous with the Neural Turing Machines (NTMs) [32] architecture, referring to the Turing-completeness of recurrent neural networks [132], demonstrating effective learning of very long distance relations, although mostly limited to sequential data structures.

From the relational learning perspective, various modifications to adapt classic neural networks have been proposed in order to cope with certain facets of relational representations. As the most simplistic case, multi-instance learning, i.e. learning from (independent) sets of examples, was addressed in [133], where the authors presented a neural model designed to learn from sets of feature-tuples. However, this model also introduced a considerable aggregation bias as it relied completely on an existence of a single tuple signature within a set.

Later, in a more involved work on using NNs for relational learning [45], followed by [134], the authors suggested that an important desired capability of statistical relational learners, that is not met in most of the systems, is to be able to combine selection and aggregation bias, i.e. to search for patterns (selection) and learn how to aggregate statistical (numerical) information out of them (aggregation) at the same time, for which they proposed Relational Neural Networks (RelNNs) [45], with a structure based on a particular relational database schema from which the samples were to be drawn. It was a combination of feed-forward and recurrent networks used to process the sample tuples together with their related tuples. Their focus was thus on learning from the related (multi-instance) tuple-sets, and the particular way they used to deal with the key problem of feeding the possibly unlimited number of tuples into the limited network structure was by feeding them in a sequence. To learn the appropriate aggregation function, they repeatedly fed these sequences into (parts of) the recurrent network, while continuously reshuffling in order to avoid the unwanted order-sensitive bias.

A similarly motivated method was then proposed in the (original) Graph Neural Network model (GNN) [57], a simplified version of which recently became very popular in the deep learning community (Section 2.4). The original GNNs can actually be seen as a generalization of the RelNNs, removing the restriction of the latter posed on the input graphs to be acyclic and rooted in a single node. The two methods were further reviewed and compared in [135] and [56]. An interesting primer on the topic can also be found in the work of [136], introducing a common framework unifying some ideas of coping with the structured data learning. For further background on related works in this category we refer to [137], providing a comprehensive review of relational learning with (knowledge) graphs.

Other, more loosely related, lines of work in the deep learning community include the CNNs [9] and techniques of indirect encoding [138], exploiting patterns and regularities in neural connections to create more compressed representations of large neural networks, for which they can be perceived as a (simplified) instance of the proposed lifting (templating) strategy. However these works are naturally geared towards learning from propositional, rather than relational, data.

4.2.1 *Distinctions*

The distinction to all the methods based on propositionalization [16, 120, 121], or any other type of fixed-form data preprocessing, is clear. In these methods, some information about the original structure is always lost as long as the structure is nontrivial, and the features have to be somehow crafted in advance. A salient aspect of our framework is that it allows learning straight from relational samples, rather than creating intermediate attribute vectors.

The (dynamically) structured embedding-based models [55, 57, 128–130], directly following the structural bias of the input relational samples in the form of symbols, trees and graphs, are very closely related. The main distinctions w.r.t. these can be clarified in terms of expressiveness. Particularly, virtually all of these models can be emulated within the proposed LRNN framework (as

we show later in Chapter 7.2), but not vice versa. The underlying reason is that these models are missing the means for definition of the lifted template, as utilized in LRNNs, which introduces the additional flexibility and expressiveness. Consequently, all of these structure-embedding methods are limited to learning from only certain types of relational data structures.

A distinction from the advanced neural architectures utilizing differentiable memory operations [32] is much more ambiguous, but can be viewed in terms of efficiency. While the Turing completeness of the underlying recurrent NNs can be seen as a strong argument, similarly to the universal approximation theorem [139], it by no means addresses the efficiency of learning [34], which tends to be prohibitively difficult in these [35]. The distinction of our proposal, backed directly by the relational subset of FOL, is in the explicit logical reasoning, which does not need to be approximated through the neural emulation. Thus, with the integrative approach, we do not attempt to reinvent existing capabilities of logical, or combinatorial, reasoning within neural networks, but try to directly provide extensions of both.

4.3 NEURAL-SYMBOLIC INTEGRATION

On the border of the neural network computation and symbolic representation areas, there has been a (rather small) community of Neural-Symbolic Integration (NSI) for learning and reasoning [140], working towards combination of the neural learning and logic reasoning principles. Naturally, this is highly relevant to the proposed topic, and so we introduce the evolution of the NSI domain in a bit more detail in the following (sub-)sections.

4.3.1 *Propositional*

Within the NSI area, early attempts to integrate the logic and connectionist systems have mainly been restricted to propositional logic. These go back all the way to the pioneering work by McCulloch and Pitts [21] from 1943, presenting the fundamental principles of encoding propositional logic statements into neural networks and, from the opposite side, extracting the network's behavior back in the form of propositional expressions. The idea was based on the, now commonly known, fact that logical connectives such as conjunction, disjunction and negation can be easily encoded by binary threshold units with weights. By those means they proved that there is a one-to-one correspondence between such networks and finite state automata, and an extension to the weighted automata was given later [141].

There have been a number of systems following this paradigm of translating logic programs into neural networks developed in the '80s and '90s. The idea was to perform the translation in a way such that the network will settle in a state corresponding to a valid model of the original program $\mathcal{P}$ [142]. The key proposal was to represent logic programs by the means of their associated semantic consequence operator T_p [92] (Section 3.1.2), which was implemented in a network N_p rather than encoded directly by the programs. It was shown that every logic program can be implemented using a 3-layer network of binary thresholds. The function of T_p, encoded in the corresponding network N_p, then maps encodings of Herbrand interpretations ω onto itself $T_p(\omega) \to \omega$, producing connectionist computation of the semantics of $\mathcal{P}$ when applied in an iterative manner (Section 3.2.1.1). This original idea resulted into many works investigating its different possibilities, mainly by replacing the originally used threshold units [142] with differentiable sigmoidal activation functions, which in turn made the network model trainable by standard gradient descent methods.

This approach also covers the popular Knowledge-Based Artificial Neural Network (KBANN) system [67]. KBANN is a hybrid learning system built on top of connectionist learning techniques that maps, in the presented spirit, problem-specific "domain theories", represented by propositional logic programs, into neural networks, and then refines this reformulated knowledge using backpropagation. A closely related approach was presented in Connectionist Inductive Lerarning

and Logic Programming system (CILP) [120, 121], completing the learning and reasoning cycle by providing corresponding algorithms for knowledge extraction back from the network.

Some further works to mention in this category include symmetric neural networks [143], and automated inferencing and connectionist model [144], and a work on a sound approach to symbolic knowledge extraction from trained neural networks [145].

4.3.2 *Intermediate*

A completely different strategy that emerged within NSI was trying to simulate forms of "reflexive reasoning", inspired by the human ability to perform variety of cognitive tasks with extreme efficiency, providing decisions in almost instantaneous time. A prominent approach from this category was presented in the SHRUTI system [146], showing how a connectionist network can encode facts and rules involving n-ary predicates and variables, and how to perform a "reflexive" type of inferences upon those. The relational knowledge base in SHRUTI was encoded by clusters of cells connected into a sort of dependency network. The binding of variables in the model could then be obtained by time-synchronization of activities of neurons and the inference mechanism was encoded by means of rhythmic activity over them. While proposed as a (very non-traditional) first-order reasoner and learner, the phase-encoding mechanism suggested in the work actually restricted the first-order language [147] to contain only constants and "multi-place" relation symbols [148]. This meant that a single rule could not be be freely instantiated in multiple places within the network, considerably limiting the capabilities of SHRUTI, which was later addressed in [149]. Moreover, it did not originally address the learning part of NSI, and only very basic learning capabilities were added later [150].

Another system combining restricted subsets of FOL with neural networks was ROBIN [151], a structured neural network model capable of partial inference requiring variable bindings and rule application. It was somewhat similar to SHRUTI, but used signatures instead of phase coding, and was also unable to truly capture structured objects (only constants).

A work presenting parallel unification algorithm and an automated reasoning system for horn clauses, implemented in a feed-forward neural network, was then presented in the Connectionist Horn Clause Logic (CHCL) system [152]. However, this system also suffered from similar restrictions on copying formulas (or parts thereof), again limiting its ability to generate new terms.

The introduced CILP [120, 121] system was also followed by an extension to CILP++[16], a neural learning system utilizing the FOL representation. The representation was however converted into a propositional form through a selected (bottom-clause) propositionalization technique [15], preventing CILP++ from actually capturing relational information in data structures, and so the learning and reasoning part of the system was still effectively propositional. While the number of existing, mostly propositional, works attempting to reach some level of neural-symbolic integration has been vast, we refer to [153] for a more comprehensive survey over the NSI field.

4.3.3 *First order*

NSI works extending the propositional models to the relational, or full first-order, logic settings have been comparably scarce. As opposed to the propositional setting, there is the major obstacle of encoding the possibly infinite variations of the FOL interpretations into the necessarily finite neural network model [148]. A work following from [142] presented an attempt for expansion of connectionist approach to FOL programming in [154] by approximating the FOL program by its finite ground (propositional) subprograms, encoded in the same manner as [142], and showing that for certain programs this leads to arbitrarily accurate encodings.

A more direct approach to capture FOL with NNs was introduced in [155], which later became very prominent in the NSI community. The work was again based on the idea that logic

programs can be represented by their associated single-step immediate consequence operators T_P (Section 3.1.2). However, while in the propositional case the variables of the finite interpretation were possible to be simulated by the (finitely many) nodes of the network directly, for the FOL case, infinite interpretations had to be treated. The core solution to deal with this problem was introduced in [155], where the authors proposed to encode the (possibly infinite) interpretations of a logic program $\mathcal{P}$ by real numbers $\mathbb{R}$, in such a way so as to exploit the natural ability of neural networks for processing these. That is the operator T_P was mapped to a function $f : \mathbb{R} \mapsto \mathbb{R}$ defined on the (subset of) real numbers encoding interpretations of the program $\mathcal{P}$, which could in turn, under certain conditions, be approximated by a (sufficiently large) feed-forward network with sigmoidal activation functions. This idea was then exploited and broadened into increasingly expressive classes of programs and logic formalisms in a number of subsequent works [12, 156, 157].

A very different work, more closely related to our proposal, designed a technique forming a class of Radial Basis Function networks directly from a flat relational rule set in a system called First-Order Neural Networks (FONN) [39], which was further exploited in [158, 159]. The relational capabilities in this model operated very naturally by binding the network structure to the relational samples through substitution (Section 3.1.2.2), similarly to the proposed LRNNs. However the inputs in FONN were directly aggregated to produce a shallow, single-layer network. Also FONN did not deal with aggregative type of reasoning as the patterns grounded from the used formulae were all existentially quantified.

Some more exotic, and so far not much investigated, proposals to model FOL programs included [160], using insights from fractal geometry to construct iterated function systems with attractors corresponding to fixed points of the semantic consequence operators, and works exploring fibring neural networks for a similar purpose [161]. Finally, works focusing on fuzzy logic interpretations of the necessary transformation of logical theories into the numerical domain of neural networks, such as detailed in [162], are generally related to the proposed semantics of LRNNs (Section 5.1.3), as well as many of the works within NSI.

4.3.4 *Distinctions*

Although there have been many works proposed as neural (first-order) logic reasoners and/or learners within the NSI community, all the methods introduced before the LRNN framework differ from it quite fundamentally. The main common theme in NSI was trying to find some *uniform* model of a logic program (or the semantic operator T_P thereof) represented by some sort of *static* neural network architecture, where full symbolic processing functionalities should emerge from the underlying neural structure or inference processes. In contrast, the most salient property of our approach is that the ground network structure depends not only on the relational rule set but also on a particular learning example, i.e. different networks are dynamically constructed for different examples in order to exploit their particular relational properties. This lifted modeling strategy was known in SRL but unexploited in NSI. Consequently, it is somewhat difficult to categorize the LRNN framework within the proposed NSI categorization dimensions [12], studying the correspondences between logic programs and (static) neural models. From one perspective, it might be considered as a hybrid, technical, application-oriented approach [70] due to our focus on practicality, utilizing existing (SRL and DL) techniques. Yet from a different perspective, it actually provides complete integration in the original NSI sense of [21].

We note that for the propositional case, the NSI idea of direct correspondence between logic programs and static neural networks, as introduced already by McCulloch and Pitts [21], is completely feasible, elegant and efficient. Consequently, the propositional systems such as KBANN relate very closely to our framework (i.e. they can be emulated within LRNNs as shown in Section 7.2.1.1). Nevertheless, with the unbound interpretations of the FOL programs, exploitation of this direct translation of a program $\mathcal{P}$ into a static network model N_P (e.g. with the CORE method [155]) has

so far been of only theoretical interest. To capture the principally infinite variety of the FOL program interpretations, encoding into (a vector of) real numbers was proposed in NSI, relying on the universality of function approximation by a neural network [139]. Yet this universality, similarly to the discussed Turing universality of recurrent NNs (Section 4.2.1), does not always come with efficiency [34] (many other models have been proved as universal approximators [163]), and examples of effective extrapolation (generalization) of this encoding of FOL interpretations by some neural network architecture are yet to be presented. Consequently, most of the previously introduced first-order NSI methods focused solely on deductive reasoning, whilst learning capabilities in these were either very limited or not present at all.

As opposed to the approach of numeric interpretation encoding, we actually follow the original (propositional) program interrelation very closely. The ability of coping with the possibly infinite variations of the FOL interpretations then comes directly from the fact that we do not use a single static neural model, as commonly done in NSI, but rather utilize the lifted modeling strategy (Section 1.4.2). This effectively resolves a key issue identified in the NSI community as the "variable binding problem" [148]. As opposed to most of the introduced, interesting yet rather abstract, approaches proposing temporal synchronization [146] with matching in firing networks, binding of activation functions [164] or transformations to approximate variable-free representations [156], each coming with a respective class of restrictions, we simply directly bind the variables to ground structures by the standard logical means (Section 3.1.2.2). Consequently, we retain the correctness of the (purely) logical inference, as well as the efficiency of neural representation learning of the symbolic structures (Section 4.2.1).

Part III

THE FRAMEWORK

In this part we discuss the proposed framework itself. Firstly in Chapter 5, we introduce the formalism and inner workings of the framework, and demonstrate some novel modeling paradigms enabled by integrating the neural learning with relational logic templating. Secondly in Chapter 6, we extend the framework with structure learning of the logical templates, allowing for a completely automated learning process. Thirdly in Chapter 7, we demonstrate the framework from a new perspective of differentiable logic programming to show benefits of such an approach, in contrast to standard deep learning methods, with a particular focus on graph neural network models and frameworks. Lastly in Chapter 8, we discuss the more recent body of work related to the framework.

5

LIFTED RELATIONAL NEURAL NETWORKS

In this chapter, we describe the Lifted Relational Neural Networks (LRNNs) – a machine learning framework that uses the lifted modeling paradigm (Section 1.4.2) for deep relational learning, which is the main contribution of this thesis (Section 1.5). Similarly to the other lifted models known from SRL (Section 3.4), LRNNs are represented as sets of weighted relational logic rules, used to describe the structure of a given learning setting. Together with a set of weighted relational facts, commonly used to describe a learning sample, these weighted rules then define a standard, also referred to as "ground", neural network, whose outputs can be used for predicting truth-value of given (relational) queries, corresponding to target labels in standard supervised learning.

During learning, a separate ground neural network is dynamically unfolded from the template for each training and testing sample. The structure of these networks is obtained through grounding (Section 3.1.2.2) of the relational rules w.r.t. the constants that appear in the data samples. The weights of the ground networks are then determined by the weights of the relational rules. Crucially, the weights of different ground neurons that were constructed from the same relational rule are tied in our framework, similarly to how weights are shared in the lifted graphical models from SRL (Section 3.4.2), or how weights are tied together by application of the convolutional filters in CNNs (Section 2.2). Naturally, the weights are also shared across the different networks, and so weight-updates performed for one training example are reflected in networks produced for other examples. This allows the lifted model to be trained directly from arbitrary relational data using efficient gradient-based optimization routines known from standard deep learning (Chapter 2).

The framework then provides a flexible way for implementing and combining a wide variety of modeling concepts. In particular, the use of relational logic allows for a declarative specification of *latent relational structures*, which can then be efficiently discovered in a given data set using deep learning. Besides the increased (relational) expressiveness, an important advantage of classic lifted models, including LRNNs, is that they can make explicit which symmetries exist in a domain, and thus reduce the number of weights that have to be learned. Another advantage is that the logic formulas can be provided by domain experts, which also offers a convenient way of incorporating background knowledge (Section 3.3.1) and guiding the learning process towards certain paradigms[1], although the formulas can also be learned from data, e.g. through the ILP methods.[2]

The most salient properties of LRNNs then stem from the used strategy of lifted modeling, where different ground neural networks are unfolded from the relational template to exploit each example's particular relational properties. While there have been previous works on dynamically embedding varying relational input sample structures (Section 4.2), and works mapping logic programs (templates) into neural architectures (Section 4.3), to our best knowledge none of the previous works exploited both at the same time, as we do in LRNNs. Note that while the background on related works preceding the LRNN framework was already introduced in Chapter 4, we will also further discuss related works that appeared more recently in Chapter 8.

This chapter is further structured as follows. Section 5.1 formally introduces LRNNs, explains how a ground network can be constructed for a given sample, and how the weights of an LRNN

1 as further detailed in Chapter 7
2 as further detailed in Chapter 6

can be learned. Subsequently, Section 5.2 illustrates some of the modeling constructs that can be encoded using LRNNs. Finally, Section 5.3 presents experimental evaluation, with conclusions in the subsequent Section 5.4.

5.1 THE FRAMEWORK

In this section, we formally introduce the framework of LRNNs. We define the representation language, describe the translation into neural models and how to use them for inference. We then discuss variant semantics with the choice of activation functions and negation, and detail the learning process.

5.1.1 *Definition*

A lifted relational neural network (LRNN) $\mathcal{N}$ is a set of weighted definite clauses, i.e. a set of pairs (R_i, w_i) where R_i is a definite clause and $w_i \in \mathbb{R}$. For an LRNN $\mathcal{N}$, we write $\mathcal{N}^*$ to denote the corresponding set of definite clauses without weights, i.e. $\mathcal{N}^* = \{C \mid (C, w) \in \mathcal{N}\}$.

5.1.1.1 *Grounding*

As already mentioned in the introduction, LRNNs are seen as templates for creating *ground* neural networks, which will be obtained by standard grounding $G(\mathcal{N}^*)$ (Section 3.1.2.2) of the logical view $\mathcal{N}^*$ of the weighted LRNN template $\mathcal{N}$. The resulting ground networks will (among others) contain a neuron for each considered ground clause from $\overline{\mathcal{N}^*}$. Since it is clearly beneficial not to create unnecessarily large networks, it is important to avoid including any ground clauses that are not relevant (i.e. those that are not active in the least Herbrand model). In practice, however, most of the rules in the standard grounding $G(\mathcal{N}^*)$ will be irrelevant, as their body can never be satisfied. For that we define *restricted grounding* as follows.

Definition 6 *A restricted grounding* $G^R(\mathcal{N}^*)$ *limits the grounding* $G(\mathcal{N}^*)$ *to those rules which are "active", i.e. whose body is satisfied in the least Herbrand model* $\mathcal{H}$ *of* $\mathcal{N}$*. It is defined by* $G^R(\mathcal{N}^*) = \{h\theta \leftarrow b_1\theta \wedge \cdots \wedge b_k\theta \mid (h \leftarrow b_1 \wedge \cdots \wedge b_k) \in \mathcal{N}^*$ *and* $\{h\theta, b_1\theta, \ldots, b_k\theta\} \subseteq \mathcal{H}\}$.

We further denote such restricted grounding of a LRNN template $\mathcal{N}$ by $\overline{\mathcal{N}}$.

Example 6 *Let the LRNN* $\mathcal{N}$ *be defined as follows:*

$$\begin{aligned}\mathcal{N} = \{&(\mathit{mother}(C, M) \leftarrow \mathit{parent}(C, M) \wedge \mathit{female}(M), 1),\\ &(\mathit{father}(C, F) \leftarrow \mathit{parent}(C, F) \wedge \mathit{male}(F), 2),\\ &(\mathit{female}(\mathit{alice}), 1), (\mathit{parent}(\mathit{bob}, \mathit{alice}), 1), (\mathit{parent}(\mathit{eve}, \mathit{alice}), 1)\}.\end{aligned}$$

The grounding of $\mathcal{N}$ *is then given by:*

$$\begin{aligned}\overline{\mathcal{N}} = \{&(\mathit{mother}(\mathit{bob}, \mathit{alice}) \leftarrow \mathit{parent}(\mathit{bob}, \mathit{alice}) \wedge \mathit{female}(\mathit{alice}), 1),\\ &(\mathit{mother}(\mathit{eve}, \mathit{alice}) \leftarrow \mathit{parent}(\mathit{eve}, \mathit{alice}) \wedge \mathit{female}(\mathit{alice}), 1),\\ &(\mathit{female}(\mathit{alice}), 1), (\mathit{parent}(\mathit{bob}, \mathit{alice}), 1), (\mathit{parent}(\mathit{eve}, \mathit{alice}), 1)\}.\end{aligned}$$

Note that $\overline{\mathcal{N}}$ *does not contain the predicates* $\mathit{male}/1$ *or* $\mathit{father}/2$ *as they do not appear in least Herbrand model.*

Procedurally, the grounding $\overline{\mathcal{N}}$ of an LRNN $\mathcal{N}$ is based on the generic (semi-naive) bottom-up[3] routine introduced in Section 3.2.1.1, with some additional optimizations. Particularly, while building the least Herbrand model of $\mathcal{N}^*$, we employ an efficient θ-subsumption engine to find the valid

3 We note that there is also a top-down version of the CSP-inspired grounder, that has been used in some previous experiments [97], the original version of which is detailed in [96].

substitutions of each relevant rule in the current T_p iteration. This engine, adopted from [108], is inspired by CSP techniques, including backtracking search with forward checking, a variable selection heuristic and randomized restarting strategies.

5.1.2 *Neural Networks*

The neural network corresponding to an LRNN $\mathcal{N}$ is then constructed as follows.

5.1.2.1 *Neurons*

The neurons that appear in the network correspond to logical constructs, such as atoms, facts and rules. Specifically, the network contains the following types of neurons:

- For each ground atom h occurring in $\overline{\mathcal{N}}$, there is a neuron A_h, called an *atom neuron*.
- For each ground fact $(h, w) \in \overline{\mathcal{N}}$, there is a neuron $F_{(h,w)}$, called a *fact neuron*.
- For every ground rule $(c\theta \leftarrow b_1\theta \wedge \cdots \wedge b_k\theta, w) \in \overline{\mathcal{N}}$, there is a neuron $R_{(c\theta \leftarrow b_1\theta \wedge \cdots \wedge b_k\theta, w)}$, called a *rule neuron*.
- For every (possibly non-ground) rule $(c \leftarrow b_1 \wedge \cdots \wedge b_k, w) \in \mathcal{N}$ and every grounding $h = c\theta$ of c that occurs in $\mathcal{H}$, there is a neuron $Agg^h_{(c \leftarrow b_1 \wedge \cdots \wedge b_k, w)}$, called an *aggregation neuron*.

5.1.2.2 *Weights and Connections*

We now describe how the different neurons are connected, and how their outputs are defined. Intuitively, the neural network computes for each ground atom h a truth value, which is encoded by the output of the atom neuron A_h. To obtain these truth values, the network propagates values in a way which closely mimics the immediate consequence operator[4]. In particular, when using the immediate consequence operator, there are two ways in which h can become true: if h corresponds to a fact, or if h is the head of a rule whose body is already satisfied. Similarly, the inputs of the atom neuron A_h consist of the fact neurons of the form $F_{(h,w)}$ and aggregation neurons of the form $Agg^h_{(c \leftarrow b_1 \wedge \cdots \wedge b_k, w)}$. The output of an atom neuron with inputs $i_1, ..., i_m$ is given by $g_\vee(i_1, ..., i_m)$, where $g_\vee$ is an activation function that maps the inputs to a real-valued output. Different choices are possible for $g_\vee$, as will be discussed in detail in Section 5.1.3.

A fact neuron $F_{(h,w)}$ has no input and has the value w as its output. The output of the aggregation neuron $Agg^h_{(c \leftarrow b_1 \wedge \cdots \wedge b_k, w)}$ intuitively expresses how strongly h can be derived using the rule $c \leftarrow b_1 \wedge \cdots \wedge b_k$. Note that there can be several groundings of the rule that have the atom h in the head, when the body of the rule contains variables that do not appear in the head. This is why we need to distinguish rule neurons, which correspond to individual groundings of rules, from aggregation neurons, which group together all groundings of a rule that have the same head. Specifically, the inputs of the aggregation neuron $Agg^h_{(c \leftarrow b_1 \wedge \cdots \wedge b_k)}$ are all rule neurons $R_{(c\theta \leftarrow b_1\theta \wedge \cdots \wedge b_k\theta, w)}$ for which $c\theta = h$. The output of this aggregation neuron is given by $w \cdot g_*(i_1, ..., i_m)$, where $i_1, ..., i_m$ are its inputs, g_* is an activation function, and w is the weight of the corresponding rule. Note that while we will assume throughout this chapter that the weight of a rule and the value $g_*(i_1, ..., i_m)$ are combined using multiplication, in principle other combination operators are also possible. The rule neuron $R_{(c\theta \leftarrow b_1\theta \wedge \cdots \wedge b_k\theta, w)}$ intuitively needs to fire if the atoms $b_1\theta, ..., b_k\theta$ are all true. Accordingly, its inputs $i_1, ..., i_k$ are given by the atom neurons $A_{b_1\theta}, ..., A_{b_k\theta}$, and its output is $g_\wedge(i_1, ..., i_k)$, with $g_\wedge$ being a third type of activation function. An overview of the different types of neurons and their interconnections is shown in Table 5, where we write $o(N)$ to denote the output of neuron N.

4 In fact, it is possible to precisely characterize the behavior of the neural network by using extensions of the immediate consequence operator for multi-valued logics [165, 166].

Table 1: Overview of the process of constructing a neural network from a given LRNN $\mathcal{N}$.

Type of neuron	Notation	Output
Atom neuron	A_h	$g_\vee(o(F_{(h,w)}), o(Agg^h_{\alpha_1}), ..., o(Agg^h_{\alpha_m}))$
Fact neuron	$F_{(h,w)}$	w
Rule neuron	$R_{(c\theta \leftarrow b_1\theta \wedge ... \wedge b_k\theta, w)}$	$g_\wedge(o(A_{b_1\theta}), ..., o(A_{b_1\theta}))$
Aggregation neuron	$Agg^h_{(c \leftarrow b_1 \wedge \cdots \wedge b_k, w)}$	$w \cdot g_*(o(R_{\alpha_1}), ..., o(R_{\alpha_m}))$

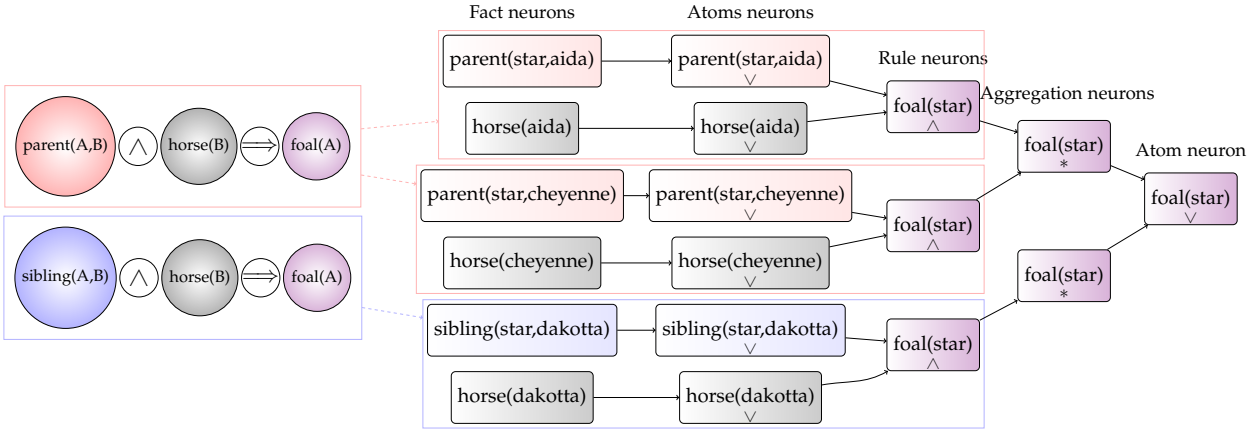

Figure 3: Left: visualization of the rules from the LRNN template $\mathcal{N}$ from Example 7 (omitting the ground facts). Right: the corresponding ground neural network. Atom neurons are denoted by "$\vee$", rule neurons by "$\wedge$" and aggregation neurons by "$*$", the remaining neurons are fact neurons.

Example 7 *Let us consider the following LRNN:*

$$\mathcal{N} = \{(foal(A) \leftarrow parent(A,B) \wedge horse(B), w_m), (foal(A) \leftarrow sibling(A,B) \wedge horse(B), w_n),$$
$$(horse(dakotta), w_1), (horse(cheyenne), w_2), (horse(aida), w_3),$$
$$(parent(star, aida), w_4), (parent(star, cheyenne), w_5), (sibling(star, dakotta), w_6)\}.$$

Figure 3 shows the LRNN $\mathcal{N}$ from the example, together with its ground neural network. To visualize the creation of the neural network, colors are used to denote unique predicate signatures, while rectangles group the neurons that have been derived from the same ground rule.

5.1.2.3 *LRNNs as Neural Network Templates*

To use LRNNs in practice, we typically start from a set of rules $\mathcal{P}$ and some labelled examples. The labelled examples are used to learn weights for the general rules in $\mathcal{P}$, as will be explained in Section 5.1.6. This process leads to a trained LRNN $\mathcal{N}$ which contains weighted rules, but does not contain any ground facts. Each time we want to use this LRNN, we then first add a set of weighted ground facts $\mathcal{M}$ that describe the specific example about which we want to make a prediction. In this way, for each prediction we make, a different ground neural network is constructed, with a potentially very different structure. This process makes LRNNs similar in spirit to lifted graphical models, and rather different from normal neural network frameworks.

Example 8 *We consider an LRNN $\mathcal{N}$ containing rules for predicting the toxicity of molecules, based on conformations of the atoms contained in them and their bonds. For example, it may be beneficial to assign*

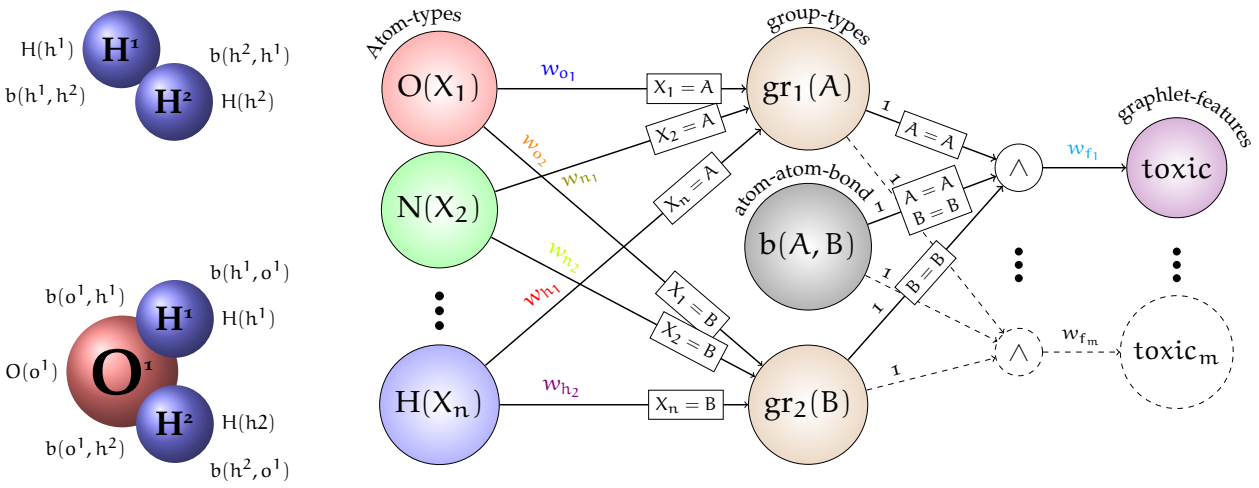

Figure 4: Two example molecules (left), described by surrounding sets of ground facts $\mathcal{M}_1$ and $\mathcal{M}_2$, and a (loose) visualization the LRNN template $\mathcal{N}$ (right) from Example 8, composed of weighted rules aimed at describing general relational features of molecules, such as $\mathtt{graph}_1$, to be learned through the corresponding weights.

atoms to some latent groups. Assuming we want to consider two latent groups, this can be accomplished using the following rules:

$$w_{o_1} : gr_1(X) \leftarrow O(X) \qquad w_{n_1} : gr_1(X) \leftarrow N(X) \qquad \ldots \qquad w_{h_1} : gr_1(X) \leftarrow H(X)$$
$$w_{o_2} : gr_2(X) \leftarrow O(X) \qquad w_{n_2} : gr_2(X) \leftarrow N(X) \qquad \ldots \qquad w_{h_2} : gr_2(X) \leftarrow H(X)$$

The weight w_{o_1}, for instance, represents the degree to which oxygen atoms (O) belong to the first latent group. This membership degree is learned based on training data, i.e. all we need to specify is that we want to consider two latent groups as the basis for making predictions and that none of the atoms $O, N, \ldots, H$ is excluded a priori from belonging to these groups. From the latent groups of atoms, we can now construct relational patterns, such as small chains, trees or general graphlets. For instance, the following rule describes a small relational pattern where an atom from gr_1 is connected to an atom from gr_2 through a bond (represented by the predicate b):

$$w_{f_1} : toxic \leftarrow gr_1(A) \wedge b(A, B) \wedge gr_2(B) \tag{3}$$

In general, there would be different rules with toxic in the head, each encoding a different relational pattern in the body. Which of these relational patterns is actually predictive of toxicity is then again learned from training data. For example, if (3) was found to be predictive, then w_{f_1} would receive a high value after training; otherwise, it might be set as 0. A rough visualization of the idea behind this template $\mathcal{N}$ is displayed in Figure 4 - right.

Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be two sets of (weighted) facts, each describing a given molecule, i.e. the particular conformations of specific atoms and bonds (Figure 4 - left). To use the LRNN $\mathcal{N}$ for predicting the toxicity of these molecules (after its weights have been trained), we construct the ground networks of $\overline{\mathcal{N} \cup \mathcal{M}_1}$ and $\overline{\mathcal{N} \cup \mathcal{M}_2}$, and for both of the resulting ground neural networks we compute the output of an atom neuron corresponding to the literal toxic. The ground neural networks for the two example molecules and template, visualized in Figure 4, are then displayed in Figure 5. As can be seen from this figure, the neural networks for the two cases are different in both size and structure, which is a result of the fact that $\mathcal{N}^* \cup \mathcal{M}_1^*$ and $\mathcal{N}^* \cup \mathcal{M}_2^*$ have different Herbrand models.

5.1.3 *Activation Functions*

The behavior of an LRNN crucially depends on how the activation functions $g_\wedge$, $g_\vee$ and g_* are chosen. Intuitively, $g_\wedge$ should behave like a conjunction, in the sense that the atom neuron for h

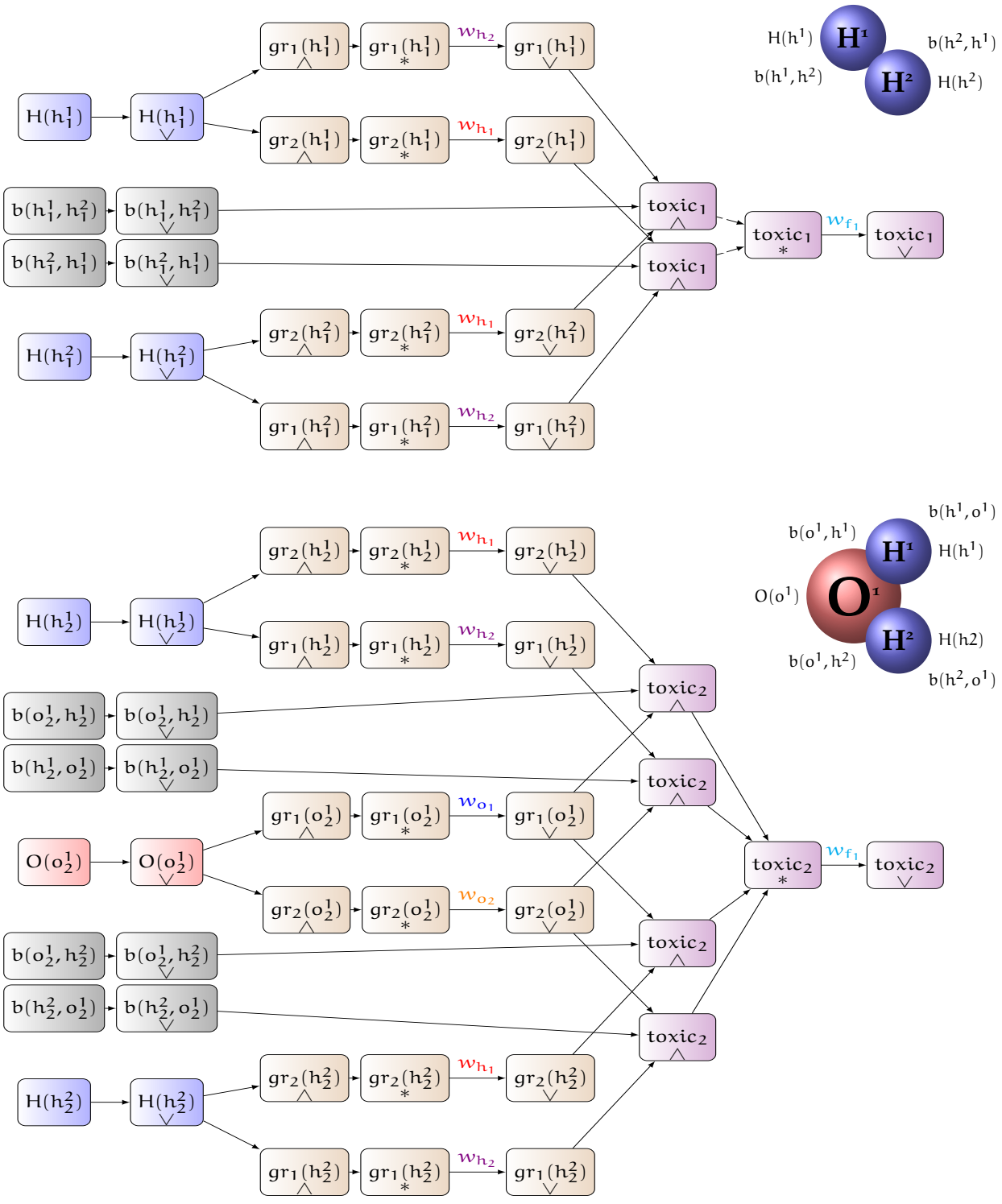

Figure 5: The two neural networks $\overline{\mathcal{N} \cup \mathcal{M}_1}$ and $\overline{\mathcal{N} \cup \mathcal{M}_2}$ grounded from the single LRNN $\mathcal{N}$ and the two example molecules from Example 8. Atom neurons are denoted by "$\vee$", rule neurons by "$\wedge$" and aggregation neurons by "$*$", the remaining neurons are fact neurons.

should only receive a high weight from the rule neuron for $h \leftarrow b_1 \wedge \cdots \wedge b_k$ if the atom neurons for $b_1, ..., b_k$ all have a high output value. Similarly, $g_\vee$ and g_* should intuitively behave like disjunctions; the reason why we need two types of disjunctive activation functions will become clear below. One possibility for choosing the activation functions is to draw inspiration from the field of fuzzy logic, where extensions of logical connectives to real-valued arguments have been widely studied [167]. In particular, if all input weights are between 0 and 1, such fuzzy logic connectives could straightforwardly be used. For example, in accordance with Gödel logic, we could choose the activation functions as follows:

$$g_\wedge(b_1, ..., b_k) = \min(b_1, ..., b_k)$$
$$g_*(b_1, ..., b_m) = g_\vee(b_1, ..., b_m) = \max(b_1, ..., b_k)$$

Alternatively, using the connectives from product logic, we obtain:

$$g_\wedge(b_1, ..., b_k) = b_1 \cdot ... \cdot b_k$$
$$g_*(b_1, ..., b_m) = g_\vee(b_1, ..., b_m) = 1 - (1 - b_1) \cdot ... \cdot (1 - b_k)$$

Another popular alternative are the Łukasiewicz connectives:

$$g_\wedge(b_1, ..., b_k) = \max(b_1 + ... + b_k - k + 1, 0)$$
$$g_\vee(b_1, ..., b_m) = \min(b_1 + ... + b_m, 1)$$
$$g_*(b_1, ..., b_m) = \max(b_1, ..., b_k)$$

Note that $g_\vee$ and g_* in this case correspond to the two types of disjunctions that are used in Łukasiewicz logic. An advantage of using fuzzy logic connectives is that LRNNs can then be seen as fuzzy logic programs. In particular, the predictions of an LRNN $\mathcal{N}$ are then precisely given by the truth degrees of the corresponding atoms in the least Herbrand model of $\mathcal{N}$, viewed as a fuzzy logic program. However, using these fuzzy logic connectives also has two important drawbacks. First, gradient-based learning is considerably less effective with such operations, compared to e.g. sigmoidal activation functions. Second, it would be useful to consider parametrized families of activation functions, such that a specific activation function could be chosen based on training data. The latter issue could in principle be addressed by using continuous families of fuzzy logic connectives, such as the Frank family of t-norms, which is parametrized by a real value, and has the three aforementioned examples of fuzzy logic conjunctions as special cases. However, as this would further complicate gradient-based learning, in this chapter we will focus on more standard activation functions, and sigmoidal functions in particular.

Specifically, we will consider two classes of activation functions, which will be useful in slightly different types of applications. The first class is defined as follows.

Definition 7 (Max-Sigmoid Activation Functions) *The Max-Sigmoid (MS) collection of activation functions are defined as:*

$$g_\wedge(b_1, \ldots, b_k) = sigm\left(a \cdot \left(\sum_{i=1}^{k} b_i - k + 1 + b_0\right)\right)$$
$$g_*(b_1, \ldots, b_m) = \max_i b_i$$
$$g_\vee(b_1, \ldots, b_k) = sigm\left(a \cdot \left(\sum_{i=1}^{k} b_i + b_0\right)\right)$$

where $a, b_0 \in \mathbb{R}$ *are parameters.*

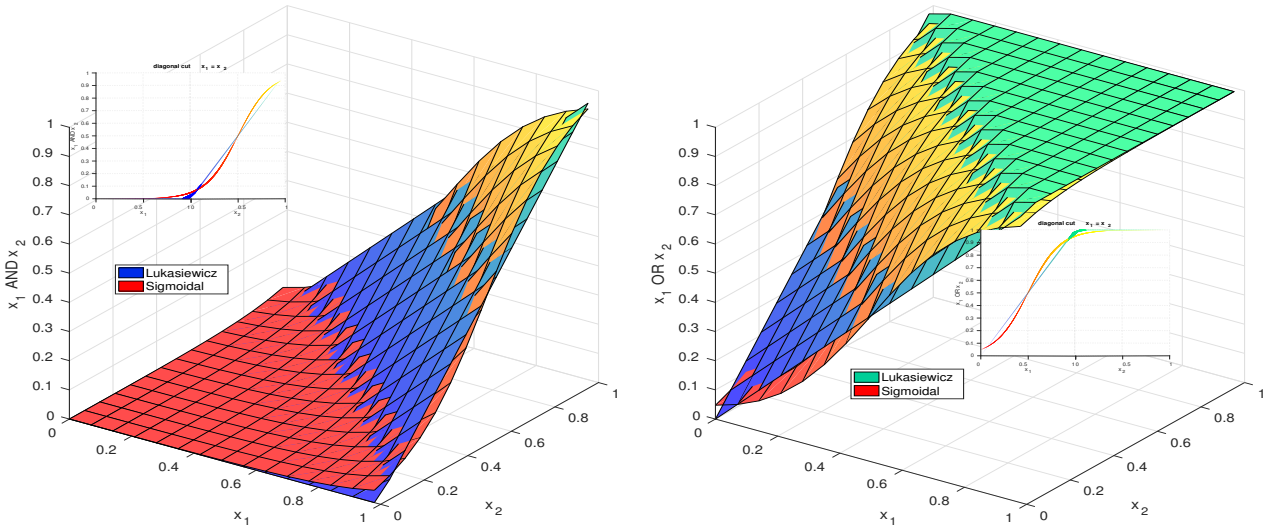

Figure 6: A crude approximation of Łukasiewicz conjunction (left) and disjunction (right) by respective sigmoidal activation functions for the use in LRNNs. A diagonal view of $x_1 = x_2$ is embedded to detail the level of approximation of each operator.

The parameters a and b_0 are typically fixed, although they could also be determined using training data (see Section 5.1.6). When learned from training data, they help to ensure that we select an activation function that is appropriate for the domain being modelled. When the inputs are between 0 and 1, the parameters a and b_0 in the definition of $g_\wedge$ and $g_\vee$ can be chosen such that these activation functions approximate the Łukasiewicz connectives. To illustrate this, the functions $sigm\left(a \cdot (b_1 + \cdots + b_k - k + 1 + b_0)\right)$ and $a \cdot sigm\left(a \cdot (b_1 + \cdots + b_k + b_0)\right)$, with $b_0 = -0.5$ and $a = 6$, are compared with the Łukasiewicz conjunction and disjunction, respectively, in Figure 6. The g_* activation function groups different groundings of the same rule with the same head. If we think of the bodies of these rules as patterns, choosing g_* as the maximum means that we are simply looking for the best match. This view is illustrated in the following example.

Example 9 *Let us consider the following LRNN with the activation functions from the Max-Sigmoid family:*

$$\begin{aligned}\mathcal{N} = \{&(\mathit{hasBrightEdge} \leftarrow \mathit{isBright}(E), 1),\\ &(\mathit{isBright}(E) \leftarrow \mathit{edge}(E, U, V) \wedge \mathit{bright}(U) \wedge \mathit{bright}(V), 1),\\ &(\mathit{bright}(U) \leftarrow \mathit{yellow}(U), 2), (\mathit{bright}(U) \leftarrow \mathit{red}(U), 1), (\mathit{bright}(U) \leftarrow \mathit{blue}(U), 0.5)\}.\end{aligned}$$

Note that hasBrightEdge is a predicate of arity 0. Let us also consider a set $\mathcal{G}$ describing a specific graph with colored vertices.

$$\begin{aligned}\mathcal{G} = \{&(\mathit{edge}(e_1, v_1, v_2), 1), (\mathit{edge}(e_2, v_2, v_3), 1), (\mathit{edge}(e_3, v_3, v_4), 1), (\mathit{edge}(e_4, v_4, v_1), 1),\\ &(\mathit{red}(v_1), 1), (\mathit{blue}(v_2), 1), (\mathit{yellow}(v_3), 1), (\mathit{yellow}(v_4), 1)\}\end{aligned}$$

The output of the atom neuron $A_{\mathit{hasBrightEdge}}$ only depends on the "brightest edge", which in this case is e_3. This is because the aggregation function g_* is $g_*(b_1, \ldots, b_m) = \max_i b_i$. Note that any colored graph that contains an edge connecting two yellow vertices would lead to the same output. Similar kinds of LRNNs could be useful in practice, for instance, to detect molecules which contain a substructure that is similar to a prescribed pattern. Instead of colors, we would then model physico-chemical properties of atoms (e.g. partial charge) and instead of graph edges we would describe bonds in molecules.

In the next example, we illustrate how using a sigmoidal function for $g_\vee$ leads us intuitively to accumulate the evidence coming from different rules with the same head.

Example 10 *Consider the following LRNN with the activation functions from the Max-Sigmoid family:*

$$\begin{aligned}\mathcal{N} =&\{(\mathit{highPressure}(X) \leftarrow \mathit{stressed}(X), 1), (\mathit{highPressure}(X) \leftarrow \mathit{obese}(X), 1),\\ &(\mathit{highPressure}(X) \leftarrow \mathit{exercises}(X), -1)\}\end{aligned}$$

and the set of weighted facts $\mathcal{P} = \{(\mathit{stressed}(\mathit{alice}), 1), (\mathit{obese}(\mathit{alice}), 1), (\mathit{stressed}(\mathit{bob}), 1), (\mathit{exercises}(\mathit{bob}), 1)\}$. *Outputs of aggregation neurons for rules with the same head are combined using the activation function* $g_\vee$. *In this particular example, if we construct the ground LRNN of* $\overline{\mathcal{N} \cup \mathcal{P}}$ *then the output of the atom neuron* $A_{\mathit{highPressure}(\mathit{alice})}$ *will be higher than the output of the atom neuron* $A_{\mathit{highPressure}(\mathit{bob})}$, *because alice is stressed and obese whereas bob is just stressed and exercises.*

We now give an example of a scenario where the Max-Sigmoid class of activation functions is not appropriate.

Example 11 *Let us consider the following simple LRNN for predicting which individuals are infected with the flu:*

$$\mathcal{N} =\{(\mathit{hasFlu}(A) \leftarrow \mathit{friends}(A, B) \wedge \mathit{hasFluDiagnosed}(B), 1)\}$$

and a set of weighted ground facts $\mathcal{P}$ *about a group of people and their friendships. If we constructed the ground neural networks of* $\overline{\mathcal{N} \cup \mathcal{P}}$ *using the activation functions from the Max-Sigmoid family then the prediction of whether an individual has the flu would be entirely based on the existence of at least one person who was already diagnosed with the flu. It would obviously be more meaningful to base the predictions on the fraction of one's friends who were diagnosed with the flu.*

The next definition introduces a family of activation functions that are appropriate for situations such as the one in the previous example.

Definition 8 (Avg-Sigmoid Activation Functions) *The Avg-Sigmoid (AS) collection of activation functions are defined as:*

$$g_\wedge(b_1, \ldots, b_k) = \mathit{sigm}\left(a \cdot \left(\sum_{i=1}^{k} b_i - k + b_0\right)\right)$$

$$g_*(b_1, \ldots, b_m) = \frac{1}{m}\sum_{i=1}^{m} b_i$$

$$g_\vee(b_1, \ldots, b_k) = \mathit{sigm}\left(a \cdot \left(\sum_{i=1}^{k} b_i + b_0\right)\right)$$

An advantage of the Avg-Sigmoid family over the Max-Sigmoid family is that the functions from the Avg-Sigmoid family are everywhere differentiable, which simplifies learning. Of course, a wide variety of other families of activation functions can be used for LRNN learning, but in this chapter we will limit ourselves to the two considered families.

Finally note that, in principle, a different activation function could be used for every neuron. For example, we may have some rules that are aimed at detecting the existence of a pattern, suggesting the use of a maximum, while for other rules we may want to accumulate the evidence provided by different rules, suggesting the use of a summation or average. We could also consider a setting where the different activation functions are learnable, e.g. by tuning the value of the parameter b_0. Note, however, that the latter could also be simulated in the basic framework, by adding an additional atom to each rule body and learning the weight of that atom instead.

5.1.4 *Negation As Failure*

There is a close connection between LRNNs, on the one hand, and (multi-valued extensions of) logic programs, on the other hand. However, from a logic programming point of view, the syntax of LRNNs is quite limited, as negation is not considered. In particular, *negation-as-failure* is a central notion in most logic programming frameworks [91]. The idea is to use rules such as $a \leftarrow b \wedge \textbf{not}\ c$, which intuitively encodes that we can derive a if b is true, unless we can prove that c is true (i.e. c is an explicit exception to the default rule "if b then normally a"). In general, logic programs with negation-as-failure may no longer have a unique least Herbrand model. This effectively makes these logic programs non-deterministic, and thus unsuitable as a basis for LRNNs in general.

A logic program with negation-as-failure $\mathcal{P}$ is called stratified if there exists a partition $\mathcal{P} = \mathcal{P}_1 \cup ... \cup \mathcal{P}_n$ such that for each rule $a \leftarrow b_1 \wedge ... \wedge b_k \wedge \textbf{not}\ b_{k+1} \wedge ... \wedge \textbf{not}\ b_{k+l}$ in $\mathcal{P}_i$ it holds that the predicates used in the atoms $b_{k+1}, ..., b_{k+l}$ do not occur in the head of any of the rules in $\mathcal{P}_i \cup ... \cup \mathcal{P}_n$ [168]. It is easy to verify that stratified logic programs have a unique least Herbrand model. It turns out that we can easily generalize LRNNs to cases where the corresponding logic program $\mathcal{N}^*$ is a stratified logic program with negation-as-failure. To this end, we define the grounding of such an LRNN $\mathcal{N}$ as

$$\overline{\mathcal{N}} = \{(h\theta \leftarrow b_1\theta \wedge \cdots \wedge b_k\theta \wedge \textbf{not}\ b_{k+1}\theta \wedge \ldots, \wedge \textbf{not}\ b_{k+l}\theta, w) : \{h\theta, b_1\theta, \ldots, b_k\theta\} \subseteq \mathcal{H} \\ \text{and } (h \leftarrow b_1 \wedge \cdots \wedge b_k \wedge \textbf{not}\ b_{k+1} \wedge \textbf{not}\ b_{k+l}, w) \in \mathcal{N}\}$$

where $\mathcal{H}$ is the least Herbrand model of $\mathcal{N}^*$. Note that we do not put any constraints on the containment of the atoms $b_{k+1}\theta \wedge \cdots \wedge b_{k+l}\theta$ in the Herbrand model $\mathcal{H}$. In a classical logic programming setting, we can omit the rule $h\theta \leftarrow b_1\theta \wedge \cdots \wedge b_k\theta \wedge \textbf{not}\ b_{k+1} \wedge \textbf{not}\ b_{k+l}$ if one of the atoms $b_{k+1}, ..., b_{k+l}$ is in $\mathcal{H}$. However, in accordance with most multi-valued extensions of negation-as-failure, in LRNNs the atom neuron corresponding to **not** b_i may have a non-zero output even if b_i has a non-zero output. Intuitively, if we can only derive that b_i is partially true, a rule with **not** b_i in the body will still partially fire.

To construct the ground network of an LRNN with negation-as-failure, we consider an additional type of neuron called *negation neuron*. For every ground atom **not** a that occurs in one of the rules of the grounding $\overline{\mathcal{N}}$, we add one such negation neuron $\textbf{Not}_a$, which has the atom neuron for a as input, if it exists (i.e. if $a \in \mathcal{H}$), and the constant o otherwise. This negation neuron is then added as one of the inputs of the corresponding rule neuron. As with the other types of neurons, the activation function $g_{\textbf{not}}$ associated with a negation neuron can be chosen in different ways.

Example 12 *Let us consider an LRNN $\mathcal{N}$ consisting of the rule*

$$w_1 : \mathit{flies}(X) \leftarrow \mathit{bird}(X) \wedge \textbf{not}\ \mathit{antarctic}(X)$$

and the fact $\mathit{bird}(\mathit{tweety})$. The grounding $\overline{\mathcal{N}}$ contains the rule

$$\mathit{flies}(\mathit{tweety}) \leftarrow \mathit{bird}(\mathit{tweety}) \wedge \textbf{not}\ \mathit{antarctic}(\mathit{tweety}).$$

Therefore there will be a neuron $\textbf{Not}_{\mathit{antarctic}(\mathit{tweety})}$ in the ground network. The input to this neuron is the constant 0 in this case because $\mathit{antarctic}(\mathit{tweety})$ is not in the least Herbrand model of the corresponding logic program $\mathcal{N}^*$. Assuming that the activation function of negation neurons is $1 - x$, the output of this neuron will be 1. The rule neuron $R_{\mathit{flies}(\mathit{tweety}) \leftarrow \mathit{bird}(\mathit{tweety}) \wedge \textbf{not}\ \mathit{antarctic}(\mathit{tweety})}$ then has two inputs, the neuron $\textbf{Not}_{\mathit{antarctic}(\mathit{tweety})}$ and the atom neuron $A_{\mathit{bird}(\mathit{tweety})}$. The rest of the network is constructed as for normal LRNNs, following the rules described in Section 5.1.1.

In LRNNs where all the outputs are between 0 and 1, a natural choice for this activation function, which is in line with the semantics of several existing multi-valued logic programming frameworks [166], is $g_{\textbf{not}}(x) = 1 - x$. In other contexts, where positive and negative values are associated with positive and negative support, respectively, the choice $g_{\textbf{not}}(x) = -x$ could be used.

5.1.5 *Recursive Rules*

As mentioned, LRNNs are designed for lifted modeling of *feed-forward* neural networks. This technical limitation was established in order to avoid work with recurrent neural networks, since these are much more difficult to train than feed-forward neural networks. It is easy to see that this is the case when the rules in $\mathcal{N}^*$ are free from cycles, i.e. when there exists a strict ordering $\prec$ of the predicates such that for each predicate p_2 occurring in the body of a rule with predicate p_1 in the head, it holds that $p_1 \prec p_2$.

Note, however, that there are many cases where the logic program $\mathcal{N}^*$ associated with an LRNN contains cycles, but where the resulting ground neural network does not. Thus, in general, recursive rules do not pose any problem to the LRNN framework as long as they do not induce directed cycles in the resulting ground neural networks. For instance, rules defining directed paths (Example 2) in acyclic graphs would not lead to directed cycles in the resulting ground neural networks, despite being recursive. The feed-forwardness of the neural models will cover all the learning scenarios presented later in this chapter, as well as most templates modeling common practical scenarios. A technical treatment of a scenario where this is not true, and the ground neural networks may contain directed cycles, will be presented later in the application Part v in Chapter 11.

An interesting consequence that might be seen as a potential complication caused by allowing LRNNs to have recursion, even in the absence of directed cycles, is that the weights may be shared among neurons that lie on a single directed path from the input to the output of the network. This might intuitively seem to complicate the computation of gradients via backpropagation, since that relies on the chain rule and linearity of partial differentiation of the function represented by the underlying network. However, as we explain subsequently in Section 5.1.6, the resulting networks can still be easily trained, using standard gradient-descent techniques, even in this case[5].

5.1.6 *Weight Learning*

We consider an LRNN $\mathcal{N}$ whose weights we want to train. To this end, we assume that we also have access to a list of training examples $\mathcal{E} = (\mathcal{E}^1, \ldots, \mathcal{E}^m)$, where each $\mathcal{E}^j$ is an LRNN. In applications, these LRNNs encoding training examples would typically only contain ground facts. In such cases, each training example intuitively describes some relational structure using a set of weighted atoms (e.g. as displayed in Figure 5). We also assume that we are given a list $\mathcal{Q} = (\{(q_1^1, t_1^1), \ldots, (q_{k_1}^1, t_{k_1}^1)\}, \ldots, \{(q_1^m, t_1^m), \ldots, (q_{k_m}^m, t_{k_m}^m)\})$ where each q_i^j is a ground atom, which we call a *training query atom*, and t_i^j is its *target value*. For a query atom q_i^j, let y_i^j denote the output of the atom neuron $A_{q_i^j}$ in the ground neural network of $\overline{\mathcal{N} \cup \mathcal{E}^j}$. The goal of the learning process is to find the weights w_h of the rules (and possibly facts) in $\mathcal{N}$ for which the loss J on the training query atoms $J(\mathcal{Q}) = \sum_{j=1}^{m} \sum_{i=1}^{k_j} \mathit{cost}(y_i^j, t_i^j)$ is minimized, where *cost* is some predefined loss function that measures the discrepancy between the output of the neurons corresponding to the training query atoms and their desired target values.

Example 13 *Let us again consider the LRNN $\mathcal{N}$ from Example 8 and the sets of facts $\mathcal{M}_1$ and $\mathcal{M}_2$ describing the two molecules (shown in Figure 5). We recall that $\mathcal{N}$ contains the rule for toxicity of molecules w_1 : $\mathit{toxic} \leftarrow gr_1(A) \wedge b(A,B) \wedge gr_2(B)$ where $b(.,.)$ is a predicate for representing atomic bonds in the molecules, and the predicates $gr_1(.)$ and $gr_2(.)$ are defined using the rules listed in Example 8. Let us suppose that we want to learn the weights of this theory based on the knowledge that $\mathcal{M}_1$ is toxic and $\mathcal{M}_2$ is not. In other words, the list of training examples in this case is given by $\mathcal{E} = (\mathcal{M}_1, \mathcal{M}_2)$. We also need to specify the corresponding list of training query atoms $\mathcal{Q}$, which in this case is given by $\mathcal{Q} = \{\{\mathit{toxic}, 1\}, \{\mathit{toxic}, 0\}\}$. The weight learning task is to optimize the weights so as to minimize the discrepancy between the toxicity according to the training*

5 The situation is analogous to the popular Recursive Neural Networks, introduced earlier in the background Section 2.3.

query atoms and the toxicity predicted by the LRNNs $\mathcal{N} \cup \mathcal{M}_1$ and $\mathcal{N} \cup \mathcal{M}_2$, where the predicted toxicity is the output of the atom neuron A_{toxic}.

Similarly to conventional neural networks, weight adaptation is performed by gradient descent steps:

$$w_h \leftarrow w_h - \gamma \frac{\partial J(\mathcal{Q})}{\partial w_h}$$

where $\gamma > 0$ is some given learning rate. Different from conventional neural networks, in the case of LRNNs, we end up with different ground networks for the different learning examples $\mathcal{E}^j$. This is not problematic, however, because the weights for all the ground neural networks $\overline{\mathcal{N} \cup \mathcal{E}^j}$ are fully specified in the LRNN $\mathcal{N}$.

It is then possible to learn the weights of an LRNN using standard gradient descent techniques, based on gradients computed by backpropagation, except that the increments for the shared weights must be accumulated. The same principle is exploited e.g. in CNNs, except that in LRNNs, the weight-sharing is not limited to individual layers. However, this is not a problem for the gradient computation, as demonstrated with the following backpropagation example.

Remark 1 *Let us consider a ground $\overline{\mathcal{N} \cup \mathcal{E}^j}$ as a regular feed forward neural network N_j with some weights $w_k \in \mathcal{W}^j$ in the network being shared, i.e. bound to the same value, even across the layers, but with the restriction that each particular weight w_k appears at most once on any simple path from the inputs $\mathcal{E}_j$ to the outputs y_j. Let the activation functions of layers l of N_j be g^l, chosen from some set of differentiable functions. Let further $w_k^{a,b,\cdots}$ denote particular occurrences of some shared weight w_k. Then we may express the output of the network as*

$$y_j = g^1\left(\ldots + w_k^a g^2(\ldots) + \ldots + w_m g^2\left(\cdots + w_k^b g^3(\ldots) + \ldots\right) + \ldots\right)$$

where "..." correspond to expressions with no w_k occurrence. Considering each w_k occurrence separately as an independent variable, we have

$$\frac{\partial y_j}{\partial w_k^a} = \frac{\partial\left(g^1\left(w_k^a g^2(\ldots)\right)\right)}{\partial w_k^a} = g^{1\prime}(\ldots) g^2(\ldots)$$

$$\frac{\partial y_j}{\partial w_k^b} = \frac{\partial\left(g^1\left(w_m g^2\left(w_k^b g^3(\ldots)\right)\right)\right)}{\partial w_k^b} = g^{1\prime}(\ldots)\, w_m g^{2\prime}(\ldots)\, g^3(\ldots)$$

Considering all occurrences $w_k^{a,b,\cdots}$ as a single variable w_k, we then have

$$\frac{\partial y_j}{\partial w_k} = \frac{\partial\left(g^1\left(w_k g^2(\ldots) + w_m g^2\left(w_k g^3(\ldots)\right)\right)\right)}{\partial w_k} = g^{1\prime}(\ldots)\left(g^2(\ldots) + w_m g^{2\prime}(\ldots)\, g^3(\ldots)\right)$$

i.e., we see that $\frac{\partial y_j}{\partial w_k} = \frac{\partial y_j}{\partial w_k^a} + \frac{\partial y_j}{\partial w_k^b}$ which follows also directly from additivity of the differentiation operator (keeping in mind that there is only one occurrence of w_k on any simple path from an atom neuron to a fact neuron). Therefore the gradient can be computed for the ground neural networks created from a given LRNN with standard backpropagation and then the components corresponding to a particular weight w_k can be simply accumulated.

In the previous remark, we went beyond standard weight sharing in CNNs, but still limited ourselves to networks without recursive application of the same weight on a single path in the network, which is beyond linearity of partial differentiation. Nevertheless, even in that case the gradient can be computed easily, as we clarify with the following, more general reasoning applicable to arbitrary weight-sharing schemes.

Remark 2 *Let $J(u_1, u_2, \ldots, u_n)$ be a loss function of a given ground neural network. To encode which of these n weights are shared, we can consider a function $f : \mathbb{R}^m \to \mathbb{R}^n$, where $R = \{w_1, ..., w_m\}$ is the set of distinct weight variables ($m \leqslant n$). For a given vector $v \in \mathbb{R}^m$, the vector $f(v)$ is obtained by copying the values of the shared weights. For instance, if $n = 3$, $R = \{w_1, w_2\}$ and the first two weights are shared, then $f(w_1, w_2) = (w_1, w_1, w_2)$. It follows from elementary multivariate calculus that $\frac{\partial}{\partial w_i}(J \circ f) = \nabla J \cdot \frac{\partial f}{\partial w_i}$, where $\cdot$ denotes scalar product (noting that both ∇J and $\frac{\partial f}{\partial w_i}$ are n-dimensional vectors). Since $\frac{\partial f}{\partial w_i}$ is a vector which has 1 at position j iff the j-th weight is assigned to the shared weight w_i, it follows that $\frac{\partial}{\partial w_i}(J \circ f)$ can be computed as the sum of the components of ∇J that correspond to the positions j to which w_i is assigned. In other words, this again means that in order to compute the partial derivative w.r.t. a shared weight w_i, we can simply compute the gradient using standard backpropagation, without assuming any weight sharing, and then sum the individual terms corresponding to the given shared weight.*

The complete weight learning algorithm then works as follows. First, the given LRNN $\mathcal{N}$ is grounded w.r.t. every example $\mathcal{E}^j$, leading to a set of ground neural networks with shared weights; information about the origin of each weight is explicitly stored, such that the respective weights in the template can be updated correctly. The algorithm then iterates over the ground networks in a random order. Each time, it computes the gradient of the loss function for the current example given the current weights in the template, and updates the weights accordingly. It continues iterating these steps, following the standard stochastic gradient descent procedure. To reduce the risk of getting stuck in local optima, we can also employ a restart strategy for this algorithm, restarting the search with randomly initialized weights after a given number of SGD epochs has been reached. There are many restart sequences that we can use, e.g. Luby's universal strategy [169].

For the Max-Sigmoid family of activation functions, learning is complicated by the fact that the max operator introduces non-differentiable points to the optimization problem. As a consequence, some weights, corresponding to ground rules which never contribute to the output value (because they never give maximal output), may never be updated by SGD because the respective partial derivatives of the loss function are zero. This may then lead to poor solutions. The restart strategies mentioned above help to partially alleviate this problem[6].

A NOTE ON STRUCTURE LEARNING For many types of lifted models, the learning process is separated into two steps, called structure learning and weight learning (Section 3.4). In the case of MLNs, for instance, the aim of structure learning consists of determining relevant first-order formulas, whose associated weights are then determined in the subsequent weight learning step. While LRNNs could, in principle, be used in a similar way, one of the main strengths of LRNNs is the fact that they can learn predictive latent relational structures in an efficient way. When we use LRNNs for this purpose, as we do in this chapter, the first-order rules are completely *generic*. Their purpose is then to encode what types of latent structures we want to find, rather than to encode domain knowledge. All domain knowledge is then obtained through weight learning. This way of using LRNNs is illustrated in detail in the next section.

5.2 ILLUSTRATIVE EXAMPLES OF LRNN MODELING CONSTRUCTS

In this section we describe several modelling constructs that can straightforwardly be encoded using LRNNs, but which would be difficult or impossible to implement in frameworks such as CILP++ [16], which also combines logic and neural networks. The considered modelling constructs correspond to different kinds of latent relational structures that can be effectively learned using LRNNs. Frameworks such as CILP++ do not allow simultaneous learning of target and auxiliary predicates, and are thus not well-suited for learning latent structures. Instead, they rely on propositionalization [15] and are thus only capable of learning latent features over the corresponding

6 However note that this is essentially the same situation as with max-pooling in CNNs (Section 2.2), which is commonly simply ignored in practice.

propositionalized representations. While somewhat similar modelling constructs can, in principle, be used in MLNs and in probabilistic logic programming systems such as Problog [52], they would require EM algorithms which repeatedly need to perform computationally expensive probabilistic inference.

5.2.1 *Implicit Soft Clustering*

In many learning tasks, it has been observed that good results can be obtained by generating (soft) clusters of objects. The idea is then to make predictions based on the membership degrees of a given object in these clusters, rather than trying to make predictions directly from the features describing the object itself. As an example, let us consider the problem of predicting adverse effects of drugs. For this problem, the use of auxiliary clusters of similar drugs has been found to lead to significant improvements in predictive accuracy [170]. However, existing methods to generate these auxiliary clusters rely on a form of greedy discrete clustering, which can often be too crisp. Using LRNNs, on the other hand, we can simply define predicates that represent these *soft* clusters, and then automatically train the corresponding weights, which represent the membership degrees. This is illustrated in the following example.

Example 14 *Let us suppose that, similarly to the work of Davis, Costa, Berg, Page, Peissig, and Caldwell 2012, we have temporal data about patients, the drugs they took, and the time instants when changes in their health occurred. We want to predict adverse effects of drugs or their combinations. Let us also assume that we have a set of general rules like:*

$$
\begin{aligned}
w_1^{(1)} : \mathit{effect}(\mathsf{P},\mathsf{AE},\mathsf{T2}) \quad &\leftarrow \quad \mathit{took}(\mathsf{P},\mathsf{D1},\mathsf{T1}) \wedge \mathit{period}(\mathsf{T1},\mathsf{T2},\mathsf{T}) \wedge \mathit{shortPeriod}(\mathsf{T}) \wedge \\
&\qquad \wedge \mathit{took}(\mathsf{P},\mathsf{D2},\mathsf{T2}) \wedge \mathit{drugGroup1}(\mathsf{D1}) \wedge \mathit{drugGroup2}(\mathsf{D2}) \wedge \\
&\qquad \wedge \mathit{effectGroup1}(\mathsf{AE}) \\
&\qquad \ldots \\
w_1^{(2)} : \mathit{effectGroup1}(\mathsf{E}) \quad &\leftarrow \quad \mathit{headache}(\mathsf{E}) \\
w_2^{(2)} : \mathit{effectGroup1}(\mathsf{E}) \quad &\leftarrow \quad \mathit{sneezing}(\mathsf{E}) \\
&\ldots
\end{aligned}
$$

Here, $\mathit{effect}(\mathit{Patient}, \mathit{AdverseEffect}, \mathit{Time})$ is a predicate whose meaning is that the patient Patient had the adverse effect AdverseEffect at time Time. Similarly, $\mathit{took}(\mathit{Patient}, \mathit{Drug}, \mathit{Time})$ states that the patient Patient took the drug Drug at time Time, $\mathit{Period}(\mathsf{T1},\mathsf{T2},\mathsf{T})$ is true when $\mathsf{T} = \mathsf{T2} - \mathsf{T1}$, i.e. it expresses that T is the length of the time period from $\mathsf{T1}$ to $\mathsf{T2}$. For simplicity, we assume that time is discretized (e.g. it may be enough to measure the time in days for certain types of adverse effects), which means that we may represent these predicates extensively as facts. The predicate shortPeriod is a predicate which defines what "short period" means in the given context; here we assume that it is specified by an expert[7]. Finally, the predicates drugGroup1, ..., drugGroupN and effectGroup1, ..., effectGroupM are latent predicates which will be learnt by the LRNN. For convenience, we use the superscript to index the latent predicates and subscript to index the rules defining the latent predicates, starting from 1 for each of the predicates.

Intuitively, the adverse effects that happen to have high soft membership *in the same clusters of effects, defined by the effect-group predicates, are effects that tend to frequently occur together. The groups of drugs are supposed to represent similar drugs. The intuition behind the rules for the predicate effect/3 is as follows. Using these rules, we want to be able to capture the adverse effects which result from interactions of certain drugs that were taken by the patient in a short time interval. Let us assume that we already have definitions of the effectGroup/1 and drugGroup/1 in accordance with the described intuition. Then each of the rules essentially says that if a person took a drug from group 1 and shortly after a drug from group 2 then the*

7 The predicate *shortPeriod* could be learned using techniques analogical to those described later for the predicate $\mathit{Similar}(\mathsf{X},\mathsf{Y})$ in Section 5.3.2.1.

patient will get (all the) adverse effects from the adverse-effect group 1. Note that the weights of these rules can also be negative, encoding the fact that the drugs that were taken actually prevent the adverse effects from the corresponding group.

Using the Avg-Sigmoid family of activation functions, weight learning in this LRNN can implicitly discover clusters of drugs which interact adversely with other clusters of drugs, clusters of adverse effects corresponding to these combinations of drugs, and an appropriate definition for the predicate shortPeriod. To obtain these weights, we only require examples consisting of ground facts describing patients, the drugs they have taken, when they have taken these drugs, and what adverse effects they experienced.

In Section 11.4 we will provide experimental results showing that LRNNs can indeed learn useful soft clusters, using a scenario which is similar to the one from the previous example, in the domain of organic chemistry. Note that we could not perform experiments for the problem setting from example 14, as the required data are not publicly available for privacy reasons.

In the above example, soft clustering was essentially used to group related predicates. However, the underlying idea can also be applied to more complex relational structures. To illustrate this, the following example shows how we can group related (hyper)graphs in a similar way.

Example 15 *Let us consider the problem of predicting properties of organic molecules (e.g. toxicity) that depends on the presence of substructures from some rather large set. In this example, we will consider such substructures based on aromatic six-rings. The basic aromatic six-ring is the benzene ring, which is a ring of six carbon atoms, each connected to a hydrogen atom, connected by aromatic bonds. If some carbon atom is replaced by another atom, we speak of a* substitution*.*

If the patterns capturing classes of substructures in the molecules have the same structure, e.g. they are all aromatic six-rings with substitutions at some positions, one could in principle use probabilistic modeling to approximate them. The main idea would then be to estimate a probability distribution on the sets of substitutions, such that sets of substitutions which are jointly occurring in the individual patterns would have high probability. While such a probabilistic modeling approach is possible in principle, it would typically require us to introduce latent concepts, in which case we would have to resort to EM. Using LRNNs, on the other hand, learning latent representation patterns is straightforward. For instance, if we want to capture pairwise dependencies between the substitutions in neighboring atoms, we can first define auxiliary binary predicates of the following form:

$$w_1^{(1)} : s_1(\mathtt{carbon}, \mathtt{nitrogen})$$
$$w_2^{(1)} : s_1(\mathtt{carbon}, \mathtt{oxygen})$$
$$\dots$$

Each s_i is supposed to represent a group of substitution pairs which often appear together in discriminative substructures (because the weights will be learned in this way). Then, we can define a predicate sixRing as follows:

$$w_1^{(2)} : \mathit{sixRing}(A, B, C, D, E, F) \leftarrow \mathit{ring}(A, B, C, D, E, F) \wedge s_1(A, B) \wedge s_2(B, C) \wedge \dots s_6(F, A)$$

together with the following rule:

$$w_1 : \mathit{toxic}(M) \leftarrow \mathit{atom}(M, A) \wedge \mathit{atom}(M, B) \wedge \dots \wedge \mathit{atom}(M, F) \wedge \mathit{sixRing}(A, B, C, D, E, F)$$

Intuitively, sixRing then represents a class of substructures whose presence in a molecule suggests that it is toxic. We can similarly define predicates for five-rings and other structures. For each of the these classes of substructures, we then add a rule, whose weight encodes how predictive that substructure is of toxicity, e.g.:

$$w_2 : \mathit{toxic}(M) \leftarrow \mathit{atom}(M, A) \wedge \mathit{atom}(M, B) \wedge \dots \wedge \mathit{atom}(M, E) \wedge \mathit{fiveRing}(A, B, C, D, E) \dots$$

When learning the weights of this LRNN (based on examples of toxic and non-toxic molecules), we then simultaneously discover the appropriate weights of the auxiliary predicates (e.g. s_1, sixRing) as well as the weights of the rules that predict the target predicate toxic. In other words, we jointly learn what the latent substructures represent and how predictive they are.

Finally, as an illustration of a more elaborate use of LRNNs for learning soft clusters, we refer to the work of [171], where a method is proposed to simultaneously learn soft clusters of predicates and soft clusters of entities, based on a reified representation of predicates.

5.2.2 *Approximate Matching*

If the body of a given rule is only approximately satisfied, it often makes sense to still derive the head of that rule, but with a lower degree. Using LRNNs we can easily learn how the different ways in which the body can be approximately satisfied should affect the degree to which we want to derive the head. We refer to this modelling construct as approximate matching.

Example 16 *Let us again consider the example about predicting who has the flu. Let us consider the following rule, expressing that if* X *is in a group of 4 people who are mutual friends and all of them have flu symptoms, then* X *has the flu:*

$$w_1^{(1)} : hasFlu(X) \leftarrow clique(W,X,Y,Z) \wedge fluSymptoms(W) \wedge fluSymptoms(X) \wedge fluSymptoms(Y) \wedge fluSymptoms(Z). \tag{4}$$

The requirement that the friendship graph of W, X, Y, Z *is a clique seems unnecessarily strict. For instance, the rule is still meaningful if two of these four people are not actually friends, although in such a case we may prefer to derive the conclusion that* X *has the flu with lower certainty. In general, the more of the friendship relations are missing, the lower the certainty of the conclusion should intuitively be. This can easily be expressed using LRNNs, e.g. by defining the predicate clique as a soft concept and automatically learning the respective weights:*

$$\begin{aligned}
w_1^{(2)} &: clique(W,X,Y,Z) \leftarrow f(W,X) \wedge f(W,Y) \wedge f(W,Z) \wedge f(X,Y) \wedge f(X,Z) \wedge f(Y,Z)\\
w_1^{(3)} &: f(X,Y) \leftarrow friends(X,Y) \wedge friends(Y,X)\\
w_2^{(3)} &: f(X,Y) \leftarrow friends(X,Y)\\
w_3^{(3)} &: f(X,Y).
\end{aligned}$$

where the predicate friends is specified in the description of the examples. Note how the predicate f *enables a flexible definition of the predicate clique, and how this allows us to draw conclusions in situations which only partially match the body of the rule* (4)*. Using the activation functions from the Max-Sigmoid family for the predicates hasFlu and f, we can obtain the desired behavior.*

Other concepts which we do not describe in detail for the sake of brevity include e.g. lifted CNNs and relational auto-encoders.

5.3 EXPERIMENTS

In this section, we describe experiments performed on 78 datasets about organic molecules: the Mutagenesis dataset [172], four datasets from the predictive toxicology challenge, and 73 NCI datasets [173]. The Mutagenesis dataset contains information about 188 molecules, with labels denoting their mutagenicity. A number of published results for this dataset have relied on an extended set of features, providing additional expert knowledge about relational properties of molecules. Since we want to focus on the learning capabilities of our model, we will not rely on these additional features, and only use atom bond information. The predictive toxicology challenge dataset (PTC) [174] is composed of four datasets about molecules, labeled by their toxicity for female rats (fr) and mice (fm) and male rats (mr) and mice (mm). Each of the NCI-GI datasets contains several thousands of molecules, labeled by their ability to inhibit growth of different types of tumors. Detailed statistics of these datasets are in Table 2.

Table 2: Statistics of the three groups of molecular datasets used in the experiments. Except for the number of datasets, the remaining numbers are averages over the datasets in the given group.

	#datasets	avg. #examples	avg. #bonds	avg. #atoms
NCI	73	3031	50	23
PTC	4	342	51	25
MUTA	1	188	56	26

We compare the performance of LRNNs with the state-of-the-art[8] relational learners kFOIL [175] and nFOIL [176], which respectively combine relational rule learning with support vector machines and with naive Bayes learning. We also compare LRNNs with MLN-boost [177] and RDN-boost [114], which are both based on functional gradient boosting [178] together with Markov logic networks and relational dependency networks [119], respectively. We also attempted a comparison with CILP++ [16], but were not able to transform the datasets into the propositional representation which is used by CILP++ using the publicly available part of the CILP++ implementation. In addition we performed experiments with Aleph [179], which we used both in its abductive and inductive modes. In the abductive mode we gave Aleph the same graphlet defining rules as we give to LRNNs in the experiments with the soft clustering modelling construct that we report in Section 5.2.1. In theory, Aleph could learn definitions of crisp clusters by abduction, although it cannot learn soft clusters. In practice we found that it was not effective. Interestingly, the inductive mode of Aleph did not achieve competitive results on the NCI datasets either; it rarely exceeded majority-class accuracy.

To demonstrate the versatility of LRNNs, we perform experiments with different templates, representing some of the modeling constructs that were discussed in Section 5.2. Each time, we only make use of generic templates, ensuring that the rules that are provided are not predictive by themselves, and that the weight learning must thus create useful latent relational concepts in order to be succesful. In particular, the considered templates do not relate to any specific property of molecules and might be equally useful for other classification tasks. The idea is that useful latent relational concepts emerge from the gradient descent based weight learning process, rather than by explicit enumeration, in contrast to propositional approaches and ILP [19]. Nonetheless, in real applications the fact that declaratively specified expert knowledge can be provided is of course an important strength of LRNNs. Table 3 lists statistics of the ground neural networks for LRNNs based on the different modelling constructs.

For all the reported experiments, we set the learning rate to 0.3 and training epochs to 3000. In general, we found that training was not very sensitive to the learning rate (with the effective range being up to 0.5) as long as a sufficient number of learning steps is used. We set the learning rate relatively high so as to keep the number of necessary epochs to converge reasonable. The time[9] for training an LRNN on a standard commodity machine with one CPU was in the order of a few hours for the larger NCI-GI datasets, and in the order of a few minutes for the smaller datasets such as Mutagenesis.

5.3.1 *Soft Clustering*

We start with a simple hand-crafted LRNN template which is based on the idea of implicit soft clustering that was described in Section 5.2.1. The template defines 3 predicates for soft clusters of

8 For this specific task of molecules classification, these general relational learners have been recently surpassed by graph neural networks, which witch we compare later in Chapter 7.

9 The computation performance of the framework has been recently greatly improved with the new version described in Chapter 7, utilizing, among others, an optimization technique detailed later in Chapter 9.

Table 3: Average sizes of ground neural networks (average number of atom neurons per network) corresponding to the LRNNs based on the different modelling constructs.

	Soft Clusters	App. Matching	Atom Embeddings	Charge	Charge+Soft
NCI	520	1145	1241	1373	1396
PTC	598	1619	1292	1414	1435
MUTA	696	1800	1338	1445	1482

atom types and 2 predicates for soft clusters of bond types. The predicates *atgr1*, *atgr2*, and *atgr3*, representing soft clusters of atom types are defined by considering one rule for every atom type occurring in the dataset, e.g.:

$$w_1^{(1)} : atgr1(\mathrm{X}) \leftarrow o(\mathrm{X})$$
$$w_2^{(1)} : atgr1(\mathrm{X}) \leftarrow br(\mathrm{X})$$
$$\ldots$$

Similarly, the predicates *bondgr1* and *bondgr2* representing soft clusters of bond types are defined by considering one rule for every bond type occurring in the dataset. These predicates are then used to define rules for different types of atom chains of length 3, one for each group choice for each of the 3 atoms' soft clusters and each of the 2 bonds' soft clusters, i.e. 243 rules in total:

$$\begin{aligned} w^{(2)}_{(1,1,1;1,1)} : chain3 \leftarrow\ & atgr1(\mathrm{X}) \wedge bond(\mathrm{X},\mathrm{Y},\mathrm{B1}) \wedge atgr1(\mathrm{Y}) \wedge bond(\mathrm{Y},\mathrm{Z},\mathrm{B2}) \\ & \wedge atgr1(\mathrm{Z}) \wedge bondgr1(\mathrm{B1}) \wedge bondgr1(\mathrm{B2}), \\ & \ldots \\ w^{(2)}_{(3,3,3;2,2)} : chain3 \leftarrow\ & atgr3(\mathrm{X}) \wedge bond(\mathrm{X},\mathrm{Y},\mathrm{B1}) \wedge atgr3(\mathrm{Y}) \wedge bond(\mathrm{Y},\mathrm{Z},\mathrm{B2}) \\ & \wedge atgr3(\mathrm{Z}) \wedge bondgr2(\mathrm{B1}) \wedge bondgr2(\mathrm{B2}). \end{aligned}$$

The predicate *chain3* then represents the, possibly varying, learning target for each of the molecular datasets (e.g. toxicity or mutagenicity). A comparison between the results obtained with this LRNN template and those obtained with kFoil, nFoil, MLN-boost and RDN-boost is shown in Figure 36. As can clearly be seen from this figure, the LRNN method consistently outperforms the four baselines.

The learned weights of the rules defining the predicates *atgr1*, *atgr2* and *atgr3* can be interpreted as membership degrees of the atom types to the three soft clusters. These degrees might be interpreted as defining a three-dimensional vector space embedding of the atom types. The first two principal components of these embeddings, for different atom types from the Mutagenesis dataset, are shown in Figure 35. Note that the atom types in the Mutagenesis dataset have been enriched with contextual information, which is why there are different atom types *c21*, *c22*, ..., *c195* which all refer to carbon atoms. The LRNNs are not given any explicit information about how these different atom types are related, and thus have to reconstruct this information from the available training data. It is therefore interesting to see that in the embedding from Figure 35, the nitrogen atoms are mostly in the top left corner, carbons are mostly in the bottom right corner and the rest of the atoms are around the center of the plot (where some further noticeable patterns can be observed, such as halogen atoms being clustered together).

Next we demonstrate the importance of relational information in the molecule classification tasks. Specifically, we show that a model which captures conformations of particular atom types leads to better classification accuracy than a model which is only based on a soft clustering of atom types. To this end, we created 3 templates with increasing complexity. The first template is based purely

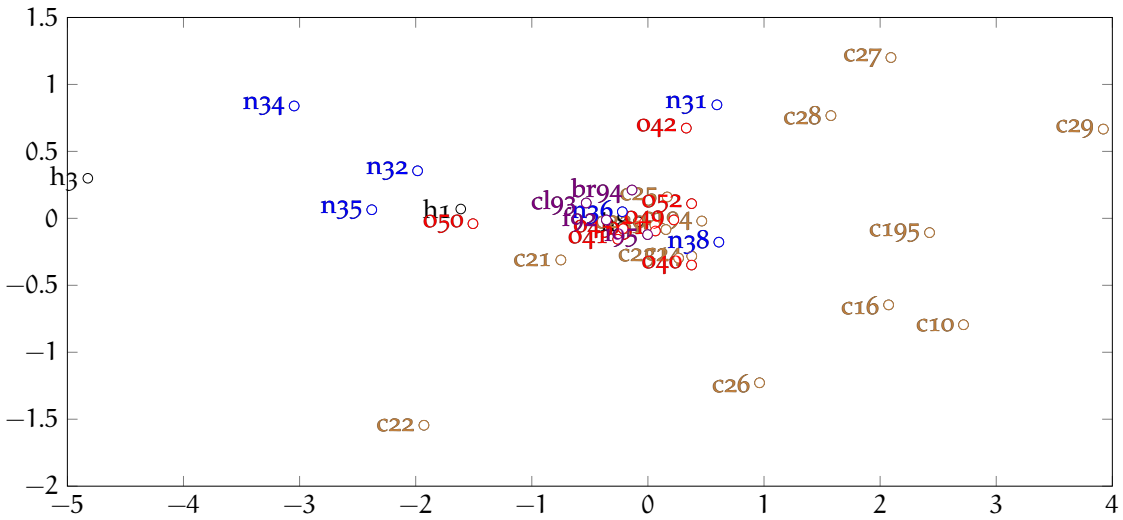

Figure 7: PCA projection of vector embeddings of atom types corresponding to the learned weights of soft clusters in the Mutagenesis dataset. Brown denotes the group 14 of the periodic table (carbon group), blue the group 15 (pnicotgens), red the group 16 (chalcogens), violet the group 17 (halogens) and black the group 1 (hydrogen).

on soft clustering, i.e. it only considers relational chains of size 0. For each $i \in \{1,2,3\}$ we consider the following rules, one for each soft cluster of atom types:

$$w^{(2)}_{(i)} : \mathit{chain1} \leftarrow \mathit{atgr}i(X)$$

The second template involves relational chains composed of two atoms. It contains the following rule for each $i,j,k \in \{1,2,3\}$:

$$w^{(2)}_{(i,j,k)} : \mathit{chain2} \leftarrow \mathit{atgr}i(X) \wedge \mathit{bond}(X,Y,B1) \wedge \mathit{atgr}j(Y) \wedge \mathit{bondgr}k(B1)$$

Finally, we consider the template with the *chain3* predicate, describing chains of 3 atoms, which we used in the previous experiment.

Results for the three templates *chain1*, *chain2* and *chain3* are shown in Figure 9. While most of the performance is clearly due to the use of soft clustering, using non-trivial relational chains does lead to improved predictive accuracy. It is evident from the graph that relational chains of length greater than 1 are better than relational chains of length 1. The difference between chains of lengths 2 and 3 is smaller but still statistically significant ($p = 0.002$, using binomial test).

5.3.2 *Alternative Modeling Constructs*

Beyond learning soft clusters of atom types and bond types, a wide variety of other constructs can be used to solve the considered learning tasks. In this section, we briefly discuss a number of these alternatives.

5.3.2.1 *Learnable Numerical Transformations*

In the previous section, we showed how the soft clusters that are learned by an LRNN can be interpreted as vector space embeddings. Conversely, we can also generate soft clusters from a given pre-trained embedding. To demonstrate this idea, we will use a simple vector space representation, which encodes for each atom type how its valence electrons are distributed across the *s,p,d,f* orbitals. For instance, the oxygen atom type O with electron configuration $(1s2)[2s2,2p4]$ is encoded as the vector $O := [2,4,0,0]$. To construct an LRNN that takes advantage of this external knowledge, we actually use the similarity degrees induced by these vectors, rather than the vectors themselves. For

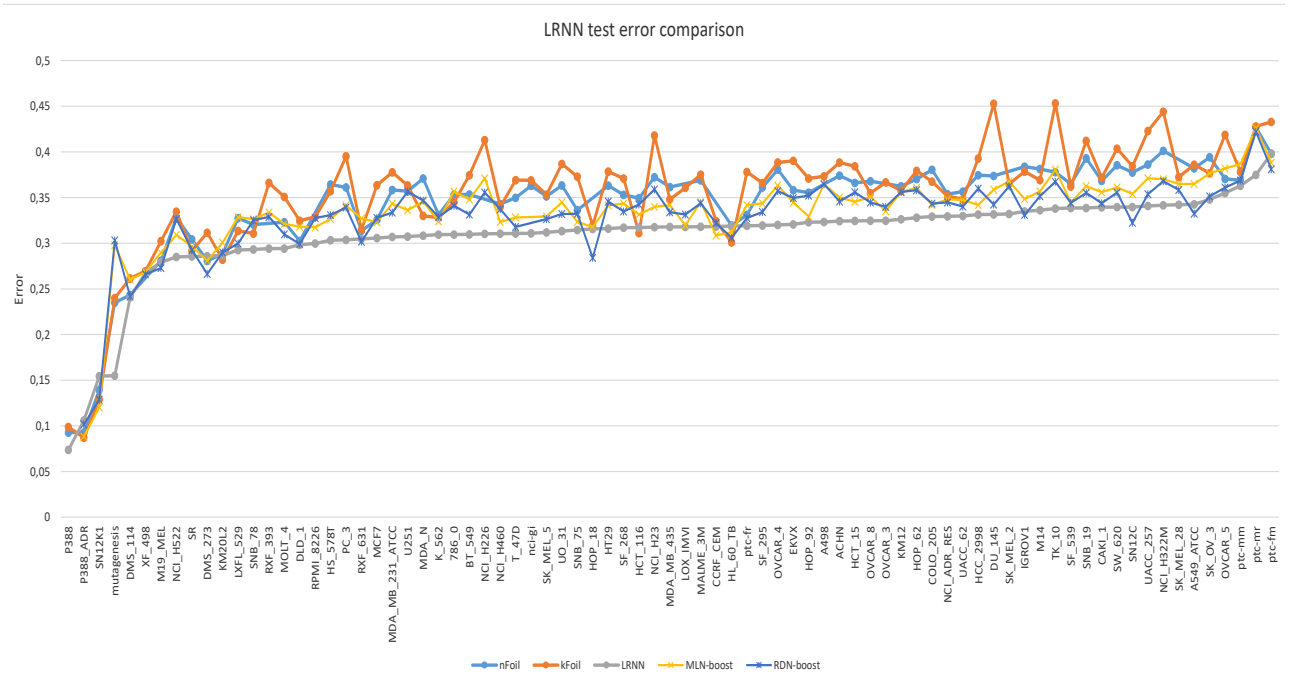

Figure 8: Prediction errors of LRNNs, kFOIL, nFOIL, MLN-boost and RDN-boost measured by cross-validation on 78 datasets about organic molecules.

this experiment, we measure the similarity between two atom types as the cosine between their vector representations. We then construct the LRNN template as follows.

For each pair of atom-types (a_1, a_2) with similarity degree s, we add the following ground fact:

$$1.0 : \mathit{Similar}(a_1, a_2, s)$$

We also add rules which encode a *learnable transformation* of the similarities into a score that is useful for the considered predictive task:

$$\begin{aligned} w_{-1} &: \mathit{Similar}(X, Y) \leftarrow \mathit{Similar}(X, Y, S), S \geqslant -1.0 \\ w_{-0.9} &: \mathit{Similar}(X, Y) \leftarrow \mathit{Similar}(X, Y, S), S \geqslant -0.9 \\ &\ldots \quad \ldots \\ w_{0.9} &: \mathit{Similar}(X, Y) \leftarrow \mathit{Similar}(X, Y, S), S \geqslant \quad 0.9 \end{aligned}$$

We then randomly sample three atom type vectors as $\mathtt{prototype1} := [2, 0, 0, 14]$, $\mathtt{prototype2} := [1, 0, 10, 0]$, $\mathtt{prototype3} := [2, 6, 10, 0]$ and modify the definition of atom groups to reflect the similarity to one of these prototypes:

$$\begin{aligned} w_1^{(1)} &: \mathit{atgr1}(X) \leftarrow \mathit{Similar}(X, \mathtt{prototype1}) \\ &\ldots \quad \ldots \\ w_3^{(1)} &: \mathit{atgr3}(X) \leftarrow \mathit{Similar}(X, \mathtt{prototype3}) \end{aligned}$$

We will refer to this method as *atomEmbeddings*.

As an alternative, we also tested how well the atom groups can be induced from the charge of each atom within a given molecule. As this information is only available in the NCI datasets, for this variant we do not consider the predictive toxicology datasets. To generate the atom groups, we

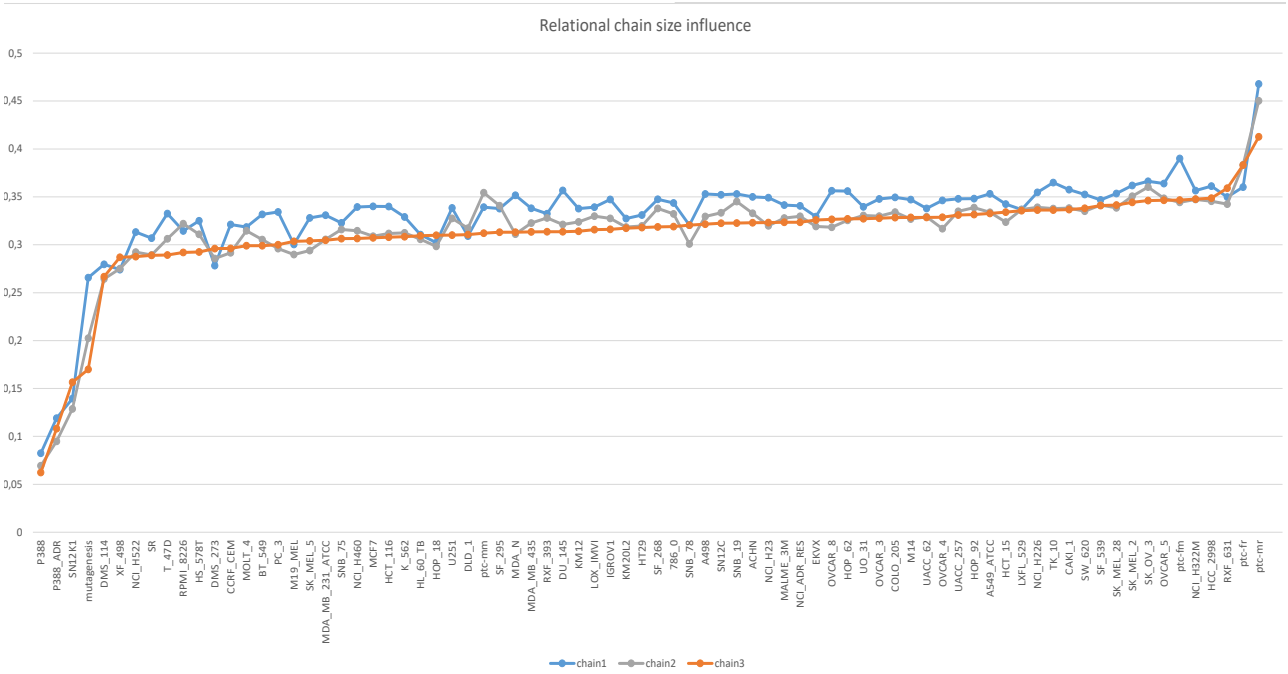

Figure 9: Test errors of three LRNN templates with growing relational chain sizes as measured by cross-validation on 78 datasets about organic molecules.

again use a learnable transformation, but this time based on partial atom charges. Noting that the partial atom charges in the datasets are always between -1 and 1, this can be done as follows:

$$\begin{aligned} w_{-1}: \ & atgr1(X) \leftarrow Charge(X, C), C \geqslant -1.0 \\ w_{-0.9}: \ & atgr1(X) \leftarrow Charge(X, C), C \geqslant -0.9 \\ \ldots \quad & \ldots \\ w_{0.9}: \ & atgr1(X) \leftarrow Charge(X, C), C \geqslant \ \ 0.9 \end{aligned}$$

This method will be referred to as *atomCharge*. Finally we also tried to combine the soft cluster definition of atom groups with the definition based on atom charges, the results of which can be seen as *charge+softCluster* in Figure 10. To construct these combined LRNNs, we simply merge the definitions of the atom groups *atgr*i from both of the LRNNs.

The experimental results for the considered methods are depicted in Figure 10. The test errors of the LRNNs based purely on the partial charges are higher than the test errors of LRNNs based purely on soft clustering, which was to be expected. Indeed, similar results for relational features based on atom types or partial charges have been previously reported [180]. However, the fact that the combined LRNNs did not outperform the soft clustering LRNNs is more surprising. It suggests that the soft clusters built from the extended atom types present in the NCI datasets (e.g. *c21*, *c22*, ...) may already capture the information present in the information about partial charges.

5.3.2.2 *Approximate Matching*

The aim of this experiment is to demonstrate the capability of LRNNs to capture structural similarities within the relational features. Following the idea of the approximate matching construct (see Section 5.3.2.2), we create a more flexible variant of the relation representing the bond between two atoms, such that more complex structural patterns can be matched, with different degrees of similarity. We use the template with chains of 3 consecutive atoms from Section 5.3.1, but replace

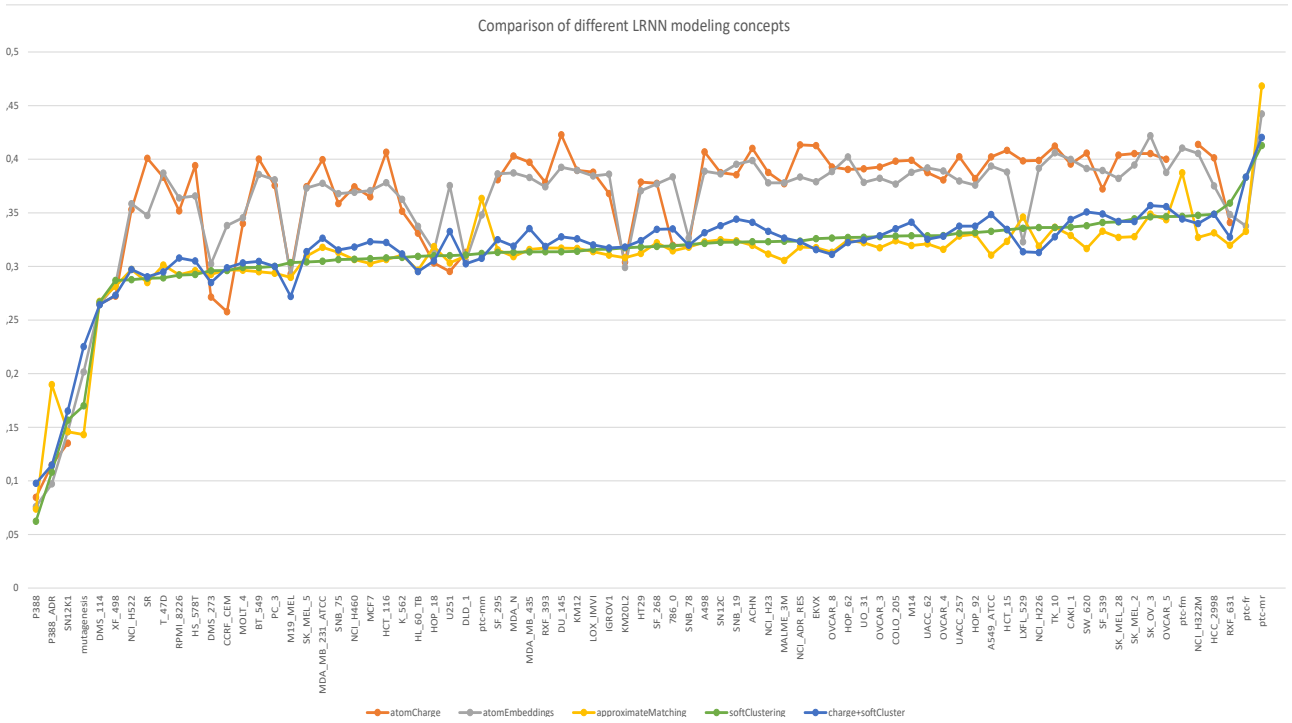

Figure 10: Prediction errors of the various introduced LRNN modeling concepts as measured by cross-validation on 78 datasets about organic molecules.

the predicate *bond* by a new predicate *bondK*. This new predicate is then defined in terms of the predicate *bond*, as follows:

$$w_1 : bondK(X, Y, B) \leftarrow bond(X, Y, B)$$
$$w_2 : bondK(X, Y, B) \leftarrow bond(X, Z, B), bond(Z, Y, B2)$$

The results of the experiments with this template are displayed as *approximateMatching* in Figure 10. Although the differences are small, the *approximateMatching* method obtained statistically significantly better accuracies than the LRNNs which only used relational chains of length 3 with soft clustering ($p = 0.008$, using binomial test).

5.4 CONCLUSIONS

In this Chapter, we have introduced LRNNs, a framework for learning from relational data. Similarly to lifted probabilistic frameworks such as Markov logic and Problog, learned LRNN models are represented as sets of weighted first-order formulas. However, while Markov logic and Problog models serve as templates for constructing probabilistic graphical models, LRNN models serve as templates for constructing feedforward neural networks. This means that we can employ neural network learning, based on backpropagation, to efficiently discover latent relational structures from structured training data. Thanks to the use of first-order logic rules, we can easily specify what kind of latent structures we want the network to explore. In the experimental results, we have shown that very general rules, essentially indicating that we want to find predictive groups of atom types and predictive groups of bond types, allow us to achieve state-of-the-art predictive accuracies on various datasets about organic molecules. Furthermore, we have discussed and evaluated several other modeling constructs, e.g. based on learning latent groups of graph patterns, approximate matching of relational patterns, and using pre-trained vector space embeddings for relational learning.

6

STRUCTURE LEARNING OF LRNNS

As introduced in the previous Chapter 5, Lifted Relational Neural Networks (LRNNs) describe relational domains using weighted relational logic rules which act as templates for constructing neural networks from varying relational structures. While in Chapter 5 we have shown that using LRNNs can lead to state-of-the-art results in various relational learning tasks (Section 5.3), these results depended on hand-crafted rules (albeit very generic). In that case, only the weights of the relational rules have to be learned from training data, which can be accomplished using a variant of back-propagation (Section 5.1.6). The use of hand-crafted rules offers a natural way to incorporate domain knowledge and guide the learning process in various ways (Section 5.2). In some applications, however, such domain knowledge is lacking.

In this chapter, we extend the framework of LRNNs with structure learning, thus enabling for a fully automated learning process, where both the rules and their weights are learned from data, akin to the structure learning task from SRL (Section 3.4). Similarly to many ILP methods (Section 3.3.1), our structure learning algorithm proceeds in an iterative fashion by top-down searching through the hypothesis space of all possible Horn clauses, considering the predicates that occur in the training examples as well as invented soft concepts entailed by the best weighted rules found so far. In each iteration, it may either learn a set of rules that intuitively correspond to a new layer of a neural network template or to learn a set of rules that intuitively correspond to creating new connections among existing layers, a strategy which we refer to as stacked structure learning. The rules that are added in a given iteration either define one of the target predicates, or they define a new predicate that may depend on predicates that were "invented" at earlier layers as well as on predicates from the considered domain. Since the actual meaning of these predicates depends on both the learned rules and their associated weights, structure learning is alternated with weight learning. Intuitively, this means that the definitions of predicates defined in earlier layers can be fine-tuned based on the rules which are added to later layers.

We present experimental result which show the ability to automatically induce useful hierarchical soft concepts, leading to deep LRNNs with a predictive power comparable to that of LRNNs based in the hand-crafted rules. We believe that this makes LRNNs a particularly convenient framework for learning in relational domains, without any need for prior knowledge nor for any extensive hyperpertuning. Somewhat surprisingly, we find that LRNNs with learned rules are often more compact than those with hand-crafted rules. Finally, we also present some initial results which suggest that the use of logical rules enable LRNNs to efficiently learn concepts which neural networks normally struggle with.

6.1 STRUCTURE LEARNING

In this section we describe a structure learning algorithm for LRNNs[1]. The algorithm receives a list of training examples and a list of training queries, and it produces a LRNN. For simplicity, we will assume that constants are only used as identifiers of objects. In particular, we will assume that

1 It is advisable to recall the basic LRNN notions from Section 5.1 of the previous Chapter 5 before continuing.

attribute values are represented using unary literals, e.g. we would use $red(o)$ instead of $color(o, red)$. Besides that we do not put any restrictions on the structure of the training examples.

6.1.1 *Structure of the Learned LRNNs*

The structure learning algorithm will create LRNNs having a generic "stacked" structure which we now describe. First, there are rules that define d new predicates, representing *soft clusters* (Section 5.2.1) of unary predicates from the dataset. These can be thought of as the first layer of the LRNN, where the weighted facts from the dataset comprise the zeroth layer. For instance, if the unary predicates in the dataset are $A, B, \ldots, Z$ then the LRNN will contain the following rules:

$$\begin{array}{llll}
w_{a_1} : \alpha_1^1(X) \leftarrow A(X) & w_{b_1} : \alpha_1^1(X) \leftarrow B(X) & \ldots & w_{z_1} : \alpha_1^1(X) \leftarrow Z(X) \\
w_{a_2} : \alpha_2^1(X) \leftarrow A(X) & w_{b_2} : \alpha_2^1(X) \leftarrow B(X) & \ldots & w_{z_2} : \alpha_2^1(X) \leftarrow Z(X) \\
\ldots & \ldots & \ldots & \ldots \\
w_{a_d} : \alpha_d^1(X) \leftarrow A(X) & w_{b_d} : \alpha_d^1(X) \leftarrow B(X) & \ldots & w_{z_d} : \alpha_d^1(X) \leftarrow Z(X)
\end{array}$$

Here each α_j^i is a latent predicate representing a soft cluster, the index i denotes the layer in which it appears (in this case, the first layer) and j indexes the individual soft clusters in that level.

In general, the second layer will consist of two types of rules. First, there may be rules introducing new latent predicates. In contrast to the unary predicates that were introduced in the first layer, here the latent predicates could be also of higher arity, although in practice an upper bound will be imposed for efficiency reasons. In the body of these rules, we may find predicates from the dataset itself, or latent predicates that were introduced in the first layer. The new latent predicates introduced in these rules may then be used in the bodies of rules in subsequent layers. Second, there may also be rules that have a predicate from the dataset in their head. These will typically be rules that were learned to predict the target predicates that we want to learn.

Example 17 *For instance, in datasets of molecules, unary predicates can be used to represent types of atoms, such as* carbon *or* hydrogen*. An example of a possible second layer rule is:*

$$w_{p_1} : \quad p_1(X, Y) \quad \leftarrow \quad bond(X, Y) \wedge \alpha_1^1(X) \wedge \alpha_2^1(Y)$$

Here p_1 is assumed to be one of the predicates from the dataset. Second layer rules that introduce a new latent predicate could look as follows.

$$\begin{array}{lll}
w_{1,1}^2 : & \alpha_1^2(V1, V2) & \leftarrow \quad bond(V1, V2) \wedge \alpha_1^1(V1) \wedge \alpha_1^1(V2) \\
w_{1,2}^2 : & \alpha_1^2(V1, V3) & \leftarrow \quad bond(V1, V2) \wedge bond(V2, V3) \wedge \alpha_1^1(V1) \wedge \alpha_1^1(V3)
\end{array}$$

The actual intuitive meaning of the predicate α_1^2 will depend on the weights $w_{1,1}^2$, $w_{1,2}^2$. For instance, if both are large enough, the (atom neurons corresponding to the) predicate will have high output whenever its arguments correspond to two atoms which are either one or two steps apart from each other in the molecule, and which have sufficiently high membership in the soft cluster α_1^1.

Any higher layers have a similar structure to the second layer, where the n^{th} layer contains rules whose bodies only contain predicates from layers 0 to $n-1$, and whose heads either contain a target predicate or introduce a new latent predicate.

6.1.2 *Structure Learning Algorithm*

The structure learning algorithm (Algorithm 1) iteratively constructs LRNNs that have the structure described in the previous section. It alternates weight learning steps with rule learning steps[2]. In the

2 Variants of this strategy are employed by many structure learning algorithms in the context of statistical relational learning, e.g. [181–183].

Algorithm 1 General schema of structure learning

1: $\mathcal{E} \leftarrow$ learning examples
2: $d \leftarrow$ latent concepts' dimension
3: $\mathcal{W}, \mathcal{V}, \mathcal{R} \leftarrow \varnothing$
4: $\mathcal{R} \leftarrow \mathtt{createLayer1Rules}(\mathcal{E}, d)$
5: $\mathcal{W} \leftarrow \mathtt{initWeights}(R)$
6: $(\mathcal{F}, \mathcal{V}) \leftarrow \mathtt{weightedFacts}(\mathcal{E}, R, W)$
7: **while** $\neg$StoppingCriterion **do**
8: $\quad \mathtt{bestRule} \leftarrow \mathtt{ruleLearning}(\mathcal{F}, \mathcal{V}, \mathcal{R})$
9: $\quad \mathtt{bestRules} \leftarrow \mathtt{predicateInvention}(\mathtt{bestRule})$
10: $\quad \mathcal{R} \leftarrow \mathcal{R} \cup \mathtt{bestRules}$
11: $\quad \mathcal{W} \leftarrow \mathtt{trainWeights}(\mathcal{R}, \mathcal{E}, \mathcal{W})$
12: $\quad (\mathcal{F}, \mathcal{V}) \leftarrow \mathtt{weightedFacts}(\mathcal{E}, \mathcal{R}, \mathcal{W})$
13: **return** $(\mathcal{R}, \mathcal{W})$

weight learning steps, the algorithm uses stochastic gradient descent to minimize the squared loss of the LRNN by optimizing the weights of the rules, as described in Section 5.1.6 of the previous Chapter 5. In the rule learning steps, the algorithm fixes the weights of all rules which define latent predicates and it searches for some *good* rule R. This rule R should be such that the squared loss of the LRNN decreases after we add R to it and and after we retrain the weights of all rules with non-latent head predicates. Next we describe this algorithm in detail.

The first step of the structure learning algorithm (lines 4–5) is the construction of the first level of the LRNN, which defines the unary predicates representing soft clusters of object properties, as described in Section 6.1.1.

After the first step, the algorithm repeats the following procedure for a given number of iterations or until no suitable rules can be found anymore. It fixes the weights of all rules defining latent predicates (line 6). Then it runs a beam search algorithm searching through the space of possible rules[3] (line 8). The scoring function which is used by the beam search algorithm is computed as follows. Given a rule R, the algorithm creates a copy of the current LRNN to which the given candidate rule R is added. It then optimises the log-loss of this new LRNN (which corresponds to maximum-likelihood estimation for logistic regression), training just the non-fixed weights, i.e. the weights of the rules with non-latent predicates in their heads. The score of the rule R is then defined to be the log-loss after training the non-fixed weights. The reason why we do not retrain all weights of the LRNN when checking score of a rule R are efficiency considerations because training the weights of the whole LRNN corresponds to training a deep neural network. After the beam search algorithm finishes, the rule R^* that it returned is added to the original LRNN.

Note that R^* contains one of the target predicates in its head. However, in addition to adding R^*, we also add a set of related rules that have latent predicates in their head (line 9), as follows. Here, we will assume for simplicity that all latent predicates have the same arity k, but the same method can still be used when the latent predicates are allowed to have different arities. Let i be the highest index such that R^* contains a latent predicate of the form α_j^i (i.e. a latent predicate from layer i) in its body, where we assume $i = 1$ if R^* does not contain any latent predicates . Then for each latent predicate α_j^{i+1} from the $(i+1)$-th layer, the algorithm adds to the LRNN all rules which have $\alpha_j^{i+1}(V_1, \ldots, V_k)$ in the head and which can be obtained by unifying $V_1, \ldots, V_k$ with the variables in R^*. This process is illustrated in the following example.

3 The space of rules is defined by two user-specified constraints: maximum rule length and maximum number of variables in a rule.

Example 18 *Revisiting the example of molecular datasets, let* $R^* = p(A,B) \leftarrow bond(A,B) \wedge \alpha_2^1(A) \wedge \alpha_5^2(B)$ *and let* $k = 1$. *Then the algorithm will add the following latent-predicate rules:*

$$
\begin{array}{lllll}
w^3_{1,1}: & \alpha^3_1(V_1) & \leftarrow & bond(V_1,B) \wedge \alpha^1_2(V_1) \wedge \alpha^2_5(B) \\
w^3_{1,2}: & \alpha^3_1(V_1) & \leftarrow & bond(A,V_1) \wedge \alpha^1_2(A) \wedge \alpha^2_5(V_1) \\
w^3_{2,1}: & \alpha^3_2(V_1) & \leftarrow & bond(V_1,B) \wedge \alpha^1_2(V_1) \wedge \alpha^2_5(B) \\
w^3_{2,2}: & \alpha^3_2(V_1) & \leftarrow & bond(A,V_1) \wedge \alpha^1_2(A) \wedge \alpha^2_5(V_1) \\
\ldots & \ldots & & \ldots \\
w^3_{d,1}: & \alpha^3_d(V_1) & \leftarrow & bond(V_1,B) \wedge \alpha^1_2(V_1) \wedge \alpha^2_5(B) \\
w^3_{d,2}: & \alpha^3_d(V_1) & \leftarrow & bond(A,V_1) \wedge \alpha^1_2(A) \wedge \alpha^2_5(V_1)
\end{array}
$$

Note that the algorithm has to add the new rules to the layer 3 because R^* *already contained predicates from the layer 2.*

After the LRNN has been extended by all these rules obtained from R^*, the weights of all the rules, including those corresponding to latent predicates, are retrained using stochastic gradient descent (line 11). Note that typically there will be some latent predicates which are not used in any rules; their weights are not considered during training. Subsequently, the algorithm again fixes the weights of the rules corresponding to the latent predicates, and repeats the same process to find an additional rule. This is repeated until a given stopping condition is met.

6.2 EXPERIMENTS

In this section we describe the results of experiments performed with the structure learning algorithm on a real-life molecular dataset and on a difficult artificial learning problem.

6.2.1 *Molecular Datasets*

We performed experiments on 72 NCI datasets [173], each of which contains several thousands of molecules, labeled by their ability to inhibit the growth of different types of tumors. We compare the performance of the proposed LRNN structure learning method with the best previously published LRNNs, which contain large generic, yet manually constructed weighted rule sets (described in Section 5.3). For further comparison, we also include the relational learners kFOIL [175] and nFOIL [176], which respectively combine relational rule learning with support vector machines and with naive Bayes learning.

The results are shown in Figure 11 and Figure 12. The automatically learned LRNNs outperform both kFOIL and nFOIL in terms of predictive accuracy (measured using cross-validation). The learned LRNNs are also competitive with the manually constructed LRNNs from Section 5.3, although they do not outperform them. They are slightly worse than the largest of the manually constructed LRNNs, based on graph patterns with 3 vertices, enumerating all possible combinations of soft cluster types of the three atoms and soft cluster types of the two bonds connecting them. Figure 13 displays statistics of the learned LRNN rule sets. These statistics show that the structure learner turned out to produce quite complex LRNNs having multiple layers of invented latent predicates.

The weights of the rules defining the latent predicates in the first layer of the LRNN can be interpreted as coordinates of a vector-space embedding of the properties (atom types in our case). In Figure 14, we plot the evolution of these embeddings as new rules are being added by the structure learning algorithm. The left panel of Figure 14 displays the evolution of the embeddings of atom types after these have been pre-trained using an unsupervised method which was originally used for statistical predicate invention in [171]. The right panel of the same figure displays the evolution

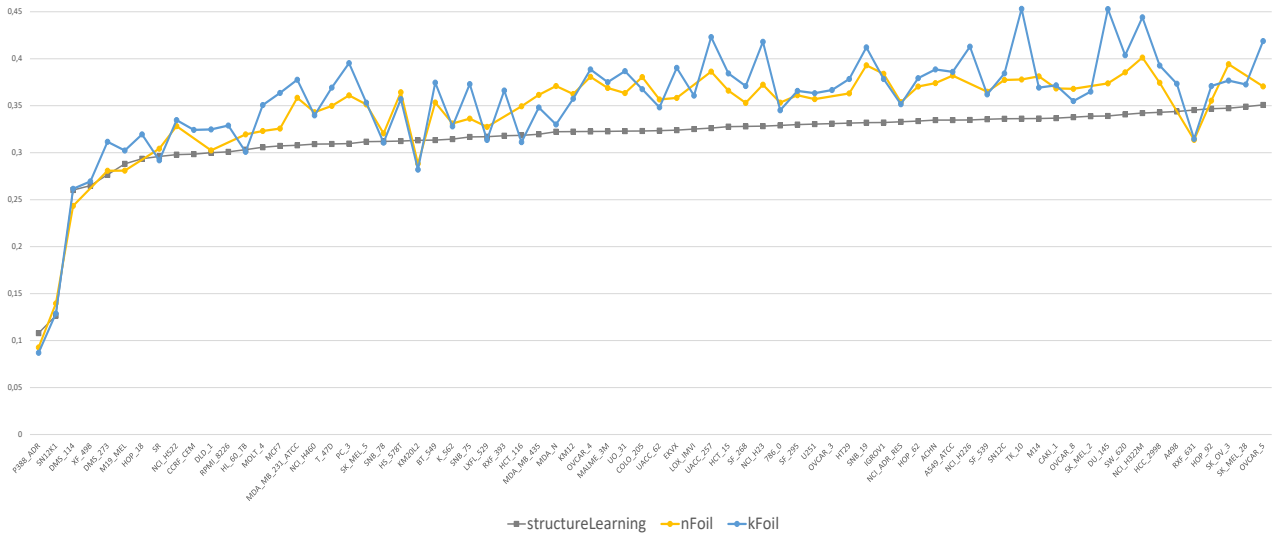

Figure 11: Comparison of crossvalidated test errors of LRNNs produced by structure learning with nFoil and kFoil learners as baselines.

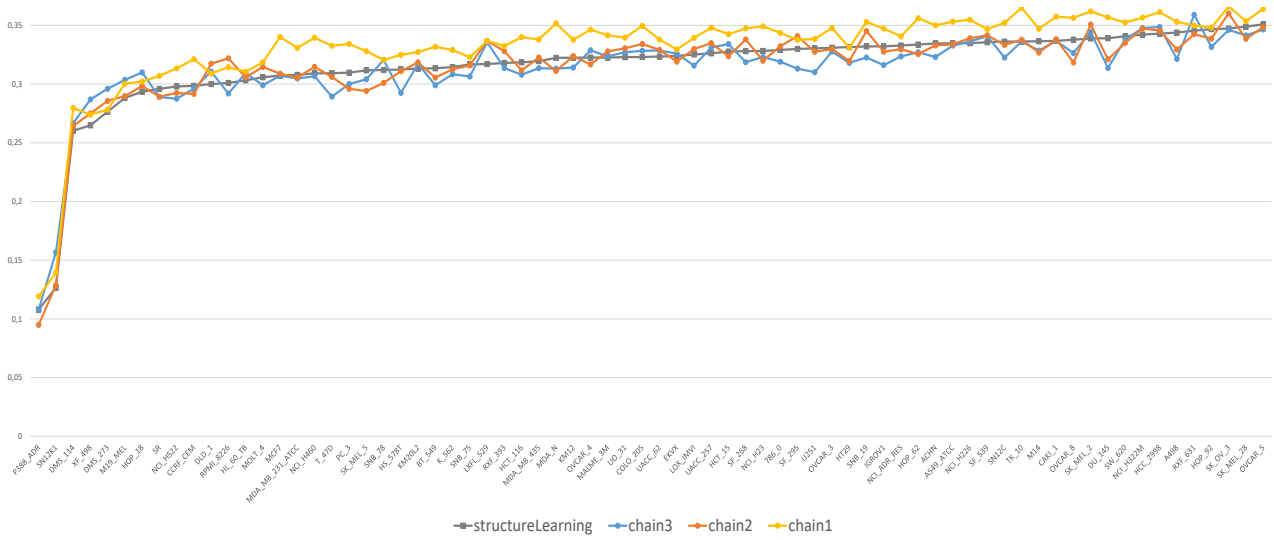

Figure 12: Comparison of test errors of LRNNs produced automatically by structure learning with 3 hand-crafted LRNNs with varying lengths of chain patterns from [97].

of the embeddings when starting from random initialization without any unsupervised pre-training. What can be seen from these figures is how, as the model becomes more complex, the atom types start to make more visible clusters. Interestingly and perhaps somewhat against intuition, the use of the unsupervised pre-training seemed to consistently decrease predictive performance (we omit details for the sake of brevity).

6.2.2 *A Hard Artificial Problem*

We consider the following variant of graph colorability, which can be seen as a relational generalization of the problem of learning the XOR function. For a graph, where each node may take on different "shades" $\{\mathtt{sh}_1 \ldots \mathtt{sh}_n\}$ of colors $\{\mathtt{col}_1 \ldots \mathtt{col}_m\}$ that are not observed (i.e. it is not given to which color each shade corresponds), the task is to learn to classify graphs that are correctly colored, i.e. where each edge in the graph connects two nodes of shades of different colors. In this problem, learning a correct representation of the colors (as sets of shades) is completely decorrelated from the target, but to correctly learn to classify correctly colored graphs, we need to learn some such

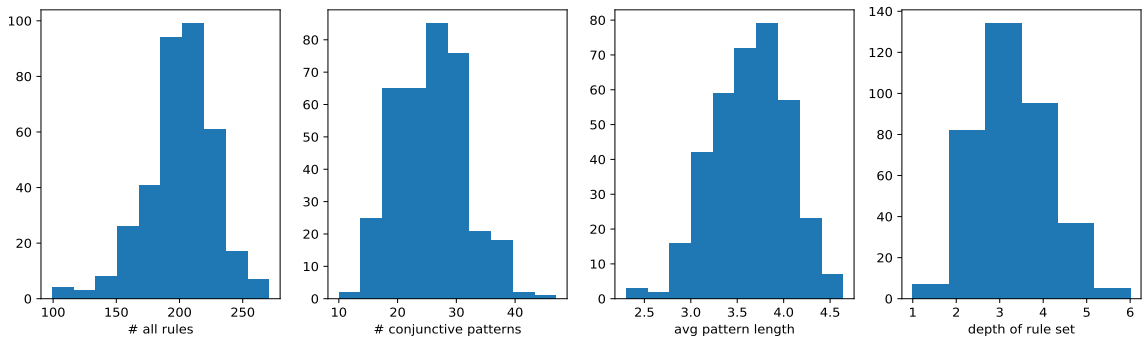

Figure 13: Statistics of the learned LRNN rule sets from experiments with the 72 NCI datasets. We display (i) the number of rules (including zeroth layer soft clusters), (ii) the number of conjunctive rules (patterns) learned, (iii) the average length of these rules (patterns), and (iv) the overall number of layers (depth of template).

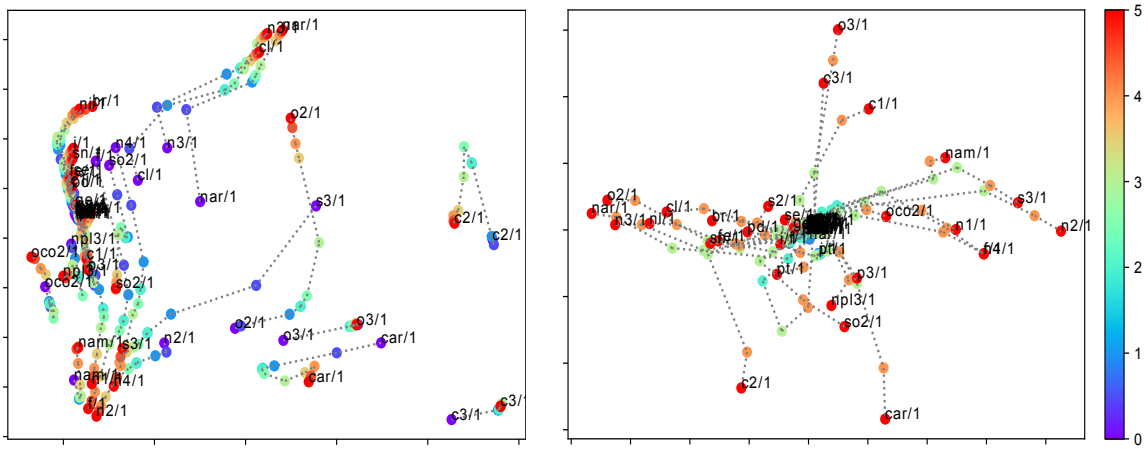

Figure 14: PCA projection of evolution of atom embeddings during first 6 iterations (denoted by colors) of structure learning of a LRNN, with initialization based on unsupervised pre-training (left) and with completely random initialization (right).

color concepts. An ideal learned LRNN correctly solving the problem could be very compact, for instance for 3 colors if might look like

$$
\begin{aligned}
w_1 : \mathtt{notCorrectlyColored} &\leftarrow \mathtt{cl}_0(X), \mathtt{edge}(X,Y), \mathtt{cl}_0(Y) \\
w_2 : \mathtt{notCorrectlyColored} &\leftarrow \mathtt{cl}_1(X), \mathtt{edge}(X,Y), \mathtt{cl}_1(Y) \\
w_3 : \mathtt{notCorrectlyColored} &\leftarrow \mathtt{cl}_2(X), \mathtt{edge}(X,Y), \mathtt{cl}_2(Y)
\end{aligned}
$$

together with rules defining the color concepts $\mathtt{cl}_0$, $\mathtt{cl}_1$ and $\mathtt{cl}_2$.

Initial experiments with these artificial problems showed that the structure learning algorithm is able to learn appropriate LRNNs. The rule sets were typically different from the rule set shown above but they also encoded correct solutions that were comparably short. The performance results, shown in Table 4, suggest that the LRNN structure learning method is able to efficiently produce accurate and compact solutions without extensive hyper-tuning.

6.3 RELATED WORK

The structure learning strategy that we employ in the methods presented in this paper is in many respects similar to structure learning methods from statistical relational learning such as [181–183]. However, what clearly distinguishes it from all these previous SRL approaches is its ability to automatically induce hierarchies of latent concepts. In this respect, it is also related to meta-interpretive

Table 4: Results of the structure learning algorithm (SL) on the graph coloring classification problem. Reported statistics are majority train error, training error, and ratio of cases with zero learning error. All problems were run 10 times with different random initialization seeds.

#colors-#shades	majority error	training error	% of perfect solutions
2c-1s	0.5	0.025	0.9
2c-2s	0.5	0.0	1
3c-1s	0.33	0.0	1
3c-2s	0.33	0.014	0.6
3c-3s	0.33	0.111	0.4
4c-1s	0.25	0.1375	0.0
4c-2s	0.25	0.160	0.0
4c-3s	0.25	0.129	0.1
4c-4s	0.25	0.044	0.1

learning [71]. However, meta-interpretive learning is only applicable to the learning of crisp logic programs. The structure learning approach is also related to works on refining architectures of neural networks [184, 185]. However, from these it differs in its ability to handle relational data.

6.4 CONCLUSIONS

In this chapter we have introduced a method for learning the structure of LRNNs, capable of learning deep weighted rule sets with invented latent predicates. The predictive accuracies obtained by the learned LRNNs were competitive with results that we obtained in our previous work using manually constructed LRNNs. The method presented in this chapter therefore has the potential to make LRNNs useful in domains where it would otherwise be difficult to come up with a rule set manually. It also makes the adoption of LRNNs by non-expert users more straightforward, as the proposed method can learn competitive LRNNs without requiring any user input (besides the dataset).

7

DIFFERENTIABLE LOGIC PROGRAMMING WITH LRNNS

The necessity to specify the relational rules can be possibly seen as a considerable limitation, or unfair advantage, of LRNNs, when viewed as "knowledge-based" models, as compared against generic, "knowledge-agnostic" models, such as classic deep neural networks. In response, the structure learning method introduced in the previous Chapter 6 was developed in order to skip the necessity of specifying these rules. In this chapter, we take a somewhat opposite approach and clearly demonstrate that the weighted relational rules do not necessarily encode any specific domain knowledge, but may be simply used to guide the learning process in a very generic way, akin to standard (deep) machine learning practices. Particularly, we view the LRNN templating as means for *differentiable logic programming*, and show how the weighted relational rules may actually be used to directly encode existing deep learning architectures through an exact correspondence.

Moreover we show how the LRNN templating can, thanks to the used declarative Datalog abstraction (Section 3.2), result into more compact and elegant learning programs, in contrast with the existing procedural deep learning paradigms operating more directly on the computational graph level. Consequently, we demonstrate on *practical examples*[1] that even very simple LRNN programs can be used to efficiently capture a wide range of deep learning models, such as MLPs and CNNs, with a particular focus on the most recently popular Graph Neural Networks (GNNs).

Additionally, we show how even the most contemporary GNN models can be not only covered, but also easily extended within LRNNs towards higher relational expressiveness. In the experiments, we then demonstrate correctness and also computation efficiency through comparison against specialized GNN deep learning frameworks, while shedding some light on the learning performance of existing GNN models.

The chapter is structured as follows. In Section 7.1, we introduce the syntax and semantics of LRNNs when viewed as a differentiable logic programming language. Subsequently, we illustrate the paradigm on a range of example models in Section 7.2. Capturing and extending GNNs is then detailed in Section 7.3. In Section 7.4, we demonstrate practicality and computation efficiency of the approach, and conclude in Section 7.6.

The content in this chapter is then generally based on the preliminaries of logic programming (Section 3.2) and deep learning (i.e. "differentiable programming", Chapter 2) as introduced in the background Part ii of the thesis. Some additional details and technical differences between this new view on LRNNs and the original framework from Chapter 5 are then detailed also in the appendix Section A. We note again that, apart from the generally related work introduced in background Chapter 4, we will also discuss the most recent related works jointly in Chapter 8.

7.1 THE PROGRAMMING LANGUAGE OF LRNNS

In this section we directly follow up on the introduced LRNNs (Section 5.1), which were originally introduced a learning framework for *lifted modeling* of neural networks oriented to relational data. Yet, from another perspective, it can be also understood as a differentiable version of simple Datalog

1 Code to reproduce experiments from this chapter is available at https://github.com/GustikS/GNNwLRNNs. The LRNNs framework itself can then be found at https://github.com/GustikS/NeuraLogic.

programming (Section 3.2), where the templates, encoding various neuro-relational learning architectures, take the form of parameterized logic programs. During learning, when presented with relational data such as various forms of graphs, the program interpreter dynamically unfolds differentiable computational graphs to be used for the program parameter optimization by standard (gradient descent) means.

This differs from the commonly used differentiable programming frameworks, such as PyTorch or Tensorflow, in its declarative, relational nature, enabling one to abstract away from the procedural details of the underlying computational graphs even further. We further explain the differences and principles of this LRNN abstraction, in the new context of differentiable (logic) programming, in the following subsections.

7.1.1 *Syntax*

The syntax of LRNN programming is derived directly from the Datalog [93] language (Section 3.2), which we further extend with numerical parameters. Note that this has been exploited in many previous works, where the parameters can signify values associated with facts [186] or rules [187]. Such extensions are typically designed to integrate standard statistical (or probabilistic [52]) modelling techniques with the high expressiveness of relational representation and reasoning (Section 3.4). In this chapter we seek to integrate Datalog with deep learning, for which we allow *each literal* in each clause of the logic program to be associated with a *tensor* weight. A parameterized program, formed by a multitude of such weighted rules, then declaratively encodes all computations to be performed in a given learning scenario.

For clarity of correspondence with standard (neural) learning scenarios, we here further split[2] the program into unit clauses (facts), constituting the learning examples, and definite clauses (rules), constituting the learning template.

7.1.1.1 *Learning Examples*

The learning examples contain factual description of a given world. For their representation we use weighted ground facts. A learning example is then a set $E = \{(V_1, e_1), \ldots, (V_j, e_j)\}$, where each V_i is a real-valued tensor and each e_i is a ground fact, i.e. expression of the form

$$\begin{aligned} &\mathbf{V_1} :: \; p_1(c_1^1, \ldots, c_q^1). \\ &\ldots \\ &\mathbf{V_j} :: \; p_n(c_1^n, \ldots, c_r^n). \end{aligned}$$

where $p_1, \ldots, p_n$ are predicates with corresponding arities $q, \ldots, r$, and c_i^j are arbitrary constants. Standard logical representation is then a special case where each $\mathbf{V_i} = 1$[3]. One can either write $1::carbon(c_1)$ or omit the weight and write, e.g., $bond(c_1, o_2)$. The values do not have to be binary and can represent a "degree of truth" to which a certain fact holds, such as $0.4::aromatic(c_1)$. The values are also not necessarily restricted to $(0, 1)$, and can thus naturally represent numerical features, such as $6::atomicNumber(c_1)$ or $2.35::ionEnergy(c_1, level_2)$. Finally the values are not necessarily restricted to scalars, and can thus have the form of feature vectors (tensors), such as $[1.0, -7, \ldots, 3.14]::features(c_1)$.

Ground facts in examples are also not restricted to unary predicates, and can thus describe not only properties of individual objects, but values of arbitrary relational properties. For example, one can assign feature values to edges in graphs, such as describing a bond between two atoms $[2.7, -1]::bond(c_1, o_2)$.

2 Note that this split is not necessary in general, and the template can also contain facts, as well as the learning examples may contain rules, such as in the original LRNNs (Section 5.1) and some ILP learning settings (Section 3.3.1).

3 Since we consider a close world assumption (CWA) and least Herbrand model, one does not enumerate false facts with zero value (Section 5.1.1).

There is no syntactical restriction on how these representations can be mixed together, and one can thus select which parts of the data are better modelled with (sub-symbolic) distributed numerical representations, and which parts yield themselves to be represented by purely logical means, and move continuously along this dimension as needed.

QUERY: Queries (q) (Section 3.2.1.2) represent the classification labels or regression targets associated with an example for supervised learning. They again utilize the same weighted fact representation such as $1::\texttt{class}$ or $4.7::\texttt{target}(c_1)$. Note that the target queries again do not have to be unary, and one can thus use the same format for different tasks. For example, for knowledge graph completion, we would use queries such as $1.0::\texttt{coworker}(\texttt{alice},\texttt{bob})$.

7.1.1.2 *Learning Programs*

The weighted logic programs written in LRNNs are then often referred to as *templates*. Syntactically, a learning template $\mathcal{P}$ is a set of weighted rules $\mathcal{P} = \{\alpha_i, \{W_j^{\alpha_i}\}\} = \{(W^i, c) \leftarrow (W_1^i, b_1), \ldots, (W_k^i, b_k)\}$ where each α_i is a definite clause and each W_j is some real-valued tensor, i.e. expressions of the form

$$
\begin{aligned}
&\mathbf{W^1} :: h_1^1(\ldots) \texttt{ :- } \mathbf{W_1^1} : b_1^1(\ldots) \texttt{ , } \ldots \texttt{ , } \mathbf{W_j^1} : b_i^1(\ldots)\texttt{ .}\\
&\mathbf{W^2} :: h_1^2(\ldots) \texttt{ :- } \mathbf{W_1^2} : b_1^2(\ldots) \texttt{ , } \ldots \texttt{ , } \mathbf{W_k^2} : b_j^2(\ldots)\texttt{ .}\\
&\ldots\\
&\mathbf{W^n} :: h_p^q(\ldots) \texttt{ :- } \mathbf{W_1^n} : b_1^n(\ldots) \texttt{ , } \ldots \texttt{ , } \mathbf{W_l^n} : b_k^n(\ldots)\texttt{ .}
\end{aligned}
$$

where h_i^j's and b_i^j's are predicates forming positive, not necessarily different, literals, and $\mathbf{W_i^j}$'s are the associated tensors (also possibly reused in different places).

The template constitutes roughly what neural *architecture* (Chapter 2) means in deep learning[4] – i.e. it does not (necessarily) encode a particular model or knowledge of the problem, but rather a *generic* mode of computation.

Example 19 *Consider a simple template for learning with molecular data, encoding a generic idea that the (distributed) representation* $(h(.))$ *of a chemical atom (e.g.* o_1*) is dependent on the chemical atoms adjacent to it. Given that a molecule can be represented by the set of contained atoms* $(a(.))$[5] *and bonds* $(b(.,.))$ *between them (see e.g. left part of Figure 15), we can encode this idea by a following rule*

$$\mathbf{W_{h_1}} :: h(X) \texttt{ :- } \mathbf{W_a} : a(Y)\texttt{, } \mathbf{W_b} : b(X,Y)\texttt{ .}$$

Moreover, one might be interested in using the representation of all atoms $(h(X))$ *for deducing the representation of the whole molecule, for which we can write*

$$\mathbf{W_q} :: q \texttt{ :- } \mathbf{W_{h_2}} : h(X)\texttt{ .}$$

to derive a single ground query atom (q)*, which can be associated with the learning target of the whole molecule. The concrete semantics of this template then follows in the next section.*

7.1.2 *Semantics*

To explain the correspondence between a relational template $\mathcal{P}$ and a "neural architecture", we now describe the mapping that takes the template and a given example description, consisting of ground facts, and produces a standard neural model. Here, "standard neural model" refers to a specific

4 We deliberately refrain from using the common term of neural "model", since a single template can have multiple logical (and neural) models.

5 e.g. $a(o_1)$ denotes a *logical* atom declaring o_1 to be a *chemical* atom. This fact can then be, e.g., associated with chemical features of the oxygen o_1 (Section 7.1.1.1). To distinguish, $h(.)$ denotes the learned distributed representation of each chemical atom.

Table 5: Correspondence between the logical ground model and computation graph.

Logical construct	Type of node	Notation
Ground atom h	Atom node	A_h
Ground fact h	Fact node	$F_{(h,\vec{w})}$
Ground rule's $\alpha\theta$ body	Rule node	$R^{c\theta}_{(W_0^c c\theta \leftarrow W_1^\alpha b_1\theta \wedge \cdots \wedge W_k^\alpha b_k\theta)}$
Rule's α ground head h	Aggregation node	$G^{h=c\theta_i}_{(W_0^c c \leftarrow W_1^\alpha b_1 \wedge \cdots \wedge W_k^\alpha b_k)}$

differentiable computational graph (Chapter 2). We note that this mapping is exactly analogous to the original LRNN semantics explained in Section 5.1.2.2.

First, let $\mathcal{N}_l$ be the set of rules and facts obtained from the template and a learning example $\mathcal{N}_l = \mathcal{P} \cup E_l$ by removing all the tensor weights. For instance, if we had a weighted rule W::h :- W_1:b_1, W_2:b_2 , we would obtain h :- b_1, b_2. Then we construct the least Herbrand model $\overline{\mathcal{N}_l}$ of $\mathcal{N}_l$, which can be done using standard theorem proving techniques (Section 3.2). One simple option is to employ a bottom-up grounding strategy by repeated application of the immediate consequence operator (Section 3.2.1.1)[6]. We note that for the consequent neural learning, the target query atom q associated with E_l must be logically entailed by $\mathcal{N}_l$, i.e. present in $\overline{\mathcal{N}_l}$[7].

Having the least Herbrand model $\overline{\mathcal{N}_l}$ containing q, we can directly construct a neural computational graph G_l. Intuitively, the structure of the graph contains all the logical derivations of the target query literal q from the example evidence E_l through the template $\mathcal{P}$. Again, this construction process is completely analogous[8] to the original LRNN translation as defined in Section 5.1.1. For the sake of clarity, we redefine the transformation mapping from $\mathcal{N}_l$ to a differentiable (neural) graph, in the new context of this chapter, as follows:

- For each weighted ground fact (V_i, e) occurring directly in E_l, there is a node $F_{(V_i,e)}$ in the computational graph, called a *fact node*.
- For each ground atom h occurring in $\overline{\mathcal{N}} \setminus E_l$, there is a node A_h in the computational graph, called an *atom node*.
- For every rule $c \leftarrow b_1 \wedge \cdots \wedge b_k \in \mathcal{P}$ and every grounding substitution $c\theta = h \in \overline{\mathcal{N}}$, there is a node $G^{c\theta=h}_{(c \leftarrow b_1 \wedge \cdots \wedge b_k)}$ in the computational graph, called an *aggregation node*.
- For every *ground* rule $\alpha_i\theta = (c\theta \leftarrow b_1\theta \wedge \cdots \wedge b_k\theta) \in \overline{\mathcal{N}}$, there is a node $R_{(c\theta \leftarrow b_1\theta \wedge \cdots \wedge b_k\theta)}$ in the computational graph, called a *rule node*.

An overview of the correspondence between the logical and the neural model, together with the used notation is reviewed in Table 5.

The nodes of the computational graph that we defined above are then interconnected so as to follow the derivation of the logical facts by the immediate consequence operator starting from E_l, i.e. starting from the *fact nodes* $F_{(V_i,e)}$ which have no antecedent inputs in the computational graph and simply output their associated values $\text{out}(F_{(V_i,e)}) = V_i$. The fact nodes are then connected into *rule nodes* $R_{\alpha\theta}$, particularly a node $F_{(V_i,e)}$ will be connected into *every* node $R_{\alpha\theta} = R_{(c\theta \leftarrow b_1\theta \wedge \cdots \wedge b_k\theta)}$

6 Another option is backward-chaining of the rules back from the associated query atom (q) through $\mathcal{P}$ into E_l. Note that this choice is purely technical and, following proper logical inference in both cases, does not affect the resulting logical (or neural) model.

7 Otherwise it stands as a "counter-example" in the ILP sense (Section 3.3.1), which are automatically considered false (or having a default value) here via CWA.

8 With the main difference being the (tensor) parameterization of the rules, leading to definition of the underlying computation on the level of nodes (akin to "layers") instead of neurons.

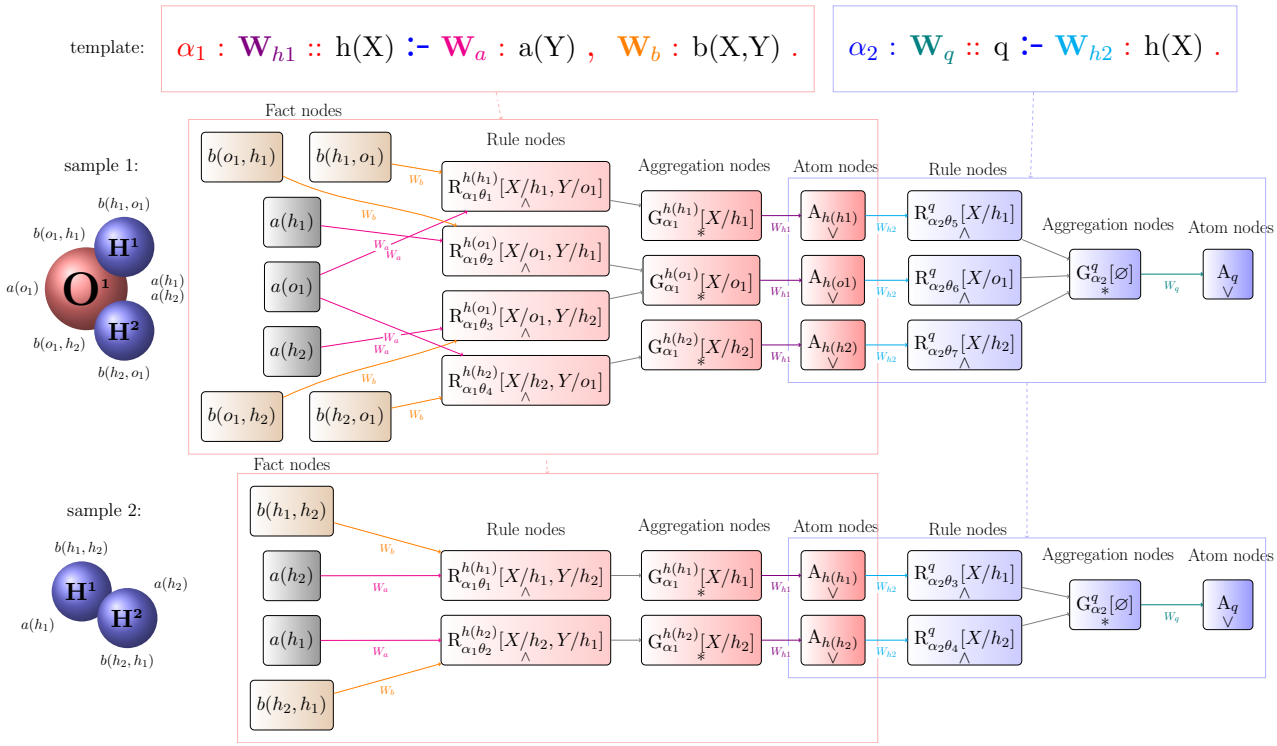

Figure 15: A simple LRNN template with 2 rules described in Example 1. Upon receiving 2 example molecules, 2 neural computation graphs get created, as prescribed by the semantics (Section 7.1.2).

where $e = b_i\theta$ for some i. Having all the inputs, corresponding to the body literals of the associated ground rule, connected, the rule node will output a value calculated as

$$\mathrm{out}(R_{\alpha\theta}) = g_{\wedge}\Big(W^{\alpha}_1 \cdot \mathrm{out}(F_{(V_1,b_1\theta)}), \ldots, W^{\alpha}_k \cdot \mathrm{out}(F_{(V_i,b_k\theta)})\Big).$$

The rule node's activation function $g_{\wedge}$ is up to user's choice. For scalar inputs, it can be for example set to mimic conjunction from Lukasiewicz logic, as detailed for the original LRNNs in Section 5.1.3. However, one can also choose to ignore the fuzzy-logical interpretation, and use completely distributed semantics and activations utilized commonly in deep learning, which is the strategy chosen in this chapter. In this case the computation follows the common (matrix) calculus by firstly aggregating the node's input values into its activation value

$$\underset{(l\times 1)}{\mathrm{act}(R_{\alpha\theta})} = \underset{(l\times n)}{W^{\alpha}_1} \cdot \underset{(n\times 1)}{\mathrm{out}(F_1)} + \cdots + \underset{(l\times m)}{W^{\alpha}_j} \cdot \underset{(m\times 1)}{\mathrm{out}(F_k)},$$

followed by an element-wise application of any differentiable function, such as logistic sigmoid

$$\underset{(l\times 1)}{\mathrm{out}(R_{\alpha\theta})} = \sigma(\underset{(l\times 1)}{\mathrm{act}(R_{\alpha\theta})}) = \sigma\Big(\mathrm{act}(R_{\alpha\theta})_1, \ldots, \mathrm{act}(R_{\alpha\theta})_l\Big).$$

In general, the rule nodes are used to represent (conjunctive) patterns to be repeatedly matched in the input (or transformed) data while reusing the same parameterization, such as the convolutional filters in CNNs[9].

The rule nodes are then connected into *aggregation nodes*. Particularly, a rule node $R_{(c\theta\leftarrow b_1\theta\wedge\cdots\wedge b_k\theta)}$ is connected into *the* aggregation node $G^{c\theta=h}_{(c\leftarrow b_1\wedge\cdots\wedge b_k)}$ that corresponds to the same ground head literal $c\theta$. Having all the inputs, corresponding to different grounding substitutions θ_i of the rule $c \leftarrow (b_1 \wedge \cdots \wedge b_k)$ with the same ground head $h = c\theta_1 = \cdots = c\theta_q$, connected, the aggregation node will output the value

$$\mathrm{out}(G^{c}_{\alpha}\theta = h) = g_*\Big(\mathrm{out}(R^{c\theta_1=h}_{\alpha\theta_1}), \ldots, \mathrm{out}(R^{c\theta_q=h}_{\alpha\theta_q})\Big).$$

9 see e.g. Figure 17 for an example use.

where g_* is some aggregation function, such as avg or max. The aggregation nodes effectively aggregate all the different ways by which a literal h can be derived from a single rule α. Their semantic intuitively corresponds to a certain quantification over free variables appearing solely in the rule's α body. The aggregation g_* is then applied in each dimension of the input values as

$$\underset{l\times 1}{\mathrm{out}(G^h_\alpha)} = g_*(\underset{l\times 1}{\mathrm{out}(R_1)},\ldots,\underset{l\times 1}{\mathrm{out}(R_q)}) = \Big(g_*\Big(\mathrm{out}(R_1)^1,\ldots,\mathrm{out}(R_q)^1\Big),\ldots, \ldots, g_*\Big(\mathrm{out}(R_1)^l,\ldots,\mathrm{out}(R_q)^l\Big)\Big).$$

Note that since all the input values are derived from a single rule α, their dimensionalities are necessarily the same. Intuitively, the aggregation nodes are used to aggregate values from the pattern matches of the underlying rule nodes, such as the pooling operation used in CNNs[9].

The aggregation nodes are then connected into *atom nodes*. In particular, an aggregation node G^h_α will be connected into *the* atom node A_h that is associated with the same atom h. The inputs of the atom node represent all the possible rules α_i through which the same atom h can be derived. Having them all connected, A_h will output the value

$$\mathrm{out}(A_h) = g_\vee\Big(W^c_1 \cdot \mathrm{out}(G^h_{\alpha_1}),\ldots,W^c_m \cdot \mathrm{out}(G^h_{\alpha_m})\Big).$$

Apart from the choice of activation function $g_\vee$, the computation of the atom node's output follows exactly the same scheme as for the rule nodes. However, the atom nodes are used to combine the aggregated values (pattern matches) from different rules (such as the combine operation in GNNs in Section 2.4).

Finally, the atom nodes are connected into rule nodes in exactly the same fashion as fact nodes, i.e. A_h will be connected into *every* $R_{(c\theta\leftarrow b_1\theta\wedge\cdots\wedge b_k\theta)}$ where $h = b_i\theta$ for some i, and the whole process continues recursively. Note that this process of transforming a learning example into a computation graph is performed only once, as the subsequent neural training can only change the values of the parameters but not the structure of the graphs.

Example 20 *Let us follow up on the Example 1 by extending the described template with two example molecules of hydrogen and water. The template will then be used to dynamically unfold two computation graphs, one for each molecule, as depicted in Figure 15. Note that the computation graphs have different structures, following from the different Herbrand models derived from each molecule's facts, but share parameters in a scheme determined by the lifted structure of the joint template.*

7.2 EXAMPLES OF COMMON NEURAL ARCHITECTURES

We demonstrate flexibility of the declarative LRNN templating, stemming from the abstraction power of Datalog, by encoding a variety of common neural architectures in very simple differentiable logic programs. For completeness, we start from simple neural models, where the advantages of templating are not so apparent, but continue to advanced deep learning architectures, where the expressiveness of relational templating stands out more clearly. Note that all templates in this chapter are actual programs that can be run and trained with the LRNN interpreter.

7.2.1 *Feed-forward Neural Networks*

Multi-layer perceptrons (MLPs) form the most simple case where the weighted logic template is restricted to propositional clauses, and its single Herbrand model thus directly corresponds to a single neural model (Section 7.1.2). In this setting, the input example information can thus be encoded merely in the *values* of their associated tensors, which is the standard deep learning scenario.

Figure 16: Demonstration of the pruning technique on a sample MLP model unfolded from a 2-rule template of $\alpha_1 = W_1{::}h_1 \texttt{:-} f.$ and $\alpha_2 = W_2{::}h_2 \texttt{:-} h_1.$

In the vector form, we can associate each example E_i with a fact proposition $[v^i_1, \ldots, v^i_n]{::}\mathtt{features}^{(0)}$, forming the input (0-th) node of the neural model. Each example is further associated with a query $v^i_q{::}q$.

In particular, an MLP with 3 layers, i.e. input layer$^{(0)}$, 1 hidden layer$^{(1)}$, and output layer$^{(2)}$, with the corresponding weight matrices $[\underset{m\times n}{W}{}^{(1)}, \underset{1\times m}{W}{}^{(2)}]$ can be directly modelled with the following rule

```
1 W[1×m]^(2) :: q^(2) :- W[m×n]^(1) : features^(0).
```

Naturally, we can extend it to a deeper MLP by stacking more rules as

```
1 W[r×m]^(2) :: hidden^(2) :- W[m×n]^(1) : features^(0).
2 ...
3 W[1×s]^(k) :: q^(k) :- W[s×r]^(k−1) : hidden^(k−2).
```

Once the template gets transformed into the corresponding neural model (Section 7.1.2), its computation graph will consist of a linear chain of nodes (Section 2.1) corresponding to standard fully-connected layers $1, \ldots, k$ with associated weight matrices $[W^{(1)}, W^{(2)} \ldots, W^{(k)}]$, and activation functions of user's choice. We note that it is also possible to specify the activation functions with each rule (layer) separately, e.g. as

```
1 W^(4) :: hidden^(4) :- W^(3) : hidden^(2) . [g∧ = ReLU, g* = AVG]
```

Note that not all the weights need to be specified, and one can thus also write, e.g., either of

```
1 W :: h^(2) :- h^(0).        h^(2) :- W : h^(0).
```

While each of these rules still encodes in essence a 3-layer MLP, either only the hidden (right) or only the output (left) layer will carry learnable parameters, respectively. Moreover, following the exact semantics (Section 7.1.2) for neural model creation, an aggregation node will be created on top of a rule node, representing the hidden layer. Since there is no need for aggregation in MLPs, i.e. only a single rule node ever gets created from each propositional rule, this introduces unnecessary operations in the graph. Since such nodes arguably do not improve learning of the model, we *prune* them out, as depicted in Figure 16. The technique is further described in more detail in the appendix Section A.1.1. Note that we assume application of pruning, where applicable, in the remaining examples described in this chapter.

7.2.1.1 *Knowledge-based Artificial Neural Networks*

As discussed in Section 4.3.1, the direct correspondence between a propositional program and a neural network has been successfully exploited in a number of previous works, particularly the original Knowledge-Based Artificial Neural Networks (KBANN) [67], and LRNNs can be possibly seen as a direct extension of KBANN into relational setting. To emulate the KBANN inference and learning, we simply fall back to scalar representation of features, e.g. $0{::}rain, 0.6{::}wet, 0.8{::}sunny$, and consider a propositional template encoding some background domain knowledge, such as $sprinkle \texttt{ :- } wet \texttt{ , } sunny$. One then needs to choose a set of proper activation functions based on desired multi-valued logic semantics, e.g. the Lukasiewicz's fuzzy operators (Section 5.1.3). Note that, choosing proper fuzzy logic activations, this still covers standard logical inference as a special case with the use of binary fact values.

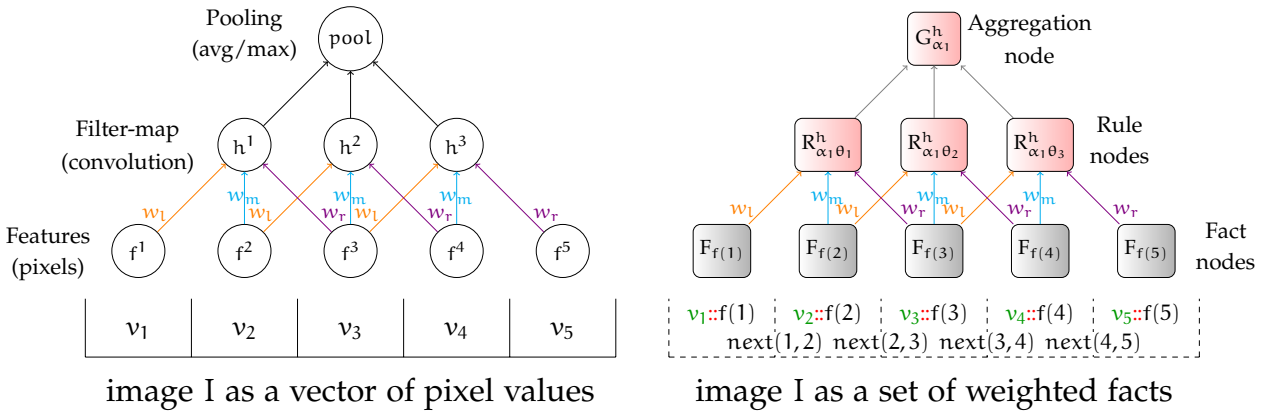

Figure 17: Left: core part of a standard CNN architecture with sparse layer composed of sequential applications of a convolutional filter (h), creating a feature-map layer, followed by a pooling operator. Right: the corresponding computation graph derived from a LRNN template.

7.2.2 *Convolutional Neural Networks*

The CNNs can no longer be represented with a propositional template. To emulate the additional parts w.r.t. the MLPs, i.e. the convolutional filters and pooling (Section 2.2), we need to move to *relational* rules (Section 3.2). Note that there is a natural, close relationship between convolutions and relational rules (or relational patterns in general), where the point of both is to exploit symmetries in data. Moreover, the point of both the aggregation nodes and the pooling layers is to enforce certain transformation invariance. Let us demonstrate this relationship with the following example.

For clarity of presentation, consider a simplistic one-dimensional "image" consisting of 5 pixels $i = 1, ..., 5$. While the regular grid structure of the image pixels is inherently assumed in CNN, we will need to encode it explicitly. Considering the 1-dimensional case, it is enough to define a linear ordering of the pixels such as $next(1,2), \ldots, next(4,5)$. The (gray-scale) value v_i of each pixel i can then be encoded by a corresponding weighted fact v_i::$f(i)$. Next we encode a convolution filter of size $[1,3]$, i.e. vector which combines the values of each three ([left,middle,right]) consecutive pixels, and a (max/avg)-pooling layer that aggregates all the resulting values. This computation can be encoded using the following template

```
h :- w_l: f(A), w_m: f(B), w_r: f(C), next(A,B), next(B,C).
```

A visualization of the CNN and the corresponding computation graph derived from the logic model of the template presented with some example pixel values $[v_1, \ldots, v_5]$ is shown in Fig 17.

While this does not seem like a convenient way to represent learning with CNNs from images, the important insight is that convolutions in neural networks correspond to weighted relational rules (patterns). The efficiency of normal CNN encoding is due to the inherent assumptions that are present in CNNs w.r.t. topology of their application domain, i.e. grids of pixel values, and similarly complete, ordered structures. While with LRNNs we need to state all these assumptions explicitly, it also means that we are not restricted to them – an advantage which will become clearer in the subsequent sections.

7.2.3 *Recursive and Recurrent Neural Networks*

A *recursive* network also exploits the principle of convolution, however the input is no longer a grid but a regular tree of an unknown structure. This prevents us from creating computation schemes customized to a specific structure, as in the CNNs. Instead, we need to resort to a general convolutional pattern that can be applied over any k-regular tree.

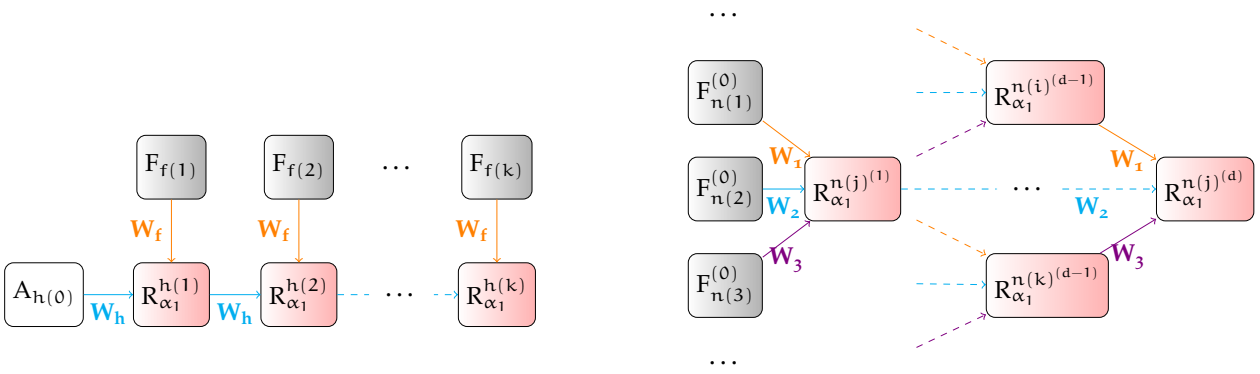

Figure 18: Simple recurrent (left) and recursive (right) neural structures encoded through LRNNs.

For that purpose, we again utilize the expressiveness of relational logic. Firstly, we encode the k-regular tree structure itself by providing a fact connecting each parent node in the tree to its child-nodes, i.e. $parent(node_j^{i+1}, node_l^i, \ldots, node_{l+k}^i)$. Secondly, we associate all the leaf nodes in the tree with their embedding vectors $[\mathbf{v}_1^i, \ldots, \mathbf{v}_n^i]$::$n(leaf_i)$. Finally, a single relational rule can then be used to encode the recursive composition of representations in the, for instance 3-regular, tree as

```
n(P) :- W1:n(C1), W2:n(C3), W3:n(C3), parent(P,C1,C2,C3).
```

which directly forms the whole learning template. Given a particular example tree, this rule translates to a computation graph recursively combining the children node representations ($n(C)$) into respective parent node representations, until the root node is reached. The root node representation ($n(root)$) could then be fed into a standard MLP rule (Section 7.2.1) to output the value for a given target query associated with the whole tree example.

A simple *recurrent* neural network unfolded over a linear (time) structure can then be modelled in a simpler manner, where only a single (vector) input is given at each step and a linear chain of *hidden* nodes ($h(X)$) replaces the prescribed tree hierarchy. Assuming encoding of the linear example structure with predicate $next(X, Y)$ as before, such a model can then be written as

```
h(Y) :- Wf:f(Y), Wh:h(X), next(X,Y).
```

The final hidden representation ($h(k)$) could then again be fed into a MLP for a whole sequence-level prediction. Neural architectures of both these templated models are displayed in Figure 18.

7.3 GRAPH NEURAL NETWORKS IN LRNNS

Graph (convolutional) Neural Networks (GNNs) (Section 2.4) can be seen as a generalization of the introduced neural architectures (Section 7.2) to arbitrary graphs, for which they combine the ideas of latent representation learning (Section 7.2.1), convolution (Section 7.2.2), and dynamic model structure (Section 7.2.3).

While modelling CNNs in the weighted logic formalism was somewhat cumbersome (because we had to explicitly represent the pixel grid), the encoding of GNNs is very straightforward. This is due to the underlying general graph representation with no additional assumptions of its structure, which yields itself very naturally to relational logic. The computation of the layer i update in GNNs can then be represented by a single rule as follows

```
W(i) :: h(i)(V) :- h(i-1)(U), edge(V,U).
```

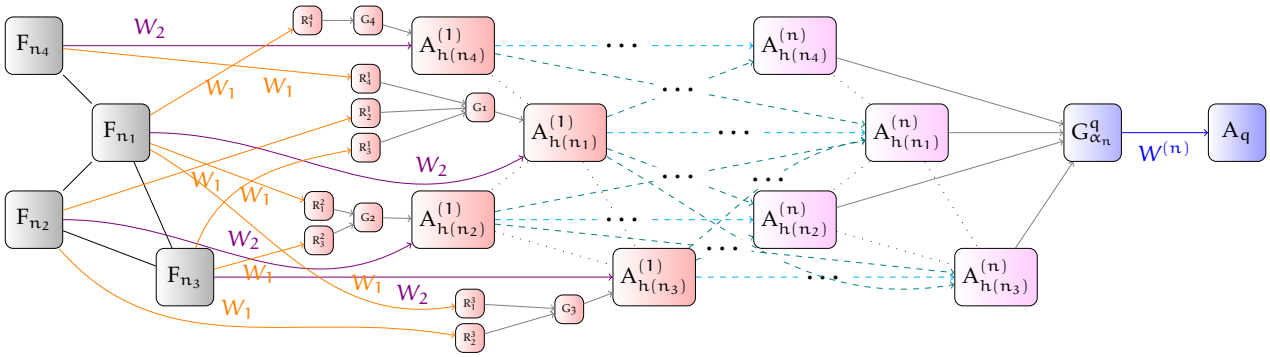

Figure 19: A computation graph of a sample (g-SAGE) GNN as encoded in LRNNs. Given an input graph of 4 (fact) nodes ($F_{n_1} \dots F_{n_4}$), neighbors of each node are firstly weighted and aggregated with rule and aggregation nodes, respectively (reduced in size in picture). The result is then combined with representation of the (central) node from the preceding layer, to form a new layer of 4 atom nodes, copying the structure of the input graph. After n such layers, each with the same structure but different parameters, a global readout (aggregation) node aggregates all the node representations, passing to the final query (atom) node's transformation.

where $edge/2$ is the binary relation of the given input graphs. With the choice of activation functions as $g_* = avg, g_\wedge = \mathrm{ReLU}$, this simple rule can be already used to model the popular Graph Convolutional Neural Networks (GCN) [75][10]. The exact same rule (up to parameterization) is then used at each layer. For the final output query (q) representing the whole graph we simply aggregate representations of all the nodes as

```
1 W^(d) :: q :- h^(d-1)(U).
```

A noticeable shortcoming of GCNs is that the representation of the "central" node (V) itself is not used in the representation update. While this can be done by extending the graph ($edge/2$) with self-loops, a novel[11] GNN model called GraphSAGE (g-SAGE) [74] was proposed to address this explicitly. To follow the architecture of g-SAGE, we thus split the template into 2 rules accordingly

```
1 h^(i)(V) :- W_1^(i) : h^(i-1)(U), edge(V,U).
2 h^(i)(V) :- W_2^(i) : h^(i-1)(V).
```

and choose $g_\wedge = \mathrm{ReLU}, g_* = max, g_\vee = identity$ for the very model (g-SAGE), the depiction of which can be seen in Figure 19.

Another popular extension taken from neural architectures for image recognition are residual (skip) connections, where one effectively adds links to preceding layers at arbitrary depth (instead of just the preceding layer), i.e. we simply add one or more rules in the form

```
1 W_skip^(i) :: h^(i)(V) :- h^(i-skip)(V).
```

This technique is also used in the Graph Isomorphism Network (GIN) [61], which is a theoretically substantiated GNN based on the expressive power of the Weisfeiler-Lehman test (WL) [59]. Firstly, the GIN model differs in that it adds residual connections from all the preceding layers to the final layer (which the authors refer to as "jumping knowledge" [76]). Secondly, the particularity of GIN is to add a 2-layered MLP on top of each aggregation to harvest its universal approximation power. Particularly, update formula derived from the WL-correspondence [61] is

$$h^{(i)}(v) = \mathrm{MLP}^{(i)}\left((1+\epsilon^{(i-1)}) \cdot h^{(i-1)}(v) + \sum_{u \in \mathcal{N}(v)} h^{(i-1)}(u)\right)$$

10 where the authors also denoted the rule as *convolution*, since it forms a linear approximation of a localized spectral convolution [75].

11 Note that, differently from GCN with self-loops, the central node is parameterized differently from the neighbors.

where MLP is the 2-layered MLP (Section 2.1). To accommodate the extra MLP layer, we thus extend the template as follows

```
mlp_tmp^(i)(V) :- h(U)^(i-1), edge(V,U).
mlp_tmp^(i)(V) :- (1+›^(i-1)) : h^(i-1)(V).
W_2^(i) :: h^(i)(V) :- W_1^(i) : mlp_tmp^(i)(V).
```

Note that, considering that such a single rule actually already models a 2-layer[12] MLP (as described in Section 7.2.1), a very similar computation can be carried out more simply with

```
W_2a^(i) :: h^(i)(V) :- W_1a^(i) : h(U)^(i-1), edge(V,U).
W_2b^(i) :: h^(i)(V) :- W_1b^(i) : h^(i-1)(V).
```

corresponding to a GIN version without the special $(1+\epsilon^{(i)})$ coefficient, which the authors refer to as "GIN-0" [61] and actually find performing better[13]. Finally they choose $g_* = sum$ as the function to aggregate the neighborhood representations. The authors proved the GIN model to belong to the most "powerful" class of GNN models, i.e. no other GNN model is more expressive than GIN, and demonstrated the GIN-0 model to provide state-of-the-art performance in various graph classification and completion tasks [61].

7.3.1 *Extending GNNs*

While the GIN model presents the most "powerful" version of the basic GNN idea, there is a large number of ways in which the GNN approach can be extended. We discuss some of the direct, natural extensions in this subsection.

7.3.1.1 *Edge Representations*

Originally aimed at single-relation graphs, GNNs do not adequately utilize the information about the possibly different types of edges. While it is straightforward to associate edges with scalar weights in the adjacency matrix, instead of using just binary edge indicators [75], extending to richer edge representations is not so direct, and has only been explored recently [189–191].

In the templating approach, addressing edges is very simple, since we do not operate directly on the graph but on the ground logical model, where each edge ($edge(n_1,n_2)$) forms an *atom* in exactly the same way as the actual nodes ($node(n_1)$) in the graph itself (similarly to an extra transformation introduced in line-GNNs [192]). We can thus directly associate edges corresponding to different relations with arbitrary features ($[v_1,\ldots,v_n]:: edge(n_1,n_2)$), learn their distributed representations, and predict their properties (or existence), just like GNNs do with the nodes. For basic learning with edge representations, there is no need to change anything in the previously introduced templates. However, one might want to associate extra transformations for edge and node representation learning [190], in which case we would simply write

```
W^(i) :: h^(i)(V) :- h(U)^(i-1), W_e:edge(V,U).
```

A large number of structured data then come in the form of multi-relational graphs, where the edges can take on different types. A straightforward extension is to learn a separate node representation of the nodes for each of the relations, e.g. as

12 or 3-layer, depending on inclusion of the input layer in the count.

13 We note there is a slight difference, where GIN-0 firstly aggregates the neighbors and weights the result, while this template aggregates the neighbors after weighting. Nevertheless we note that GNN authors often switch this order themselves, for instance GraphSAGE in [188] performs weighting before aggregation, while it is vice-versa in [61].

```
W^(i) :: h_x^(i)(V) :- h_x(U)^(i-1), W_e:edge_type=x(V,U).
```

and to choose from the different representations depending on context, such as in multi-sense word embeddings [193], or simply directly combine [194] these representations in the template.

7.3.1.2 *Heterogeneous Graphs*

The majority of current GNNs then assume homogeneous graphs, and learning from heterogeneous graphs has just been marked as one of the future directions for GNNs [58]. In LRNNs, various heterogeneous graphs [195] can be directly covered without any modification, since there is no restriction to the types of nodes and relations to be used in the same template (and so we do not have to e.g. split the graphs [196] or perform any extra operation [197] for such a task). In the context of heterogeneous information networks, a similar "templating" idea has already become popular as defining *"meta-paths"* [198, 199], which can be directly covered by a single LRNN rule and, importantly, differentiated through.

We can further represent the relations as actual objects to be operated by logical means, by reifying them into logical constants as

```
W_1^(i) :: h^(i)(V) :- h(U)^(i-1), h(E)^(i-1), edge(V,U,E).
```

where variable E represents the edge object and h(E) is its hidden representation. The learned embeddings of the nodes and relations can then be directly used for predicting triplets of (Object,Relation,Subject) in KBC, again with a simple template extension, e.g. for an MLP-based KBE [200], as

```
W :: edge(O,R,S) :- W_o:h(O), W_r:h(R), W_s:h(S).
```

7.3.1.3 *Hypergraphs*

Naturally, the GNN idea can be extended to hypergraphs, too, as was recently also proposed [201]. While extending to hypergraphs from the adjacency matrix form used for simple graphs can be somewhat cumbersome, in the relational Datalog, hypergraphs are first-class citizens, so we can just directly write

```
W_1^(i) :: h^(i)(U_1) :- h(U_1)^(i-1), ..., h(U_n)^(i-1), edge(U_1,...,U_n).
...
W_1^(i) :: h^(i)(U_n) :- h(U_1)^(i-1), ..., h(U_n)^(i-1), edge(U_1,...,U_n).
```

and possibly combine with all the other extensions.

There are many other simple ways in which GNNs can be extended towards higher expressiveness and there is a wide variety of emerging works in this area. While reaching beyond the standard, single adjacency matrix format, each of the novel extensions typically requires extra transformations (and libraries) to create their necessary intermediate representations [192, 198]. Many of these extensions are often deemed complex from the graph (GNN) point of view, but are rather trivial template modifications with LRNNs, as indicated in the preceding examples (and following in Section 7.3.2). This is due to the adopted *declarative* relational abstraction, as opposed to the procedural manipulations on ground graphs, defined often on a per basis. On the other hand we note that LRNNs currently cannot cover recent non-isotropic GNNs [188] with computation constructs such as attention, gating, or LSTMs [58]. While such construct could be included on an ad-hoc basis, they do not yield themselves naturally to the LRNN semantics.

7.3.2 *Beyond GNN architectures*

While we discussed possible ways for direct extensions of GNNs, there are more substantial alterations that break beyond the core principles of GNNs. One of them is the "message passing" idea, where the nodes are restricted to "communicate" with neighbors through the existing edges (WL label propagation [59]). Obviously, there is no such restriction in LRNNs, and we can design templates for arbitrary message passing schemes, corresponding to more complex and expressive convolutional filters. For instance, consider a simple extension beyond the immediate neighborhood aggregation by defining edges as weighted paths of length 2 (introduced as "soft edges" in [202]):

```
W:: edge2(U,W) :- W1:edge(U,V), W2:edge(V,W).
```

We can also easily compose the edges into small subgraph patterns of interest (also known as "graphlets" or "motifs" used in, e.g., social network analysis [203]), such as triangles and other small cliques (alternatively conveniently representable by the hyper-edges (Section 7.3.1.3)), and operate on the level of these instead:

```
W:: node(U,V,W) :- W1:edge(U,V), W2:edge(V,W), W3:edge(W,U).
```

Since both nodes and edges can be treated uniformly as logic atoms, we can easily alter the GNN idea to hierarchically propagate latent representations of the edges, too. In other words, each edge can aggregate representations of "adjacent" edges from previous layers:

```
W:: h^(i)_edge(E) :- WF:h^(i−1)_edge(F), WU,V:edge(U,V,E), WV,W:edge(V,W,F).
```

Naturally, this can be further combined with the standard learning of the latent node representations (as we do in experiments in Section 7.4).

Moreover, the messages do not have to spread homogeneously through the graph and a custom logic can drive the diffusion scheme. This can be, for instance, naturally put to work in the heterogeneous graph environments (Section 7.3.1.2) with explicit types, which can then be used to control communication and representation learning of the nodes:

```
W:: h(i)(X) :- h(i−1)(Y), edge(X,Y,E), type(E,type^e_1).
```

Besides being able to represented the *types* explicitly as objects (as opposed to the vector embeddings), we can actually induce new types, for instance into latent hierarchical categories (such as in [171]):

```
isa(edge1,type^e_1).
...
W(1):: isa(supertype^(1)_e,type^e_1).
W(k):: isa(A,C) :- W1^(k−1): isa(A,B), W2^(k−1): isa(B,C).
```

Importantly, there is no need to directly follow the input graph structure in each layer. We can completely abstract away from the graph representation in the subsequent layers and reason on the level of the newly invented, logically derived, entities, such as, e.g., the various graphlets, latent types, and their combinations:

```
W:: node^(1)_motif(T1,T2,T3) :- W1:node(X), W2:node(Y), W3:node(Z),
                                W4:type(X,T1), W5:type(Y,T2), W6:type(Z,T3),
                                edge(X,Y), edge(Y,Z), edge(X,Z).
```

Finally, the models can be directly extended with external relational background knowledge. Note that such knowledge can be specified declaratively, with the same expressiveness as the templates themselves, since they are consequently simply merged together, for instance:

```
ring6(A,...,F) :- Ve1:edge(A,B), ..., Ve6:edge(F,A),
                  Vn1node(A), ..., Vn6node(F).
W:: node(n)(X) :- V1: ring6(X,...,F).
```

We further detail extending GNNs with such declarative background knowledge in a practical case study in Section 7.5.1. Note that this is very different from the standard GNNs, where one can only input ground information, in the form of numerical feature vectors along with the actual nodes (and possibly edges). Nevertheless this does not mean that LRNNs cannot work with numerical representations. On the contrary, besides the standard neural means, one can also directly interact with it by the logical means, e.g. by arithmetic predicates to define learnable numerical transformations (such as in [202]) over some given (or learned) node similarities:

```
W:: edgesim(N1,N2) :- similar(N1,N2,Sim), W0.3:⩾(Sim,0.3).
```

7.4 EXPERIMENTS

The preceding examples were meant to demonstrate high expressiveness and encoding efficiency of declarative LRNN templating. The main purpose of the experiments is to assess correctness and efficiency of the actual learning. For that purpose, we select GNNs as the most general and flexible of the commonly used neural architectures, since they encompass building blocks of all the other introduced architectures. Given the focus on GNNs, we compare against two most popular[14] GNN frameworks of Pytorch Geometric (PyG) [204] and Deep Graph Library (DGL) [205]. Both these frameworks contain reference implementations of many popular GNN models, which makes them ideal for such a comparison. Note also that both these frameworks are highly contemporary and were specifically designed and optimized for creation and training of GNNs. For clarity of presentation, we restrict ourselves to a single task of structure property prediction, but perform experiments across a large number of datasets.

7.4.1 *Datasets*

For clarity of presentation, we restrict ourselves to a single task of graph classification, but perform experiments across a large number of datasets to obtain statistically significant results. Particularly, we assembled a collection of 78 popular molecular datasets, ranging from small instances, such as the infamous Mutagenesis [172], to medium, such as the Predictive Toxicology Challenge [174], and large, particularly various datasets from the National Cancer Institute NCI [206][15]. Note that these include popular datasets such as NCI1 [61, 62, 207] or NCI109 [207–209] that are commonly picked by GNN authors, but we also include the 70 others. The tasks with these are generally to recognize molecules w.r.t. their mutagenicity, toxicity or ability to inhibit growth of different types of tumors. On average, each of these datasets contains app. 3000 samples each with app. 24 atoms and 51 bonds. Note we only use the basic (Mol2 [210]) types of atoms and bonds without extra chemical features.

7.4.2 *Modern GNN frameworks*

While popular deep learning frameworks such as TensorFlow or Pytorch provide ways for efficient acceleration of standard neural architectures such as MLPs and CNNs, implementing GNNs is more challenging due to the irregular, dynamic, and sparse structure of the input graph data. Nevertheless, following the success of vectorization of the classic neural architectures, both PyG

14 with, as of date, PyG having 7.3K stars and DGL having 4.7K stars on Github, respectively.
15 available at ftp://ftp.ics.uci.edu/pub/baldig/learning/nci/gi50/

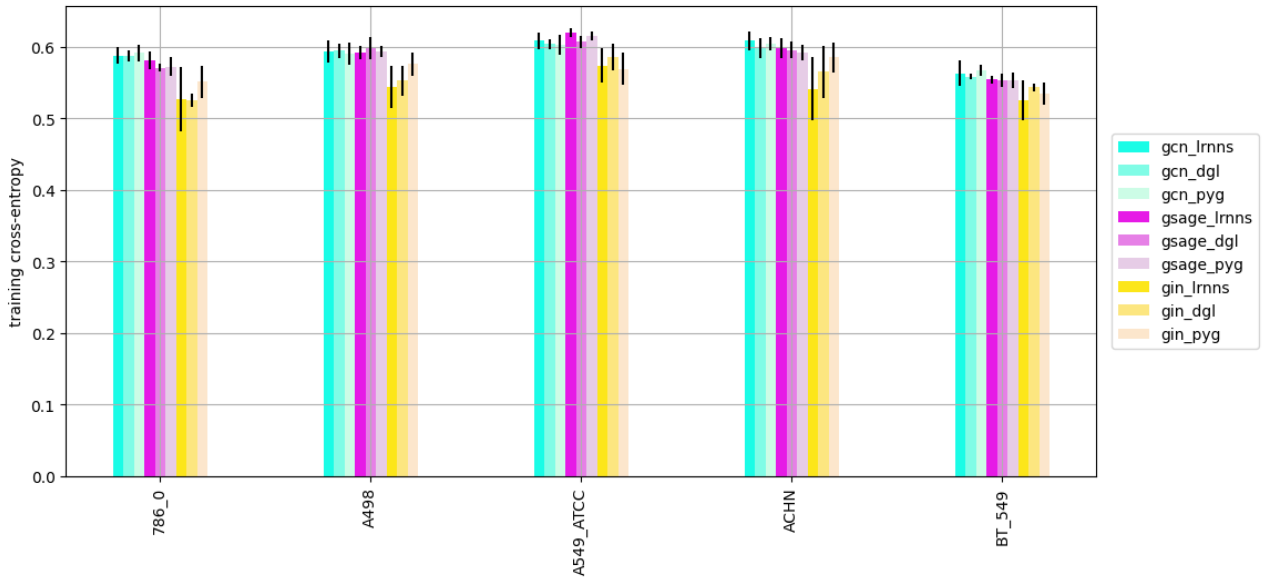

Figure 20: Alignment of training errors of the 3 models (GCN, g-SAGE, GIN), as implemented in the 3 different frameworks (LRNNs, DGL, PyG), over first 5 (alphabetically) datasets.

and DGL adopt the standard (sparse) tensor representation of all the data to leverage vectorized operations upon these. This includes the graphs themselves, which are then represented by their sparse adjacency matrices G^i_i. Further, each node index i can be associated with a feature vector $([f_1, \ldots, f_j]_i)$ through an additional matrix F^j_i associated with each input graph.

Following the standard procedural differentiable programming paradigm, both frameworks then represent model computations explicitly through a predefined graph of tensor transformations applied directly to the input graph matrices, creating an updated feature tensor $F^{j^{(k)}}_i$ at each step k. The same tensor transformations are then applied to each input graph.

Both frameworks are based on similar ideas of message passing between the nodes (neighborhood aggregation) and its respective acceleration through optimized sparse tensor operations and batching (gather-scatter). DGL then seems to support a wider range of operations (and backends), with high-level optimizations directed towards larger scale data and models (and a larger overhead), while PyG utilizes more efficient low-level optimizations stemming from its tighter integration with Pytorch.

7.4.3 *Model and Training Correctness*

Firstly, we evaluate correctness of the templated GNN architectures via correspondence to their reference implementations in PyG and DGL. For this we select some of the most popular GNN models introduced in previous chapters, particularly the original GCN [75], highly used GraphSAGE (g-SAGE) [74] and the "most powerful" GIN [61] (particularly GIN-0). Each of the models comes with a slightly different aggregate-combine scheme and particular aggregation/activation functions (detailed in Sections 2.4 and 7.3). Moreover, we keep original GCN and g-SAGE as 2-layered models, while we adopt 5-layers for GIN (as proposed by the authors)[16]. We further use the same latent dimension $d = 10$ for all the weights in all the models. Finally we set average-pooling operation, followed by a single linear layer, as the final graph-level readout for prediction in each of the models.

While the declarative templating takes a very different approach from the procedural GNN frameworks, for the specific case of GNN templates it is easy to align their computations, as they are

16 Obviously the number of layers could be increased/equalized for all of the models, however we keep them distinct to also accentuate their learning differences further.

Table 6: Average Mahalanobis distances between the training errors of the respective models and frameworks over all the datasets.

	|LRNN,PyG|	|LRNN,DGL|	|DGL,PyG|
GCN	0.20 ± 0.10	0.22 ± 0.19	0.26 ± 0.23
G-SAGE	0.25 ± 0.17	0.33 ± 0.35	0.27 ± 0.20
GIN	0.49 ± 0.30	0.54 ± 0.40	0.52 ± 0.35

mostly simple sequential applications of the (i) neighborhood aggregation, (ii) weighting, and (iii) non-linear activation, which can be covered altogether by a single rule (Section 7.3)[17].

For the comparison, we choose the NCI [173] molecular datasets[18], each containing thousands of molecules labeled by their ability to inhibit growth of different types of tumors. Note we only use the basic (Mol2 [210]) types of atoms and bonds without extra chemical features. For visual clarity we present results only for the first 5 of the total 73 datasets in alphabetical order (while we note that the results are very similar over the whole set). We use the same 10-fold crossvalidation splits for all the models. We further use Glorot initialization scheme [211] where possible, and optimize using ADAM with a learning rate of $lr = 1.5e^{-5}$ (betas and epsilon kept the usual defaults) against binary crossentropy over 2000 epochae. Note that some other works propose a more radical training scheme with $lr = 0.01$ and exponential decay by 0.5 every 50 epochae [61], however we find GNN training in this setting highly unstable[19], and thus unsuitable for the alignment of the different implementations. We then display the actual training errors (as opposed to accuracy) as the most consistent evaluation metric for the alignment purpose in Figure 20 over the first 5 datasets for demonstration. Additionally, we report aggregated Mahalanobis pair-wise distances [212] between the training errors of the respective frameworks and models calculated over all the datasets in Table 6. While it is very difficult to align the training perfectly due to the underlying stochasticity, we can see that the performances are tightly aligned within a margin of standard deviation calculated over the folds and datasets. The differences are generally highest for the most complex GIN model, which also exhibits most unstable performance over the folds. Importantly, the differences between LRNNs and the other frameworks is generally not greater than between PyG and DGL themselves, which both utilize the exact same PyTorch modules and operations.

7.4.4 *Computing Performance*

The main aim of the declarative LRNN framework is quick prototyping of models aiming to integrate deep and relational learning capabilities, for which it generally provides more expressive constructs than that of GNNs (Section 7.3) and does not contain any specific optimizations for computation over graph data. Additionally, it introduces a startup model compilation overhead as the particular models are not specified by the user but rather automatically induced by the theorem prover. Moreover, it implements the neural training in a rather direct (but flexible) fashion of actual traversal over each network (such as in Dynet [213]), and does so without batching, efficient tensor multiplication or GPU support[20]. Nevertheless, we show that the increased expressiveness does not come at the cost of computation performance.

17 However for a precise correspondence, care must be taken to respect the same order of the (i-iii) operations, which often varies across different reports and implementations.

18 available at ftp://ftp.ics.uci.edu/pub/baldig/learning/nci/gi50/

19 as is e.g. also visible in the respective Figure 4 reported in [61].

20 However it is possible to export the networks to be trained by any neural backend rather than the native Java engine.

Table 7: Training times *per epocha* across the different models and frameworks. Additionally, the startup model creation time (theorem proving) overhead of LRNNs is displayed.

model/engine	LRNNs (s)	PyG (s)	DGL (s)	LRNNs startup (s)
GCN	**0.25 ± 0.01**	3.24 ± 0.02	23.25 ± 1.94	35.2 ± 1.3
g-SAGE	**0.34 ± 0.01**	3.83 ± 0.04	24.23 ± 3.80	35.4 ± 1.8
GIN	**1.41 ± 0.10**	11.19 ± 0.06	52.04 ± 0.41	75.3 ± 3.2

We evaluate the training times of a GCN over 10 folds of a single dataset (containing app. 3000 molecules) over the different models. We set Pytorch as the DGL backend (to match PyG), and train on CPU[21] with a vanilla SGD (i.e. batch size=1) in all the frameworks. From the results in Table 7, we see that LRNNs surprisingly train significantly faster than PyG, which in turn runs significantly faster than DGL. While the performance edge of PyG over DGL generally agrees with [204][22], the (app. 10x) edge of LRNN seems unexpected, even accounting for the startup theorem proving overhead for the model creation (giving PyG a head start of app. 10 epochae). We account the superior performance of LRNNs to the rather sparse, irregular, small, dynamic graphs for which the common vectorization techniques, repeatedly transforming the tensors there and back, often create more overhead than speedup, making it more efficient to traverse the actual spatial graph representations [213]. Additionally, LRNNs are implemented in Java, removing the Python overhead, and contain some generic novel computation compression [214] techniques (providing about 3x speedup for the GNN templates).

Note we also prevented the frameworks from batching over several graphs, which they do by embedding the adjacency matrices into a block-diagonal matrix. While (mini) batching has been shown to deteriorate model generalization [215, 216], it still remains the main source of speedup in deep learning frameworks [217], and is a highlighted feature of PyG, too. We show the additional PyG speedup gained by batching in Figure 21. While batching truly boosts the PyG performance significantly, it still lacks behind the non-batched LRNNs[23]. For illustration, we additionally include an inflated version of the GCN model by a 10x increase of all the tensor dimensionalities. In this setting we can finally observe a performance edge from mini-batching, due to vectorization and GPU[24], over the non-batched LRNNs[25].

A NOTE ON THE STARTUP OVERHEAD Recall that in the LRNNs workflow, the computation graphs are only created once for each of the learning samples during startup (Section 7.1.2). The subsequent training times are thus completely independent of reported the startup overhead stemming from the involved theorem proving (grounding) and neural network creation. An estimate of the total learning time can thus be obtained by $\mathtt{time}_{\mathtt{startup}} + 2000 \cdot \mathtt{time}_{\mathtt{epocha}}$, rendering the overhead practically negligible in all the reported experiments. Some further details on the theorem proving overhead can then be found in the appendix Section A.3.

21 We evaluated the training on CPU as in this problem setting the python frameworks run *slower* on GPU (Figure 21).

22 We note that we run both frameworks in default configurations, and there might be settings in DGL for which it does not lag behind PyG so rapidly. Note that for the small models of GCN and g-SAGE it is 10x slower, while for the bigger GIN model only 5x, which is in agreement with DGL's focus on large scale optimization.

23 While LRNNs currently do not support batching natively, it can be emulated by outsourcing the training to Dynet.

24 Also note that we used a non-high-end Ge-Force 940m, and the performance boost could thus be even more significant.

25 On the other hand note that $\mathtt{dim} = 100$ is considerably high. Most implementations we found were in range {8,16,32} and we did not observe any *test* performance improvement beyond $\mathtt{dim} = 5$ with the reported datasets and models.

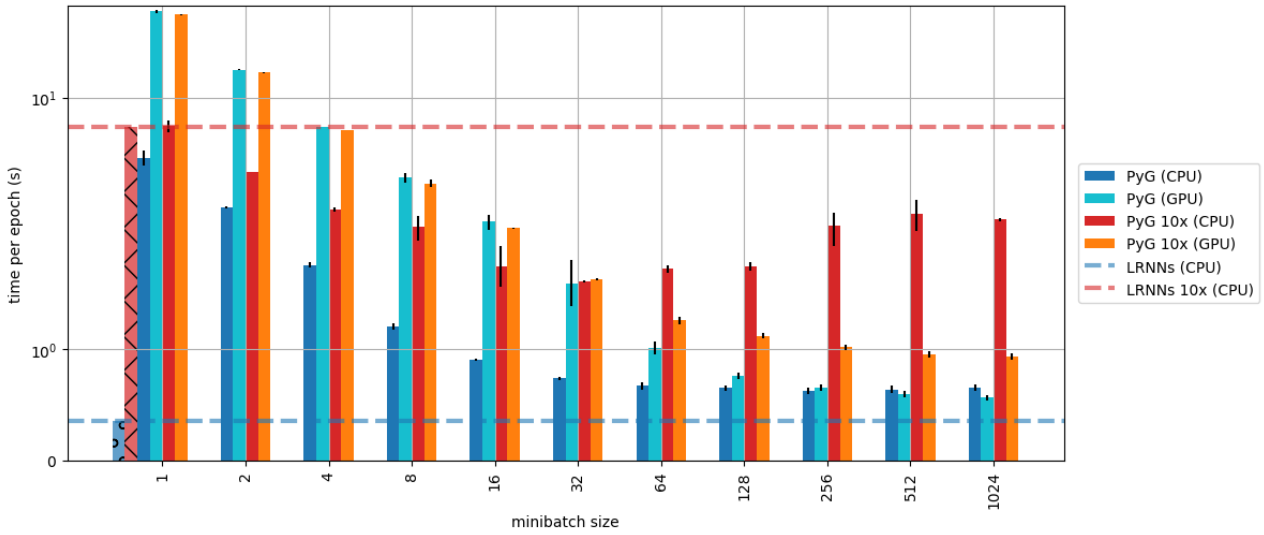

Figure 21: Improving the computing performance of PyG via mini-batching and model size blow-up. The actual GCN model (10×10 parameter matrices) and 10x inflated version (100×100 parameter matrices) as run on CPU and GPU. Compared to a non-batched (batch=1) LRNN run on CPU.

7.4.5 *Model Generalization*

Finally we evaluate learning performances of the different models. We select the discussed GNN models of GCN, g-SAGE and GIN (we keep only the PyG implementation for clarity), and we further include some example relational templates for demonstration. Particularly, we extend GIN with edge (bond) representations and associate all literals in all rules with learnable matrices (Section 7.3), denoted as "gin*". In a second template we add a layer of graphlets (motifs) of size 3, aggregating jointly representations of *three* neighboring nodes, on top of GIN, denoted as "graphlets". Lastly, we introduce latent bond learning (Section 7.3.2) into GIN, where bond (edge) representations are also aggregated into latent hierarchies, denoted as "latent_bonds". Note that we restrict these new relational templates to the same tensor dimensionalities and number of layers as GIN. For statistical soundness, we increase the number of datasets to the first 10 (alphabetically). We run all the models on the same 10-fold crossvalidation splits with a 80:10:10 (train:val:test) ratio, and keep the same, previously reported, training hyperparameters. We display the aggregated *training* errors in Figure 22, and the cross-validated *test* errors, corresponding to the best validation errors in each split, in Figure 23.

We can observe that the training performance follows intuition about capacity of each model, where the more complex models generally dominate the simpler ones. However, the increased capacity does not seem to consistently translate to better test performance (contrary to the intuition stated in [61]). While we could certainly pick a subset of datasets to support the same hypothesis on test sets, we can generally see that none of the models actually performs consistently better than another, and even the simplest models (e.g. GCN) often outperform the "powerful" ones (GIN and its modifications), and the test performances are thus generally inconclusive[26]. While this is in contrast with the self-reported results accompanying the diverse GNN proposals on similar-sized graph datasets, it is in agreement with another (much larger) recent benchmark [188].

Additionally, we include results of an old LRNN template reported in [97], denoted as "lrnn(2015)". It was based on small graphlets of size 3, similarly to the "graphlets" template (and similarly to some other works [218, 219]), however it only contained a single layer of learnable parameters for the atom and bond representations. Note that we use results from the original paper [97] experi-

26 We note that the conclusion could be different for different types of datasets.

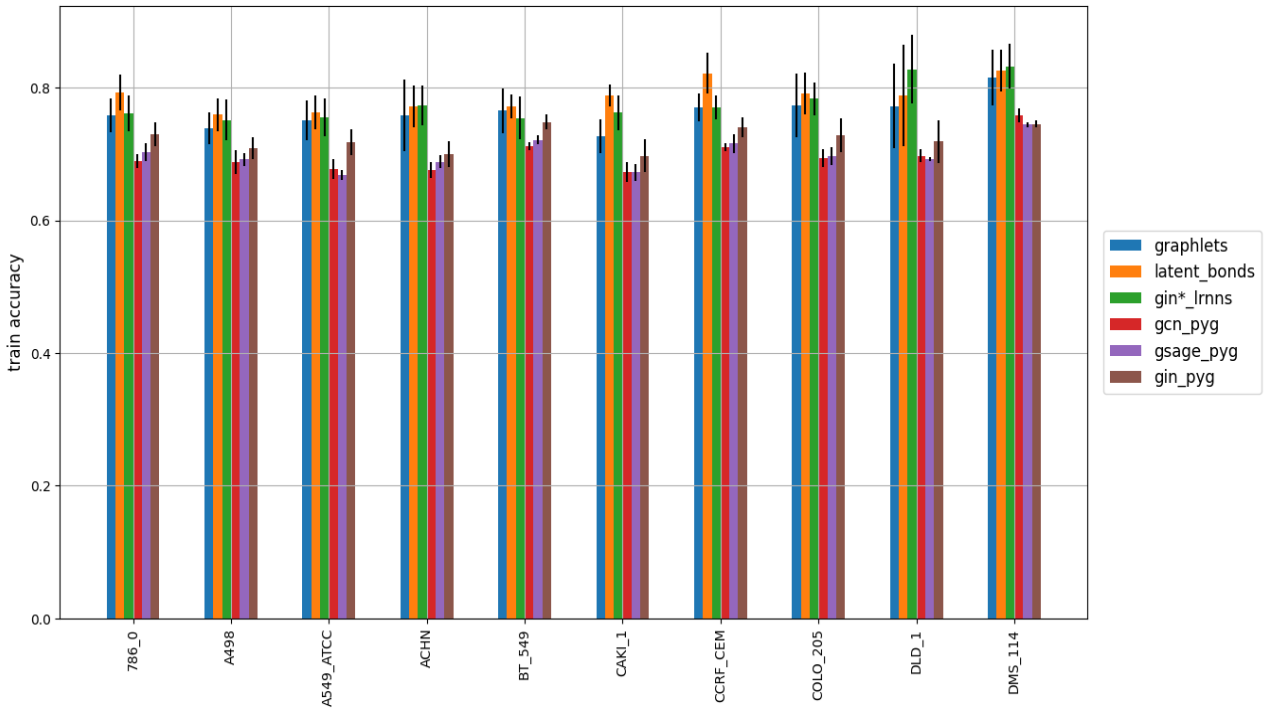

Figure 22: Comparison of *train* accuracies of selected models across 10 datasets.

ments, which were run with different hyperparameters and splits. Nevertheless, we can see that it is again generally on par with performance of the more recent, deeper, and bigger GNN models.

7.5 CASE STUDY: EXTENDING GNNS WITH RINGS FOR MOLECULE CLASSIFICATION

For their ability of direct (deep) learning from graph data, GNNs have recently become extremely popular in structured prediction tasks, such as classification of molecules [58, 78], which can be understood as attributed graphs of atoms connected via chemical bonds. Since the standard GNNs can be viewed as a continuous, differentiable version of the Weisfeiler-Lehman (WL) label propagation algorithm [59] (Section 2.4), there are also considerable limitations to this class of models, stemming from the limited expressiveness of the WL test, which is only based on the immediate neighborhood information gathered in each iteration [61, 62]. Consequently, information about more complex relational substructures, such as node rings[27], cannot be properly extracted. Naturally, however, one might ask whether such increased expressiveness actually translates into some measurable performance increase in practice.

So far in this chapter, we demonstrated how GNNs and their extensions can be easily captured in the framework of LRNNs, when viewed as a differentiable logic programming language. In this section, we extend the, mostly theoretical, discussion from Section 7.3.2 with a complete, practical case study on how to extend the GNNs by modifying the underlying propagation rules to capture *atom rings* in molecules, and experimentally evaluate the benefits of such an extension.

7.5.1 *Molecular Rings*

We build on the simple encoding of standard GNNs as introduced in Example 19. We again note that all the example templates discussed in this chapter are actual code that can be run very efficiently. Now we provide a short, but complete, demonstration on how to extended the GNNs from Example 19 with the atom rings. These rings are normally beyond the scope of the underlying WL

27 with length $\geqslant 3$

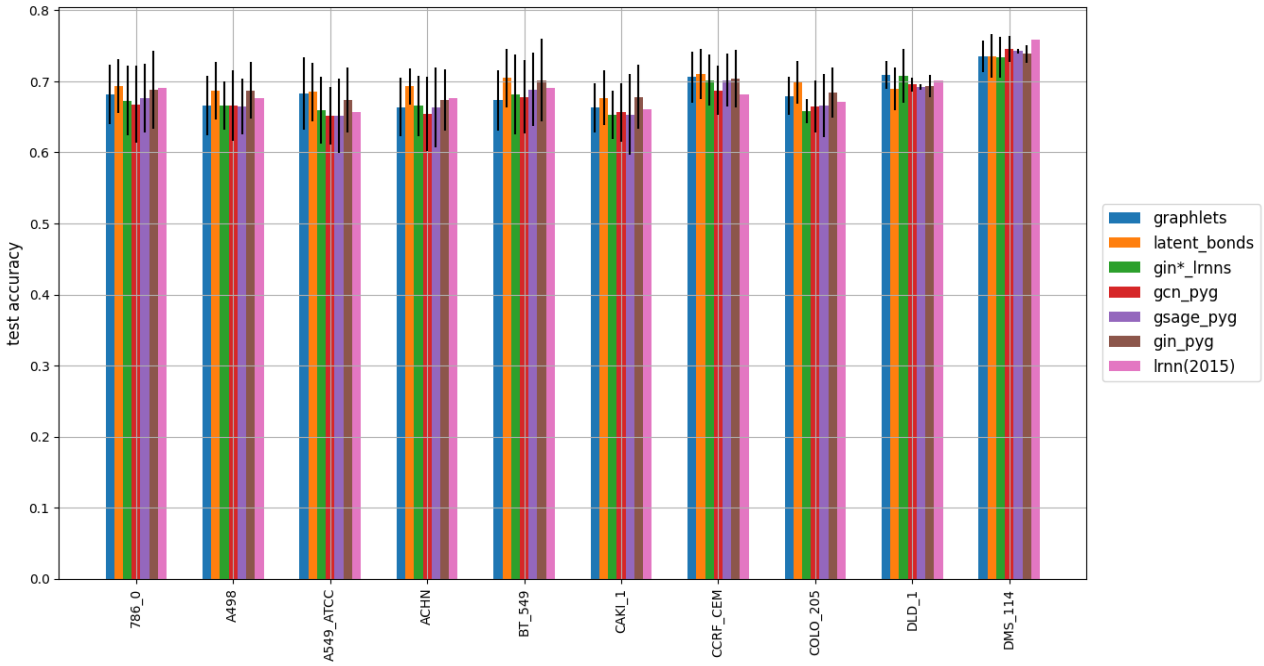

Figure 23: Comparison of *test* accuracies of selected models across 10 datasets.

expressiveness, but commonly present in aromatic molecules, and can thus be expected to be of importance in organic molecular datasets, which we target in this short case study.

Using the language of relational logic, a ring can be easily defined as a crisp pattern based on the existing bonds as

$$\_ring_6(A,\ldots,F) \text{ :- } bond(A,B), \ldots, bond(E,F), bond(F,A).$$

A distributed representation of a ring can then be declared by aggregating all the contained atoms:

$$\mathbf{W_r}::ring_6^{(n)}(A,\ldots,F) \text{ :- } \_ring_6(A,\ldots,F), \mathbf{W_a}:atom^{(n)}(A), \ldots, \mathbf{W_f}:atom^{(n)}(F).$$

These can be further stacked hierarchically, replacing the crisp $\_ring_6$ pattern by the respective distributed representation $ring_6^{(n-1)}$ from the preceding layer. Similarly, we can then also propagate the representation from the ring back into all the contained atoms by simply adding the following rule

$$\mathbf{W_{a'}}::atom^{(n)}(A) \text{ :- } \mathbf{W_{r'}}: ring_6^{(n-1)}(A,\ldots,F).$$

Note that each ring pattern will be automatically instantiated in all possible revolutions, and so it is enough to specify propagation into the "first" atom (A) only. We further refer to this template as `rings`. We note that it is a good practice to add some basic transformation to produce at least some output should a molecule contain no rings at all[28], e.g. simply aggregating all pairs of connected atoms:

$$\mathbf{W_q}::molecule \text{ :- } \mathbf{W_1}:atom(A), \mathbf{W_2}:atom(B), bond(A,B).$$

Naturally, this can be further combined with the standard GNN propagation scheme (Ex. 19), where we aggregate the direct neighbor representations:

$$\mathbf{W_h}::atom^{(n)}(X) \text{ :- } \mathbf{W_1}':atom^{(n-1)}(Y), bond(X,Y).$$

In each layer n, the representation of an atom will thus get updated based on the neighbors as well as all the atoms occupying the same rings. The particular contributions to the representation update are then parameterized separately. We further refer to this joint template as `rings + gnn`.

28 Otherwise such an un-entailed sample would be considered as a counter-example (in the ILP sense, Section 3.3.1).

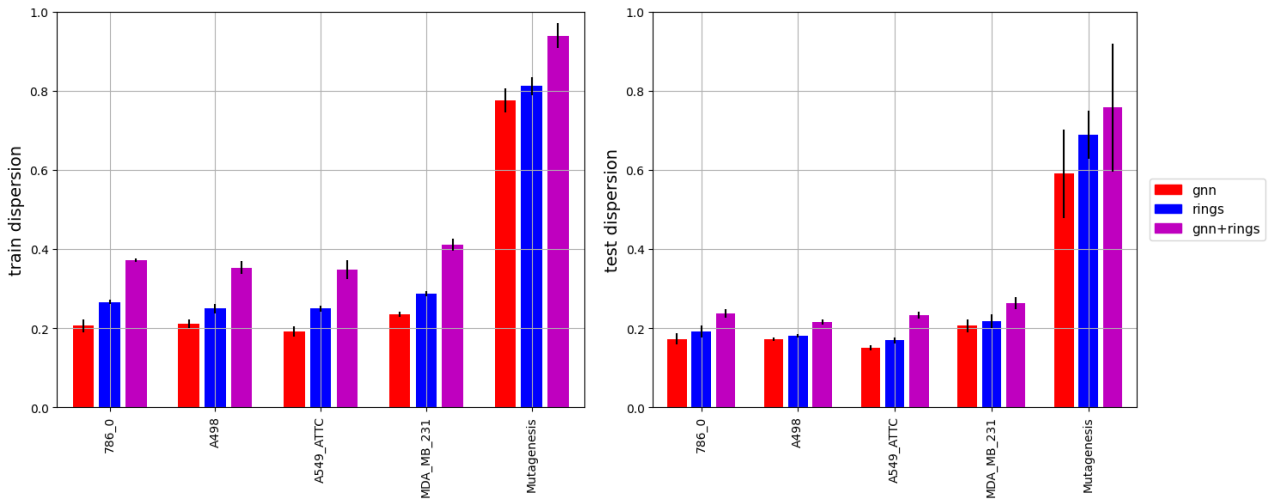

Figure 24: Comparison of the gnn, rings, and the joint gnn + rings learners across several molecule classification datasets w.r.t. training (left) and testing (right) dispersion (higher is better).

7.5.2 *Experiments*

For the experiments we encoded both 5 and 6 rings, added 3 layers for both the gnn and rings embedding propagation, and set all the learnable tensor to randomly initialized 3×3 matrices. We then evaluated the architectures across several datasets for molecule classification [172–174], ranging from 188 to app. 3500 molecules. We note we only used the basic information on atom types and their conformations. We left all the hyperparameters of the framework on their defaults, learning with tanh activations, avg aggregations, and ADAM optimizer, trained for 2000 steps, and evaluated with a 5-fold cross-validation. The results are displayed in Figure 24. We can see that the rings template indeed performs better than the basic gnn learner, while both benefit from their mutual combination.

7.5.3 *Discussion*

We have demonstrated the practical side of the LRNN encoding on a small learning scenario designed to extend the basic GNN idea to capture atom rings in molecules. Note that all we had to do was to declare a couple of simple rules specifying (i) what a ring is and (ii) how to update representation of its contained atoms. The derivation of the particular dynamic computation graphs, encoding the respective propagation scheme for each of the differently structured molecules, the corresponding gradients, training and evaluation were then all performed completely automatically. This allows for quick prototyping of diverse relational modelling ideas, where one can stack complex structural patterns to be trained end-to-end with gradient descent. In the particular showcase just demonstrated, we were then able to quickly assess the contribution of atom rings w.r.t basic GNNs.

7.6 CONCLUSIONS

In this chapter, we re-introduced[29] LRNNs (Chapter 5) as a declarative, differentiable logic programming approach for specification of advanced deep learning architectures. We demonstrated how simple parameterized logic programs, also called templates, can be efficiently used for declaration and optimization of complex convolutional models, with a particular focus on Graph Neural

29 Some technical details on the differences between this new view on LRNNs and the original framework from Chapter 5 are discussed in the appendix Section A.

Networks (GNNs). In contrast with the commonly used procedural (Python) frameworks, LRNNs abstract away the creation of the specific computation graphs, which are dynamically unfolded from the template by an underlying theorem prover. As a result, creating a diverse class of complex neural architectures reduces to rather trivial modifications of the templates, distilling only the high level idea of each architecture. We illustrated versatility of the approach on a number of examples, ranging gradually from simple neural models to complex GNNs, including very recent GNN models and their extensions. Finally we showed how the existing models can be easily extended to even higher relational expressiveness.

In the experiments, we then demonstrated correctness and computation efficiency by the means of comparison against modern deep learning frameworks. We showed that while LRNNs are designed with main focus on expressiveness, flexibility and abstraction, they do not suffer from computation inefficiencies for the simpler (GNN) models, as one might expect. On the contrary, we demonstrated that for a range of existing GNN models and their practical parameterizations, LRNNs actually outperform the existing frameworks optimized specifically for GNNs. Additionally, we also demonstrated practical usefulness of the LRNN encoding approach in a case study of extending GNNs with atom rings for molecule classification.

While there is a number of related works targeting the integration of deep and relational learning, to our best knowledge, capturing advanced convolutional neural architectures in an exact manner, as exemplified in this chapter, would not be possible with the other approaches. The proposed relational upgrades can then be understood as proper extensions of the existing, arguably popular, GNN models.

However, we also showed that generalization performance of the various state-of-the-art GNN models is somewhat peculiar, as they actually performed with rather insignificant test error improvements, when measured uniformly over a large set of medium-sized, molecular structure-property prediction datasets, which is actually in agreement with another recent benchmark [188].

8

RELATED WORK

In this chapter, we discuss the more recent[1], works related to the framework. In contrast to the prior work discussed in Chapter 4, the more recent body of work covered in this chapter can also be distinguished through the noticeable influence by the "modern deep learning" practices. In the recent years, we have also seen much more fusion between the different communities of NSI (Section 4.3), SRL (Section 4.1), and also increased interest in relational learning from the "main stream" deep learning community (Section 4.2), especially with the recent resurgence of the GNNs. The respective categorization of the related work w.r.t. these communities (Chapter 4) thus does not seem useful any longer.

Instead, we here view the modern streams of the related works roughly as (i) "Deep Relational Learning", representing models focused more on enhancing neural networks with relational expressiveness, akin to the Chapter 5, and (ii) "Differentiable Logic Programming", with works exhibiting some aspects of the declarative differentiable encoding of relational learning scenarios, akin to the Chapter 7. Nevertheless, we note that the two research streams are naturally tightly interconnected and overlapping.

8.1 DEEP RELATIONAL LEARNING

Shortly after LRNNs, two other frameworks for combining relational logic and neural networks have been introduced [220, 221]. Neural Theorem Provers (NTPs) [220, 222] share a very similar approach with LRNNs through the use of a definite clause logic templates with an underlying theorem prover to derive ground computation graphs, which are differentiable under certain semantics inspired by fuzzy logic. The use of parameterization slightly differs between the frameworks, where NTPs are focused on learning embeddings of constants and LRNNs on embeddings of whole relational constructs[2]. Consequently, NTPs represent *all* constants implicitly as (embedding) vectors, for which the theorem prover cannot perform standard unification, and NTPs thus introduce "soft-unification" [223], returning a value which is determined by dot-products of the underlying embeddings fed into a sigmoid activation. While nicely relaxing classic unification, this leads to effectively trying all possible constant combinations in the inference process. Additionally, this prevents from using NTPs for explicit modeling of the exemplified convolutional neural architectures (Section 7.3), and also severely limits NTP's scalability, where the latter has been partially addressed by some recent NTP extensions [224, 225]. LRNNs are more flexible in this sense as one can use the parameterization to specify which parts of the program keep the logical structure, enabling for much more efficient evaluation (e.g. with magic sets [95]), and which parts should succumb themselves to the exhaustive numerical optimization (and to combine them arbitrarily), possibly enabling to find a more fine-grained neural-logic trade-off.

For interest, we describe how to implement a similar modeling concept to NTPs within the LRNNs (Chapter 7).[3] Particularly, the soft unification can be modeled by extending the template

1 i.e published after the original LRNNs [97] (Chapter 5)
2 Note that this includes learning embeddings of constants, too, as demonstrated in a number of the example templates.
3 We note that this could not have been done in full with the older version of LRNNs from Chapter 5.

with an additional rule in the form of: $match(X,Y) :- emb(X), emb(Y)$ with the values of the facts $emb(c_i)$ being the embeddings of the (soft) constants c_i (Section 7.1.1.1). For this particular rule, we then set the $g_\wedge$ as the dot product and $g_\vee$ as the sigmoid, making the $match(X,Y)$ literal yield the soft unification values for all the (relevant) pairs of constants substituted for X, Y. The $match(X,Y)$ literal can then be used in the rules requiring the soft-unification, where we further choose both the activation functions $g_\vee$ and g_* as the maximum, since in NTPs, any situation where different unifying substitutions to the same variables need to be aggregated, is resolved by taking the maximum value. Hence, it should be possible to represent models from the NTP framework in LRNNs. However, other modeling constructs with distributed representation learning are actually more natural to encode in LRNNs (Section 5.2, Section 7.2).

Another system introduced about the same time is called TensorLog [221, 226], which is a differentiable probabilistic database based on belief propagation. TensorLog implements a subset of Datalog (Section 3.2). It restricts the factor graphs constructed for the belief propagation step to be tree-like. Further restrictions include the fact that only unary and binary predicates are allowed, and only certain types of queries are supported. Because of these restrictions, it would be difficult to directly compare TensorLog with LRNNs. Both approaches seem to be tailored towards different types of tasks. One advantage of TensorLog is that it does not require a complete grounding of the set of rules to perform inference. While we have relied on complete groundings in the original framework[4], even for LRNNs it would be sufficient to limit the grounding to the proofs of the given query formula.[5].

More loosely related are methods for incorporating domain knowledge in neural networks [227, 228], which can mostly be seen as a form of regularization [229, 230]. For instance, these include "lifted rule injection" [231], allowing to incorporate implication rules into distributed representations for the task of knowledge base construction. While somewhat related (Section 11.4.4), it is restricted to binary predicates, and only considers rules with a single atom in the body.

A well-substantiated line of research called Deep Relational Machines [46] also utilizes knowledge to enhance NNs. It is, however, based on utilizing ILP [232] and other advanced discrete search-space techniques to generate relational features for neural networks [233, 234][6]. In a somewhat similar fashion, "Neural Networks for Relational Data" [235] generate relational features from path-constrained random walks over input data. The relational features here thus take the form of linear chains of binary predicates, the instantiations of which in the data are then weighted, aggregated and sent to a logistic regression. The authors then compare this architecture against custom implementation of some other models, including similar propositionalization-based NNs, such as what the authors present as the "LRNN", to demonstrate superiority of their architecture. We note that the paper contains a number of misguided claims[7], and the model used for the comparisons does not actually represent the LRNN concept at all. We encourage the authors of [235] to compare with the actual LRNN system. While the authors of [235] themselves claim relatedness (and overall superiority) of their architecture over LRNNs, we take the liberty to clarify the main differences here, too. Firstly, the architecture of [235] is actually limited to a single (shallow) layer of rules, simi-

4 However we introduce an optimization technique to compress the groundings in Chapter 9.

5 However, this requires a fast top-down inference engine (Section 3.2.1.2) and in our experiments, we have found such top-down grounding of LRNNs to be actually slower than the bottom-up grounding (Section 5.1.1.1).

6 Recently, the Deep Relational Machines [46], when combined with an appropriate background knowledge, actually achieved better performance than that of the original LRNNs for the datasets reported in Section 5.3.

7 particularly, regarding LRNNs, statements such as "While Šourek et al. [97] exploit tied parameters across facts, we share parameters across multiple instances of the same rule". Obviously that is *exactly* what LRNNs are doing as their main characteristic, as explained in the paper used as a reference [97] by the authors. Similarly, "The formulation of this [aggregation] layer is much more general and subsumes the approach of Sourek et al (2018) [97], which uses a max combination layer". It is obvious that the choice of any NN activation/aggregation function is arbitrary in this sense, nevertheless, even in the original paper being referenced [97] we already presented the two example choices of both max and avg (standard pooling), which is actually what the authors of [235] used, *too*. We advise the authors to actually read the papers [97, 202] and compare with the actual system (with more than 1 layer, which is, obviously, a very degenerate case [4]) in order to beat it on benchmarks, which certainly is possible (see e.g. the Deep Relational Machines [46]).

larly to [39], restricting from modeling any latent (neural) relational concept (Section 5.2), which is a defining LRNN characteristic. Secondly, these rules are limited to linear chains of binary predicates. Thirdly, all the literals from the same rule share the same single weight here which, in combination with the shallow rule architecture, is not actually a "relational parameter tying" feature [235] but a considerable limitation meaning that there is actually no useful parameter tying in the neural model at all, restricting from expressing any interesting convolutional pattern (Section 7.2.2), and making the neural model (logistic regression) effectively propositional. On the other hand, the rules here are automatically extracted by the relational random walks, as opposed to some other propositionalization schemes, which is a neat contribution to this class of models.

The described concept then seems architecturally very close to a (prior) model called Relational Logistic Regression [236], which has been later extended into another relational neural network model called RelNN[8] [237] by stacking multiple of these relational regressors on top of each other. This model, finally, indeed resembles the LRNNs idea very closely, while the architecture of the RelNN concept seems practically equivalent to the soft-clustering concept (Section 5.2.1) introduced in the original LRNNs [97]. Unfortunately the paper [237], while technically sound, again contains unsubstantiated (false) statements about LRNNs[9], and we encourage the readers to check the claimed "flexibility" and modularity of RelNNs[10] over the LRNNs[11] themselves (which is convenient, as they are both standalone Java projects). Actually, lacking explicit representation of the template and general logic program inference (Section 3.2.1) the, notably trivial[12], RelNN model is *hardcoded* to the exact scenario of unary-predicate soft-clustering and the three simplified datasets presented in the paper. Interestingly, even the future works proposed in the RelNN paper [237], such as the structure learning, parallelization, database querying and relational dropout, had already been implemented in the LRNNs. On the other hand, the authors provide interesting connections to the probabilistic models and MLNs (Section 3.4.2).

There are also interesting works utilizing standard, purely sub-symbolic, deep learning techniques, aimed at learning semantics of logic programs from their raw (character-level) encoding [34]. Inspired by the (recurrent) architectures with differentiable memory [32, 33] (Section 4.2), these advanced engineering feats provide useful insights into representation learning of the dynamic symbolic structures within the common fixed-size embedding spaces, and the resulting disadvantages w.r.t. the logic-based systems for representation learning [238, 239]. However, there are also some highly interesting deep learning works utilizing even more advanced neural architectures, such as the transformer networks [86] in combination with the GNNs and gating, successfully addressing similarly complex problems based on dynamic graph representations [49]. These can be again seen along the lines of the advanced models with differentiable memory representations, however, they address many of their shortcomings thanks to the explicit graph representation of the internal state.

Much of the most recent work using neural networks to target relational data is then based on the GNNs. As discussed in Section 7.3.2, LRNNs can be seen as a generalization of GNNs. From the graph-level perspective, the most similar idea to the introduced relational templating has become popular in the knowledge discovery community as *"meta-paths"* [198, 199] defined on the schema-level of a heterogeneous information network. A meta-path is simply a sequence of types, the concrete instantiations of which are then searched for in the ground graphs (similarly to the re-

8 not to be confused with the prior RelNN model [56] (Section 4.2)

9 Quotation: "Sourek et al. (2015)'s models are the closest proposals to RelNNs, but RelNNs are more flexible in terms of adding new types of layers in a modular way. These works are also limited in one or more of the following ways: 1- the model is limited to only one input relation, 2- the structure of the model is highly dependent on the input data, 3- the model allows for only one hidden layer, 4- the model cannot learn hidden object properties through general rules, or 5- the model does not scale to large domains." – all of which are literally, one by one, *false*. We encourage the authors to actually check their claims (next time).

10 https://github.com/Mehran-k/RelNN

11 https://github.com/GustikS/NeuraLogic - further also described in Chaoter B.

12 The presented RelNN [237] system [10] (2018) is actually a trivial fragment of even the very limited 2013's precursor [96] of LRNNs, which came before the relational logistic regression [236] itself.

lational walks from [235]). Such ground sequences can then be used to define node similarities [240, 241], random walks [198] as well as node embeddings [242, 243]. An extension from paths to small directed acyclic graphs was then proposed as "meta-graph" (or "meta-structure") [244, 245]. We note that any meta-path or meta-graph can be understood as a conjunctive rule in a LRNN template (Section 3.2). Naturally, we can stack multiple meta-graphs to create deep hierarchies and, importantly, differentiate them through to jointly learn all the parameters, and provide further extensions towards relational expressiveness, as exemplified in Section 7.3.1.

In a similar way to GNNs, so-called discriminative Gaifman models [246] were introduced. These are models that aggregate information from locally sampled neighborhoods, motivated by Gaifman's locality theorem [247]. In another work [248], the authors used GNNs to address the scalability issue of MLNs (Section 3.4.2). The GNN representation here is used for an efficient stochastic training of MLNs within the framework of variational EM, very similarly to an earlier work of [249], which used flat NN embeddings for the same purpose.

An interesting theoretical work connecting GNNs and logic was then recently presented in [250], where the authors studied their expressiveness in terms of (logical) boolean node classifiers (as opposed to the WL test itself).

Encouragingly, there is also a growing number of application-oriented works utilizing the integrated learning paradigm, ranging from the, closely related, planning domain [251, 252], to natural language processing [253] and understanding [254, 255]. Most recently, the paradigm has even brought attention of big corporations [256].

8.2 DIFFERENTIABLE LOGIC PROGRAMMING

The most closely related works here naturally comprise of other differentiable programming languages with relational expressiveness[13], and there is a number of works targeting similar abilities by extending logic programming with numerical parameters [24]. The most prominent framework in this category is the language of Problog [52], where the parameters and values further posses probabilistic interpretation (Section 3.4.1). The extension to Deep-Problog [69] then incorporates "neural predicates" into Problog programs. Since probabilistic logic programs can be differentiated [259] and trained as such, the gradients can be passed from the logic program to the neural modules and trained jointly. While this is somewhat similar to LRNNs, Deep-Problog introduces a clear separation line between the neural and logical parts of the program, which communicate merely through the gradient values (and so any gradient-based learner could be used instead). The logical part with relational expressiveness is thus completely oblivious of structure of the gradient-ingesting learner and vice versa, and it is thus impossible to model complex convolutional patterns (i.e. relational patterns in the neural part) as demonstrated in Chapter 7. On the other hand LRNNs do not have probabilistic interpretation, which is elegantly incorporated in Deep-Problog. Related is also an extension of kProblog [260], proposing integration of algebraic expressions into logic programs towards more general tensor-algebraic and ML algorithms. A very interesting extension from the language formalism perspective is targeting the increased answer-set programming expressiveness with neural networks [261]. Within the classic SRL, apart from neuralization of the Problog, a neural variant of the MLNs (Section 3.4.2) has also been introduced by defining the potential functions as generic neural networks which are trained together with the model parameters in a tightly integrated (sub-symbolic) manner [262].

13 Note that encoding computation graphs in common differentiable programming frameworks, such as PyTorch or TensorFlow, is effectively propositional. These frameworks provide sets of evaluation functions (modules), with predefined hooks for backward differentiation, that can be assembled by users into differentiable programs in a *procedural* fashion. In contrast, with relational programming, such programs are firstly automatically assembled from the *declarative* template (by a theorem prover or grounder), and only then evaluated and differentiated in the same fashion. Such an approach can also be understood as "meta-programming" [257, 258] from the perspective of the current procedural frameworks.

Another line of work is focused on inducing Datalog programs with the help of numerical relaxation (Chapter 6). While such a task has traditionally been addressed by the means of Inductive Logic Programming (ILP) [19], extending the rules with weights can help to relax the combinatorial search into a gradient descent optimization, while providing robustness to noise. An example of such an approach is δILP [27]. Similarly to LRNNs, the Datalog programs here are unfolded by chaining the rules, where the associated parameters are trained against given target to be solved by the program. The parameterization in these approaches is used differently as its purpose is to determine the right *structure* of the template (in contrast to our structure learning from Chapter 6). This is typically done by exhaustive enumeration from some restricted set of possible literal combinations (particularly 2 literals with arity at most 2 and no constants for δILP), where each combination is then associated with a weight to determine its appropriateness for the program via gradient descent. The differentiability is again based on replacing the logical connectives with fuzzy logic operators (particularly product t-norm). A very similar rule-inducing system was also proposed in [263] to target comparable toy ILP tasks. Another recently proposed related system is called Difflog [264], where the candidate rules are also exhaustively generated w.r.t. a more narrow language bias, thanks to which it seem to scale beyond the previous systems. A different way to scale up differentiable ILP even more is by skipping the exhaustive enumeration part, while giving up on the explicit logical representations even further, and relaxing the symbolic inference in distributed embedding spaces processed through advanced neural architectures [265].

Closely related class of approaches here target full FOL expressiveness by providing mapping of all the logical constructs into numerical (tensor) spaces (also referred to as "tensorization" [36]). For instance, one can cast constants to vectors, functions terms to vector functions of the corresponding dimensionality, and similarly predicates to tensors of the corresponding arity-dimension [266, 267]. Again adopting a fuzzy logic interpretation of the logical connectives [268], the learning problem can then be cast as a constrained numerical optimization problem, including works such as LYRICS [269] and Logic Tensor Networks [42]. While the distributed representation of the logical constructs is the subject of learning, in contrast with the discussed Datalog program structure learning approaches, the weight (strength) of each rule needs to be specified apriori – a limitation which was recently addressed in [44, 270]. Other recent works based on the idea of fully dissolving the logic into tensors, moving even further from the logical interpretation, include e.g. Neural Logic Machines [43]. While these frameworks are theoretically even more expressive than LRNNs (lacking the function terms and non-definite clauses), the whole logic interpretation is only approximate and completely dissolved in the tensor weights in these frameworks. Consequently, they again lack the capability of precise relational logic inference chaining, based on the underlying theorem prover, which we use to explicitly model the advanced convolutional neural structures, such as the GNNs, in Chapter 7.

Part IV

OPTIMIZATIONS

In this part of the thesis, we introduce two principled methods for improving scalability of the LRNN framework introduced in the previous part. Firstly in Chapter 9, we show how to speedup the parameter learning part via lossless compression of the induced neural networks. The technique, inspired by lifted inference, is then generally useful for speeding up of any other structured, weight-sharing models, such as the GNNs, too. The second technique, which we introduce in Chapter 10, then improves complexity of the structure learning part via lossless pruning of the searched hypothesis space. Again, the technique is also generally useful for any other relational logic learner following the classic ILP strategy.

9

LOSSLESS MODEL COMPRESSION VIA LIFTING

As introduced in the respective SRL Section 3.4.2, lifting is an efficient technique to scale up graphical models generalized to relational domains by exploiting the underlying symmetries. Concurrently, neural models (Chapter 2) are continuously expanding from grid-like tensor data into structured representations, such as various attributed graphs and relational databases. To address the irregular structure of the data, these models typically extrapolate on the idea of convolution, effectively introducing parameter sharing in their, dynamically unfolded, computation graphs (e.g. Section 2.3 and Section 2.4). The computation graphs themselves then reflect the symmetries of the underlying data, similarly to the lifted graphical models.

Inspired by lifting, this chapter introduces a simple and efficient technique to detect the symmetries and compress the neural models without loss of any information. We then demonstrate through experiments that such compression can lead to significant speedups of various "structured convolutional models" across diverse tasks, such as molecule classification and knowledge-base completion. This technique is applicable not only to the introduced LRNNs (Part iii) but, importantly, also various popular models such as standard Graph Neural Networks.

9.1 INTRODUCTION

Lifted, often referred to as *templated*, models use highly expressive representation languages, typically based in weighted predicate logic, to capture symmetries in relational learning problems [271]. This includes learning from data such as chemical, biological, social, or traffic networks, and various knowledge graphs, relational databases and ontologies. As detailed in Section 3.4.2, the idea has been studied extensively in probabilistic settings under the notion of lifted graphical models [63], with instances such as Markov Logic Networks (MLNs) [51] or Bayesian Logic Programs (BLPs) [50].

In a wider view, *convolutions* can be seen as instances of the templating idea in neural models, where the same parameterized pattern is being carried around to exploit the underlying symmetries, i.e. some forms of shared correlations in the data. In this analogy, the popular Convolutional Neural Networks (CNN) (Section 2.2) themselves can be seen as a simple form of a templated model, where the template corresponds to the convolutional filters, unfolded over regular spatial grids of pixels. But the symmetries are further even more noticeable in structured, relational domains with discrete element types. With convolutional templates for regular trees, the analogy covers Recursive Neural Networks (Section 2.3), popular in natural language processing. Extending to arbitrary graphs, the same notion covers works such as the Graph Convolutional Networks (Section 2.4) and their variants [58], as well as various Knowledge-Base Embedding methods [272]. Extending even further to relational structures, there are works integrating parameterized relational logic templates with neural networks [69, 222, 262], including the introduced LRNN framework (Part iii).

The common underlying principle of templated models is a joint parameterization of the symmetries, allowing for better generalization. However, standard lifted models, such as MLNs, provide another key advantage that, under certain conditions, the model computations can be efficiently carried out without complete template unfolding, often leading to even exponential speedups [63]. This is known as "lifted inference" [273] and is utilized heavily in lifted graphical models as well

as database query engines [274]. However, to our best knowledge, this idea has been so far unexploited in the neural (convolutional) models. The main contribution of this chapter is thus a "lifting" technique to compress symmetries in convolutional models applied to structured data, which we refer to generically as "structured convolutional models".

9.1.1 *Related Work*

The idea for the compression is inspired by lifted inference [273] used in templated graphical models. The core principle is that all equivalent sub-computations can be effectively carried out in a single instance and broadcasted into successive operations together with their respective multiplicities, potentially leading to significant speedups. While the corresponding "liftable" template formulae (or database queries) generating the isomorphisms are typically assumed to be given [63], we explore the symmetries from the unfolded ground structures, similarly to the approximate methods based on graph bisimulation [275]. All the lifting techniques are then based in some form of first-order variable elimination (summation), and are inherently designed to explore *structural* symmetries in graphical models. In contrast, we aim to additionally explore *functional* symmetries, motivated by the fact that even structurally different neural computation graphs may effectively perform identical function.

The learning in neural networks is also principally different from the model counting-based computations in lifted graphical models in that it requires many consecutive evaluations of the models as part of the encompassing iterative training routine. Consequently, even though we assume to unfold a complete computation graph before it is compressed with the proposed technique, the resulting speedup due to the subsequent training is still substantial. From the deep learning perspective, there have been various model compression techniques proposed to speedup the training, such as pruning, decreasing precision, and low-rank factorization [276]. However, to our best knowledge, all the existing techniques are lossy in nature and do not exploit the model computation symmetries. The most relevant line of work here are the LRNNs (Chapter 5) which however, despite the name, so far provided only templating capabilities without lifted inference, i.e. with complete, uncompressed ground computation graphs, as demonstrated in Section 5.1.2.3.

9.2 BACKGROUND

The compression technique described in this chapter is applicable to a number of structured convolutional models, ranging from simple recursive (Section 2.3) to fully relational neural models (Part iii) The common characteristic of the targeted learners is the utilization of convolution (templating), where the same parameterized pattern is carried over different subparts of the data (representation) with the same local structure, effectively introducing repetitive sub-computations in the resulting computation graphs, which we exploit in this work.

The most prominent representatives within this category are currently Graph neural networks (GNNs), which is why we choose them for brevity of demonstration of the proposed compression technique. Note that in this chapter, we directly build on the GNN computation principles detailed in the background Section 2.4. For now, recall that GNNs work by dynamically unfolding a parameterized computation template, interleaving convolution, aggregation and combination, over input graphs of varying structure. An example computation graph of a generic GNN unfolded over an example molecule of methane can then be seen in Figure 26.

9.2.1 *Computation Graphs*

The general idea of a computation graph has been introduced in the deep learning Chapter 2. For the sake of this chapter, let us now redefine the notion of a computation graph more formally. A

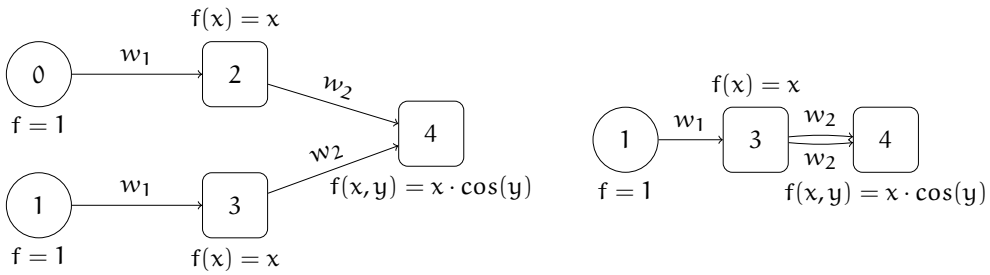

Figure 25: Depiction of the computation graph (left) compression (right) from Example 21.

computation graph is a tuple $G = (\mathcal{N}, \mathcal{E}, \mathcal{F})$, where $\mathcal{N} = (1, 2, \ldots, n)$ is a list of *nodes* and $\mathcal{E} \subseteq \mathcal{N}^2 \times \mathbb{N}$ is a list of directed labeled *edges*. Each labeled edge is a triple of integers (n_1, n_2, l) where n_1 and n_2 are nodes of the computation graph and l is the *label*. The labels are used to assign weights to the edges in the computation graph. Note this allows to define the weight sharing scheme as part of the graph (cf. Example 21 below). Finally, $\mathcal{F} = \{f_1, f_2, \ldots, f_n\}$ is the list of *activation functions*, one for each node from $\mathcal{N}$. As usual, the graph is assumed to be acyclic. Children of a node N are naturally defined as all those nodes M such that $(M, N, L) \in \mathcal{E}$, and analogically for parents. Note that since $\mathcal{E}$ is a list, edges contained in it are ordered, and the same edge may appear multiple times (which will be useful later). Children of each node are also ordered – given two children C and C' of a node N, C precedes C' iff (C, N, L) precedes (C', N, L') in the list of edges $\mathcal{E}$. We denote the lists of children and parents of a given node N by $\mathit{Children}(N)$ and $\mathit{Parents}(N)$, respectively. Computation graphs are then evaluated bottom up from the leaves of the graph (nodes with no children) to the roots of the graph (nodes with no parents). Given a list of weights $\mathcal{W}$, we can now define the *value* of a node $N \in \mathcal{N}$ recursively as:

$$\mathit{value}(N; \mathcal{W}) = f_N\Big(\mathcal{W}_{L_1} \cdot \mathit{value}(M_1; \mathcal{W}), \ldots, \mathcal{W}_{L_m} \cdot \mathit{value}(M_m; \mathcal{W})\Big),$$

where $(M_1, \ldots, M_m) \equiv \mathit{Children}(N)$ is the (ordered) list of children of the node N, and $L_1, \ldots, L_m$ are the labels of the respective edges $(M_1, N, L_1), \ldots, (M_m, N, L_m) \in \mathcal{E}$, and $\mathcal{W}_{L_i}$ is the L_i-th component of the list $\mathcal{W}$. Note that with the structured convolutional models, such as GNNs, we assume dynamic computation graphs (Chapter 2) where each learning sample S_j generates a separate G_j. Consequently, we can associate the leaf nodes in each G_j with constant functions[1], outputting the corresponding node (feature) values from the corresponding structured input sample S_j.

9.3 PROBLEM DEFINITION

The problem of detecting the symmetries in computation graphs can then be formalized as follows.

Definition 9 (Problem Definition) *Let $G = (\mathcal{N}, \mathcal{E}, \mathcal{F})$ be a computation graph. We say that two nodes N_1, N_2 are equivalent if, for any $\mathcal{W}$, it holds that $\mathit{value}(N_1; \mathcal{W}) = \mathit{value}(N_2; \mathcal{W})$. The problem of detecting symmetries in computation graphs asks to partition the nodes of the computation graph into equivalence classes of mutually equivalent nodes.*

Example 21 *Consider the computation graph $G = (\mathcal{N}, \mathcal{E}, \mathcal{F})$, depicted in Figure 25, where*

$$\mathcal{N} = \{0, 1, 2, 3, 4\},\ \mathcal{E} = ((0, 2, 1), (1, 3, 1), (2, 4, 2), (3, 4, 2)),$$
$$\mathcal{F} = \{f_0 = f_1 = 1, f_2(x) = f_3(x) = x, f_4(x, y) = x \cdot \cos(y)\}.$$

1 in contrast to static computation graphs where these functions are identities requiring the features at input.

Let $\mathcal{W} = (w_1, w_2)$ be the weight list. The computation graph then computes the function $(w_1 w_2) \cdot \cos(w_1 w_2)$. It is not difficult to verify that the nodes $\{0, 1\}$, and $\{2, 3\}$ are functionally equivalent. This also means, as we discuss in more detail in the next section, that we can "merge" them without changing the function that the graph computes. The resulting reduced graph then has the form

$$\mathcal{N} = \{1, 3, 4\},\ \mathcal{E} = \{(1, 3, 1), (3, 4, 2), (3, 4, 2)\},$$
$$\mathcal{F} = \{f_1 = 1, f_3(x) = x, f_4(x, y) = x \cdot \cos(y)\}.$$

In the example above, the nodes $\{0, 1\}$ and $\{2, 3\}$ are in fact also isomorphic in the sense that there exists an automorphism (preserving weights and activation functions) of the computation graph that swaps the nodes. Note that our definition is less strict: all we want the nodes to satisfy is *functional* equivalence, meaning that they should evaluate to the same values for any initialization of $\mathcal{W}$.

We will also use the notion of *structural-equivalence* of nodes in computational graphs. Two nodes are structurally equivalent if they have the same outputs for any assignment of weights $\mathcal{W}$ and for any replacement of any of the activation functions in the graph.[2] That is if two nodes are structurally equivalent then they are also functionally equivalent but not vice versa. Importantly, the two nodes do not need to be automorphic[3] in the graph-theoretical sense while being structurally equivalent, which also makes detecting structural equivalence easier from the computational point of view. In particular, we describe a simple polynomial-time algorithm in Section 9.4.2.

9.4 TWO ALGORITHMS FOR COMPRESSING COMPUTATION GRAPHS

In this section we describe two algorithms for compression of computation graphs: a non-exact algorithm for compression based on functional equivalency (cf. Definition 9) and an exact algorithm for compression based on detection of structurally-equivalent nodes in the computation graph. While the exact algorithm will guarantee that the original and the compressed computation graphs represent the same function, that will not be the case for the non-exact algorithm. Below we first describe the non-exact algorithm and then use it as a basis for the exact algorithm.

9.4.1 *A Non-Exact Compression Algorithm*

The main idea behind the non-exact algorithm is almost embarrassingly simple. The algorithm first evaluates the computation graph with n randomly sampled parameter lists $\mathcal{W}_1, \ldots, \mathcal{W}_n$, i.e. with n random initializations of the (shared) weights, and records the values of all the nodes of the computation graph (i.e. n values per node). It then traverses the computation graph from the output nodes in a breadth-first manner, and whenever it processes a node N, for which there exists a node N′ that has not been processed yet and all n of its recorded values are the same as those of the currently processed node N, the algorithm replaces N by N′ in the computation graph. In principle, using larger n will decrease the probability of merging nodes that are not functionally equivalent as long as there is a non-zero chance that any two non-equivalent nodes will have different values (this is the same as the "amplification trick" normally used in the design of randomized algorithms).

It is easy to see that any functionally equivalent nodes will be mapped by the above described algorithm to the same node in the compressed computation graph. However, it can happen that the algorithm will also merge nodes that are not functionally equivalent but just happened (by chance) to output the same values on all the random parameter initializations that the algorithm used. We acknowledge that this can happen in practice, nevertheless it was not commonly encountered in

2 Here, we add that in this definition, obviously, when we replace a function f by function f', we have to replace all occurrences of f in the graph also by f'.

3 Here, when we say "automorphic nodes", we mean that there exists an automorphism of the graph swapping the two nodes.

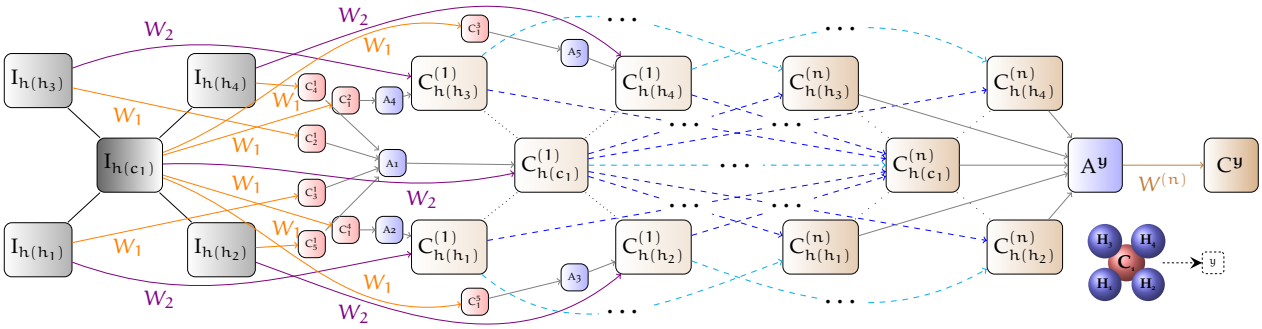

Figure 26: A multi-layer GNN model with a global readout unfolded over an example molecule of methane. Colors are used to distinguish the weight sharing, as well as different node types categorized w.r.t. the associated activation functions, denoted as input (I), convolution (C), and aggregation (A) nodes, respectively.

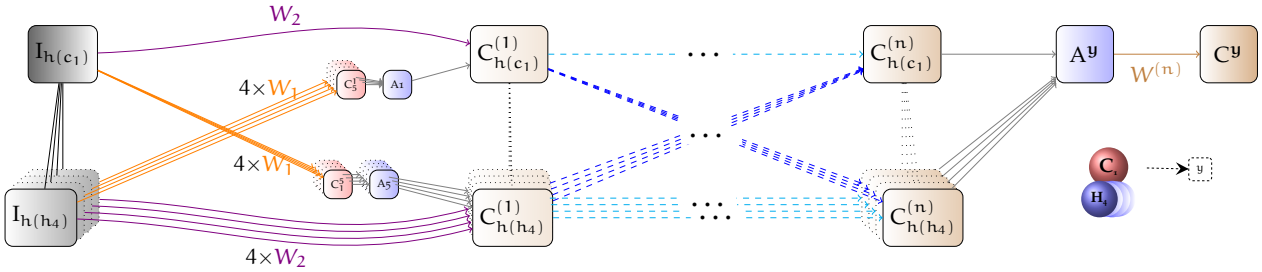

Figure 27: A compressed version of the GNN from Figure 26, with the compressed parts dotted.

our experiments (Section 9.5), unless explicitly emulated. To do that, we decreased the number of significant digits used in each equivalence check between $\mathit{value}(N; \mathcal{W}_i)$ and $\mathit{value}(N'; \mathcal{W}_i)$. This allows to compress the graphs even further, at the cost of sacrificing fidelity w.r.t. the original model.

There are also cases when we can give (probabilistic) guarantees on the correctness of this algorithm. One such case is when the activation functions in the computation graph are all polynomial. In this case, we can use DeMillo-Lipton-Schwartz-Zippel Lemma [277] to bound the probability of merging two nodes that are not functionally equivalent. However, since the activation functions in the computation graphs that we are interested in are usually not polynomial, we omit the details here. In particular, obtaining similar probabilistic guarantees with activation functions such as ReLU does not seem doable.[4]

9.4.2 *An Exact Compression Algorithm*

The exact algorithm for compressing computation graphs reuses the evaluation with random parameter initializations while recording the respective values for all the nodes. However, the next steps are different. First, instead of traversing the computation graph from the output nodes towards the leaves, it traverses the graph bottom-up, starting from the leaves. Second, rather than merging the nodes with the same recorded value lists right away, the exact algorithm merely considers these as candidates for merging. For that it keeps a data structure (based on a hash table) that indexes the nodes of the computation graph by the lists of the respective values recorded for the random parameter initializations. When, while traversing the graph, it processes a node N, it checks if there is any node N' that had the same values over all the random parameter initializations and

4 In particular, the proof of DeMillo-Lipton-Schwartz-Zippel Lemma relies on the fact that any single variable polynomial is zero for only a finite number of points, which is not the case for computation graphs with ReLUs.

has already been processed. If so it checks if N and N' are *structurally equivalent* (which we explain in turn) and if they are it replaces N by N'. To test the structural equivalence of two nodes, the algorithm checks the following conditions:

1. The activation functions of N and N' are the same.
2. The lists of children of both N and N' are the same (not just structurally equivalent but identical, i.e. $\mathit{Children}(N) = \mathit{Children}(N')$), and if C is the i-th child of N and C' is the i-th child of N', with (C, N, L_1) and (C', N, L_2) being the respective edges connecting them to N, then the labels L_1 and L_2 must be equal, too.

One can show why the above procedure works by induction. We sketch the main idea here. The base case is trivial. To show the inductive step we can reason as follows. When we are processing the node N, by the assumption, the node N' has already been processed. Thus, we know that the children of both N and N' must have already been processed as well. By the induction hypothesis, if any of the children were structurally equivalent, they must have been merged by the algorithm, and so it is enough to check identity of the child nodes. This reasoning then allows one to easily finish a proof of correctness of this algorithm.

There is one additional optimization that we can do for symmetric activation functions. Here by "symmetric" we mean symmetric with respect to permutation of the arguments. An example of such a symmetric activation function is any function of the form $f(x_1, \ldots, x_k) = h\left(\sum_{i=1}^{k} x_k\right)$; such functions are often used in neural networks. In this case we replace the condition 2 above by:

2′. There is a permutation π such that $\pi(\mathit{Children}(N)) = \mathit{Children}(N')$), and if C is the i-th child of N and C' is the $\pi(i)$-th child of N', with (C, N, L_1) and (C', N, L_2) being the respective edges connecting them to N, then the labels L_1 and L_2 must be equal.

It is not difficult to implement the above check efficiently (we omit the details). Note also that the overall asymptotic complexity of compressing a graph G is simply the same as the n evaluations of G.

Finally, to illustrate the effect of the lossless compression, we show the GNN model (Section 2.4), unfolded over a sample molecule of methane from Figure26, compressed in Figure 27.

9.5 EXPERIMENTS

To test the proposed compression in practice, we selected some common structured convolutional models, and evaluated them on a number of real datasets from the domains of (i) molecule classification and (ii) knowledge-base completion. The questions to be answered by the experiments are:

1. How numerically efficient is the non-exact algorithm in achieving lossless compression?
2. What improvements does the compression provide in terms of graph size and speedup?
3. Is learning accuracy truly unaffected by the, presumably lossless, compression in practice?

MODELS We chose mostly GNN-based models as their dynamic computation graphs encompass all the elements of structured convolutional models (convolution, pooling, and recursive layer stacking). Particularly, we choose well-known instances of GCNs and graph-SAGE (Section 2.4), each with 2 layers. Additionally, we include Graph Isomorphism Networks (GIN) [61], which follow the same computation scheme with 5 layers, but their particular operations ($C_{W_1} = \mathit{identity}$, $A = \mathit{sum}$, $C_{W_2} = \mathrm{MLP}$) are theoretically substantiated in the expressiveness of the Weisfeiler-Lehman test [59]. This is interesting in that it should effectively distinguish non-isomorphic substructures in the data by generating consistently distinct computations, and should thus be somewhat more resistant to our proposed compression. Finally, we include a relational template ("graphlets") introduced in [202], which generalizes GNNs to aggregate small 3-graphlets instead of just neighbors.

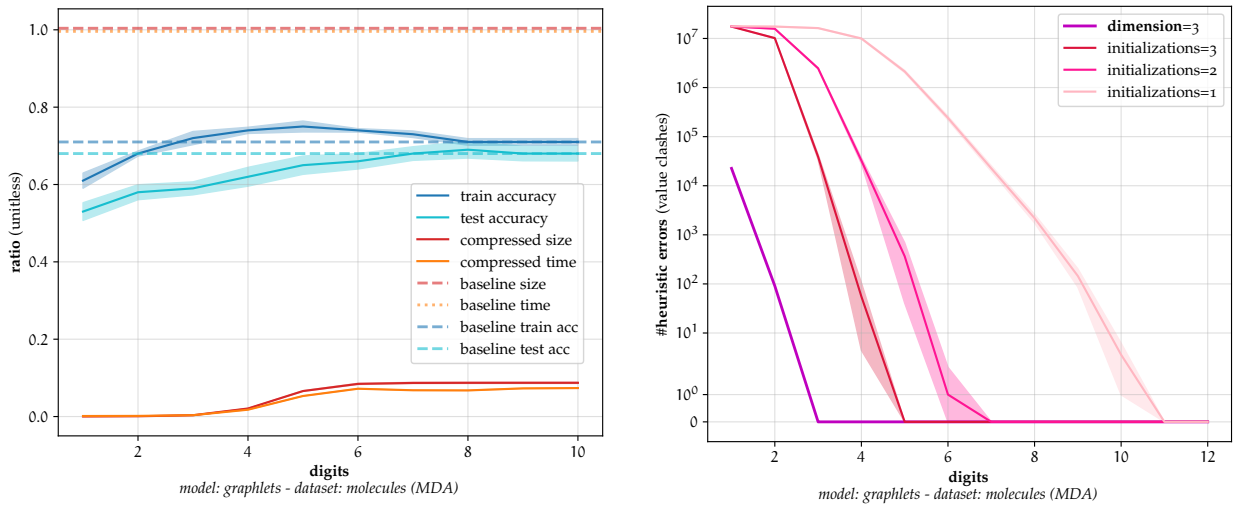

Figure 28: Compression of a *scalar*-parameterized graphlets model on a molecular dataset. We display progression of the selected metrics w.r.t. increasing number of significant digits (inits=1) used in the value comparisons (left), and number of non-equivalent subgraph value clashes detected by the exact algorithm w.r.t. the digits, weight re-initializations, and increased weight dimension (right).

DATASETS For structure property prediction, we used 78 organic molecule classification datasets reported in previous works [172–174]. Nevertheless, we show only the (alphabetically) first 3 for clarity, as the target metrics were extremely similar over the whole set. We note we also extended GCNs with edge embeddings to account for the various bond types, further *decreasing* the symmetries. For knowledge base completion (KBC), we selected commonly known datasets of Kinships, Nations, and UMLS [66] composed of different object-predicate-subject triplets. We utilized GCNs to learn embeddings of all the items and relations, similarly to R-GCNs [194], and for prediction of each triplet, we fed the three embeddings into an MLP, such as in [200], denoted as "KBE".

EXPERIMENTAL PROTOCOL We approached all the learning scenarios under simple unified setting with standard hyperparameters, none of which was set to help the compression (sometimes on the contrary). We used the (re-implemented) LRNN framework to encode all the models, and also compared with popular GNN frameworks of PyTorch Geometric (PyG) [204] and Deep Graph Library (DGL) [205]. If not dictated by the particular model, we set the activation functions simply as $C_W = \frac{1}{1+e^{-W \cdot x}}$ and $A = avg$. We then trained against MSE using 1000 steps of ADAM, and evaluated with a 5-fold crossvalidation.

9.5.1 *Results*

Firstly, we tested numerical efficiency of the non-exact algorithm itself (Section 9.4), for which we used scalar weight representation in the models to detect symmetries on the level of individual "neurons" (rather than "layers"). We used the (most expressive) graphlets model, where we checked the functional symmetries to overlap with the structural symmetries. The results in Figure 28 then show that the non-exact algorithm is already able to perfectly distinguish all structural symmetries with but a single weight initialization within less than 12 significant digits. While more initializations indeed improved the efficiency rapidly, in the end they proved unnecessary (but could be used in cases where the available precision would be insufficient). Moreover this test was performed with the actual low-range logistic activations. The displayed (10x) training time improvement (Figure28 - left) in the scalar models was then directly reflecting the network size reduction, and could be pushed further by decreasing the numeric precision at the expected cost of degrading the learning performance.

Table 8: Training times *per epocha* across different models and frameworks over 3000 molecules. Additionally, the startup graphs creation time of LRNNs (including the compression) is reported.

Model	Lifting (s)	LRNNs (s)	PyG (s)	DGL (s)	LRNN startup (s)
GCN	**0.25 ± 0.01**	0.75± 0.01	3.24 ± 0.02	23.25 ± 1.94	35.2 ± 1.3
g-SAGE	**0.34 ± 0.01**	0.89± 0.01	3.83 ± 0.04	24.23 ± 3.80	35.4 ± 1.8
GIN	**1.41 ± 0.10**	2.84± 0.09	11.19 ± 0.06	52.04 ± 0.41	75.3 ± 3.2

Secondly, we performed similar experiments with standard tensor parameterization, where the isomorphisms were effectively detected on the level of whole neural "layers", since the vector output values (of dim=3) were compared for equality instead. This further improved the precision of the non-exact algorithm (Figure 28 - right), where merely the first 4 digits were sufficient to achieve lossless compression in all the models and datasets (Figure 29). However, the training (inference) time was no longer directly reflecting the network size reduction, which we account to optimizations used in the vectorized computations. Nevertheless the speedup (app. 3x) was still substantial.

We further compared with established GNN frameworks of PyG [204] and DGL [205]. We made sure to align the exact computations of GCN, graph-SAGE, and GIN, while all the frameworks performed equally w.r.t. the accuracies (as detailed in Section 7.4.3[5]). For a more fair comparison, we further increased all (tensor) dimensions to a more common dim=10. The compression effects, as well as performance edge of the implemented LRNN framework itself, are displayed in Table 8 for a sample molecular dataset (MDA). Note that the compression was truly least effective for the aforementioned GIN model, nevertheless still provided app. 2x speedup.

Finally, the results in Figure 30 confirm that the proposed lossless compression via lifting, with either the exact algorithm or the non-exact algorithm with a high-enough numeric precision used, indeed does not degrade the learning performance in terms of training and testing accuracy (both were close within margin of variance over the crossvalidation folds).

Note that the used templated models are quite simple and do not generate any symmetries on their own (which they would, e.g., with recursion), but rather merely reflect the symmetries in the data themselves. Consequently, the speedup was overall lowest for the sparse knowledge graph of Nations, further decomposed by the 2 distinct relation types, and higher for the Kinships dataset, representing a more densely interconnected social network. The improvement was then generally biggest for the highly symmetric molecular graphs where, interestingly, the compression often reduced the neural computation graphs to a size even smaller than that of the actual input molecules. Note we only compressed symmetries within individual computation graphs (samples), and the results thus cannot be biased by the potential existence of isomorphic samples [278], however, potentially much higher compression rates could be also achieved with (dynamic) batching.

9.6 CONCLUSIONS

We introduced a simple, efficient, lossless compression technique for structured convolutional models inspired by lifted inference. The technique is very light-weight and can be easily adopted by any neural learner, but is most effective for structured convolutional models utilizing weight sharing schemes to target relational data, such as in various GNNs. We have demonstrated with existing models and datasets that a significant inference and training time reduction can be achieved without affecting the learning results, and possibly extended beyond for additional speedup.

5 We note that these results have already been demonstrated in Chapter 7, Section 7.4.4, Table 7, demonstrating the performance of the LRNNs versus the specialized GNN frameworks. Here we additionally demonstrate the contribution of the incorporated optimization technique (Lifting) itself.

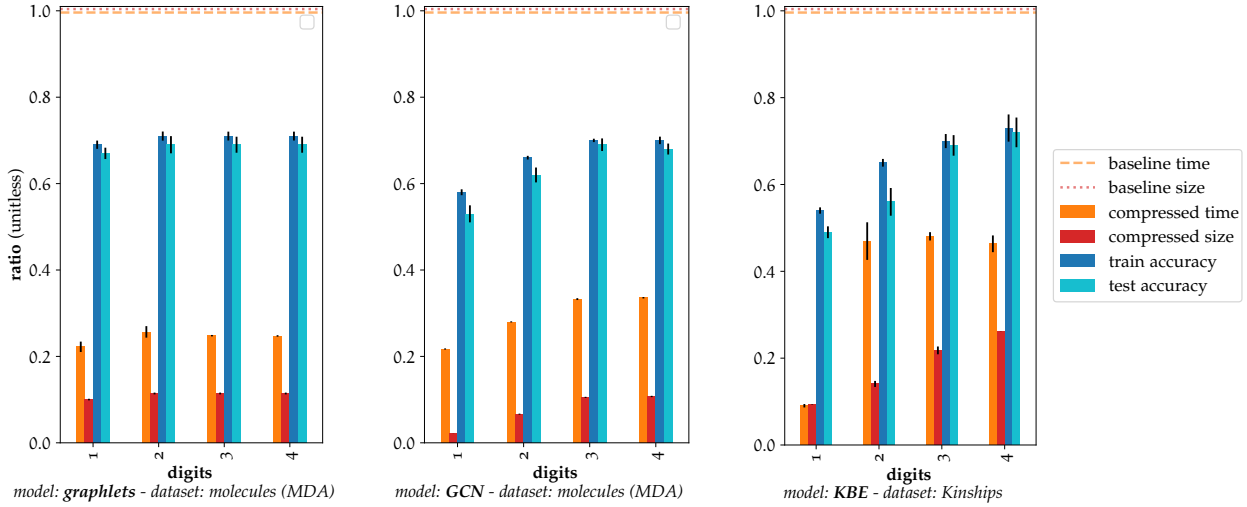

Figure 29: Compression of 3 *tensor*-parameterized models of graphlets (left), GCNs (middle) and KBEs (right) over the molecular (left, middle) and Kinships (right) datasets, with progression of selected metrics against the increasing number of significant digits used for equivalence checking.

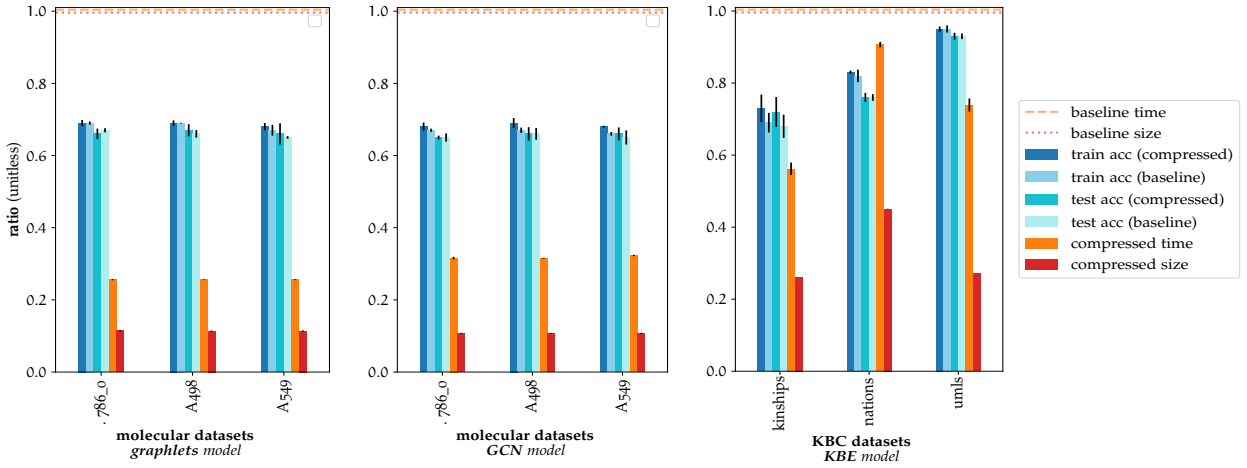

Figure 30: Comparison of 3 different baseline models of graphlets (left), GCNs (middle), and KBEs (right) with their compressed versions over molecule classification (left, middle) and KBC (right).

We note that some further details on the introduced algorithms, reported experiments, and application to the original LRNNs can also be found in the appendix Section A.2.

10

LOSSLESS HYPOTHESIS SPACE PRUNING

In this chapter, we present a generic method to prune hypothesis spaces in search-based, inductive logic programming (ILP) strategies (Section 3.3.1), such as the structure learning scenario from Chapter 6. The main strategy of our method consists in removing hypotheses that are equivalent to already considered hypotheses. The distinguishing feature of our method is that we use learned domain theories to check for *equivalence*, in contrast to existing approaches which only prune isomorphic hypotheses. Specifically, we use such learned domain theories to saturate hypotheses and then check if these saturations are isomorphic. While conceptually simple, we experimentally show that the resulting pruning strategy can be surprisingly effective in reducing both computation time and memory consumption when searching for long clauses, compared to approaches that only consider isomorphism.

10.1 INTRODUCTION

A key challenge for ILP algorithms (e.g. Progol [101]) is the fact that they typically have to search through large hypothesis spaces. Methods for pruning this search space have the potential to dramatically improve the quality of learned hypotheses and the runtime of the algorithms. One way of doing this is by filtering isomorphic hypotheses, which is the strategy used, for instance, in the relational pattern mining algorithm Farmr [279]. However, pruning isomorphic hypotheses is often not optimal, in the sense that it may be possible to prune hypotheses which are not isomorphic, but which are nonetheless equivalent in the considered domain. For example, the hypothesis that "if X is the father of Y then X and Y have the same last name" is equivalent to the hypothesis that "if X is male and a parent of Y then X and Y have the same last name".

In this chapter, we introduce a method which explicitly tries to prune equivalent hypotheses that are created during the hypothesis search, while still maintaining completeness[1]. One important challenge is that we need a quick way of testing whether a new hypothesis is equivalent to a previously considered one. To this end, we propose a saturation method which, given a first-order-logic clause, derives a longer *saturated* clause that is equivalent to it modulo a domain theory. This saturation method has the important property that two clauses are equivalent, given a domain theory, whenever their saturations are isomorphic. This means that we can use saturations to detect equivalent hypotheses as follows. We compute saturations of all hypotheses as they are being constructed, as well as certain invariants[2] of these saturations. Then we use the invariants to compute hashes for the saturated hypotheses, which allows us to use hash tables to efficiently narrow down the set of previously constructed hypotheses against which equivalence needs to be tested. In this way, we can avoid the need to explicitly compare new hypotheses with all previously constructed ones, which would clearly be infeasible in spaces with possibly millions of hypotheses. Note that this technique crucially relies on the use of saturations, and would not be possible with e.g. just a no-

1 As we show later in the chapter, the completeness requirement disqualifies relative subsumption [103] as a candidate for such a pruning method.

2 We use invariants based on a generalized version of Weisfeiler-Lehman procedure [59].

tion of relative subsumption modulo a background theory. To avoid the need for any prior domain knowledge, our method learns the required domain theories from the training data.

We experimentally show that our method can be orders of magnitude faster than methods which merely check for isomorphism, even when taking into account the time needed for learning domain theories.

10.2 BACKGROUND

In this chapter, we directly build on the notations and terminology of relational logic, as introduced in the background Section 3.1. Additionally, we introduce some simple extensions for the ease of presentation of the proposed technique. Particularly, for the set of variables occurring in a clause A we will write $vars(A)$, and we will denote the set of all respective terms as $terms(A)$. For a clause A, we then define the *sign flipping* operation as $\widetilde{A} \stackrel{def}{=} \bigvee_{l \in A} \tilde{l}$, where $\tilde{a} = \neg a$ and $\tilde{\neg a} = a$ for an atom a. In other words, the sign flipping operation simply replaces each literal by its negation.

As discussed in Section 3.3.1.1, we will also commonly identify a clause A with the corresponding set of literals $\{\phi_1, ..., \phi_k\}$. A clause $A = \{\phi_1, ..., \phi_n\}$ is then satisfied by a possible world ω, written $\omega \models A$, if for each grounding substitution θ, it holds that $\{\phi_1\theta, ..., \phi_n\theta\} \cap \omega \neq \emptyset$. The satisfaction relation $\models$ is then extended to (sets of) propositional combinations of clauses in the usual way, as detailed in Section 3.1.2.

Recall also from Section 3.3.1.1 that if A and B are clauses, we say that A θ-subsumes B (denoted $A \preceq_\theta B$) if and only if there is a substitution θ such that $A\theta \subseteq B$. In this chapter, we will further call A and B θ-*equivalent* (denoted $A \approx_\theta B$) if $A \preceq_\theta B$ and $B \preceq_\theta A$. Note that the $\approx_\theta$ relation is indeed an equivalence relation (i.e. it is reflexive, symmetric and transitive). Clauses A and B are then said to be *isomorphic* (denoted $A \approx_{iso} B$) if there exists an injective substitution θ such that $A\theta = B$. Finally, we say that A OI-subsumes B (denoted $A \preceq_{OI} B$ [280]) if there is an injective substitution such that $A\theta \subseteq B$. Note that A is isomorphic to B iff $A \preceq_{OI} B$ and $B \preceq_{OI} A$.

Example 22 *Let us consider the following four clauses:*

$$C_1 = p_1(A, B) \vee \neg p_2(A, B)$$
$$C_2 = p_1(A, B) \vee \neg p_2(A, B) \vee \neg p_2(A, C)$$
$$C_3 = p_1(X, Y) \vee \neg p_2(X, Y) \vee \neg p_2(X, Z)$$
$$C_4 = p_1(A, B) \vee \neg p_3(A, B)$$

Then we can easily verify that $C_1 \approx_\theta C_2 \approx_\theta C_3$ *(and thus also* $C_i \preceq_\theta C_j$ *for* $i, j \in \{1, 2, 3\}$*). We also have* $C_1 \not\approx_{iso} C_2$, $C_1 \not\approx_{iso} C_3$, $C_2 \approx_{iso} C_3$, *as well as* $C_i \not\preceq_\theta C_4$ *and* $C_4 \not\preceq_\theta C_i$ *for any* $i \in \{1, 2, 3\}$*. Finally we also have* $C_1 \preceq_{OI} C_i$ *for* $i \in \{1, 2, 3\}$, $C_2 \preceq_{OI} C_3$ *and* $C_3 \preceq_{OI} C_2$.

10.2.1 *Learning Setting*

In this chapter we will work in the classical setting of *learning from interpretations* (Section 3.3.1). Recall that in this setting, examples are interpretations and hypotheses are clausal theories (i.e. conjunctions of clauses). An example e is said to be *covered* by a hypothesis H if $e \models H$ (i.e. e is *covered* by H if it is a model of H). Given a set of positive examples $\mathcal{E}^+$ and negative examples $\mathcal{E}^-$, the training task is then to find a hypothesis H from some class of hypotheses $\mathcal{H}$ which optimizes a given scoring function (e.g. training error). For the ease of presentation, we will restrict ourselves to classes $\mathcal{H}$ of hypotheses in the form of clausal theories without constants, as constants can be emulated by unary predicates (since we do not consider functions).

The covering relation $e \models H$ can then be checked using a θ-subsumption solver (Section 3.3.1.1) as follows. Each hypothesis H can be written as a conjunction of clauses $H = C_1 \wedge \cdots \wedge C_n$. Clearly, $e \not\models H$ if there is an i in $\{1, \ldots, n\}$ such that $e \models \neg C_i$, which holds precisely when $C_i \preceq_\theta \neg(\bigwedge e)$.

Example 23 *Let us consider the following example, inspired by the Michalski's East-West trains datasets [281]:*

$$\begin{aligned} e = \{&\mathit{eastBound}(\mathit{car1}), \mathit{hasCar}(\mathit{car1}), \mathit{hasLoad}(\mathtt{car1}, \mathtt{load1}), \mathit{boxShape}(\mathtt{load1}),\\ &\neg\mathit{eastBound}(\mathit{load1}), \neg\mathit{hasCar}(\mathit{load1}), \neg\mathit{hasLoad}(\mathit{load1}, \mathit{car1}),\\ &\neg\mathit{hasLoad}(\mathit{load1}, \mathit{load1}), \neg\mathit{hasLoad}(\mathit{car1}, \mathit{car1}), \neg\mathit{boxShape}(\mathtt{car1})\} \end{aligned}$$

and two hypotheses H_1 *and* H_2

$$\begin{aligned} H_1 &= \mathit{eastBound}(C) \vee \neg\mathit{hasLoad}(C, L) \vee \neg\mathit{boxShape}(L)\\ H_2 &= \neg\mathit{eastBound}(C) \vee \neg\mathit{hasLoad}(C, L) \end{aligned}$$

To check if $e \models H_i$, $i = 1, 2$, *using a* θ*-subsumption solver, we construct*

$$\begin{aligned} \neg(\bigwedge e) = \neg&\mathit{eastBound}(\mathit{car1}) \vee \neg\mathit{hasCar}(\mathit{car1}) \vee \neg\mathit{hasLoad}(\mathtt{car1}, \mathtt{load1}) \vee\\ &\vee \mathit{boxShape}(\mathtt{load1}) \vee \mathit{eastBound}(\mathit{load1}) \vee \mathit{hasCar}(\mathit{load1}) \vee\\ &\vee \mathit{hasLoad}(\mathit{load1}, \mathit{car1}) \vee \mathit{hasLoad}(\mathit{load1}, \mathit{load1}) \vee \mathit{hasLoad}(\mathit{car1}, \mathit{car1})\\ &\vee \mathit{boxShape}(\mathtt{car1}) \end{aligned}$$

It is then easy to check that $H_1 \not\preceq_\theta \neg(\bigwedge e)$ *and* $H_2 \preceq_\theta \neg(\bigwedge e)$*, from which it follows that* $e \models H_1$ *and* $e \not\models H_2$.

In practice, when using a θ-subsumption solver to check $C_i \preceq_\theta \neg(\bigwedge e)$, it is usually beneficial to flip the signs of all the literals, i.e. to instead check $\widetilde{C_i} \preceq_\theta \bigvee e$, which is clearly equivalent. This is because θ-subsumption solvers often represent negative literals in interpretations implicitly to avoid excessive memory consumption[3], relying on the assumption that most predicates in real-life datasets are sparse.

10.2.2 *Theorem Proving Using SAT Solvers*

The methods described in this chapter will require access to an efficient theorem prover for clausal theories. Since we restrict ourselves to function-free theories (without equality), we can rely on a simple theorem-proving procedure based on propositionalization, which is a consequence of the following well-known result[4] [282].

Theorem 1 (Herbrand's Theorem) *Let* $\mathcal{L}$ *be a first-order language without equality and with at least one constant symbol, and let* $\mathcal{T}$ *be a set of clauses. Then* $\mathcal{T}$ *is unsatisfiable iff there exists some finite set* $\mathcal{T}_0$ *of* $\mathcal{L}$*-ground instances of clauses from* $\mathcal{T}$ *that is unsatisfiable.*

Here $A\theta$ is called an $\mathcal{L}$-ground instance of a clause A if θ is a grounding substitution that maps each variable occurring in A to a constant from the language $\mathcal{L}$.

In particular, to decide if $\mathcal{T} \models C$ holds, where T is a set of clauses and C is a clause (without constants and function symbols), we need to check if $\mathcal{T} \wedge \neg C$ is unsatisfiable. Since Skolemization preserves satisfiability, this is the case iff $\mathcal{T} \wedge \neg C_{Sk}$ is unsatisfiable, where $\neg C_{Sk}$ is obtained from $\neg C$ using Skolemization. Let us now consider the restriction $\mathcal{L}_{Sk}$ of the considered first-order language $\mathcal{L}$ to the constants appearing in C_{Sk}, or to some auxiliary constant s_0 if there are no constants in C_{Sk}. From Herbrand's theorem, we know that $\mathcal{T} \wedge \neg C_{Sk}$ is unsatisfiable in $\mathcal{L}_{Sk}$ iff the grounding of this formula w.r.t. the constants from $\mathcal{L}_{Sk}$ is satisfiable, which we can efficiently check using

3 This is true for the θ-subsumption solver based on [108] which we use in our implementation.

4 The formulation of Hebrand's theorem used here is taken from notes by Cook and Pitassi: `http://www.cs.toronto.edu/~toni/Courses/438/Mynotes/page39.pdf`.

a SAT solver. Moreover, it is easy to see that $\mathcal{T} \wedge \neg C_{Sk}$ is unsatisfiable in $\mathcal{L}_{Sk}$ iff this formula is unsatisfiable in $\mathcal{L}$ [5].

In practice, it is not always necessary to completely ground the formula $\mathcal{T} \wedge \neg C_{Sk}$. It is often beneficial to use an incremental grounding strategy similar to cutting plane inference in Markov logic [283]. To check if a clausal theory $\mathcal{T}$ is satisfiable, this method proceeds as follows.

STEP 0: start with an empty Herbrand interpretation $\mathcal{H}$ and an empty set of ground formulas $\mathcal{G}$.

STEP 1: check which groundings of the formulas in $\mathcal{T}$ are not satisfied by $\mathcal{H}$ (e.g. using a CSP solver). If there are no such groundings, the algorithm returns $\mathcal{H}$, which is a model of $\mathcal{T}$. Otherwise the groundings are added to $\mathcal{G}$.

STEP 2: use a SAT solver to find a model of $\mathcal{G}$. If $\mathcal{G}$ does not have any model then $\mathcal{T}$ is unsatisfiable and the method finishes. Otherwise replace $\mathcal{H}$ by this model and go back to Step 1.

10.3 PRUNING HYPOTHESIS SPACES USING DOMAIN THEORIES

In this section we show how domain theories can be used to prune the search space of ILP systems. Let us start with two motivating examples.

Example 24 *Let us consider the following two hypotheses for some target concept* x*:*

$$H_1 = x(A) \vee \neg animal(A) \vee \neg cod(A)$$
$$H_2 = x(A) \vee \neg fish(A) \vee \neg cod(A)$$

Intuitively, these two hypotheses are equivalent since every cod *is a* fish *and and every* fish *is an* animal*. Yet ILP systems would need to consider both of these hypotheses separately because* H_1 *and* H_2 *are not isomorphic, they are not* θ*-equivalent and neither of them* θ*-subsumes the other.*

Example 25 *Problems with redundant hypotheses abound in datasets of molecules, which are widespread in the ILP literature. For instance, consider the following two hypotheses:*

$$H_1 = x(A) \vee \neg carb(A) \vee \neg bond(A, B) \vee \neg bond(B, C) \vee \neg hydro(C)$$
$$H_2 = x(A) \vee \neg carb(A) \vee \neg bond(A, B) \vee \neg bond(C, B) \vee \neg hydro(C)$$

These two hypotheses intuitively represent the same molecular structures (a carbon and a hydrogen both connected to the same atom of unspecified type). Again, however, their equivalence cannot be detected without the domain knowledge that bonds in molecular datasets are symmetric[6]*.*

In the remainder of this section we will describe how background knowledge can be used to detect equivalent hypotheses. First, we introduce the notion of *saturations* of clauses in Section 10.3.1. Subsequently, in Section 10.3.2 we show why pruning hypotheses based on these saturations does not hurt the completeness of a refinement operator. In Section 10.3.3, we then explain how these saturations can be used to efficiently prune search spaces of ILP algorithms. In Section 10.3.4 we describe a simple method for learning domain theories from the given training data. Finally, in Section 10.3.5 we show why using relative subsumption is not sufficient.

5 Indeed, if $\mathcal{T} \wedge \neg C_{Sk}$ is unsatisfiable in $\mathcal{L}$, then there is a set of corresponding $\mathcal{L}$-ground instances of clauses that are unsatisfiable. If we replace each constant appearing in these ground clauses which does not appear in C_{sk} by an arbitrary constant that does appear in C_{sk}, then the resulting set of ground clauses must still be inconsistent, since T does not contain any constants and there is no equality in the language, meaning that $\mathcal{T} \wedge \neg C_{Sk}$ cannot be satisfiable in $\mathcal{L}_{Sk}$.

6 In the physical world, bonds do not necessarily have to be symmetric, e.g. there is an obvious asymmetry in polar bonds. However, it is a common simplification in data mining on molecular datasets to assume that bonds are symmetric.

10.3.1 *Saturations*

The main technical ingredient of the proposed method is the following notion of *saturation*.

Definition 10 (Saturation of a clause) *Let $\mathcal{B}$ be a clausal theory and C a clause (without constants or function symbols). If $\mathcal{B} \not\models C$, we define the saturation of C w.r.t. $\mathcal{B}$ to be the maximal clause C' satisfying: (i) $vars(C') = vars(C)$ and (ii) $\mathcal{B} \wedge C'\theta \models C\theta$ for any injective grounding substitution θ. If $\mathcal{B} \models C$, we define the saturation of C w.r.t. $\mathcal{B}$ to be* **T**, *where* **T** *denotes tautology.*

When $\mathcal{B}$ is clear from the context, we will simply refer to C' as the saturation of C.

Definition 10 naturally leads to a straightforward procedure for computing the saturation of a given clause. Let $\mathcal{P} = \{l_1, l_2, \ldots, l_n\}$ be the set of all literals which can be constructed using variables from C and predicate symbols from $\mathcal{B}$ and C. Let θ be an arbitrary injective grounding substitution; note that we can indeed take θ to be arbitrary because $\mathcal{B}$ and C do not contain constants. If $\mathcal{B} \not\models C$, the saturation of C is given by the following clause:

$$\bigvee\{l \in \mathcal{P} : \mathcal{B} \models \neg l\theta \vee C\theta\} \tag{5}$$

This means in particular that we can straightforwardly use the SAT based theorem proving method from Section 10.2.2 to compute saturations. The fact that (5) correctly characterizes the saturation can be seen as follows. If C' is the saturation of C then $\mathcal{B} \wedge C'\theta \models C\theta$ by definition, which is equivalent to $\mathcal{B} \wedge \neg(C\theta) \models \neg(C'\theta)$. We have $\neg(C'\theta) = \bigwedge\{\tilde{l}\theta : \mathcal{B} \wedge \neg(C\theta) \models \tilde{l}\theta\} = \bigwedge\{\tilde{l}\theta : \mathcal{B} \wedge l\theta \models C\theta\}$, and thus $C'\theta = \bigvee\{l\theta : \mathcal{B} \wedge l\theta \models C\theta\}$. Finally, since θ is injective, we have[7] $C' = (C'\theta)\theta^{-} = \bigvee\{l : \mathcal{B} \wedge l\theta \models C\theta\}$.

Example 26 *Let us consider the following theory*

$$\mathcal{B} = \{\neg friends(X, Y) \vee friends(Y, X)\}$$

which expresses the fact that friendship is a symmetric relation and a clause

$$C = \neg friends(X, Y) \vee happy(X).$$

To find the saturation of this clause, we first need a suitable injective substitution θ; let us take $\theta = \{X \mapsto c_1, Y \mapsto c_2\}$. Then we have

$$\begin{aligned}\mathcal{B} \cup \neg(C\theta) &= \mathcal{B} \cup \{friends(c_1, c_2) \wedge \neg happy(c_1)\} \\ &\models friends(c_1, c_2) \wedge friends(c_2, c_1) \wedge \neg happy(c_1),\end{aligned}$$

After negating the latter formula and inverting the substitution (noting that it is injective) we get the following saturation:

$$C' = \neg friends(X, Y) \vee \neg friends(Y, X) \vee happy(X).$$

Now, let us consider another clause $D = \neg friends(X, Y) \vee happy(Y)$. This clause is not isomorphic to C. However, it is easy to see that its saturation

$$D' = \neg friends(X, Y) \vee \neg friends(Y, X) \vee happy(Y)$$

is isomorphic to the saturation C' of C.

The next proposition will become important later in the chapter as it will allow us to replace clauses by their saturations when learning from interpretations.

Proposition 1 *If C' is a saturation of C w.r.t. $\mathcal{B}$ then $\mathcal{B} \wedge C' \models C$.*

7 Note that we are slightly abusing notation here, as θ^{-1} is not a substitution.

Proof 1 *We have* $\mathcal{B} \wedge C' \models C$ *iff* $\mathcal{B} \wedge C' \wedge \neg C$ *is unsatisfiable. Skolemizing* $\neg C$*, this is equivalent to* $\mathcal{B} \wedge C' \wedge \neg(C\theta_{Sk})$ *being unsatisfiable, where* θ_{Sk} *is a substitution representing the Skolemization. As in Section 10.2.2, we find that the satisfiability of* $\mathcal{B} \wedge C' \wedge \neg(C\theta_{Sk})$ *is also equivalent to the satisfiability of the grounding of* $\mathcal{B} \wedge C' \wedge \neg(C\theta_{Sk})$ *w.r.t. the Skolem constants introduced by* θ_{Sk}*. In particular, this grounding must contain the ground clause* $C'\theta_{Sk}$*. From the definition of saturation, we have that* $\mathcal{B} \wedge C'\theta_{Sk} \wedge \neg(C\theta_{Sk}) \models \mathbf{F}$*, where* $\mathbf{F}$ *denotes falsity (noting that* θ_{Sk} *is injective). It follows that* $\mathcal{B} \wedge C' \wedge \neg C \models \mathbf{F}$*, and thus also* $\mathcal{B} \wedge C' \models C$*.* □

The next proposition shows that saturations cover the same examples as the clauses from which they were obtained, when $\mathcal{B}$ is a domain theory that is valid for all examples in the dataset.

Proposition 2 *Let* $\mathcal{B}$ *be a clausal theory such that for all examples* e *from a given dataset it holds that* $e \models \mathcal{B}$*. Let* C *be a clause and let* C' *be its saturation w.r.t.* $\mathcal{B}$*. Then for any example* e *from the dataset we have* $(e \models C) \Leftrightarrow (e \models C')$*.*

Proof 2 *From the characterization of saturation in (5), it straightforwardly follows that* $C \models C'$*, hence* $e \models C$ *implies* $e \models C'$*. Conversely, if* $e \models C'$*, then we have* $e \models \mathcal{B} \wedge C'$*, since we assumed that* $e \models \mathcal{B}$*. Since we furthermore know from Proposition 1 that* $\mathcal{B} \wedge C' \models C$*, it follows that* $e \models C$*.* □

Finally, we define *positive* and *negative saturations*, which only add positive or negative literals to clauses. Among others, this will be useful in settings where we are only learning Horn clauses.

Definition 11 *A positive (resp. negative) saturation of* C *is defined as* $C'' = C \cup \{l \in C' : l$ *is a positive (resp. negative) literal*$\}$ *where* C' *is a saturation of* C*.*

Propositions 1 and 2 are also valid for positive or negative saturations; their proofs can be straightforwardly adapted. When computing the positive (resp. negative) saturation, we can restrict the set $\mathcal{P}$ of candidate literals to the positive (resp. negative) ones. This can speed up the computation of saturations significantly.

10.3.2 *Searching the Space of Saturations*

In this section we show how saturations can be used together with refinement operators to search the space of clauses ordered by OI-subsumption[8]. Specifically, we show that if we have a refinement operator that can completely generate some set of clauses then we can use the same refinement operator, in combination with a procedure for computing saturations, to generate the set of all saturations of the considered set of clauses. Since this set of saturations is typically smaller than the complete set of clauses (as many clauses can lead to the same saturated clauses), this is already beneficial for reducing the size of the hypothesis space. In Section 10.3.3, we show that it also allows us to very quickly prune equivalent clauses. First we give a definition of *refinement operator* [284].

Definition 12 (Refinement operator) *Let* $\mathcal{L}$ *be a first-order language. A refinement operator*[9] *on the set* $\mathcal{C}$ *of all* $\mathcal{L}$*-clauses is a function* $\rho : \mathcal{C} \to 2^{\mathcal{C}}$ *such that for any* $C \in \mathcal{C}$ *and any* $D \in \rho(C)$ *it holds* $C \preceq_{OI} D$*. A refinement operator* ρ *is complete if for any two clauses* C *and* D *such that* $C \preceq_{OI} D$*, a clause* E *isomorphic to* D *(*$D \approx_{iso} E$*) can be obtained from* C *by repeated application of the refinement operator (i.e.* $E \in \rho(\rho(\ldots\rho(C)\ldots))$*).*

Most works define refinement operators w.r.t. θ-subsumption instead of OI-subsumption [284]. We need the restriction to OI-subsumption as a technical condition for Proposition 3 below. It should be

8 Note that we only use OI-subsumption to partially order the constructed hypotheses, not to check the entailment relation.
9 What we call refinement operator in this chapter is often called downward refinement operator. Since we only consider downward refinement operators in this chapter, we omit the word downward.

noted, however, that our results remain valid for many refinement operators that are not specifically based on OI-subsumption, including all refinement operators that only add new literals to clauses. Also note that we do not use OI-subsumption as a covering operator but only to structure the space of hypotheses. Therefore there is no loss in what hypotheses can be learnt.

The next definition formally introduces the combination of refinement operators and saturations.

Definition 13 (Saturated refinement operator) *Let $\mathcal{L}$ be a first-order language. Let ρ be a refinement operator on the set $\mathcal{C}$ of all $\mathcal{L}$-clauses containing at most n variables. Let $\mathcal{B}$ be a clausal theory. Let $\sigma_{\mathcal{B}} : \mathcal{C} \to \mathcal{C}$ be a function that maps a clause C to its saturation C' w.r.t. $\mathcal{B}$. Then the function $\rho_{\mathcal{B}} = \sigma_{\mathcal{B}} \circ \rho$ is called the saturation of ρ w.r.t. $\mathcal{B}$.*

Clearly, the saturation of a refinement operator w.r.t. some clausal theory $\mathcal{B}$ is a refinement operator as well. However, it can be the case that ρ is complete whereas its saturation is not. As we will show next, this is not a problem for completeness w.r.t. the given theory $\mathcal{B}$ in the sense that saturations of all clauses from the given class $\mathcal{C}$ are guaranteed to be eventually constructed by the combined operator, when ρ is a complete refinement operator.

Proposition 3 *Let $\mathcal{L}$ be a first-order language. Let ρ be a complete refinement operator on the set of $\mathcal{L}$-clauses $\mathcal{C}$, $\mathcal{B}$ be clausal theory, $\sigma_{\mathcal{B}}$ a function that maps a clause C to its saturation C' w.r.t. $\mathcal{B}$ and let $\rho_{\mathcal{B}}$ be the saturation of ρ w.r.t. $\mathcal{B}$. Let $C \in \mathcal{C}$ be a clause, $\mathcal{S}_C$ and let $\mathcal{S}_C^{\mathcal{B}}$ be the sets of clauses that can be obtained from C by repeated application of ρ and $\rho_{\mathcal{B}}$, respectively. Then for any clause $D \in \mathcal{S}_C$ there is a clause $D' \in \mathcal{S}_C^{\mathcal{B}}$ such that $\sigma_{\mathcal{B}}(D) \approx_{iso} D'$.*

Proof 3 *We first note that if $A \preceq_{OI} B$ then $\sigma_{\mathcal{B}}(A) \preceq_{OI} \sigma_{\mathcal{B}}(B)$ (assuming an extended definition of OI-subsumption such that $A \preceq_{OI} \mathbf{T}$ for any A), which follows from the monotonicity of the entailment relation $\models$. Let us define $\mathcal{X} = \{\sigma_{\mathcal{B}}(A) | A \in \mathcal{S}_C\}$. Note that $\mathcal{X}$ and $\mathcal{S}_C^{\mathcal{B}}$ are not defined in the same way ($\mathcal{X}$ is the set of saturations of clauses in $\mathcal{S}_C$ whereas $\mathcal{S}_C^{\mathcal{B}}$ is the set of clauses that can be obtained by the saturated refinement operator $\rho_{\mathcal{B}}$ from the clause C). We need to show that these two sets are* equivalent. *Clearly, $\mathcal{S}_C^{\mathcal{B}} \subseteq \mathcal{X}$. To show the other direction, let us assume (for contradiction) that there is a clause $X \in \mathcal{X}$ for which there is no clause $Y \in \mathcal{S}_C^{\mathcal{B}}$ which is isomorphic to X. Let us assume that X is a minimal clause with this property, meaning that for any clause X' contained in the set $\mathcal{Z}_X = \{Z \in \mathcal{X} | Z \preceq_{OI} X \wedge X \not\approx_{iso} Z\}$ there is a clause $Y' \in \mathcal{S}_C^{\mathcal{B}}$ which is isomorphic to X'. Clearly, if there is one such clause X then there is also a minimal one which follows from the fact that all the considered clauses are finite and $\preceq_{OI}$ is a partial order. Let us take a clause $X' \in \mathcal{Z}_X$ which is maximal[10] w.r.t. the ordering induced by $\preceq_{OI}$ and let Y' be the respective isomorphic clause from $\mathcal{S}_C^{\mathcal{B}}$. Then $\rho(Y')$ must contain a clause Y'', $Y' \not\approx_{iso} Y''$, that OI-subsumes X, which follows from completeness of the refinement operator ρ. However, then $\sigma_{\mathcal{B}}(Y'')$ must be contained in $\mathcal{S}_C^{\mathcal{B}}$. It must also hold that $\sigma_{\mathcal{B}}(Y'') \preceq_{OI} \sigma_{\mathcal{B}}(X) = X$. Here, $\sigma_{\mathcal{B}}(Y'') \preceq_{OI} \sigma_{\mathcal{B}}(X)$ follows from the already mentioned observation that if $A \preceq_{OI} B$ then $\sigma_{\mathcal{B}}(A) \preceq_{OI} \sigma_{\mathcal{B}}(B)$, and the equality $\sigma_{\mathcal{B}}(X) = X$ follows from the idempotence of $\sigma_{\mathcal{B}}$, noting that X is already a saturation of some clause. However, this is a contradiction with the maximality of X' and the corresponding Y'.* □

10.3.3 *Pruning Isomorphic Saturations*

When searching the space of clauses or, in particular, saturations of clauses, we should avoid searching through isomorphic clauses. It is easy to see that the sets of clauses generated by a (saturated) complete refinement operator ρ from two isomorphic clauses C and C' will contain clauses that are isomorphic (i.e. for any clause in the first set there will be an isomorphic clause in the second set and vice versa). Therefore it is safe to prune isomorphic clauses during the search.

When searching through the hypothesis space of clauses, most ILP algorithms maintain some queue of candidate clauses. This is the case, for instance, in algorithms based on best-first search

10 If we ordered the set of clauses by θ-subsumption instead of OI-subsumption then there would not have to exist a maximal clause with this property.

(Progol, Aleph [101]). Other algorithms, e.g. those based on level-wise search, maintain similar data structures (e.g. Warmr [285]). Many of the clauses that are stored in such queues or similar data structures will be equivalent, even if they are not isomorphic. Existing methods, even if they were removing isomorphic clauses during search[11], have to consider each of these equivalent clauses separately, which may greatly affect their performance. This is where using saturations of clauses w.r.t. some background knowledge is most useful because it can replace the different implicitly equivalent clauses by their saturation.

In theory, one could try to test isomorphism of all pairs of clauses currently in the queue data structures. However, this would be prohibitively slow in most practical cases. To efficiently detect equivalences by checking isomorphism of saturations, we replace the queue data structure (or a similar data structure used by the given algorithm) by a data structure that is based on hash tables. When a new hypothesis H is constructed by the algorithm, we first compute its saturation H'. Then, we check whether the modified queue data structure already contains a clause that is isomorphic to H'. To efficiently check this, we use a straightforward generalization of the Weisfeiler-Lehman labeling procedure [59]. We then only need to check whether two clauses are isomorphic if they have the same hash value. We similarly check whether H' is isomorphic to a clause in the so-called closed set of previously processed hypotheses. If H' is neither isomorphic to a clause in the queue nor to a clause in the closed set, it is added to the queue.

Example 27 *Let us again consider the two clauses from Example 24:* $H_1 = x(A) \vee \neg animal(A) \vee \neg cod(A)$ *and* $H_2 = x(A) \vee \neg fish(A) \vee \neg cod(A)$*. Suppose that the theory* $\mathcal{B}$ *encodes the taxonomy of animals and contains the rules* $\neg cod(X) \vee fish(X)$ *and* $\neg fish(X) \vee animal(X)$*. Computing the saturations of* H_1 *and* H_2*, we obtain* $H_1' = x(A) \vee \neg animal(A) \vee \neg cod(A) \vee \neg fish(A)$ *and* $H_2' = x(A) \vee \neg fish(A) \vee \neg cod(A) \vee \neg animal(A)$*, which are isomorphic. Therefore both of them can be replaced by the same saturations while the corresponding algorithm keeps searching the hypothesis space.*

Similarly as shown above for the two clauses from Example 24, saturations could be used to detect equivalence of the two clauses from Example 25 w.r.t. the corresponding background knowledge theory $\mathcal{B}$.

In addition to equivalence testing, saturations can be used to filter trivial hypotheses, i.e. hypotheses covering every example, without explicitly computing their coverage on the dataset (which would be very costly on large datasets). We illustrate this use of saturations in the next example.

Example 28 *Consider a domain theory* $\mathcal{B} = \neg professor(X) \vee \neg student(X)$ *which states that no one can be both a student and a professor. Let us also consider a hypothesis* $H = employee(X) \vee \neg professor(X) \vee \neg student(X)$*. If the domain theory* $\mathcal{B}$ *is correct,* H *should cover all examples from the dataset and is thus trivial. Accordingly, the saturation of* H *contains every literal, and is in particular equivalent to* $\mathbf{T}$*.*

10.3.4 *Learning Domain Theories for Pruning*

The domain theories that we want to use for pruning hypothesis spaces can be learned from the given training dataset. Every clause C in such a learned domain theory should satisfy $e \models C$ for all examples e in the dataset. We construct such theories using a level-wise search procedure, starting with an empty domain theory. The level-wise procedure maintains a list of candidate clauses (modulo isomorphism) with i literals. If a clause C in the list of candidate clauses covers all examples (i.e. $e \models C$ for all e from the dataset) then it is removed from the list and if there is no clause in the domain theory which θ-subsumes C, then C is also added to the domain theory. Each of the remaining clauses in the list, i.e. those which do not cover all examples in the dataset, are then extended in all possible ways by the addition of a literal. This is repeated until a threshold on the maximum

11 For instance, Farmr [279] or RelF [286] remove isomorphic clauses (or conjunctive patterns), but many existing ILP systems do not attempt removing isomorphic clauses.

number of literals is reached. The covering of examples by the candidate clauses is checked using θ-subsumption as outlined in Section 10.2.1.

It is worth pointing out that if we restrict the domain theories, e.g. to contain only clauses of length at most 2 or only Horn clauses, the saturation process will be guaranteed to run in polynomial time (which follows from the polynomial-time solvability of 2-SAT and Horn-SAT).

10.3.5 *Why Relative Subsumption is Not Sufficient*

Although the motivation behind relative subsumption [103] is similar to ours, relative subsumption has two main disadvantages that basically disqualify it for the purpose of pruning the hypothesis space. The first problem is that pruning hypotheses that are equivalent w.r.t. relative subsumption may not guarantee completeness of the search. This is the same issue as with pruning based on plain θ-subsumption which, unlike pruning based on isomorphism, may lead to incompleteness of the search. Note that this is already the case in the more restricted setting of graph mining under homomorphism [287]. The second issue with relative subsumption is that it would need to be tested for all pairs of candidate hypotheses, whereas the pruning based on saturations and isomorphism testing allows us to use the more efficient hashing strategy based on the Weisfeiler-Lehman procedure.

10.4 EXPERIMENTS

In this section we evaluate the usefulness of the proposed pruning method on real datasets. We test it inside an exhaustive feature construction algorithm which we then evaluate on a standard molecular dataset KM20L2 from the NCI GI 50 dataset collection [173]. This dataset contains 1207 examples (molecules) and 94263 facts.

10.4.1 *Methodology and Implementation*

The evaluated feature construction method is a simple level-wise algorithm which works similarly to the Warmr frequent pattern mining algorithm [285]. It takes two parameters: maximum depth d and maximum number of covered examples t (also called "maximum frequency"). It returns all connected[12] clauses which can be obtained by saturating clauses containing at most d literals, and which cover at most t examples. Unlike in frequent conjunctive pattern mining where minimum frequency constraints are natural, when mining in the setting of learning from interpretations, the analogue of the minimum frequency is the maximum frequency constraint[13].

The level-wise algorithm expects as input a list of interpretations (examples) and the parameters t and $d > 0$. It proceeds as follows:

STEP 0: set $i := 0$ and $L_0 := \{\square\}$ where $\square$ denotes the empty clause.

STEP 1: construct a set L_{i+1} by extending each clause from L_i with a negative literal (in all possible ways).

STEP 2: replace clauses in L_{i+1} by their negative saturations and for each set of mutually isomorphic clauses keep only one of them.

STEP 3: remove from L_{i+1} all clauses which cover more than t examples in the dataset.

12 A clause is said to be connected if it cannot be written as disjunction of two non-empty clauses. For instance $\forall X, Y : p_1(X) \vee p_2(Y)$ is not connected because it can be written also as $(\forall X : p_1(X)) \vee (\forall Y : p_2(Y))$ but $\forall X, Y : p_1(X) \vee p_2(Y) \vee p_3(X, Y)$ is connected. If a clause is connected then its saturation is also connected.

13 Frequent conjunctive pattern mining can be emulated in our setting. It is enough to notice that the clauses that we construct are just negations of conjunctive patterns.

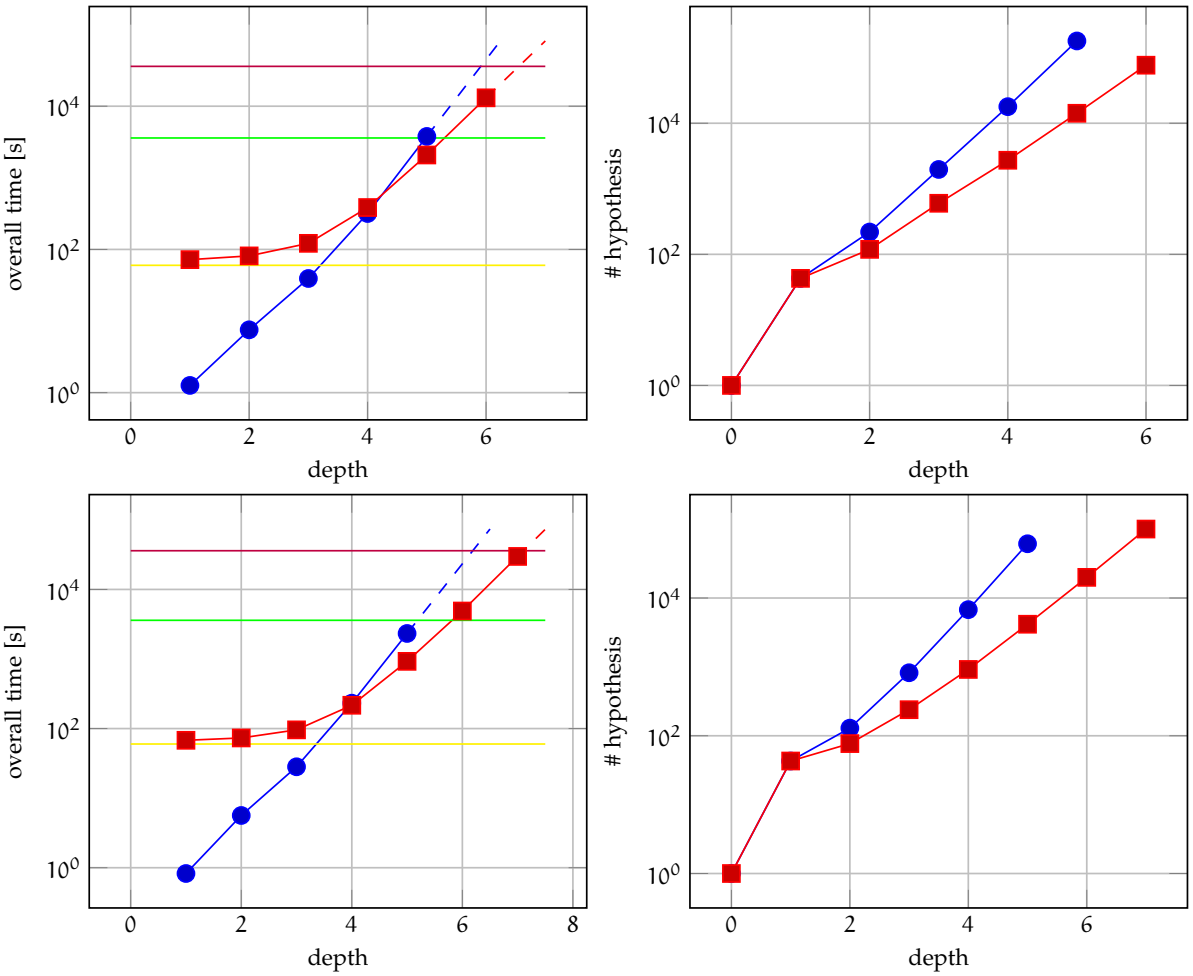

Figure 31: **Left panels:** Runtime of the level-wise algorithm using saturations for pruning (red) and without using saturations (blue). **Right panels:** Number of clauses constructed by the algorithm using saturations (red) and without using saturations (blue). Top panels display results for maximal number of covered examples equal to dataset size minus one and bottom panels for this parameter set to dataset size minus 50, which corresponds to minimum frequency of 50. One minute, one hour, and ten hours are highlighted by yellow, green, and purple horizontal lines. Runtimes are extrapolated by exponential function and shown in dashed lines.

STEP 4: if L_{i+1} is empty or $i+1 > d$ then finish and return $\bigcup_{j=0}^{i+1} L_j$. Otherwise set $i := i+1$ and go to step 1.

As can be seen from the above pseudocode, we restricted ourselves to mining clauses which contain only negative literals. This essentially corresponds to mining positive conjunctive queries, which is arguably the most typical scenario. Nonetheless, it would be easy to allow the algorithm to search for general clauses, as the θ-subsumption solver used in the implementation actually allows efficient handling of negations.

We implemented the level-wise algorithm and the domain theory learner in Java[14]. To check the coverage of examples using θ-subsumption, we used an implementation of the θ-subsumption algorithm from [108]. For theorem proving, we used an incremental grounding solver which relies on the Sat4j library [288] for solving ground theories and the θ-subsumption engine from [108].

10.4.2 *Results*

We measured runtime and the total number of clauses returned by the level-wise algorithm without saturations and with saturations. Both algorithms were exactly the same, the only difference being that the second algorithm first learned a domain theory and then used it for computing the saturations. Note in particular that both algorithms used the same isomorphism filtering. Therefore any differences in computation time must be directly due to the use of saturations.

We performed the experiments reported here on the NCI dataset KM20L2. The learned domain theories were restricted to contain only clauses with at most two literals. We set the maximum number of covered examples equal to the number of examples in the dataset minus one (which corresponds to a minimum frequency constraint of 1 when we view the clauses as negated conjunctive patterns). Then we also performed an experiment where we set it equal to the number of examples in the dataset minus 50 (which analogically corresponds to a minimum frequency constraint of 50). We set the maximum time limit to 10 hours.

The results of the experiments are shown in Figure 31. The pruning method based on saturations turns out to pay off when searching for longer clauses where it improves the baseline by approximately an order of magnitude and allows it to search for longer hypotheses within the given time limit. When searching for smaller clauses, the runtime is dominated by the time for learning the domain theory, which is why the baseline algorithm is faster in that case. The number of generated clauses, which is directly proportional to memory consumption, also becomes orders of magnitude smaller when using saturations for longer clauses. Note that for every clause constructed by the baseline algorithm, there is an equivalent clause constructed by the algorithm with the saturation-based pruning. We believe these results clearly suggest the usefulness of the proposed method, which could potentially also be used inside many existing ILP systems.

10.5 RELATED WORK

The works most related to our approach are those relying on a special case of Plotkin's relative subsumption [103] called generalized subsumption [289]. Generalized subsumption was among others used in [290]. In Section 10.3.5 we discussed the reasons why relative subsumption is not suitable for pruning. Background knowledge was also used to reduce the space of hypotheses in the Progol 4.4 system [101], which uses Plotkin's relative clause reduction. Note that the latter is a method for *removing* literals from bottom clauses, whereas in contrast our method is based on *adding* literals to hypotheses. Hence, the Progol 4.4 strategy is orthogonal to the methods presented in this chapter. Another key difference is that our approach is able to learn the background knowledge from the training data whereas all the other approaches use predefined background knowledge. Finally, our approach is not limited to definite clauses, which is also why we do not use SLD resolution. On the other hand, as our method is rooted in first-order logic (due to the fact that we use the learning from interpretations setting) and not directly in logic programming, it lacks some of the expressive power of logic programming.

10.6 CONCLUSIONS

In this chapter, we introduced a generally applicable method for pruning hypotheses in ILP, which goes beyond mere isomorphism testing. We showed that the method is able to reduce the size of the hypothesis space by orders of magnitudes, and also leads to a significant runtime reduction. An interesting aspect of the proposed method is that it combines induction (domain theory learning) and deduction (theorem proving) for pruning the search space.

14 available at https://github.com/martinsvat/Pruning-Hypotheses.

Part V

APPLICATIONS

In the following part, we demonstrate some use cases and applications of the (original) LRNN framework from Chapter 5. While the application range is wide, for the sake of brevity we select only two, highly diverse, examples. First in Chapter 11, we introduce novel LRNN modeling concepts for knowledge base completion based on learning of latent predictive categories. As a second application in Chapter 12, we demonstrate the use of LRNNs for relational soccer team representation learning used in match outcome prediction. A short summary of some other applications can also be found in the appendix Part C.

11

LEARNING PREDICTIVE CATEGORIES

In Part iii, we introduced the LRNN framework and showed how it can be used across various relational learning scenarios (Chapter 5) and enhancement of existing models (Chapter 7). However, these were mostly limited to classification, particularly molecule classification. In this chapter, we show how LRNNs can be easily used for enhancing knowledge-base completion (KBC), too. Particularly, we use the original LRNNs (Chapter 5) to declaratively specify and solve learning problems in which latent categories of entities, properties and relations need to be jointly induced from a knowledge base.

11.1 INTRODUCTION

In this chapter, we first show how LRNNs can be used to learn a latent category structure that is predictive in the sense that the properties of a given entity can be largely determined by the category to which that entity belongs, and dually, the entities satisfying a given property can be largely determined by the category to which that property belongs. This enables a form of transductive reasoning which is based on the idea that similar entities have similar properties. We then extend this model into a relational setting, in which entities not only have properties but can also be linked by arbitrary relations.

As discussed in previous chapters, LRNNs were heavily inspired by lifted models, such as MLNs (Section 3.4.2). Likewise, the approach proposed in this chapter is somewhat similar to the MLN's authors work on Statistical Predicate Invention [66], which uses crisp clustering based on second-order MLNs. However, the use of LRNNs has several important advantages for learning latent concepts. Firstly, LRNNs do not need to invoke costly EM algorithms and hence can be more efficient than latent variable probabilistic models. Secondly, the learnt soft clusters can naturally be interpreted as vector space embeddings of entities, properties and relations. Finally, the flexibility of LRNNs means that the considered form of transductive reasoning can be extended in a natural way to take into account various forms of prior domain knowledge, as well as alternative types of heuristic reasoning (e.g. reasoning by analogy, modelling persistence or periodic behaviour).

The remainder of this chapter is structured as follows. Firstly, we slightly modify the original LRNN network creation process (Section 5.1.2.2) to cope with recurrent connections in ground networks. We then introduce the predictive templates for a non-relational setting in Section 11.2.1 and for a relational setting in Section 11.2.2. Next, in Section 11.3, we describe a simple model encoded as a LRNN which is based on similarity-based reasoning. In Section 11.4 we evaluate the method experimentally. Finally, we discuss related work in Section 11.5 and conclude the chapter in Section 11.6.

11.1.1 *Handling Recursion in LRNNs*

In this chapter, we build directly on the formalization of the original LRNN framework, as introduced in Section 5.1. There, in Section 5.1.5, we discussed how recursive rules do not pose problems to the LRNN templating, as long as their grounding does not lead to directed cycles in the resulting

neural networks. In this chapter, however, we *will* use recursive rule sets that may potentially lead to *recurrent* ground neural networks. In order to maintain the feed-forward nature of the resulting ground neural networks, and avoid training of recurrent neural networks, we modify the strategy for constructing ground networks as follows. First we construct the ground network exactly as described in Section 5. If this network contains directed cycles, we then proceed as follows. Let Q be a given ground query atom[1]. We find the respective atom neuron corresponding to Q in the ground network. If no such atom neuron exists, the output value for Q is 0. If there is such an atom neuron, we perform a breadth-first search from this atom neuron (traversing the connections between neurons in reverse, i.e. from output to input) and whenever we find an edge pointing *from* an already visited atom neuron, we delete it. The resulting ground neural network is then feed-forward. While this process enables us to stick with feed-forward neural networks, it comes at the price of a slightly less intuitive semantics, in which the inference and output for non-query atom neurons may also depend on the used queries. This is not problematic for any of the applications considered in this chapter, as non-query atoms are not used within these.

11.2 LEARNING PREDICTIVE CATEGORIES

In this section, we introduce a class of LRNN models that are aimed at learning predictive categories of entities, properties and relations. We first introduce a model for attribute-valued data in Section 11.2.1, which is extended to cope with relational data in Section 11.2.2.

11.2.1 *Predictive Categories for Attribute-Value Data*

Let a set of entities be given, and for each entity, a list of properties that it satisfies. The basic assumption underlying our model is that there exist some (possibly overlapping) categories, such that every entity can be described accurately enough by its soft membership to each of these categories. We furthermore assume that these categories can themselves be organised in a set of higher-level categories. The idea is that the category hierarchy should allow us to predict which properties a given entity has, where the properties associated with higher-level categories are typically (but not necessarily) inherited by their sub-categories. To improve the generalization ability of our method, we assume that a dual category structure exists for properties. The main task we consider is to learn these (latent) category structures from the given input data.

To encode the above described model in a LRNN, we proceed as follows. We use $\mathit{HasProperty}(e, \mathrm{p})$ to denote that the entity e has the property p. For every entity e and for each category c at the lowest level of the category hierarchy, we construct the following ground rule:

$$w_{\mathrm{ec}} : \mathit{IsA}(e, \mathrm{c})$$

Note that weight w_{ec} intuitively reflects the soft membership of e to the category c; it will be determined when training the ground network. Similarly, for each category c_1 at a given level and each category c_2 one level above, we add the following ground rule:

$$w_{\mathrm{c}_1\mathrm{c}_2} : \mathit{IsA}(\mathrm{c}_1, \mathrm{c}_2)$$

In the same way, ground rules are added that link each property to a property category at the lowest level, as well as ground rules that link property categories to higher-level categories. To encode the idea that entity categories should be predictive of properties, we add the following rule for each entity category c_e and each property category c_p:

$$w_{\mathrm{c}_e\mathrm{c}_\mathrm{p}} : \mathit{HasProperty}(\mathrm{A}, \mathrm{B}) \leftarrow \mathit{IsA}(\mathrm{A}, \mathrm{c}_e), \mathit{IsA}(\mathrm{B}, \mathrm{c}_\mathrm{p}).$$

1 In general LRNNs support non-ground query atoms, too, but in this chapter we will not need them. Therefore we assume only ground query atoms for simplicity.

The weights $w_{c_e c_p}$ encode which entity categories are related to which property categories, and will again be determined when training weights of the LRNN. To encode transitivity of the is-a relationship, we simply add the following rule:

$$w_{isa} : IsA(A, C) \leftarrow IsA(A, B), IsA(B, C).$$

Training examples are encoded as a set of facts of the form $(HasProperty(e, p), l)$ where $l \in \{0, 1\}$, 0 denoting a negative example and 1 a positive example. We train the model using SGD as described in [97]. In particular, in a LRNN, there is a neuron for any ground literal which is logically entailed by the rules and facts in the LRNN and the output of this neuron represents the truth value of this literal. Therefore if we want to train the weights of the LRNN, we just optimize the weights of the network w.r.t. a loss function such as the mean squared error, where the loss function is computed from the desired truth values of the query literals and the outputs obtained from the respective atom neurons.

11.2.2 *Predictive Categories for Relational Data*

The model from Section 11.2.1 can be extended to cope with relational facts. Similar to our encoding of properties, we will use a reified representation of relational facts, with e.g. $Relation(ParentOf, e_1, e_2)$ denoting that e_1 is the parent of e_2. In this way, we can induce predictive relation categories, similar to the entity and property categories considered in Section 11.2.1.

To this end, analogously as for entity and property categories, for every relation r and every (latent) relation category c we add the following ground rule:

$$w_{rc} : IsA(r, c)$$

For each relation category c_1 at a given level and each category c_2 one level above, we add the following ground rule:

$$w_{c_1 c_2} : IsA(c_1, c_2).$$

Note that a rule encoding transitivity of the *IsA* relation was already added in the first part of the model. Finally we encode that, like properties, relations among entities are typically determined by their categories. Specifically, for each triple consisting of a pair of (not necessarily distinct) entity clusters c_e, c'_e and a relation cluster c_r, we add the following ground rule:

$$w_{c_r c_e c'_e} : Relation(R, A, B) \leftarrow IsA(R, c_r), IsA(A, c_e), IsA(B, c'_e) \tag{6}$$

The LRNNs defined in this way will be referred to as *fully-connected*, as they contain rules for every *relation-entity-entity* triple. Obviously, when a high number of clusters is used, the number of rules of the form (6) may be prohibitively high. To address this, we can limit the triples for which such rules are added. In particular, we will consider LRNNs which restrict such rules to those of the following form:

$$w_{c_r c_{2i} c_{2i+1}} : Relation(R, A, B) \leftarrow IsA(R, c_r), IsA(A, c_{2i}), IsA(B, c_{2i+1}) \tag{7}$$

where c_1, c_2, ..., c_n are entity concepts. In fact the LRNNs with rules of this form can learn anything that can be learned by LRNNs with rules of the form (6) as long as they have enough rules.

In addition, to help the model learn symmetric and transitive relations (e.g. the "same-political-bloc" relation), we also add rules of the following form:

$$w_{c_r c'_i c'_i} : Relation(R, A, B) \leftarrow IsA(R, c_r), IsA(A, c'_i), IsA(B, c'_i) \tag{8}$$

Note that we do not need to explicitly consider these in the fully-connected model, as they are a special case of (6).

11.3 PREDICTION USING LEARNED SIMILARITIES

In this section we describe a LRNN model based on similarity degrees, for the same predictive task that was considered in the previous section. While the similarity degrees could be obtained from any source, we will use similarity degrees that have been obtained from the model described in the previous section, by taking advantage of the fact that the cluster membership degrees can be interpreted as defining a vector-space embedding. Rather than using the membership degrees directly, we will use the weights of the respective ground $\mathrm{IsA}(e, c)$ rules, which, unlike the membership degrees, may also be negative[2]. In particular, the similarity degree between two entities is defined as the cosine similarity between the vector representation of these entities, with the coordinates of these vectors the soft memberships of the entity in each of the categories.

For each pair of entities (e_1, e_2) with similarity degree s, we add the following ground fact:

$$1.0 : \mathit{Similar}(e_1, e_2, s)$$

We furthermore add rules which encode a learnable transformation of the similarities into a score which is useful for the given predictive task:

$$\begin{aligned}
&w_{-1} : \quad \mathit{Similar}(X, Y) \leftarrow \mathit{Similar}(X, Y, S), S \geqslant -1.0 \\
&w_{-0.9} : \mathit{Similar}(X, Y) \leftarrow \mathit{Similar}(X, Y, S), S \geqslant -0.9 \\
&\ldots \\
&w_{0.9} : \quad \mathit{Similar}(X, Y) \leftarrow \mathit{Similar}(X, Y, S), S \geqslant \quad 0.9
\end{aligned}$$

Finally we add one rule of the following type for every relation r:

$$w_r : \quad \mathit{Relation}(r, X, Y) \leftarrow \mathit{Relation}(r, V, W), \mathit{Similar}(X, V), \mathit{Similar}(Y, W).$$

Taking into account the aggregative nature of the used family of activation functions (cf. Section 5.1.3), these rules encode the intuition that in order to predict if X and Y are in relation r, we could check how similar on average the entities known to be in this relation are to X and Y.

Naturally, not all relations can be accurately predicted by a model like the one described in this section. However, this similarity based approach is quite natural, and serves as an important illustrative example of how other strategies could be encoded (e.g. interpolation/extrapolation or reasoning by analogy).

11.4 EVALUATION

11.4.1 *Evaluation of the Model for Attribute-Value data (Section 11.2.1)*

To evaluate the potential of the model proposed in Section 11.2.1, we have used the Animals dataset[3], which describes 50 animals in terms of 85 Boolean features, such as *f*ish, large, smelly, strong, and *t*imid. This dataset was originally created in [291], and was used among others for evaluating a related learning task in [66]. For both entities and properties, we have used two levels of categories, with in both cases three categories at the lowest level and two categories at the highest level.

Recall that we can view the category membership degrees as defining a vector-space embedding. Figures 32 and 33 show the first two principal components of this embedding for a number of entities and properties. We can see, for instance, that sea mammals are clustered together, and that predators tend to be separated from herbivores. In Figure 33, we have highlighted two types of properties: colours and teeth types. Note that these do not form clusters (e.g. a cluster of colours)

2 The membership degrees are simply obtained as applying sigmoids on the respective weights in this particular case, so the two representations essentially bear the same information

3 Downloaded from https://alchemy.cs.washington.edu/data/animals/

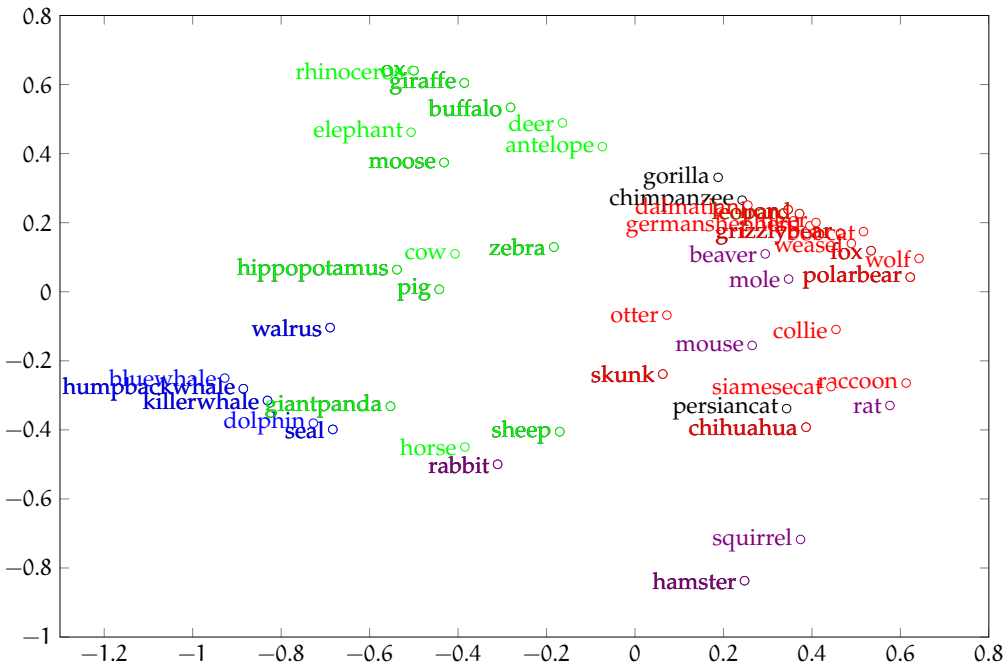

Figure 32: Embedding of entities (animals, only a subset of entities is displayed). Several homogeneous groups of animals are highlighted: sea mammals (blue), large herbivores (green), rodents (violet), and other predators (red).

but they represent, as prototypes, different clusters of properties which tend to occur together. For instance, *b*lue is surrounded by properties which typically hold for water mammals; white and red occur together with stripes, nocturnal, pads; gray occurs together with small and weak; etc. We also evaluated the predictive ability of this model. We randomly divided the facts from the dataset in two halves, trained the model on one half and tested it on the other one, obtaining AUC ROC of 0.77. We also performed an experiment with a 90-10 split, in order to be able to directly compare our results with those from [66]; we obtained the same AUC PR 0.8 as reported in [66] (and AUC ROC 0.86).

11.4.2 *Evaluation of the Model for relational data (Section 11.2.2)*

In order to evaluate the relational method proposed in Section 11.2.2 we performed experiments with two relational datasets:[4] Nations and UMLS. These datasets have previously been used to evaluate statistical predicate invention methods in [66]. The Nations dataset contains a set of relations between pairs of nations and their features [292]. It consists of relations such as *ExportsTo* and *GivesEconomicAidTo*, as well as properties such as *Monarchy*. The dataset contains 14 nations, 56 relations and 111 properties. There are 2565 true ground atoms. The UMLS dataset contains data from the Unified Medical Language System, which is a biomedical ontology [293]. It contains 49 relations and 135 biomedical entities. There are 6529 true ground atoms in this dataset.

Initial experiments have revealed two trends. First, accuracy consistently improved when we increased the size of the LRNNs (contrarily to our expectation that overfitting might be a problem when increasing the size). Second, for a fixed number of entity, property and relation categories, adding the layer of more general concepts helps, but it also increased memory consumption and runtime. Therefore, in the experiments, we created LRNNs as large as possible which still fitted in memory. A consequence of this strategy is that the LRNNs with more than one layer of categories had fewer categories in total than their single-layer counterparts. Similar effects also took place

4 Downloaded from https://alchemy.cs.washington.edu/data/nations/ and from https://alchemy.cs.washington.edu/data/umls/.

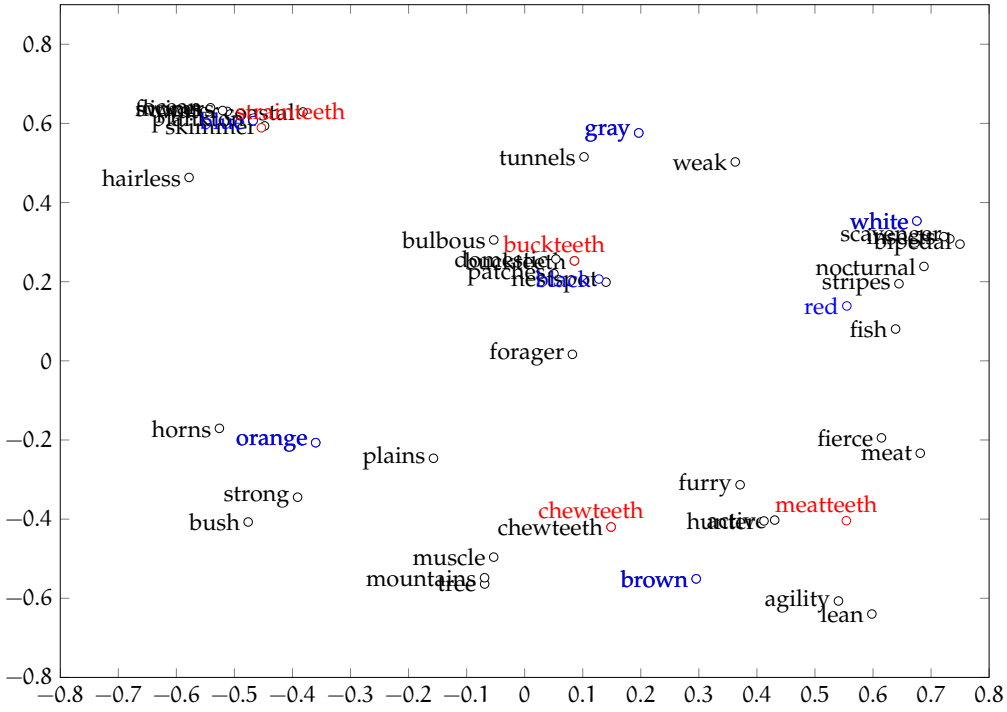

Figure 33: Embedding of properties (only a subset of properties is displayed). Two representative groups of properties are shown in colour: colours (blue) and teeth-type (red).

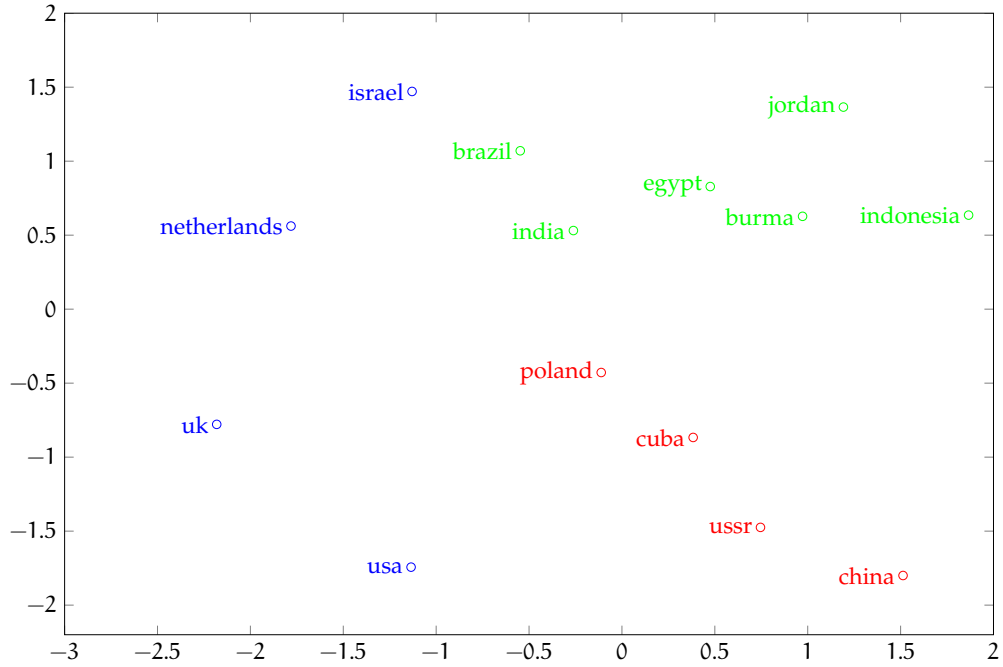

Figure 34: Embeddings of countries as induced by learning from their geopolitical relations captured in the historical dataset [292]. A possible interpretation of the projection is displayed in colors, dividing them into communist (red), western (blue), and developing nations (green).

for fully-connected LRNNs when compared to LRNNs with isolated rules of the form (7) and (8); therefore we did not consider fully connected LRNNs in our experiments.

For the Nations dataset, the largest single-layer LRNN which fitted in 40GB of memory had 100 property categories, 100 entity categories and 50 relation categories. The cross-validated AUC ROC was 0.89 and AUC PR 0.74, which is within the standard error margin of the results obtained in [66]. The largest two-layer LRNN learned on this dataset had 20 property categories, 20 entity categories and 10 relation categories. Its cross-validated AUC ROC was 0.88 and AUC PR 0.7. For comparison, we also trained a single-layer LRNN with the exact same number of each type of categories, which achieved AUC ROC 0.86 and AUC PR 0.67, which agrees with the above described general trends.

The first two principal components of the embeddings of the states are displayed in Figure 34. When interpreting this embedding, note that this dataset relates to the political situation of 1950s.

For the UMLS dataset we used a LRNN with 100 entity categories and 50 relation categories. Due to the size of the dataset, consisting of a total of 893k ground facts and memory limitations, we only performed experiments with a largely subsampled training set, obtaining test AUC ROC 0.97 and AUC PR 0.76. This is a lower AUC PR than obtained by the method from [66], but it is close to the second-best method tested there and is better than the reported results for MLN structure learning.

11.4.3 *Evaluation of the relational Model based on Similarities (Section 11.3)*

The evaluation of the model introduced in Section 11.3 primarily serves to estimate the usefulness of the embeddings learned by LRNNs. We have particulary focused on the embedding of countries, whose first two principal components are shown in Figure 34. First, we have split the nations dataset [292] into equally large training and testing parts. We trained the relational model described in Section 11.2.2, extracted the learned cluster membership degrees as vector embeddings, and calculated their pairwise cosine similarities. We included these similarities as ground facts, together with all the true statements from the training part of the dataset. On top of these facts, we added the transformation and inference rules to form the model described in Section 11.3. We then trained this composite model on the same training part of the nations dataset that we used to obtain the embeddings, and evaluated its generalization ability on the remaining testing part. We obtained AUC ROC of 0.85 and 0.49 AUC PR, which is lower than the crossvalidated performance reported for the best models, but indirectly proves that the previously learned embeddings indeed carry useful information that may be subsequently reused for different predictive scenarios.

11.4.4 *An Experiment with Real-Life Data from NELL*

We have also evaluated the method on a real-life dataset. The main idea here was to analyse whether the LRNN models described in this chapter could be used in an NLP pipeline to fill gaps in a knowledge base. To test this idea we downloaded a collection consisting of about 29k actors from NELL [294] with all their parental categories. For the experiments, we have subsampled the dataset to 2k actors. In the end, the number of different parental categories assigned to actors in this dataset turned out to be quite small. There were only 20 different categories such as *comedian* or *celebrity*, resulting into a dataset of 4k true ground facts, which we completed with their negative complement under the closed world assumption for evaluation. In the experiments, we have tested the LRNN construct described in Section 11.2.1 and obtained a test-set AUC ROC 0.84 and AUC PR 0.43. This suggests that the LRNN method is indeed able to discover plausible properties of entities in datasets obtained from text. This could be quite useful for suggesting properties or relations in settings like NELL's where feedback from users is also used to validate the predictions.

11.5 RELATED WORK

The proposed models essentially relies on the assumption that similar entities tend to have similar properties, for some similarity function which is learned implicitly in terms of category membership degrees. It is possible to augment this form of inference with other models of plausible reasoning, such as reasoning based on analogical (and other logical) proportions [295, 296]. Moreover, as in [297], we could take into account externally obtained similarity degrees, using rules such as those in Section 11.3.

The model considered in this chapter is related to statistical predicate invention [66] which relies on jointly clustering entities and relations. The dual representation of entity and property categories is also reminiscent of formal concept analysis [298]. LRNNs themselves are also related to the long stream of research in neural-symbolic integration [70], previous work on using neural networks for relational learning [45], and more recent approaches such as [220, 221].

11.6 CONCLUSIONS

We have illustrated how the declarative and flexible nature of LRNNs can be used for easy encoding of non-trivial learning scenarios. The models that we considered in this chapter jointly learn predictive categories of entities, their properties and relations between them. The main strength of this approach lies in the ease with which the model can be extended to more complicated settings, which is mainly due to the declarative nature of LRNNs. It seems remarkable that such a declarative approach is able to obtain results which are close to the state-of-the-art method from [66], without tailoring any part of the learning method to this particular problem setting.

12

LEARNING RELATIONAL TEAM EMBEDDINGS

In this chapter, we investigate the use of deep relational learning in the, somewhat unconventional, domain of predictive sports analytics. Particularly, we use the basic LRNN framework (Section 5) to learn "relational team embeddings". These are latent learning representations which directly reflect the historical relational interplay between the teams within a (soccer) league. On a large official dataset of historical soccer results, we then compare different relational learners against strong current domain-specific methods to show some very promising results of the relational approach when combined with the representation learning (embedding) within LRNNs.

12.1 INTRODUCTION

Sport analytics is a popular multi-billion dollar world-wide industry. It is a natural application domain for mathematical modelling, yet only recently we have started to see penetration of modern machine learning methods into the field, with standard predictive techniques still being geared towards simple statistical models [299]. We argue that incorporating relational learning techniques might benefit the field considerably. It only seems natural as the data arising from sport records possess interesting relational characteristics on many levels of abstraction, from the matches themselves forming relations between teams, players and seasons, to the course of the individual matches being driven by the rules of each sport with characteristic game-play patterns stemming from these.

We investigate viability of the relational approach to the domain via experimental evaluation on the task of soccer match outcome predictions based *solely on historical results*, i.e. without incorporating any features on the teams or players. This data format, composed solely of the team names and the resulting match scores, was introduced by the Soccer Prediction Challenge [300][1], from which we also adopt the respective dataset.

We then propose simple relational representations, background knowledge and modeling concepts for which we provide some interpretable insights. Particularly, we focus on expressing a concept we called "lifted relational team embeddings" in the original LRNN framework (Chapter 5). Finally, we experimentally compare different relational approaches with strong methods from the domain for their predictive performance on a large dataset of real soccer records.

12.1.1 *Predictive Sports Analytics*

In predictive sport analytics, the ultimate goal is to predict results of future matches. Given the stochastic nature of sports, the goal translates to correctly estimating probabilities of the corresponding outcomes. Particularly for a game of soccer, the aim is to estimate the probabilities of the three possible outcomes *loss*, *draw*, *win*.

The task of predicting soccer results is well established in the literature. Typical approaches include statistical models based on the Poisson distribution and its variations [301, 302], as well as various rating systems [303, 304]. An example of a relational learning approach was also introduced in [305], however the literature remains very scarce in this regard.

1 organized in conjunction with the MLJ's special issue on Machine Learning for Soccer in 2017

12.2 PREDICTIVE MODELS

We compare the proposed relational team embedding concept against a multitude of diverse learners. These consist of a simple prior probability baseline predictor, RDN-boost – a powerful SRL method for boosting Relational Dependency Networks [114], and an actual state-of-the-art model [306] that won the Soccer Prediction Challenge (2017) [300]. Note that each of these learners has been actually selected for being a strong performer in the given task.

BASELINE PREDICTOR is a simple model aggregating the prior probabilities of the individual home and away outcomes in each league. Being often surprisingly hard to beat, we include it as a baseline to serve as a natural lower bound for performance of all the learners.

RDN-BOOST LEARNER follows a functional gradient boosting strategy on top of Relational Dependency Networks [114], powerful lifted graphical models designed to learn from data with relational dependencies using pseudo-likelihood estimation techniques. Similarly to LRNNs, RDN-boost learns from Herbrand interpretations for which it utilizes fragment of relational logic for representation, where the inner nodes of the individual regression trees of the resulting ensemble model represent conjunctions of the original predicates.

STATE-OF-THE-ART MODEL is the actual winning solution [306][2] from the mentioned 2017's Soccer Prediction Challenge. It is an ensemble, gradient boosted trees-based model utilizing expert-designed features. Some of these features are derived from other, already sophisticated, models from literature, such as the *pi-ratings* [303] or *page-rank* [307]. Other features are statistics based on expert insights incorporating the home advantage, historical strength, current form, or match importance. These are further aggregated in different ways w.r.t. seasons and leagues, to finally form an input into a carefully tuned XGBoost algorithm [308].

LIFTED RELATIONAL NEURAL NETWORKS model is a custom generic template, encoding very general background knowledge on the structure of soccer matches, designed for the prediction task in the original LRNN formalism, as introduced in the respective Chapter 5. The model is further detailed in the subsequent Section 12.2.1.

12.2.1 *Knowledge Representation*

In its raw form, the match records contain merely the team names and the result, hence we tried to extract as much useful information as possible for each of the models. For the baseline this was straightforward, and for the SotA model this was already done [306]. For the approaches of RDN-boost and LRNNs we had to derive appropriate relational representation. Since they both learn from Herbrand interpretations, we firstly encoded the records with numerical outcomes into predicates, which we describe in Table 9.

12.3 LIFTED RELATIONAL TEAM EMBEDDINGS

Here we describe the proposed relational embedding model as expressed in the language of LRNNs. Firstly, we tested the hypothesis that there exists some predictive latent space embedding the teams. This is based on an intuition from various rating systems, such as the pi-ratings [303], where each team is assigned one or more parameters denoting its particular strength, possibly within different areas, such as when playing at home stadium and when playing away. However, opposite to the existing rating systems, the idea of the embedding approach is to explore meaning of these latent

2 This is also one of our contributions which is, however, omitted from this thesis for brevity, as it is only loosely related.

Table 9: Overview of predicates extracted from the data for the relational learners.

Predicate	Description
home(Tid)	Team Tid is home team w.r.t. prediction match.
away(Tid)	Team Tid is away team w.r.t. prediction match.
team(Tid, name)	Team Tid has name name.
win(Mid, Tid_1, Tid_2)	Win of home team Tid_1 over away team Tid_2 in match Mid.
draw(Mid, Tid_1, Tid_2)	Draw between home team Tid_1 and Tid_2 in match Mid.
loss(Mid, Tid_1, Tid_2)	Loss of home team Tid_1 to team Tid_2 in match Mid.
scored(Mid, Tid, n)	The team Tid scored more than n goals in match Mid.
conceded(Mid, Tid, n)	The team Tid conceded more than n goals in match Mid.
goal_diff(Mid, n)	Difference in goals scored by the teams is greater than n.
recency(Mid, n)	The match Mid was played more than n rounds ago (w.r.t. prediction match).

parameters automatically by the means of regular learning from data. We can encode this scenario in LRNNs as follows

$$w_1^{(0)} : \textit{type1}(\mathsf{T}) \leftarrow \textit{team(T,chelsea)}$$
$$w_2^{(0)} : \textit{type1}(\mathsf{T}) \leftarrow \textit{team(T,arsenal)}$$
$$\dots$$
$$w_j^{(0)} : \textit{type3}(\mathsf{T}) \leftarrow \textit{team(T,everton)}$$

where the types ($\text{type}_1 \dots \text{type}_3$) denote individual embedding dimensions of the teams. We may directly use aggregation of such embeddings for the prediction of outcome of home vs. away team matches using the following rules

$$w_{(1;1)}^{(1)} : \textit{outcome} \leftarrow \text{home}(\text{T1}) \wedge \text{type1}(\text{T1}) \wedge \text{away}(\text{T2}) \wedge \text{type1}(\text{T2})$$
$$w_{(1;2)}^{(1)} : \textit{outcome} \leftarrow \text{home}(\text{T1}) \wedge \text{type1}(\text{T1}) \wedge \text{away}(\text{T2}) \wedge \text{type2}(\text{T2})$$
$$\dots$$
$$w_{(3;3)}^{(1)} : \textit{outcome} \leftarrow \text{home}(\text{T1}) \wedge \text{type3}(\text{T1}) \wedge \text{away}(\text{T2}) \wedge \text{type3}(\text{T2})$$

This construct in principle creates a fully connected neural network with one hidden embedding layer, such as e.g. in the famous "word2vec" embedding architecture [22]. For all the historical matches we then jointly perform corresponding gradient updates of the weights to reflect the actual values of the outcome labels. We further denote this architecture as *embeddings*.

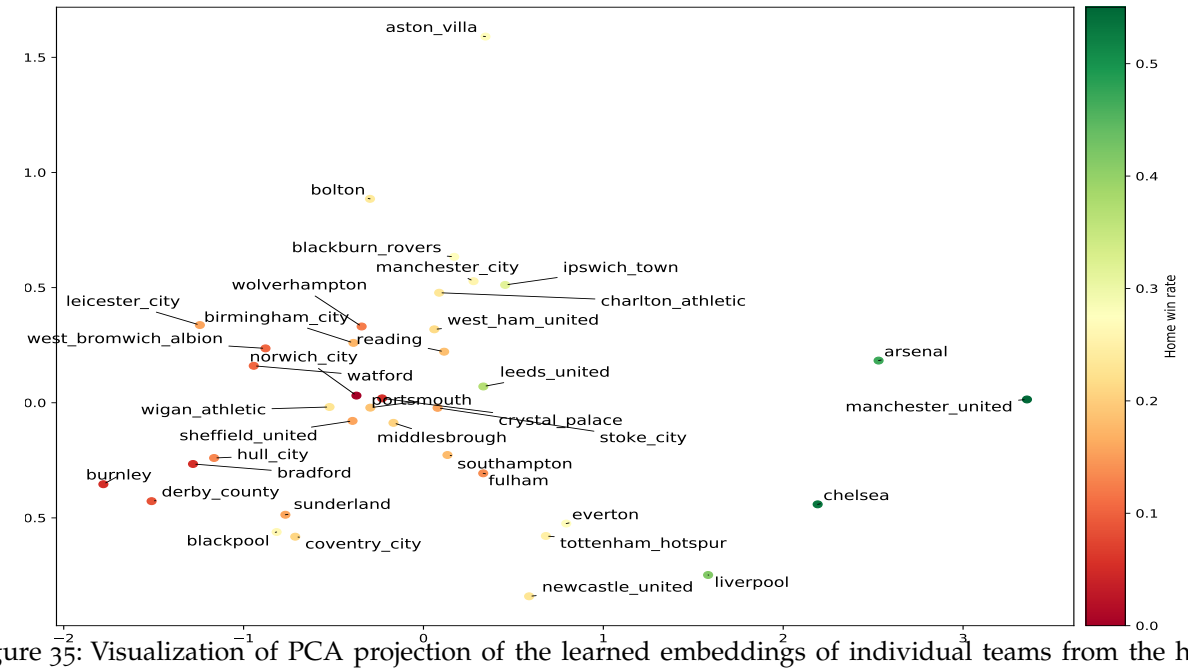

Figure 35: Visualization of PCA projection of the learned embeddings of individual teams from the home-win model. A significant relationship between the home win rate, captured by the colorscale, and the variance captured by the main X axis can be observed.

In theory, the embeddings possibly capture some information on the relational interplay between the matches as they are jointly optimized on the whole match history. However, we find this approach quite limited as it is rather naive to expect the flat, fixed-size embeddings to reflect all the possible nuances of the complex relational structure stemming from the different outcomes of different historical matches played between different teams in different orders. Fortunately with LRNNs, we can easily capture the relational structures explicitly while keeping the benefits of embedding learning. For that we first extend the template with a predicate capturing the different outcomes of *historical* matches (w.r.t. prediction match) through a learnable transformation as

$$
\begin{aligned}
w_1^{(2)}: &\quad \mathrm{outcome}(M,H,A) &\leftarrow &\quad \mathit{win}(M,H,A)\\
w_2^{(2)}: &\quad \mathrm{outcome}(M,H,A) &\leftarrow &\quad \mathit{draw}(M,H,A)\\
w_3^{(2)}: &\quad \mathrm{outcome}(M,H,A) &\leftarrow &\quad \mathrm{loss}(M,H,A)
\end{aligned}
$$

with which we accordingly extend the predictive rules as

$$
\begin{aligned}
&w^{(1)}_{h-h(1;1)} : \mathrm{outcome} \leftarrow \mathrm{home}(T1) \wedge \mathrm{type1}(T1) \wedge \mathrm{outcome}(M,T1,T2) \wedge \mathrm{type1}(T2).\\
&w^{(1)}_{h-a(1;1)} : \mathrm{outcome} \leftarrow \mathrm{home}(T1) \wedge \mathrm{type1}(T1) \wedge \mathrm{outcome}(M,T2,T1) \wedge \mathrm{type1}(T2).\\
&w^{(1)}_{h-h(1;2)} : \mathrm{outcome} \leftarrow \mathrm{home}(T1) \wedge \mathrm{type1}(T1) \wedge \mathrm{outcome}(M,T1,T2) \wedge \mathrm{type2}(T2).\\
&\qquad\qquad \dots\\
&w^{(1)}_{a-a(3;3)} : \mathrm{outcome} \leftarrow \mathrm{away}(T1) \wedge \mathrm{type3}(T1) \wedge \mathrm{outcome}(M,T2,T1) \wedge \mathrm{type3}(T2).
\end{aligned}
$$

reflecting the possible settings of *historical* home and away positions of the *actual* home and away teams in all historical matches played. By grounding of this template (Section 5.1.1), the LRNN engine assures to create the corresponding relational histories transformed into the respective, differently structured, neural networks. We denote this architecture as *relational embeddings*. The actual embeddings of teams extracted from this model learned to predict home team win can then be seen in Figure 35.

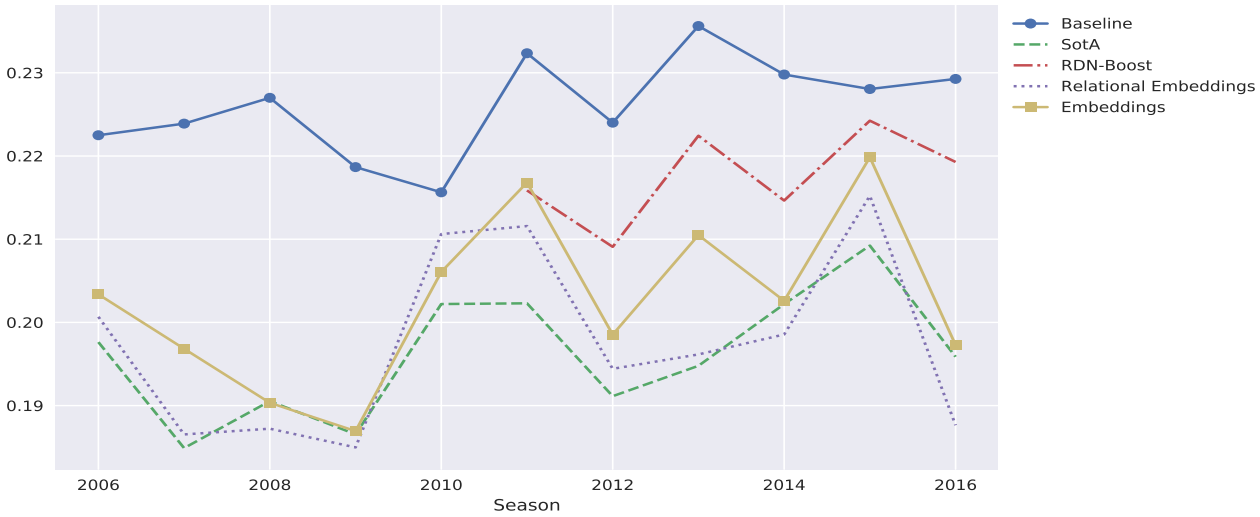

Figure 36: Comparison of performance of the learners on English Premier League as measured by the RPS metric (lower is better) from the 2017's Soccer Prediction Challenge.

12.4 EXPERIMENTS

We compared approaches discussed in this chapter on data from the 2017 Soccer Prediction Challenge [300], organized in conjunction with the MLJ's special issue on Machine Learning for Soccer. Particularly we selected the world's most prestigious English Premier League over the seasons 2006-2016 for demonstration. In the dataset, for each historical match there is merely a simple record of the team names and the resulting score. Contestants' models were then evaluated using Ranked Probability Score (RPS) [309], an evaluation metric designed for the ordinal outcomes.

For each of the historical matches, we actually extract 3 learning examples for each respective outcome $(\mathtt{loss}, \mathtt{draw}, \mathtt{win})$, and learn one corresponding model for each of the latter. For each learner we then normalize the three outputs from the three models to obtain the final predictions that form input to the RPS metric.

Calculation of the baseline involved no setup and for details on settings of the SotA and RDN-boost models we refer to our submission for the Prediction Challenge detailed in [306]. For LRNNs we set the learning rate (0.1) and number of learning steps (50), and actually utilized just a subset of the predicates (Section 12.3) used by RDN-boost. LRNNs were then trained sequentially with a history span of 5 years.

We display the final results in Figure 36. Notably, all the learners easily pass the natural baseline (mean RPS 0.2260), with LRNNs (0.1976) performing significantly better than RDN-boost (0.2175), while trailing just closely behind the state-of-the-art model (0.1961). We also see that the relational embeddings generally dominate the standard embeddings (0.2027).

12.5 CONCLUSIONS

We discussed how the domain of predictive sports analytics might benefit from relational learning approaches, and experimentally proved that even simple LRNN templates with latent relational structures may lead to surprisingly strong, competitive results in predicting soccer game outcomes.

Part VI

CONCLUSIONS

13

CONCLUSIONS

In this thesis, we addressed the problem of neural network learning with relational data and knowledge, for which we introduced a declarative deep relational learning framework called Lifted Relational Neural Networks (LRNNs) in Part iii. The main idea underlying the framework is to approach the neural networks through the lifted modeling paradigm, known otherwise from Statistical Relational Learning (SRL). Similarly to other lifted models from SRL, LRNNs are represented as sets of weighted relational logic rules, used to describe the structure of a given learning setting. Together with a set of weighted relational facts, commonly describing the learning samples, these rules define standard, or "ground", neural networks. Since, in the relational learning setting, different learning samples may compose of different sets of weighted facts, a different neural network is constructed to exploit the particular structure of each. Crucially, the weights of different ground neurons that were constructed from the same relational rule are tied both within and across the networks. Consequently, we can employ efficient optimization techniques known from standard deep learning for parameter training of the rules, which compose the lifted model.

As discussed throughout the thesis, this approach offers many benefits. As set out, it allows for end-to-end (deep) learning straight from relational data and knowledge, i.e. without any preprocessing. The (indirect) encoding through the relational rules then provides flexible means for implementing and combining a wide variety of novel modeling concepts, such as a declarative specification of latent relational pattern types which are to be learned. Besides the increased (relational) expressiveness, an important advantage of lifted models, including the LRNNs, is that they can make explicit which symmetries exist in a domain, reducing the overall number of parameters and complexity of training. Another advantage is that the rules can be provided by domain experts to incorporate additional background knowledge.

Additionally, we introduced several enhancements to the LRNN framework. Firstly, we developed a structure learning method, removing the need for specification of the rules, thus enabling for a fully automated learning process. The method is inspired by meta-interpretive learning, known from Inductive Logic Programming (ILP), in which new predicates are being "invented" on top of the previous relational patterns, for which we referred to the method as stacked structure learning. Secondly, we demonstrated the LRNNs as a differentiable logic programming language to contrast it directly with some advanced deep learning models and frameworks. Here we showed that very simple relational programs (rules) can be elegantly used to encode advanced deep learning models, with a particular focus on Graph Neural Networks (GNNs). We illustrated how to use our encoding to easily generalize the existing state-of-the-art models, while also demonstrating computational efficiency of the LRNN framework.

From the computational perspective, we then introduced two principled optimization techniques to improve scalability of the framework in Part iv. Firstly, we developed a lossless compression technique designed to scale up training of the resulting weight-sharing (dynamic) neural computation graphs via a technique inspired by lifted inference. We note that this technique is not limited to the LRNNs, but is also applicable to standard deep learning models, such as the GNNs, leading to significant speedups. Secondly, we introduced a technique incorporating learned domain theories

for lossless pruning of the logical hypothesis search space used in the structure learning. Again, this technique is also directly applicable to any classic ILP learner.

Lastly, we illustrated the framework on two selected applications in Part v. In the first use case, we demonstrated how LRNNs can be used to jointly learn hierarchical latent predictive categories of attributes, entities and relations for the task of knowledge-base completion. In the second application, we demonstrated how we successfully used LRNNs for a, somewhat less ordinary, task of soccer match outcome prediction based on learning of relational team embeddings.

13.1 FUTURE WORK

The framework already provides considerable expressiveness w.r.t. practical application domains of relational learning. We now intend to push the computational efficiency (Part iv) of the paradigm even further, using additional optimization techniques, language compilers, and emerging AI hardware, such as the Graphcore's IPU[1] architecture. This should allow to directly embrace the inherent irregularity and sparsity present in relational learning, instead of trying to circumvent it via the various approximation techniques based on mapping of logic into dense embedding spaces (Section 8). Following up on the explicit relational logic encoding, our aim is then to not only explore novel capabilities of the integrative approach to learning, but to also provide effective means to use the weighted logic programming paradigm to improve "main-stream" SotA deep learning concepts, as we did with the GNNs (Chapter 7), which currently seems as the most salient feature of our framework w.r.t. the related works (Section 8).

Finally, and perhaps most importantly, we intend to make the actual system more user friendly for the community, so as to encourage other researchers to take advantage of the concise weighted Datalog encoding, and accelerate their own exploration of novel deep relational learning concepts.

1 https://www.graphcore.ai/

Part VII

APPENDIX

A

TECHNICAL DETAILS

In this appendix part, we provide some additional technical details clarifying the descriptions introduced in the main framework Part iii, and some further content on the optimization technique described in Chapter 9.

A.1 DIFFERENCES BETWEEN THE LRNNS FROM CHAPTER 5 AND CHAPTER 7

The view on LRNNs as a differentiable logic programming language, discussed in Chapter 7, closely follows the original LRNNs framework introduced in Chapter 5. In fact, the main semantic difference is "merely" in the parameterization of the rules, where one can now include weights within the bodies (conjunctions), too, e.g.

$$w_1^{(2)} :: \mathrm{neuron}_1^{(2)}(X) \leftarrow w_1^{(1)} \cdot \mathrm{neuron}_1^{(0)}(X) \wedge w_2^{(1)} \cdot \mathrm{neuron}_2^{(0)}(X)$$

We note that this could be in essence emulated in the original LRNNs through the use of auxiliary predicates, such as in [202], as follows:

$$\begin{aligned} w_1^{(2)} :: \mathrm{neuron}_1^{(2)}(X) &\leftarrow \mathrm{neuron}_1^{(1)}(X) \wedge \mathrm{neuron}_2^{(1)}(X) \\ w_1^{(1)} :: \mathrm{neuron}_1^{(1)}(X) &\leftarrow \mathrm{neuron}_1^{(0)}(X) \\ w_2^{(1)} :: \mathrm{neuron}_2^{(1)}(X) &\leftarrow \mathrm{neuron}_2^{(0)}(X) \end{aligned}$$

which might be more appropriate in scenarios where the neurons correspond to actual logical concepts under the fuzzy logic semantics[1] (Section 5.1.3), while the second representation is arguably more suitable to exploit the correspondence with standard deep learning architectures (Section 7.2).

Another difference is that in Chapter 7 we allow tensor weights and values. While these could be again modeled in the original LRNNs, too, such as in the "soft-clustering" (embedding) construct [97] used for atom representation learning:

$$\begin{aligned} w_{o_1} :: gr_1(X) &\leftarrow O(X) \quad \ldots \quad w_{h_1} :: gr_1(X) \leftarrow H(X) \\ w_{o_2} :: gr_2(X) &\leftarrow O(X) \quad \ldots \quad w_{h_2} :: gr_2(X) \leftarrow H(X) \end{aligned}$$

the explicit tensor-valued weights offer an arguably more elegant representation of the same construct:

$$[\mathbf{w}_{o_1}, \mathbf{w}_{o_2}] :: gr(X) \text{ :- } O(X) \quad \ldots \quad [\mathbf{w}_{h_1}, \mathbf{w}_{h_2}] :: gr(X) \text{ :- } H(X)$$

In general, the new representation can be thought of as merging the scalar weights of individual neurons into tensors used by the *nodes* (prev. referred to as "neurons" in the original LRNNs) in the computation graph. Note that it is possible to process any ground LRNN network into this form, i.e. turn individual neurons into matrix layers, in a similar manner, as outlined in Algorithm 2.

1 Note that any model from the original LRNNs can still be directly encoded in the new formalism.

Algorithm 2 Transforming ground neural network into vectorized form

```
1:  function VECTORIZE(neurons)
2:      𝒩 ← ⋃neurons
3:      (depth, 𝒩) ← topologicOrder(𝒩)
4:      Layers = ∅
5:      for i = 1 : depth do
6:          M_i ← initMatrix()
7:          ℳ ← neuronsAtLevel(i, 𝒩)
8:          for neuron ∈ ℳ do
9:              (Inputs, Weights) ← inputs(neuron)
10:             for (input, weight) ∈ (Inputs, Weights) do
11:                 if getLevel(input) = i + 1 then
12:                     M_i(neuron, input) = weight
13:                 else
14:                     skipConnect ← void(neuron, i + 1)
15:                     M_i(neuron, skipConnect) = 1
16:         Layers = Layers ∪ M_i
17:     return Layers
```

Algorithm 3 Pruning linear chains of unnecessary transformations

```
1:  function PRUNE(neurons)
2:      𝒩 ← ⋃neurons
3:      for neuron ∈ 𝒩 do
4:          (inputs, weights) ← inputs(neuron)
5:          if inputs.size = 1 ∧ weights = ∅ then
6:              outputs ← outputs(neuron)
7:              for output ∈ outputs do
8:                  input = inputs[0]
9:                  output.replaceInput(neuron, input)
10:                 input.replaceOutput(neuron, output)
11:     return connectedComponent(𝒩)
```

Being heavily utilized in deep learning, such transformation can significantly speed up the training of the, more regularly structured and dense, networks. Moreover, by the resulting reduction of the number of rules, effectively merging together semantically equivalent rules which do not differ up to their (scalar) parameterization, we now also alleviate much of the complexity during network creation, i.e. calculation of the least Herbrand model (Section 3.1.2), by avoiding repeated calculations. This results into a significant speedup during the model creation process.

A.1.1 *Network Pruning*

Following the computation graph creation procedure from Section 7.1.2, we might end up with unnecessary trivial neural transformations through auxiliary predicates in cases, where the original rules have only a single literal in their body and are unweighted. For mitigation, we can apply a straightforward procedure for detection and removal of linear chains of these trivial operations, as described in Algorithm 3. While such an operation arguably changes the inference and logical semantics of the original model, these structures do not contribute to learning capacity of the model and, on the contrary, cause gradient diminishing. This technique is thus particularly suited for improving training performance in correspondence with standard deep learning architectures.

While this form of pruning can be theoretically performed directly in the template, it is easier to execute as a post-processing step in the resulting neural networks, which is what we do.

A.2 LOSSLESS MODEL COMPRESSION VIA LIFTING

In this section we enclose some additional information on the lossless compression optimization techniques introduced in Chapter 9, and describe the compression algorithms from a more practical perspective.

THE NON-EXACT ALGORITHM The idea behind the heuristic algorithm, is to pre-calculate the equivalence classes of neurons $N \in \mathcal{N}$ via random initialization of the weights $\mathcal{W}$ (Algorithm 5), and then to iterate the neurons in the computation graphs G, starting from *output nodes*, while continuously replacing each node N by some, so far *unprocessed*, equivalent node $N' \equiv N$, if there are any (Algorithm 4).

Algorithm 4 Processing Neurons

```
1: function Compression(G, W)
2:     (N, E, F) ← G
3:     M, M̄ ← EquiClasses(N, W)
4:     N' ← topologicOrder(N)
5:     for n ∈ N' do
6:         values ← M[n]
7:         if values ∈ M̄ then
8:             n' ← M̄[values]
9:             replace n by n' in G
```

Algorithm 5 Equivalence Classes of Neurons

```
1: function EquiClasses(N, W)
2:     M[neuron ↦ valueList] ← ∅
3:     i ← 0
4:     while i < repetitions do
5:         W ← randomInit (W)
6:         (N, V) ← inferValues (N, W)
7:         for (neuron, value) ∈ (N, V) do
8:             valueList ← M[neuron]
9:             valueList = valueList ∪ value
10:        i ← i + 1
11:    M̄[valueList ↦ N'] ← reverseMap (M)
12:    return M, M̄
```

Iterating the graph in a top-down fashion while replacing the neurons by their effectively last equivalents, in the given processing order, then ensures maximal possible compression. The repeated initialization of weights, as inspired by the polynomial identity testing [310], is then used to minimize the chances of two different nodes accidentally yielding the same value. Apart from these repetitions, the efficiency of the routine can be also simply increased by replacing the standard NN activation functions with injective mappings to a wider output value range.

THE EXACT ALGORITHM The idea is again to pre-calculate the equivalence classes of neurons in $\mathcal{N}$ via random initializations of $\mathcal{W}$, however, this time it is only used for speedup. In this algorithm, we iterate the neurons in G, starting from the *leaf nodes*, while continuously replacing each node N by some *structurally equivalent* node $N' \underset{\text{struct}}{\equiv} N$, which has *already* been processed. The structural equivalence can then be simply seen as defined recursively by the following statements:

1. every 2 nodes with no inputs and the same output value are necessarily equivalent
2. every 2 nodes with the same activation function and equivalent sets of inputs (including the weights) are necessarily equivalent, too.

Processing the nodes in the bottom-up order then ensures that gradually bigger subgraphs can be checked for the structural equivalence in linear time w.r.t. number of the root neuron input connections.

Note that the whole compression process is in both cases linear w.r.t. size of the network, and is thus no more (asymptotically) demanding than a single gradient descent step.

A NOTE ON THE LOSSY COMPRESSION Note that if two subgraphs are structurally equivalent in the sense introduced in Section 9.3, they necessarily lead to the same computations, including parameter updates, and can thus be interchanged safely. However, even if two subgraphs producing the same value are not structurally equivalent, they may still be functionally equivalent, i.e. perform effectively the same computation, which happens e.g. in (linear) arithmetic circuits, rendering the structural equivalence check too strict. Moreover if the values for 2 subgraphs differ just slightly, one might expect to also observe similar learning behavior. This led to the idea of the lossy compression, which can be easily emulated by conditioning the value equality check between nodes with some numeric precision. The experimental results of such a setting were then discussed in Section 9.5.

A.2.1 *Graph-Based Model Encoding in LRNNs*

Here we detail the LRNN encoding of the models reported in the experiments in Chapter 9.

MOLECULE CLASSIFICATION For all the molecular models (templates), we first map all the atom and bond types into (3-dimensional) embedding vectors as

$$\begin{array}{lll} \{3\} : \mathtt{atom}_{emb}(X) :- \mathtt{O}(X). & & \{3\} : \mathtt{bond}_{emb}(X) :- \mathtt{1}(X) \\ \dots \quad \dots \quad \dots & & \dots \quad \dots \quad \dots \\ \{3\} : \mathtt{atom}_{emb}(X) :- \mathtt{H}(X) & & \{3\} : \mathtt{bond}_{emb}(B) :- \mathtt{ar}(B). \end{array}$$

The GNN templates were then encoded analogically to the explanation in Section 7.3. For instance, the GNN corresponding to a 2-layered spatial GCN [58] (Section 2.4), was encoded as

$$\begin{aligned} \mathtt{hidden}_{emb}(X) :- \ & \{3\}\ \mathtt{bond}(X,Y,B),\ \{3,3\}\ \mathtt{atom}_{emb}(Y),\ \{3,3\}\ \mathtt{bond}_{emb}(B). \\ \{3\}\ \underset{\mathtt{molecule}}{\mathtt{goal}} :- \ & \{3\}\ \mathtt{bond}(X,Y,B),\ \{3,3\}\ \mathtt{hidden}_{emb}(Y),\ \{3,3\}\ \mathtt{bond}_{emb}(B). \end{aligned}$$

and similarly for the other GNN models, which differ merely in the activation functions and number of layers. The relational "graphlets" template then directly corresponds to Example 8 with no input weights but a "cross-product" application of $g_\wedge$, encoded as

$$\begin{aligned} 243\ \underset{\mathtt{molecule}}{\mathtt{goal}} :- \ & \mathtt{bond}(X,Y,B),\ \mathtt{bond}(Y,Z,C),\ \mathtt{bond}_{emb}(B),\ \mathtt{bond}_{emb}(C), \\ & \mathtt{atom}_{emb}(X),\ \mathtt{atom}_{emb}(Y),\ \mathtt{atom}_{emb}(Z). \end{aligned}$$

KNOWLEDGE BASE COMPLETION In the selected, commonly used, KBC datasets, particularly the Kinships, Nations, and UMLS [66], the tuples are represented in a reified form as $\mathtt{r}(R,O,S)$. Analogously to the molecules, we again started by mapping all items into embeddings, which are then utilized by similar MLP-based KBE (Section 2.4) templates for each of the 3 datasets as

$$\begin{aligned} \mathtt{person1}_{emb}(O) :- \ & \{3\}\ \mathtt{r}(R,O,S),\ \{3,3\}\ \mathtt{rel}_{emb}(R),\ \{3,3\}\ \mathtt{person}_{emb}(S). \\ \mathtt{person2}_{emb}(S) :- \ & \{3\}\ \mathtt{r}(R,O,S),\ \{3,3\}\ \mathtt{person}_{emb}(O),\ \{3,3\}\ \mathtt{rel}_{emb}(R). \\ \{3\}\ \underset{\mathtt{kinships}}{\mathtt{goal}}\ (O,R,S) :- \ & \{3,3\}\ \mathtt{person1}_{emb}(O), \{3,3\}\ \mathtt{rel}_{emb}(R),\ \{3,3\}\ \mathtt{person2}_{emb}(S). \end{aligned}$$

$$\begin{aligned} \{3\}\ \underset{\mathtt{nation}_{att}}{\mathtt{goal}}\ (N,A) :- \ & \{3,3\}\ \mathtt{nation}_{emb}(N),\ \{3,3\}\ \mathtt{att}_{emb}(A). \\ \mathtt{nat1}_{emb}(O) :- \ & \{3\}\ \mathtt{r}(R,O,S),\ \{3,3\}\ \mathtt{rel}_{emb}(R),\ \{3,3\}\ \mathtt{nation}_{emb}(S). \\ \mathtt{nat2}_{emb}(S) :- \ & \{3\}\ \mathtt{r}(R,O,S),\ \{3,3\}\ \mathtt{nation}_{emb}(O),\ \{3,3\}\ \mathtt{rel}_{emb}(R). \\ \{3\}\ \underset{\mathtt{nations}}{\mathtt{goal}}\ (O,R,S) :- \ & \{3,3\}\ \mathtt{nat1}_{emb}(O),\ \{3,3\}\ \mathtt{rel}_{emb}(R),\ \{3,3\}\ \mathtt{nat2}_{emb}(S). \end{aligned}$$

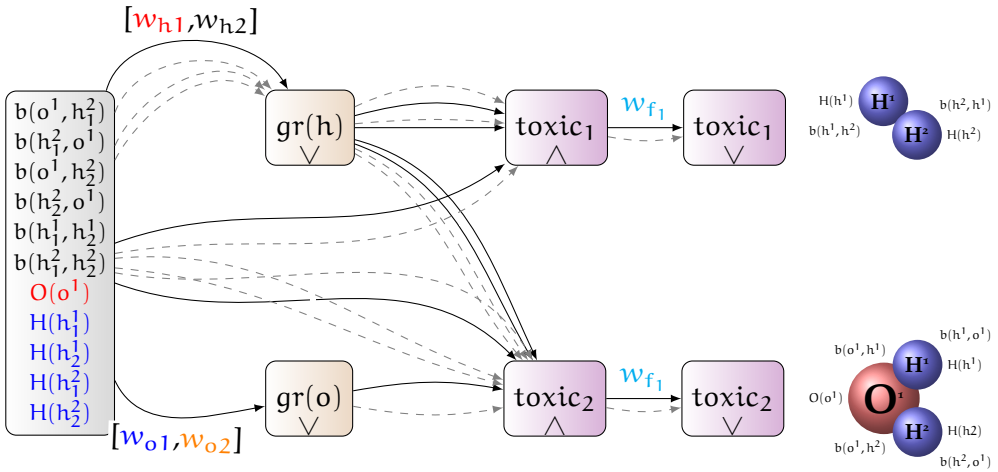

Figure 37: A compressed representation of the two networks from Figure 5, with dashed lines illustrating the compressed parts.

$$
\begin{aligned}
&\mathtt{item1}_{emb}(O) :- \{3,3\}\ \mathtt{item}_{emb}(O).\\
&\mathtt{item1}_{emb}(O) :- \{3\}\ r(R,O,S),\ \{3,3\}\ \mathtt{rel}_{emb}(R),\ \{3,3\}\ \mathtt{item}_{emb}(S).\\
&\mathtt{item2}_{emb}(S) :- \{3\}\ r(R,O,S),\ \{3,3\}\ \mathtt{item}_{emb}(O),\ \{3,3\}\ \mathtt{rel}_{emb}(R).\\
&\mathtt{item2}_{emb}(S) :- \{3,3\}\ \mathtt{item}_{emb}(S).\\
\underset{\text{UMLS}}{\{3\}}\ &\mathtt{goal}(O,R,S) :- \{3,3\}\ \mathtt{item1}_{emb}(O),\ \{3,3\}\ \mathtt{rel}_{emb}(R),\ \{3,3\}\ \mathtt{item2}_{emb}(S).
\end{aligned}
$$

A.2.2 *Compression of the LRNN Templates*

Previously, in Section 9.4.1, we demonstrated the compression effect on a classic GNN model in Figures 26 and 27, respectively. Let us now do the same for a relational LRNN template encoding the "graphlets" idea introduced in the Example 8, with the corresponding ground networks displayed in Figure 5. The resulting effect of all the compression techniques, here also including the vectorization and pruning, as introduced in Section A.1, is displayed in Figure 37.

A.3 A NOTE ON THE THEOREM PROVING AND MODEL COMPLEXITY

Naturally, the performance of the theorem prover, or grounder, depends heavily on the complexity of the template, and can theoretically lead to excessively large models. Nevertheless for LRNNs, we only need to construct the least Herbrand model, which is typically much smaller in the real-world (sparse) templates, such as the GNN computation schemes (Chapter 7). Also, excessively large models would be translated into neural computation graphs too large to be trainable in practice, anyway. Consequently, the overhead of the theorem prover (grounder) is commonly negligible w.r.t. the overall learning time for practical model classes.

Note that the same also applies for large input graphs, such as the knowledge-bases, which can be learned from with LRNNs, too (such as in [171]). For instance, the grounding and network creation overhead of related (multi-relational) GNN models combining 1 layer of GNN with KBE (described in Sections A.2.1, 9.5 and reported in [214] [2]) on the KBC datasets of Nations, Kinships and UMLS [66] takes 1s, 8s and 22s, corresponding to Herbrand models with app. 14314, 281552, and 845511 atoms, translated into 11681, 42196, and 27506 neurons, leading to train epocha times of app. 1.2s 0.440s, and 0.660s, respectively.

2 available at https://github.com/GustikS/NeuraLifting

B

IMPLEMENTATION

The somewhat unorthodox and novel learning paradigm, as introduced in this thesis, made it practically impossible to build on top of existing popular frameworks, such as PyTorch or TensorFlow, which are inherently based on the common tensor-based, attribute-value machine learning principles. Consequently, we had to implement the whole functionality from scratch[1].

In contrast to most of the contemporary Python machine learning frameworks, the LRNN framework is implemented in Java. This unpopular[2] choice was based in the crucial need for overall computation performance, which here is not limited to bottlenecks in the form of matrix (tensor) transformations, which can be easily outsourced from Python to C++, but includes complex structured (recursive) operations stemming from the relational aspects of the proposed learning paradigm. Consequently, fast algorithms have been required in all stages, ranging from the logic language parsing and grounding, tightly integrated with the automated neural network creation, to the subsequent parameter optimization. The typical two-language barrier present in the common Python/C++ frameworks would thus be even more problematic here[3]. That being said, there are two (very) high-level Python front-ends[4], wrapping the input user interface into the framework.

B.1 USER WORKFLOW

One way or another, the user workflow with the framework starts by providing the (i) learning examples, (ii) target queries, and (iii) the learning template. The learning examples are expected to be encoded in the discussed language of weighted Datalog (Section 7.1.1.1), which covers a wide range of possible application domains (Section 1.2). For instance, one can provide list of molecular graphs with rich annotations encoded, e.g., as follows

```
0.0 toxic :- [0.1,7] bond(c_1,h_2,b_type_1), [0.4,-3,6] atom(c_1), [0.1,-2,3] atom(h_2),
             [1,2] type(b_type_1), [2.1,-0.3] bond(h_2,c_3,b_type_4), ... .
...
0.9 toxic :- [0.2,-1] bond(c_1,f_1,b_type_6), [0.4,-1,2] atom(c_1),
             [0.1,-3,0] atom(f_1), ... .
```

each of which can be associated with some query, such as the simple ground fact "toxic" here, representing the value of the target label (toxicity) of each molecule. Note that there may be multiple queries associated with each example (provided as a comma-separated list), and these queries also do not even need to be ground, which creates space for interesting learning settings, such as collective classification [311]. This format of input files generally corresponds to standard supervised machine learning scenarios.

1 Also, there were no such frameworks at the time we started with the project. An analysis of problems with the existing deep learning frameworks can then be found in Section C.3

2 unpopular in classic ML but actually quite popular in SRL, for the aforementioned reasons.

3 Also, Julia was in infancy by that time.

4 One for the original version of LRNNs (Chapter 5) stemming from the work described in Section C.2 is available at https://github.com/jendas1/jupyter_neurologic. Another one for the more current version (Chapter 7), currently still in progress, is available at https://github.com/LukasZahradnik/PyNeuraLogic.

One may also wish to learn in the knowledge-base completion mode, which is very similar, except that there is but a single (large) structure provided as the example input. In this case, the queries are expected to be provided in a separate file, e.g. as a list

```
1 holds(amino_acid_peptide,affects,cell_function) .
...
0 holds(reptile,result_of,vitamin) .
```

of true and false facts to be deduced from the given knowledge-base, provided again in the same (weighted) Datalog format.

The (relational) inference process is then encoded by the template. This can be also understood as encoding of the deep learning architectures, and used as such (as shown in Chapter 7). For example, to learn to classify the molecules with a GNN model, one could provide a template in the form of

```
{10,10} layer1(X) :- {10,3} atom(X), {10,3} atom(Y), bond(X,Y,B), {10,2} type(B) .
{5,5}   layer2(X) :- {5,10} layer1(X), {5,10} layer1(Y), bond(X,Y,B), {10,2} type(B) .
{1,5}   toxic     :- {5,5} layer2(X) . [activation=sigmoid, aggregation=avg]
```

encoding a deep GNN with 6 layers[5] of matrix transformations, hierarchically embedding the features of both the atoms and the bonds between. These are interleaved with 3 layers of (average) pooling, the final of which is a global one (since there is no variable in the `toxic` literal), aggregating all the atom embeddings into a single (scalar) value corresponding to the fact (query) toxic, as prescribed by the last rule. The resulting value of `toxic` will then be automatically matched with each of the example molecule's query atom value within each of the induced networks. Finally, the parameters associated with the rules[6] will then be automatically trained to minimize the discrepancy between the two values. Note that much more complex models than the GNNs[7], can be declared in the framework just as easily (as demonstrated in Chapter 7.3.1).

We also note that there are many other (meta-data) annotations one can use to further specify various behaviors of the templates for different use cases, which is ultimately prescribed by the actual grammar of the LRNN templating language. Likewise, there are many settings which can be passed to the framework itself. For getting started with some running examples we refer to the Github project (Section B.3).

B.2 PROJECT STRUCTURE

The project is currently of a considerable size and complexity (50,000+ lines of code) but, since the original LRNN version [97] (Chapter 5), it has been completely reimplemented from scratch, with a substantial focus on flexibility and modularity. For instance it is easy to add new types of logical and non-logical constructs, functions, computation node types, node and weight-sharing schemes, modes of grounding, graph iteration and backpropagation, training strategies, etc. This is mainly based on a custom, modular execution graph-based software architecture, where the blocks in the actual workflow of the framework are wrapped in abstract nodes of the execution graph (Figure 38), which is automatically inferred at startup based on the provided settings and input data formats. This allows to adaptively incorporate the specific instances of, e.g., the grounding and training strategies (etc.), as part of the automated startup assembly of the workflow. Based

5 Note that if there are weights provided in both head and body of a rule, it actually corresponds to two consecutive transformations (Section 7.2.1). The remaining (non-specified) activation and aggregation functions here are automatically set at defaults (`tanh` and `avg`), which can also be specified in the framework settings.

6 Note that with the {...} brackets we only specify the *dimensions* of the (parameter) values. Nevertheless one can also specify the actual values with [...] brackets, which can be also additionally fixed to a specific (non-trainable) value with the < ... > brackets. For complete specification, please see the grammar of the templating language.

7 e.g. incorporating latent graphlet patterns, hyper-graphs etc. (Chapter 7.3.2)

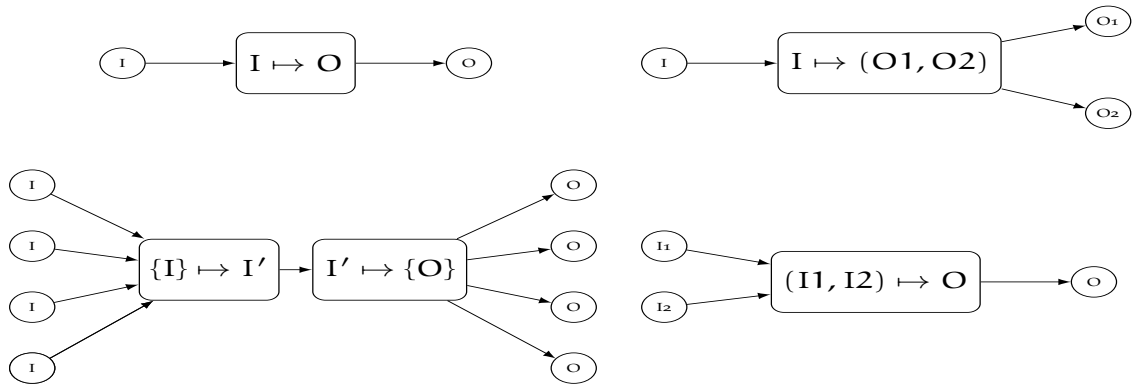

Figure 38: Different types of abstract computation blocks used in the learning workflow. Prior to execution, these block are firstly assembled into a workflow graph, the structure of which is dynamically inferred from the input data and settings. The Java generics on the respective input and output types (I/O) then provide flexibility while ensuring compile-time safety of all the possible learning scenarios, represented by the different execution graph configurations.

heavily on the Java generics and *Streams*, these reconfigurable workflow graphs then provide type-safety and efficient streaming of computation through the blocks.

The structure of the project itself then follows the standard Maven layout, consisting of the modules listed in Table 10.

B.3 REFERENCE

The LRNN framework (Part iii) which we gave a working title "*NeuraLogic*"[8], is available at Github:

- **project name**: NeuraLogic
- **home page**: `https://github.com/GustikS/NeuraLogic`
- **operating system**: platform independent
- **requirements**: Java 1.8 or higher
  - both OpenJDK and Oracle JDK tested
  - for use, the runtime environment (JRE) will suffice
- **license**: MIT License (free)

8 obviously, due to the integration of *Neural* networks and (relational) *Logic* programming

Table 10: A list of the high-level modules of the project.

Module	Description
Algebra	value definitions (scalar/vector/matrix...) with the respective operations
CLI	simple command line interface to the framework, based on Apache commons' CLI
Drawing	visualization of templates, grounding/inference and neural networks, based on the DOT language (and GraphViz engine)
Frontend	Python scripts for calling high-level functionalities, reading results, and (optionally) exporting neural networks to Dynet, based on Py4J (in progress)
Learning	high-level (supervised) machine learning definitions and functions
Logging	simple logging and (color) formatting, based on the default java library
Logic	base classes for working with logic and a subsumption engine providing efficient first order logic grounding and inference, developed by Ondrej Kuzelka
Logical	the logical part of the integration containing logic-based structures and computation - i.e. weighted logic grounding, based on the Logic module
Neural	the neural part of the integration containing neural-based structures and computation - i.e. neural networks processing, backpropagation and standard deep learning functions and operations
Neuralization	the process of conversion from the logical to the neural structures
Parsing	definition and parsing of the NeuraLogic language into internal representation, based on the ANTLR parser generator
Pipelines	high-level abstraction library for creating generic execution graphs, suitable for ML workflows (custom made, Figure 38)
Resources	test resources, templates, datasets, etc.
Settings	central configuration/validation of all the settings and input sources (files)
Utilities	generic utilities (maths, java DIY, etc.), utilizing GSON for serialization and JMH for microbenchmarking
Workflow	specific building blocks for typical ML worflows used with this framework, based on the Pipelines module

C

OTHER APPLICATIONS OF THE FRAMEWORK

C.1 "RELATIONAL LEARNING WITH NEURAL NETWORKS FOR MACHINE TRANSLATION"

This work [312] utilizes the LRNN framework for automatic evaluation of machine translation. Particularly, it aims to automate a human-based translation quality metric based on top of so-called Universal Conceptual Cognitive Annotation (UCCA) semantic representation of sentences. The designed LRNN templates are used to aggregate the information about semantic roles of constituents and sub-structures of each sentence into latent (relational) representations, which are simultaneously used for evaluation of both the leaves and internal nodes of the tree-based tectogrammatical UCCA representation. In the experiments, the approach is then compared against classic (feature-based) neural network techniques on translation data from English to four different languages.

C.2 "LEARNING RELEVANT REASONING PATTERNS WITH NEURO-LOGIC PROGRAMMING"

This introductory work [313] demonstrates the capability of the LRNN framework to capture diverse AI tasks based on different generic "reasoning patterns". It describes common examples of such reasoning patterns used in statistical and symbolic AI approaches and demonstrates how each particular pattern may be captured within LRNNs. It further details the patterns in context of learning and reasoning, with an extra focus on abilities that arise from combination of both. On selected examples from simple game environments, it illustrates how the joint, declarative approach of LRNNs broadens the scope of existing reasoning patterns through the ability to represent and reason with relational information, while keeping the benefits of neural learning.

C.3 "INTEGRATION OF RELATIONAL AND DEEP LEARNING FRAMEWORKS"

This practically oriented work [314] targets the computational efficiency of the LRNNs w.r.t. existing deep learning frameworks, with a special focus on the underlying dynamic computation graphs. As discussed in Section 7.4.4, the dynamic nature and irregular sparse structure of the computation in relational learning scenarios introduces considerable problems to the common deep learning frameworks. This works targets the associated problems via mathematical analysis, custom implementation, as well as interfacing with the existing frameworks of PyTorch, TensorFlow, and DyNet. In the experiments it then compares the various solutions and demonstrates that none of the competing frameworks was particularly well suited for the sparse, dynamic and irregular computation arising in relational learning.

BIBLIOGRAPHY

[1] Karl Friston. "The free-energy principle: a unified brain theory?" In: *Nature reviews neuroscience* 11.2 (2010), pp. 127–138.

[2] Gary Marcus. "The next decade in AI: four steps towards robust artificial intelligence." In: *arXiv preprint arXiv:2002.06177* (2020).

[3] Suresh P Sethi and Gerald L Thompson. *What is optimal control theory?* Springer, 2000.

[4] Yoshua Bengio. "Learning Deep Architectures for AI." In: *Foundations and Trends in Machine Learning* 2.1 (2009), pp. 1–127. ISSN: 1935-8237.

[5] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. "Mastering the game of Go with deep neural networks and tree search." In: *nature* 529.7587 (2016), pp. 484–489.

[6] M. Riesenhuber and T. Poggio. "Hierarchical models of object recognition in cortex." In: *Nature neuroscience* 2.11 (Nov. 1999), pp. 1019–25. ISSN: 1097-6256.

[7] Jürgen Schmidhuber. "Deep learning in neural networks: An overview." In: *Neural networks* 61 (2015), pp. 85–117.

[8] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups." In: *IEEE Signal processing magazine* 29.6 (2012), pp. 82–97.

[9] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. "Gradient-based learning applied to document recognition." In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[10] Kunihiko Fukushima. "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position." In: *Biological Cybernetics* 36.4 (1980), pp. 193–202. ISSN: 03401200.

[11] Juyang Weng, Narendra Ahuja, and Thomas S Huang. "Cresceptron: a self-organizing neural network which grows adaptively." In: *IJCNN International Joint Conference on Neural Networks*. Vol. 1. IEEE. 1992, pp. 576–581.

[12] Sebastian Bader, Pascal Hitzler, and Andreas Witzel. "Integrating First-Order Logic Programs and Connectionist Systems — A Constructive Approach." In: *Proceedings of the IJCAI-05 Workshop on Neural-Symbolic Learning and Reasoning, NeSy'05, held at IJCAI-05* (2005).

[13] Artur S. d'Avila Garcez, Tarek R Besold, Luc De Raedt, Peter Földiak, Pascal Hitzler, Thomas Icard, Kai-uwe Kühnberger, Luis C Lamb, Risto Miikkulainen, and Daniel L Silver. "Neural-Symbolic Learning and Reasoning : Contributions and Challenges." In: *Proceedings of the AAAI Spring Symposium on Knowledge Representation and Reasoning: Integrating Symbolic and Neural Approaches, Stanford* (2014), pp. 18–21.

[14] Lise Getoor and Benjamin Taskar. *Introduction to Statistical Relational Learning*. 2007, p. 586. ISBN: 0262072882.

[15] Mark Krogel, Simon Rawles, Filip Železný, Peter Flach, Nada Lavrač, and Stefan Wrobel. "Comparative evaluation of approaches to propositionalization." In: *International Conference on Inductive Logic Programming*. Springer. 2003, pp. 197–214.

[16] Manoel VM França, Gerson Zaverucha, and Artur S d'Avila Garcez. "Fast relational learning using bottom clause propositionalization with artificial neural networks." In: *Machine learning* 94.1 (2014), pp. 81–104.

[17] Johannes Kobler, Uwe Schöning, and Jacobo Torán. *The graph isomorphism problem: its structural complexity*. Springer Science & Business Media, 2012.

[18] Luc De Raedt. *Logical and relational learning*. Springer Science & Business Media, 2008.

[19] Stephen Muggleton and Luc De Raedt. "Inductive logic programming: Theory and methods." In: *The Journal of Logic Programming* 19 (1994).

[20] Raymond M Smullyan. *First-order logic*. Courier Corporation, 1995.

[21] Warren S. McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity." In: *The Bulletin of Mathematical Biophysics* 5.4 (Dec. 1943), pp. 115–133. ISSN: 00074985.

[22] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. "Distributed Representations of Words and Phrases and their Compositionality." In: *Advances in Neural Information Processing Systems* 26 (2013), pp. 3111–3119.

[23] Alex A Alemi, François Chollet, Niklas Een, Geoffrey Irving, Christian Szegedy, and Josef Urban. "Deepmath-deep sequence models for premise selection." In: *arXiv preprint arXiv:1606.04442* (2016).

[24] Luc De Raedt, Sebastijan Dumančić, Robin Manhaeve, and Giuseppe Marra. "From Statistical Relational to Neuro-Symbolic Artificial Intelligence." In: *arXiv preprint arXiv:2003.08316* (2020).

[25] Luís C. Lamb, Artur S. d'Avila Garcez, Marco Gori, Marcelo O. R. Prates, Pedro H. C. Avelar, and Moshe Y. Vardi. "Graph Neural Networks Meet Neural-Symbolic Computing: A Survey and Perspective." In: *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*. Ed. by Christian Bessiere. ijcai.org, 2020, pp. 4877–4884.

[26] Richard Evans, David Saxton, David Amos, Pushmeet Kohli, and Edward Grefenstette. "Can neural networks understand logical entailment?" In: *arXiv preprint arXiv:1802.08535* (2018).

[27] Richard Evans and Edward Grefenstette. "Learning explanatory rules from noisy data." In: *Journal of Artificial Intelligence Research* 61 (2018), pp. 1–64.

[28] Rasmus Palm, Ulrich Paquet, and Ole Winther. "Recurrent relational networks." In: *Advances in Neural Information Processing Systems*. 2018, pp. 3368–3378.

[29] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. "Machine learning for combinatorial optimization: a methodological tour d'horizon." In: *European Journal of Operational Research* (2020).

[30] Marcelo Prates, Pedro HC Avelar, Henrique Lemos, Luis C Lamb, and Moshe Y Vardi. "Learning to solve np-complete problems: A graph neural network for decision tsp." In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33. 2019, pp. 4731–4738.

[31] Chris Cameron, Rex Chen, Jason S Hartford, and Kevin Leyton-Brown. "Predicting Propositional Satisfiability via End-to-End Learning." In: *Aaai*. 2020, pp. 3324–3331.

[32] Alex Graves, Greg Wayne, and Ivo Danihelka. "Neural Turing Machines." In: *arXiv preprint* (Oct. 2014).

[33] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. "Hybrid computing using a neural network with dynamic external memory." In: *Nature* 538.7626 (2016), pp. 471–476.

[34] Nuri Cingillioglu and Alessandra Russo. "Deeplogic: Towards end-to-end differentiable logical reasoning." In: *arXiv preprint arXiv:1805.07433* (2018).

[35] Zachary C Lipton, John Berkowitz, and Charles Elkan. "A critical review of recurrent neural networks for sequence learning." In: *arXiv preprint arXiv:1506.00019* (2015).

[36] Artur d'Avila Garcez, Marco Gori, Luis C Lamb, Luciano Serafini, Michael Spranger, and Son N Tran. "Neural-symbolic computing: An effective methodology for principled integration of machine learning and reasoning." In: *arXiv preprint arXiv:1905.06088* (2019).

[37] Geoffrey G. Towell and Jude W. Shavlik. "Knowledge-based artificial neural networks." In: *Artificial Intelligence* 70.1-2 (1994), pp. 119–165. ISSN: 00043702.

[38] Paul Smolensky. "Tensor product variable binding and the representation of symbolic structures in connectionist systems." In: *Artificial intelligence* 46.1-2 (1990), pp. 159–216.

[39] M Botta, A Giordana, and R Piola. "FONN: Combining First Order Logic with Connectionist Learning." In: *Proceedings of the Fourteenth International Conference on Machine Learning* (1997), pp. 46–56.

[40] Liya Ding. "Neural prolog-the concepts, construction and mechanism." In: *1995 IEEE International Conference on Systems, Man and Cybernetics. Intelligent Systems for the 21st Century*. Vol. 4. IEEE. 1995, pp. 3603–3608.

[41] Artur S Avila Garcez and Gerson Zaverucha. "The connectionist inductive learning and logic programming system." In: *Applied Intelligence* 11.1 (1999), pp. 59–77.

[42] Luciano Serafini and Artur d'Avila Garcez. "Logic tensor networks: Deep learning and logical reasoning from data and knowledge." In: *arXiv preprint arXiv:1606.04422* (2016).

[43] Honghua Dong, Jiayuan Mao, Tian Lin, Chong Wang, Lihong Li, and Denny Zhou. "Neural logic machines." In: *arXiv preprint arXiv:1904.11694* (2019).

[44] Giuseppe Marra, Michelangelo Diligenti, Francesco Giannini, Marco Gori, and Marco Maggini. "Relational neural machines." In: *arXiv preprint arXiv:2002.02193* (2020).

[45] Hendrik Blockeel and Werner Uwents. "Using neural networks for relational learning." In: *ICML 2004 workshop on statistical relational learning and its connections to other fields*. 2004, pp. 23–28.

[46] Tirtharaj Dash, Ashwin Srinivasan, Lovekesh Vig, Oghenejokpeme I Orhobor, and Ross D King. "Large-scale assessment of deep relational machines." In: *International Conference on Inductive Logic Programming*. Springer. 2018, pp. 22–37.

[47] Seyed Mehran Kazemi and David Poole. "Bridging Weighted Rules and Graph Random Walks for Statistical Relational Models." In: *Frontiers in Robotics and AI* 5 (Feb. 2018), p. 8. ISSN: 2296-9144.

[48] Patrick Hohenecker and Thomas Lukasiewicz. "Ontology reasoning with deep neural networks." In: *Journal of Artificial Intelligence Research* 68 (2020), pp. 503–540.

[49] Daniel D Johnson. "Learning graphical state transitions." In: *International Conference on Learning Representations* (2016).

[50] Kristian Kersting and Luc De Raedt. "Towards Combining Inductive Logic Programming with Bayesian Networks." In: *Inductive Logic Programming, 11th International Conference, ILP 2001, Strasbourg, France, September 9-11, 2001, Proceedings*. 2001, pp. 118–131.

[51] Matthew Richardson and Pedro Domingos. "Markov logic networks." In: *Machine learning* 62.1-2 (2006), pp. 107–136.

[52] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. "ProbLog: A Probabilistic Prolog and Its Application in Link Discovery." In: *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence*. 2007, pp. 2462–2467.

[53] Geoffrey E. Hinton. "Mapping part-whole hierarchies into connectionist networks." In: *Artificial Intelligence* 46.1-2 (1990), pp. 47–75. ISSN: 00043702.

[54] Andreas Stolcke and Dekai Wu. "Tree matching with recursive distributed representations." In: *International Computer Science Institute* (1992).

[55] Richard Socher, Danqi Chen, Christopher D Manning, and Andrew Ng. "Reasoning with neural tensor networks for knowledge base completion." In: *Advances in neural information processing systems*. Citeseer. 2013, pp. 926–934.

[56] Werner Uwents, Gabriele Monfardini, Hendrik Blockeel, Marco Gori, and Franco Scarselli. "Neural networks for relational learning: An experimental comparison." In: *Machine Learning* 82.3 (July 2011), pp. 315–349. ISSN: 08856125.

[57] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. "The graph neural network model." In: *IEEE transactions on neural networks* 20.1 (2008), pp. 61–80.

[58] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. "A comprehensive survey on graph neural networks." In: *IEEE transactions on neural networks and learning systems* (2020).

[59] Boris Weisfeiler and AA Lehman. "A reduction of a graph to a canonical form and an algebra arising during this reduction." In: *Nauchno-Technicheskaya Informatsia* 2.9 (1968), pp. 12–16.

[60] Richard Socher, Alex Perelygin, Jean Y Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, Christopher Potts, et al. "Recursive deep models for semantic compositionality over a sentiment treebank." In: *Proceedings of the conference on empirical methods in natural language processing (EMNLP)*. Vol. 1631. Citeseer. 2013, p. 1642.

[61] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. "How powerful are graph neural networks?" In: *arXiv preprint arXiv:1810.00826* (2018).

[62] Christopher Morris, Martin Ritzert, Matthias Fey, William L Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe. "Weisfeiler and leman go neural: Higher-order graph neural networks." In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33. 2019, pp. 4602–4609.

[63] A Kimmig, L Mihalkova, and L Getoor. "Lifted graphical models: a survey." In: *Machine Learning* 99.1 (2015), pp. 1–45.

[64] Luc De Raedt, Kristian Kersting, Sriraam Natarajan, and David Poole. *Statistical Relational Artificial Intelligence: Logic, Probability, and Computation*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2016.

[65] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks." In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[66] Stanley Kok and Pedro Domingos. "Statistical predicate invention." In: *Proceedings of the 24th international conference on Machine learning*. 2007, pp. 433–440.

[67] Geofrey G Towell, Jude W Shavlik, and Michiel O Noordewier. "Refinement of approximate domain theories by knowledge-based neural networks." In: *Proceedings of the eighth National conference on Artificial intelligence*. Boston, MA. 1990, pp. 861–866.

[68] Efthymia Tsamoura and Loizos Michael. "Neural-Symbolic Integration: A Compositional Perspective." In: *arXiv preprint arXiv:2010.11926* (2020).

[69] Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. "Deepproblog: Neural probabilistic logic programming." In: *Advances in Neural Information Processing Systems*. 2018, pp. 3749–3759.

[70] Sebastian Bader and Pascal Hitzler. "Dimensions of neural-symbolic integration-a structured survey." In: *arXiv preprint cs/0511042* (2005).

[71] Stephen H. Muggleton, Dianhuan Lin, and Alireza Tamaddoni-Nezhad. "Meta-interpretive learning of higher-order dyadic datalog: predicate invention revisited." In: *Machine Learning* 100.1 (2015), pp. 49–73.

[72] Ondřej Kuželka. "Fast construction of relational features for machine learning." Ph.D. thesis. Czech Technical University in Prague, 2013.

[73] Michael M Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. "Geometric deep learning: going beyond euclidean data." In: *IEEE Signal Processing Magazine* 34.4 (2017), pp. 18–42.

[74] Will Hamilton, Zhitao Ying, and Jure Leskovec. "Inductive representation learning on large graphs." In: *Advances in neural information processing systems*. 2017, pp. 1024–1034.

[75] Thomas N. Kipf and Max Welling. "Semi-Supervised Classification with Graph Convolutional Networks." In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[76] Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. "Representation learning on graphs with jumping knowledge networks." In: *arXiv preprint arXiv:1806.03536* (2018).

[77] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. "Neural message passing for quantum chemistry." In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org. 2017, pp. 1263–1272.

[78] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. "Graph neural networks: A review of methods and applications." In: *arXiv preprint arXiv:1812.08434* (2018).

[79] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. "Graph attention networks." In: *arXiv preprint arXiv:1710.10903* (2017).

[80] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. "Gated graph sequence neural networks." In: *arXiv preprint arXiv:1511.05493* (2015).

[81] Rudolf Kadlec, Ondrej Bajgar, and Jan Kleindienst. "Knowledge base completion: Baselines strike back." In: *arXiv preprint arXiv:1705.10744* (2017).

[82] Geoffrey E Hinton. "Deep belief networks." In: *Scholarpedia* 4.5 (2009), p. 5947.

[83] Ruslan Salakhutdinov, Andriy Mnih, and Geoffrey E. Hinton. "Restricted Boltzmann Machines for Collaborative Filtering." In: *Proceedings of the 24th International Conference on Machine Learning (2007)*. Vol. pp. New York, New York, USA: ACM Press, June 2007, pp. 791–798. ISBN: 978-1-59593-793-3.

[84] Ian J Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. "Generative adversarial networks." In: *arXiv preprint arXiv:1406.2661* (2014).

[85] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory." English. In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667.

[86] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. "Attention is All you Need." In: *Neural Information Processing Systems* (2017).

[87] Chaitanya Joshi. "Transformers are Graph Neural Networks." In: *The Gradient* (2020).

[88] John Haugeland. *Artificial intelligence: The very idea*. MIT press, 1989.

[89] Herve Gallaire, Jack Minker, and Jean-Marie Nicolas. "Logic and databases: A deductive approach." In: *Readings in Artificial Intelligence and Databases* (1989), pp. 231–247.

[90] Mirco Kuhlmann and Martin Gogolla. "From UML and OCL to relational logic and back." In: *International conference on model driven engineering languages and systems*. Springer. 2012, pp. 415–431.

[91] John W Lloyd. *Foundations of logic programming*. Springer Science & Business Media, 2012.

[92] Maarten H Van Emden and Robert A Kowalski. "The semantics of predicate logic as a programming language." In: *Journal of the ACM (JACM)* 23.4 (1976), pp. 733–742.

[93] Jeffrey D Unman. "Principles of database and knowledge-base systems." In: *Computer Science Press, New York* (1989).

[94] Ivan Bratko. *Prolog programming for artificial intelligence*. Pearson education, 2001.

[95] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. "Magic sets and other strange ways to implement logic programs." In: *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*. 1985, pp. 1–15.

[96] Vojtěch Aschenbrenner. "Deep Relational Learning with Predicate Invention." M.Sc. thesis. Czech Technical University in Prague, 2013.

[97] Gustav Šourek, Vojtech Aschenbrenner, Filip Železny, and Ondřej Kuželka. "Lifted relational neural networks." In: *Proceedings of the 2015th International Conference on Cognitive Computation: CoCo @ NIPS - Volume 1583*. 2015, pp. 52–60.

[98] Wei Chu Vikas Sindhwani, Zoubin Ghahramani, and S Sathiya Keerthi. "Relational learning with gaussian processes." In: *Advances in Neural Information Processing Systems 19: Proceedings of the 2006 Conference*. Vol. 19. MIT Press. 2007, p. 289.

[99] Nir Friedman, Lise Getoor, Daphne Koller, and Avi Pfeffer. "Learning probabilistic relational models." In: *Ijcai*. Vol. 99. 1999, pp. 1300–1309.

[100] Kristian Kersting. "An inductive logic programming approach to statistical relational learning." In: *AI Communications* 19.4 (2006), pp. 389–390.

[101] Stephen Muggleton. "Inverse entailment and Progol." In: *New generation computing* 13.3-4 (1995), pp. 245–286.

[102] Luc De Raedt. "Logical settings for concept-learning." In: *Artificial Intelligence* 95.1 (1997), pp. 187–201.

[103] Gordon D Plotkin. "A note on inductive generalization." In: *Machine intelligence* 5.1 (1970), pp. 153–163.

[104] Peter Flach. "Logical approaches to machine learning-an overview." In: *Think* 1.2 (1992), pp. 25–36.

[105] Rina Dechter, David Cohen, et al. *Constraint processing*. Morgan Kaufmann, 2003.

[106] Chandra Chekuri and Anand Rajaraman. "Conjunctive query containment revisited." In: *Theor. Comput. Sci.* 239.2 (2000), pp. 211–229.

[107] Jérôme Maloberti and Michéle Sebag. "Fast Theta-Subsumption with Constraint Satisfaction Algorithms." In: *Machine Learning* 55.2 (2004), pp. 137–174.

[108] Ondřej Kuželka and Filip Železný. "A restarted strategy for efficient subsumption testing." In: *Fundamenta Informaticae* 89.1 (2008), pp. 95–109.

[109] Willem-Jan Van Hoeve. "The alldifferent constraint: A survey." In: *arXiv preprint cs/0105015* (2001).

[110] Luc De Raedt and Kristian Kersting. "Statistical relational learning." In: *Encyclopedia of Machine Learning* (2010).

[111] Stephen Muggleton et al. "Stochastic logic programs." In: *Advances in inductive logic programming* 32 (1996), pp. 254–264.

[112] T Sato. "A statistical learning method for logic programs with distribution semantics." In: *Proc. of the 12th Intl. Conf. on Logic Programming (ICLP-95)*. 1995.

[113] David Poole. "The independent choice logic for modelling multiple agents under uncertainty." In: *Artificial intelligence* 94.1-2 (1997), pp. 7–56.

[114] Sriraam Natarajan, Tushar Khot, Kristian Kersting, Bernd Gutmann, and Jude W. Shavlik. "Gradient-based boosting for statistical relational learning: The relational dependency network case." In: *Machine Learning* 86.1 (2012), pp. 25–56.

[115] Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. "Inference and learning in probabilistic logic programs using weighted Boolean formulas." In: *Theory and Practice of Logic Programming* 15.3 (2015), pp. 358–401.

[116] Finn V Jensen. *An introduction to Bayesian networks*. Vol. 210. UCL press London, 1996.

[117] Kristian Kersting and Luc De Raedt. "Bayesian Logic Programming: Theory and Tool." In: *Statistical Relational Learning* (2007), p. 291.

[118] David Heckerman, David Maxwell Chickering, Christopher Meek, Robert Rounthwaite, and Carl Kadie. "Dependency networks for inference, collaborative filtering, and data visualization." In: *Journal of Machine Learning Research* 1.Oct (2000), pp. 49–75.

[119] Jennifer Neville and David Jensen. "Relational dependency networks." In: *Journal of Machine Learning Research* 8.3 (2007).

[120] Artur S. d'Avila Garcez, Gerson Zaverucha, and Luis A. V. De Carvalho. "Logical inference and inductive learning in artificial neural networks." In: *Knowledge Representation in Neural networks* (1997), pp. 33–46.

[121] Artur S. d'Avila Garcez and Gerson Zaverucha. "Connectionist inductive learning and logic programming system." In: *Applied Intelligence* 11.1 (1999), pp. 59–77. ISSN: 0924669x.

[122] A. Paccanaro and Geoffrey E. Hinton. "Learning distributed representations of concepts using Linear Relational Embedding." In: *IEEE Transactions on Knowledge and Data Engineering* 13.2 (2001), pp. 232–244. ISSN: 10414347.

[123] A. Paccanaro and Geoffrey E. Hinton. "Learning Distributed Representation of Concepts Using Linear Relational Embedding." In: *Knowledge and Data Engineering, IEEE Transactions* 13.2 (2001), pp. 1–13. ISSN: 10414347.

[124] Jordan B. Pollack. "Recursive distributed representations." In: *Artificial Intelligence* 46.1-2 (1990), pp. 77–105. ISSN: 00043702.

[125] John F. Kolen and Stefan C. Kremer. *A Field Guide to Dynamical Recurrent Networks*. Ieee, 2009, pp. 351–374. ISBN: 9780470544037.

[126] Tony A. Plate. "Holographic Reduced Representations." In: *IEEE Transactions on Neural Networks* 6.3 (1995), pp. 623–641. ISSN: 19410093.

[127] C M Bishop. "Neural networks for pattern recognition." In: *Journal of the American Statistical Association* 92 (1995), p. 482. ISSN: 01621459.

[128] Christoph Goller and Andreas Kuechler. "Learning task-dependent distributed representations by backpropagation through structure." In: *International Conference on Neural Networks* 1 (1996), pp. 347–352.

[129] Bishan Yang, Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. "Learning multi-relational semantics using neural-embedding models." In: *arXiv preprint arXiv:1411.4072* (2014).

[130] Antoine Bordes, Jason Weston, Ronan Collobert, and Yoshua Bengio. "Learning structured embeddings of knowledge bases." In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 25. 1. 2011.

[131] Antoine Bordes, Xavier Glorot, Jason Weston, and Yoshua Bengio. "A semantic matching energy function for learning with multi-relational data." In: *Machine Learning* 94.2 (2014), pp. 233–259.

[132] Hava T Siegelmann and Eduardo D Sontag. "Turing computability with neural nets." In: *Applied Mathematics Letters* 4.6 (1991), pp. 77–80.

[133] J Ramon and L De Raedt. "Multi instance neural networks." In: *Proceedings of the ICML Workshop on Attribute-Value and Relational Learning*. 2000.

[134] Werner Uwents and Hendrik Blockeel. "Classifying relational data with neural networks." In: *Inductive Logic Programming* (2005), pp. 384–396. ISSN: 03029743.

[135] Werner Uwents, Gabriele Monfardini, Hendrik Blockeel, Franco Scarselli, and Marco Gori. "Two connectionist models for graph processing: an experimental comparison on relational data." In: *MLG 2006, Proceedings on the International Workshop on Mining and Learning with Graphs*. 2006, pp. 211–220.

[136] Paolo Frasconi, Marco Gori, and Alessandro Sperduti. "A general framework for adaptive processing of data structures." In: *IEEE Transactions on Neural Networks* 9.5 (1998), pp. 768–786. ISSN: 10459227.

[137] Maximilian Nickel, Kevin Murphy, Volker Tresp, and Evgeniy Gabrilovich. "A Review of Relational Machine Learning for Knowledge Graph." In: *Proceedings of the IEEE* 104.28 (Mar. 2015), pp. 1–23. ISSN: 00189219.

[138] Jeff Clune, Kenneth O. Stanley, Robert T. Pennock, and Charles Ofria. "On the Performance of Indirect Encoding Across the Continuum of Regularity." In: *IEEE Trans. Evolutionary Computation* 15.3 (2011), pp. 346–367.

[139] Ken Ichi Funahashi. "On the approximate realization of continuous mappings by neural networks." In: *Neural Networks* 2.3 (Jan. 1989), pp. 183–192. ISSN: 08936080.

[140] Sebastian Bader. "Neural-Symbolic Integration." Ph.D. thesis. 2009.

[141] Sebastian Bader, Steffen; Holldobler, and Alexandre Scalzitti. "Semiring Artificial Neural Networks and Weighted Automata And an Application to Digital Image Encoding." In: *KI 2004: Advances in Artificial Intelligence* (2004). ISSN: 03029743.

[142] Steffen Hölldobler, Yvonne Kalinke, Fg Wissensverarbeitung Ki, et al. "Towards a new massively parallel computational model for logic programming." In: *In ECAI'94 workshop on Combining Symbolic and Connectioninst Processing* (1991).

[143] Gadi Pinkas. "Propositional Non-Monotonic Reasoning and Inconsistency in Symetric Neural Networks." In: *Proceedings of the 12th IJCAI* (1991), pp. 525–530.

[144] Steffen Hölldobler. "Automated Inferencing and Connectionist Models." Ph.D. thesis. Fakultät Informatik, Technische Hochschule Darmstadt, 1993.

[145] Artur S. d'Avila Garcez, Krysia Broda, and Dov M. Gabbay. "Symbolic knowledge extraction from trained neural networks: A sound approach." In: *Artificial Intelligence* 125.1-2 (Jan. 2001), pp. 155–207. ISSN: 00043702.

[146] Lokendra Shastri and Venkat Ajjanagadde. "From simple associations to systematic reasoning: A connectionist representation of rules, variables and dynamic bindings using temporal synchrony." In: *Behavioral and Brain Sciences* 16.03 (1993), p. 417. ISSN: 0140-525X.

[147] Steffen Hölldobler, Yvonne Kalinke, and J. Wunderlich. "A recursive neural network for reflexive reasoning." In: *Hybrid neural systems* D (2000), pp. 46–62. ISSN: 16113349.

[148] Sebastian Bader, Pascal Hitzler, and Steffen Hoelldobler. "The Integration of Connectionism and First-Order Knowledge Representation and Reasoning as a Challenge for Artificial Intelligence." In: *Network* (2004), p. 12.

[149] Carter Wendelken and Lokendra Shastri. "Multiple instantiation and rule mediation in SHRUTI." en. In: *Connection Science* 16.3 (Sept. 2004), pp. 211–217. ISSN: 0954-0091.

[150] Carter Wendelken and Lokendra Shastri. "Acquisition of concepts and causal rules in SHRUTI Causal Hebbian learning." In: *Proceedings of Cognitive Science* (2003), pp. 1224–1229.

[151] Trent E. Lange and Michael G. Dyer. "High-level Inferencing in a Connectionist Network." en. In: *Connection Science* 1.2 (Jan. 1989), pp. 181–217. ISSN: 0954-0091.

[152] Steffen Hölldobler and Franz Kurfeß. "CHCL—A connectionist inference system." In: *Parallelization in Inference Systems*. Springer, 1992, pp. 318–342.

[153] Artur S. d'Avila Garcez, Krysia Broda, and Dov M. Gabbay. *Neural-Symbolic Learning Systems: Foundations and Applications*. Springer-Verlag London, 2012. ISBN: 1447102118.

[154] Anthony K. Seda and Máire Lane. "On approximation in the integration of connectionist and logic-based systems." In: *the Third International Conference on Information (Information'04)* (2004), pp. 297–300.

[155] Steffen Hölldobler, Yvonne Kalinke, and Hans Peter Störr. "Approximating the semantics of logic programs by recurrent neural networks." In: *Applied Intelligence* 11.1 (1999), pp. 45–58. ISSN: 0924669x.

[156] Pascal Hitzler and Anthony Karel Seda. "A note on the relationships between logic programs and neural networks." In: *4th Irish Workshop on Formal Methods 4* (2000), pp. 1–9.

[157] Pascal Hitzler, Steffen Hölldobler, and Anthony Karel Seda. "Logic programs and connectionist networks." In: *Journal of Applied Logic* 2.3 (Sept. 2004), pp. 245–272. ISSN: 15708683.

[158] M Botta, A Giordana, and R Piola. "Refining numerical terms in horn clauses." In: *AI* IA 97: Advances in Artificial Intelligence* (1997). ISSN: 16113349.

[159] M Botta, A Giordana, and R Piola. "Refining First Theories with Neural Networks." In: *Methodologies for Intelligent Systems, Proc. of the 9th International Symposium, ISMIS-96* (1997), pp. 84–93.

[160] Sebastian Bader and Pascal Hitzler. "Logic programs, iterated function systems, and recurrent radial basis function networks." In: *Journal of Applied Logic* 2.3 (2004), pp. 273–300. ISSN: 15708683.

[161] Sebastian Bader, Artur S d'Avila Garcez, and Pascal Hitzler. "Computing First-Order Logic Programs by Fibring Artificial Neural Networks." In: (2005), pp. 314–319.

[162] A. Ciaramella, R. Tagliaferri, W. Pedrycz, and A. Di Nola. "Fuzzy relational neural network." In: *International Journal of Approximate Reasoning* 41.2 (Feb. 2006), pp. 146–163. ISSN: 0888613x.

[163] Alexandre B Tsybakov. *Introduction to nonparametric estimation*. Springer Science & Business Media, 2008.

[164] Antony Browne and Ron Sun. "Connectionist variable binding." In: *Expert Systems* 16.3 (1999), pp. 189–207. ISSN: 0266-4720.

[165] Ágnes Achs and Attila Kiss. "Fuzzy extension of Datalog." In: *Acta Cybernetica* 12 (1995), pp. 153–166.

[166] Carlos Viegas Damásio and Lúýs Moniz Pereira. "Antitonic logic programs." In: *Proceedings of the International Conference on Logic Programming and Non-Monotonic Reasoning*. 2001, pp. 379–393.

[167] Erich Peter Klement, Radko Mesiar, and Endre Pap. *Triangular norms*. Vol. 8. Springer Science & Business Media, 2013.

[168] Panos Rondogiannis and William W Wadge. "Minimum model semantics for logic programs with negation-as-failure." In: *ACM Transactions on Computational Logic (TOCL)* 6.2 (2005), pp. 441–467.

[169] Michael Luby, Alistair Sinclair, and David Zuckerman. "Optimal speedup of Las Vegas algorithms." In: *Information Processing Letters* 47.4 (1993), pp. 173–180.

[170] Jesse Davis, Vítor Santos Costa, Elizabeth Berg, David Page, Peggy L. Peissig, and Michael Caldwell. "Demand-Driven Clustering in Relational Domains for Predicting Adverse Drug Events." In: *Proceedings of the 29th International Conference on Machine Learning, ICML*. 2012.

[171] Gustav Šourek, Suresh Manandhar, Filip Železný, Steven Schockaert, and Ondřej Kuželka. "Learning predictive categories using lifted relational neural networks." In: *International Conference on Inductive Logic Programming*. Springer. 2016, pp. 108–119.

[172] Huma Lodhi and Stephen Muggleton. "Is mutagenesis still challenging." In: *ILP-Late-Breaking Papers* 35 (2005).

[173] Liva Ralaivola, Sanjay J Swamidass, Hiroto Saigo, and Pierre Baldi. "Graph kernels for chemical informatics." In: *Neural networks* 18.8 (2005), pp. 1093–1110.

[174] Christoph Helma, Ross D. King, Stefan Kramer, and Ashwin Srinivasan. "The predictive toxicology challenge 2000–2001." In: *Bioinformatics* 17.1 (2001), pp. 107–108.

[175] N. Landwehr, A. Passerini, L. De Raedt, and P. Frasconi. "kFOIL: learning simple relational kernels." In: *AAAI'06: Proceedings of the 21st national conference on Artificial intelligence*. AAAI Press, 2006, pp. 389–394.

[176] Niels Landwehr, Kristian Kersting, and Luc De Raedt. "Integrating naive bayes and foil." In: *The Journal of Machine Learning Research* 8 (2007), pp. 481–507.

[177] Tushar Khot, Sriraam Natarajan, Kristian Kersting, and Jude W. Shavlik. "Learning Markov Logic Networks via Functional Gradient Boosting." In: *11th IEEE International Conference on Data Mining, ICDM 2011*. 2011, pp. 320–329.

[178] Jerome H Friedman. "Greedy function approximation: a gradient boosting machine." In: *Annals of statistics* (2001), pp. 1189–1232.

[179] Ashwin Srinivasan. *The Aleph manual*. Computing Laboratory, Oxford University. 2000.

[180] Ondřej Kuželka, Andrea Szabóová, and Filip Železný. "Relational Learning with Polynomials." In: *IEEE 24th International Conference on Tools with Artificial Intelligence, ICTAI 2012*. 2012, pp. 1145–1150.

[181] Jesse Davis, Elizabeth S. Burnside, Inês de Castro Dutra, David Page, and Vítor Santos Costa. "An Integrated Approach to Learning Bayesian Networks of Rules." In: *Proceedings of the 16th European Conference on Machine Learning*. 2005, pp. 84–95.

[182] Stanley Kok and Pedro Domingos. "Learning the structure of Markov logic networks." In: *Proceedings of the 22nd International Conference on Machine Learning*. 2005, pp. 441–448.

[183] Quang-Thang Dinh, Matthieu Exbrayat, and Christel Vrain. "Generative structure learning for Markov logic networks based on graph of predicates." In: *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*. Vol. 22. 1. 2011, p. 1249.

[184] Scott E Fahlman and Christian Lebiere. "The Cascade-Correlation Learning Architecture." In: *Neural Information Processing Systems* (1989).

[185] David W Opitz and Jude W Shavlik. "Heuristically expanding knowledge-based neural networks." In: *IJCAI*. Citeseer. 1993, pp. 1360–1365.

[186] Stefano Bistarelli, Fabio Martinelli, and Francesco Santini. "Weighted datalog and levels of trust." In: *2008 Third International Conference on Availability, Reliability and Security*. Ieee. 2008, pp. 1128–1134.

[187] Jason Eisner and Nathaniel W Filardo. "Dyna: Extending datalog for modern AI." In: *International Datalog 2.0 Workshop*. Springer. 2010, pp. 181–220.

[188] Vijay Prakash Dwivedi, Chaitanya K Joshi, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. "Benchmarking graph neural networks." In: *arXiv preprint arXiv:2003.00982* (2020).

[189] Thomas Kipf, Ethan Fetaya, Kuan-Chieh Wang, Max Welling, and Richard Zemel. "Neural relational inference for interacting systems." In: *arXiv preprint arXiv:1802.04687* (2018).

[190] Liyu Gong and Qiang Cheng. "Exploiting edge features for graph neural networks." In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2019, pp. 9211–9219.

[191] Jongmin Kim, Taesup Kim, Sungwoong Kim, and Chang D Yoo. "Edge-labeling graph neural network for few-shot learning." In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2019, pp. 11–20.

[192] Zhengdao Chen, Xiang Li, and Joan Bruna. "Supervised community detection with line graph neural networks." In: *arXiv preprint arXiv:1705.08415* (2017).

[193] Jiwei Li and Dan Jurafsky. "Do multi-sense embeddings improve natural language understanding?" In: *arXiv preprint arXiv:1506.01070* (2015).

[194] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. "Modeling relational data with graph convolutional networks." In: *European Semantic Web Conference*. Springer. 2018, pp. 593–607.

[195] Xiao Wang, Houye Ji, Chuan Shi, Bai Wang, Yanfang Ye, Peng Cui, and Philip S Yu. "Heterogeneous graph attention network." In: *The World Wide Web Conference*. 2019, pp. 2022–2032.

[196] Shichao Zhu, Chuan Zhou, Shirui Pan, Xingquan Zhu, and Bin Wang. "Relation structure-aware heterogeneous graph neural network." In: *2019 IEEE International Conference on Data Mining (ICDM)*. Ieee. 2019, pp. 1534–1539.

[197] Ziqi Liu, Chaochao Chen, Xinxing Yang, Jun Zhou, Xiaolong Li, and Le Song. "Heterogeneous graph neural networks for malicious account detection." In: *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*. 2018, pp. 2077–2085.

[198] Yuxiao Dong, Nitesh V Chawla, and Ananthram Swami. "metapath2vec: Scalable representation learning for heterogeneous networks." In: *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*. 2017, pp. 135–144.

[199] Zhipeng Huang and Nikos Mamoulis. "Heterogeneous information network embedding for meta path based proximity." In: *arXiv preprint arXiv:1701.05291* (2017).

[200] Xin Dong, Evgeniy Gabrilovich, Geremy Heitz, Wilko Horn, Ni Lao, Kevin Murphy, Thomas Strohmann, Shaohua Sun, and Wei Zhang. "Knowledge vault: A web-scale approach to probabilistic knowledge fusion." In: *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2014, pp. 601–610.

[201] Yifan Feng, Haoxuan You, Zizhao Zhang, Rongrong Ji, and Yue Gao. "Hypergraph neural networks." In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33. 2019, pp. 3558–3565.

[202] Gustav Šourek, Vojtěch Aschenbrenner, Filip Železný, Steven Schockaert, and Ondřej Kuželka. "Lifted relational neural networks: Efficient learning of latent relational structures." In: *Journal of Artificial Intelligence Research* 62 (2018), pp. 69–100.

[203] Gustav Šourek, Ondrej Kuzelka, and Filip Železný. "Predicting top-k trends on twitter using graphlets and time features." In: *ILP 2013 Late Breaking Papers* (2013), p. 52.

[204] Matthias Fey and Jan Eric Lenssen. "Fast graph representation learning with PyTorch Geometric." In: *arXiv preprint arXiv:1903.02428* (2019).

[205] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, et al. "Deep graph library: Towards efficient and scalable deep learning on graphs." In: *arXiv preprint arXiv:1909.01315* (2019).

[206] George WA Milne, Marc C Nicklaus, John S Driscoll, Shaomeng Wang, and D Zaharevitz. "National Cancer Institute drug information system 3D database." In: *Journal of chemical information and computer sciences* 34.5 (1994), pp. 1219–1224.

[207] Marion Neumann, Roman Garnett, Christian Bauckhage, and Kristian Kersting. "Propagation kernels: efficient graph kernels from propagated information." In: *Machine Learning* 102.2 (2016), pp. 209–245.

[208] Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. "Learning convolutional neural networks for graphs." In: *International conference on machine learning*. 2016, pp. 2014–2023.

[209] Martin Simonovsky and Nikos Komodakis. "Dynamic edge-conditioned filters in convolutional neural networks on graphs." In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 3693–3702.

[210] L Tripos. "Tripos Mol2 file format." In: *St. Louis, MO: Tripos* (2007).

[211] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010, pp. 249–256.

[212] Goshu Nagino and Makoto Shozakai. "Distance measure between Gaussian distributions for discriminating speaking styles." In: *Ninth International Conference on Spoken Language Processing*. 2006.

[213] Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, et al. "Dynet: The dynamic neural network toolkit." In: *arXiv preprint arXiv:1701.03980* (2017).

[214] Gustav Šourek, Filip Železný, and Ondřej Kuželka. "Lossless Compression of Structured Convolutional Models via Lifting." In: *International Conference on Learning Representations* (2021).

[215] Dominic Masters and Carlo Luschi. "Revisiting small batch training for deep neural networks." In: *arXiv preprint arXiv:1804.07612* (2018).

[216] D Randall Wilson and Tony R Martinez. "The general inefficiency of batch training for gradient descent learning." In: *Neural networks* 16.10 (2003), pp. 1429–1451.

[217] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. "On large-batch training for deep learning: Generalization gap and sharp minima." In: *arXiv preprint arXiv:1609.04836* (2016).

[218] Kun Tu, Jian Li, Don Towsley, Dave Braines, and Liam D Turner. "gl2vec: Learning feature representation using graphlets for directed networks." In: *Proceedings of the 2019 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*. 2019, pp. 216–221.

[219] Aravind Sankar, Xinyang Zhang, and Kevin Chen-Chuan Chang. "Motif-based convolutional neural network on graphs." In: *arXiv preprint arXiv:1711.05697* (2017).

[220] Tim Rocktäschel and Sebastian Riedel. "Learning knowledge base inference with neural theorem provers." In: *Proceedings of the 5th Workshop on Automated Knowledge Base Construction*. 2016, pp. 45–50.

[221] William W Cohen. "Tensorlog: A differentiable deductive database." In: *arXiv preprint arXiv:1605.06523* (2016).

[222] Tim Rocktäschel and Sebastian Riedel. "End-to-end differentiable proving." In: *Advances in Neural Information Processing Systems*. 2017.

[223] Nuri Cingillioglu and Alessandra Russo. "Learning Invariants through Soft Unification." In: *Advances in Neural Information Processing Systems* 33 (2020).

[224] Pasquale Minervini, Matko Bosnjak, Tim Rocktäschel, and Sebastian Riedel. "Towards neural theorem proving at scale." In: *arXiv preprint arXiv:1807.08204* (2018).

[225] Leon Weber, Pasquale Minervini, Jannes Münchmeyer, Ulf Leser, and Tim Rocktäschel. "Nlprolog: Reasoning with weak unification for question answering in natural language." In: *arXiv preprint arXiv:1906.06187* (2019).

[226] Fan Yang, Zhilin Yang, and William W Cohen. "Differentiable learning of logical rules for knowledge base reasoning." In: *Advances in Neural Information Processing Systems*. 2017, pp. 2319–2328.

[227] Wenya Wang and Sinno Jialin Pan. "Integrating deep learning with logic fusion for information extraction." In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. 05. 2020, pp. 9225–9232.

[228] Tirtharaj Dash, Sharad Chitlangia, Aditya Ahuja, and Ashwin Srinivasan. "Incorporating Domain Knowledge into Deep Neural Networks." In: *arXiv preprint arXiv:2103.00180* (2021).

[229] Jingyi Xu, Zilu Zhang, Tal Friedman, Yitao Liang, and Guy Broeck. "A semantic loss function for deep learning with symbolic knowledge." In: *International Conference on Machine Learning*. Pmlr. 2018, pp. 5502–5511.

[230] Yaqi Xie, Ziwei Xu, Mohan S Kankanhalli, Kuldeep S Meel, and Harold Soh. "Embedding symbolic knowledge into deep networks." In: *arXiv preprint arXiv:1909.01161* (2019).

[231] Thomas Demeester, Tim Rocktäschel, and Sebastian Riedel. "Lifted Rule Injection for Relation Embeddings." In: *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, EMNLP 2016*. 2016, pp. 1389–1399.

[232] Andrew Cropper and Sebastijan Dumančić. "Inductive logic programming at 30: a new introduction." In: *arXiv preprint arXiv:2008.07912* (2020).

[233] Ashwin Srinivasan and Lovekesh Vig. "Mode-Directed Neural-Symbolic Modelling." In: *Submitted to: The 27th International Conference on Inductive Logic Programming (ILP2017)*. 2017, p. 44.

[234] Tirtharaj Dash, Ashwin Srinivasan, Ramprasad S Joshi, and A Baskar. "Discrete stochastic search and its application to feature-selection for deep relational machines." In: *International Conference on Artificial Neural Networks*. Springer. 2019, pp. 29–45.

[235] Navdeep Kaur, Gautam Kunapuli, Saket Joshi, Kristian Kersting, and Sriraam Natarajan. "Neural networks for relational data." In: *International Conference on Inductive Logic Programming*. Springer. 2019, pp. 62–71.

[236] Seyed Mehran Kazemi, David Buchman, Kristian Kersting, Sriraam Natarajan, and David Poole. "Relational Logistic Regression: The Directed Analog of Markov Logic Networks." In: *AAAI Workshop: Statistical Relational Artificial Intelligence*. Citeseer. 2014.

[237] Seyed Mehran Kazemi and David Poole. "RelNN: A deep neural model for relational learning." In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 32. 1. 2018.

[238] Sebastijan Dumancic and Hendrik Blockeel. "Clustering-based relational unsupervised representation learning with an explicit distributed representation." In: *arXiv preprint arXiv:1606.08658* (2016).

[239] Sebastijan Dumancic, Tias Guns, Wannes Meert, and Hendrik Blockeel. "Learning relational representations with auto-encoding logic programs." In: *arXiv preprint arXiv:1903.12577* (2019).

[240] Yizhou Sun, Jiawei Han, Xifeng Yan, Philip S Yu, and Tianyi Wu. "Pathsim: Meta path-based top-k similarity search in heterogeneous information networks." In: *Proceedings of the VLDB Endowment* 4.11 (2011), pp. 992–1003.

[241] Jingbo Shang, Meng Qu, Jialu Liu, Lance M Kaplan, Jiawei Han, and Jian Peng. "Meta-path guided embedding for similarity search in large-scale heterogeneous information networks." In: *arXiv preprint arXiv:1610.09769* (2016).

[242] Chuan Shi, Binbin Hu, Wayne Xin Zhao, and S Yu Philip. "Heterogeneous information network embedding for recommendation." In: *IEEE Transactions on Knowledge and Data Engineering* 31.2 (2018), pp. 357–370.

[243] Tao-yang Fu, Wang-Chien Lee, and Zhen Lei. "Hin2vec: Explore meta-paths in heterogeneous information networks for representation learning." In: *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. 2017, pp. 1797–1806.

[244] Zhipeng Huang, Yudian Zheng, Reynold Cheng, Yizhou Sun, Nikos Mamoulis, and Xiang Li. "Meta structure: Computing relevance in large heterogeneous information networks." In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2016, pp. 1595–1604.

[245] Lichao Sun, Lifang He, Zhipeng Huang, Bokai Cao, Congying Xia, Xiaokai Wei, and S Yu Philip. "Joint embedding of meta-path and meta-graph for heterogeneous information networks." In: *2018 IEEE International Conference on Big Knowledge (ICBK)*. Ieee. 2018, pp. 131–138.

[246] Mathias Niepert. "Discriminative gaifman models." In: *arXiv preprint arXiv:1610.09369* (2016).

[247] Haim Gaifman. "On Local and Non-Local Properties." In: *Proceedings of the Herbrand Symposium*. Ed. by J. Stern. Vol. 107. Studies in Logic and the Foundations of Mathematics Supplement C. Elsevier, 1982, pp. 105–135.

[248] Yuyu Zhang, Xinshi Chen, Yuan Yang, Arun Ramamurthy, Bo Li, Yuan Qi, and Le Song. "Efficient probabilistic logic reasoning with graph neural networks." In: *arXiv preprint arXiv:2001.11850* (2020).

[249] Meng Qu and Jian Tang. "Probabilistic logic neural networks for reasoning." In: *arXiv preprint arXiv:1906.08495* (2019).

[250] Pablo Barceló, Egor V Kostylev, Mikael Monet, Jorge Pérez, Juan Reutter, and Juan Pablo Silva. "The logical expressiveness of graph neural networks." In: *International Conference on Learning Representations*. 2019.

[251] Masataro Asai. "Unsupervised grounding of plannable first-order logic representation from images." In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 29. 2019, pp. 583–591.

[252] Sam Toyer, Sylvie Thiébaux, Felipe Trevizan, and Lexing Xie. "ASNets: Deep Learning for Generalised Planning." In: *Journal of Artificial Intelligence Research* 68 (2020), pp. 1–68.

[253] Prithviraj Sen, Marina Danilevsky, Yunyao Li, Siddhartha Brahma, Matthias Boehm, Laura Chiticariu, and Rajasekar Krishnamurthy. "Learning Explainable Linguistic Expressions with Neural Inductive Logic Programming for Sentence Classification." In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2020, pp. 4211–4221.

[254] Jiangjie Chen, Qiaoben Bao, Jiaze Chen, Changzhi Sun, Hao Zhou, Yanghua Xiao, and Lei Li. "LOREN: Logic Enhanced Neural Reasoning for Fact Verification." In: *arXiv preprint arXiv:2012.13577* (2020).

[255] Forough Arabshahi, Jennifer Lee, Mikayla Gawarecki, Kathryn Mazaitis, Amos Azaria, and Tom Mitchell. "Conversational neuro-symbolic commonsense reasoning." In: *arXiv preprint arXiv:2006.10022* (2020).

[256] Ryan Riegel, Alexander Gray, Francois Luus, Naweed Khan, Ndivhuwo Makondo, Ismail Yunus Akhalwaya, Haifeng Qian, Ronald Fagin, Francisco Barahona, Udit Sharma, et al. "Logical neural networks." In: *arXiv preprint arXiv:2006.13155* (2020).

[257] Eelco Visser. "Meta-programming with concrete object syntax." In: *International Conference on Generative Programming and Component Engineering*. Springer. 2002, pp. 299–315.

[258] Patricia Hill and John Gallagher. "Meta-programming in logic programming." In: *Handbook of Logic in Artificial Intelligence and Logic Programming* 5 (1998), pp. 421–497.

[259] Arnaud Nguembang Fadja, Evelina Lamma, and Fabrizio Riguzzi. "Deep Probabilistic Logic Programming." In: *Probabilistic logic programming workshop at International Conference on Inductive Logic Programming*. 2017, pp. 3–14.

[260] Francesco Orsini, Paolo Frasconi, and Luc De Raedt. "kProbLog: an algebraic Prolog for machine learning." In: *Machine Learning* 106.12 (2017), pp. 1933–1969.

[261] Zhun Yang, Adam Ishay, and Joohyung Lee. "Neurasp: Embracing neural networks into answer set programming." In: *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI*. 2020, pp. 1755–1762.

[262] Giuseppe Marra and Ondřej Kuželka. "Neural Markov Logic Networks." In: *arXiv preprint arXiv:1905.13462* (2019).

[263] Andres Campero, Aldo Pareja, Tim Klinger, Josh Tenenbaum, and Sebastian Riedel. "Logical rule induction and theory learning using neural theorem proving." In: *arXiv preprint arXiv:1809.02193* (2018).

[264] Mukund Raghothaman, Xujie Si, Kihong Heo, and Mayur Naik. "Difflog: Learning Datalog Programs by Continuous Optimization." In: *arXiv preprint arXiv:1906.00163* (2019).

[265] Yuan Yang and Le Song. "Learn to Explain Efficiently via Neural Logic Inductive Learning." In: *arXiv preprint arXiv:1910.02481* (2019).

[266] Tim Rocktäschel, Sameer Singh, and Sebastian Riedel. "Injecting Logical Background Knowledge into Embeddings for Relation Extraction." In: *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* (2015), pp. 1119–1129.

[267] Michelangelo Diligenti, Marco Gori, and Claudio Sacca. "Semantic-based regularization for learning and inference." In: *Artificial Intelligence* 244 (2017), pp. 143–165.

[268] Emile van Krieken, Erman Acar, and Frank van Harmelen. "Analyzing differentiable fuzzy logic operators." In: *arXiv preprint arXiv:2002.06100* (2020).

[269] Giuseppe Marra, Francesco Giannini, Michelangelo Diligenti, and Marco Gori. "Lyrics: a general interface layer to integrate AI and deep learning." In: *arXiv preprint arXiv:1903.07534* (2019).

[270] Giuseppe Marra, Francesco Giannini, Michelangelo Diligenti, and Marco Gori. "Integrating learning and reasoning with deep logic models." In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer. 2019, pp. 517–532.

[271] Daphne Koller, Nir Friedman, Sašo Džeroski, Charles Sutton, Andrew McCallum, Avi Pfeffer, Pieter Abbeel, Ming-Fai Wong, David Heckerman, Chris Meek, et al. *Introduction to statistical relational learning*. MIT press, 2007.

[272] Quan Wang, Zhendong Mao, Bin Wang, and Li Guo. "Knowledge graph embedding: A survey of approaches and applications." In: *IEEE Transactions on Knowledge and Data Engineering* 29.12 (2017), pp. 2724–2743.

[273] Kristian Kersting. "Lifted Probabilistic Inference." In: *European Conference on Artificial Intelligence*. 2012, pp. 33–38.

[274] Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch. "Probabilistic databases." In: *Synthesis lectures on data management* 3.2 (2011), pp. 1–180.

[275] Prithviraj Sen, Amol Deshpande, and Lise Getoor. "Bisimulation-based approximate lifted inference." In: *arXiv preprint arXiv:1205.2616* (2012).

[276] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. "A survey of model compression and acceleration for deep neural networks." In: *arXiv preprint arXiv:1710.09282* (2017).

[277] Richard A DeMillo and Richard J Lipton. *A Probabilistic Remark on Algebraic Program Testing.* Tech. rep. Georgia Inst. of Technology, School of Information and Computer science, 1977.

[278] Sergei Ivanov, Sergei Sviridov, and Evgeny Burnaev. "Understanding Isomorphism Bias in Graph Data Sets." In: *arXiv preprint arXiv:1910.12091* (2019).

[279] Siegfried Nijssen and Joost N. Kok. "Efficient Frequent Query Discovery in FARMER." In: *Knowledge Discovery in Databases: PKDD'03, 7th European Conference on Principles and Practice of Knowledge Discovery in Databases*. 2003, pp. 350–362.

[280] Stefano Ferilli, Nicola Fanizzi, Nicola Di Mauro, and Teresa MA Basile. "Efficient θ-subsumption under object identity." In: *AI*IA Workshop, 2002)*. 2002, pp. 59–68.

[281] Robert E Stepp and Ryszard S Michalski. "Conceptual clustering: Inventing goal-oriented classifications of structured objects." In: *Machine learning: An artificial intelligence approach* 2 (1986), pp. 471–498.

[282] Monty Newborn. *Automated theorem proving - theory and practice*. Springer, 2001. ISBN: 978-0-387-95075-4.

[283] Sebastian Riedel. "Improving the Accuracy and Efficiency of MAP Inference for Markov Logic." In: *UAI 2008, 24th Conference on Uncertainty in Artificial Intelligence*. 2008, pp. 468–475.

[284] Alireza Tamaddoni-Nezhad and Stephen Muggleton. "The lattice structure and refinement operators for the hypothesis space bounded by a bottom clause." In: *Machine Learning* 76.1 (2009), pp. 37–72.

[285] Luc Dehaspe and Luc De Raedt. "Mining Association Rules in Multiple Relations." In: *Inductive Logic Programming, 7th International Workshop, ILP-97*. 1997, pp. 125–132.

[286] Ondřej Kuželka and Filip Železný. "Block-wise construction of tree-like relational features with monotone reducibility and redundancy." In: *Machine Learning* 83.2 (2011), pp. 163–192.

[287] Jan Ramon, Samrat Roy, and Daenen Jonny. "Efficient homomorphism-free enumeration of conjunctive queries." In: *Preliminary Papers ILP 2011*. 2011, p. 6.

[288] D. Le Berre and A. Parrain. "The SAT4J library, release 2.2." In: *Journal on Satisfiability, Boolean Modeling and Computation* 7 (2010), pp. 50–64.

[289] Wray L. Buntine. "Generalized Subsumption and Its Applications to Induction and Redundancy." In: *Artif. Intell.* 36.2 (1988), pp. 149–176.

[290] Donato Malerba. "Learning recursive theories in the normal ilp setting." In: *Fundamenta Informaticae* 57.1 (2003), pp. 39–77.

[291] Daniel N Osherson, Joshua Stern, Ormond Wilkie, Michael Stob, and Edward E Smith. "Default probability." In: *Cognitive Science* 15.2 (1991), pp. 251–269.

[292] Rudolph J Rummel. *The dimensionality of nations project: attributes of nations and behavior of nations dyads, 1950-1965*. 5409. Inter-University Consortium for Political Research, 1976.

[293] Alexa T McCray. "An upper-level ontology for the biomedical domain." In: *Comparative and Functional Genomics* 4.1 (2003), pp. 80–84.

[294] Tom M. Mitchell et al. "Never-Ending Learning." In: *Proceedings of the 29th AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA.* 2015, pp. 2302–2310.

[295] Laurent Miclet, Sabri Bayoudh, and Arnaud Delhay. "Analogical dissimilarity: definition, algorithms and two experiments in machine learning." In: *Journal of Artificial Intelligence Research* 32 (2008), pp. 793–824.

[296] Henri Prade and Gilles Richard. "Reasoning with logical proportions." In: *Twelfth International Conference on the Principles of Knowledge Representation and Reasoning*. 2010.

[297] Islam Beltagy, Cuong Chau, Gemma Boleda, Dan Garrette, Katrin Erk, and Raymond Mooney. "Montague meets Markov: Deep semantics with probabilistic logical form." In: *Proc. *SEM*. 2013, pp. 11–21.

[298] Bernhard Ganter, Gerd Stumme, and Rudolf Wille. *Formal concept analysis: foundations and applications*. Vol. 3626. springer, 2005.

[299] Christophe Ley, Tom Van de Wiele, and Hans Van Eetvelde. "Ranking soccer teams on basis of their current strength: a comparison of maximum likelihood approaches." In: *arXiv preprint arXiv:1705.09575* (2017).

[300] Werner Dubitzky, Philippe Lopes, Jesse Davis, and Daniel Berrar. *The open international soccer database for machine learning*. 2019.

[301] John Goddard. "Regression models for forecasting goals and match results in association football." In: *International Journal of Forecasting* 21.2 (2005), pp. 331–340.

[302] Ian McHale and Phil Scarf. "Modelling soccer matches using bivariate discrete distributions with general dependence structure." In: *Statistica Neerlandica* 61.4 (2007), pp. 432–445.

[303] Anthony Costa Constantinou and Norman Elliott Fenton. "Determining the level of ability of football teams by dynamic ratings based on the relative discrepancies in scores between adversaries." In: *Journal of Quantitative Analysis in Sports* 9.1 (2013), pp. 37–50.

[304] Lars Magnus Hvattum and Halvard Arntzen. "Using ELO ratings for match result prediction in association football." In: *International Journal of Forecasting* 26.3 (2010), pp. 460–470.

[305] Jan Van Haaren and Guy Van den Broeck. "Relational learning for football-related predictions." In: *Latest Advances in Inductive Logic Programming*. World Scientific, 2015, pp. 237–244.

[306] Ondřej Hubáček, Gustav Šourek, and Filip Železný. "Learning to predict soccer results from relational data with gradient boosted trees." In: *Machine Learning* (2018).

[307] Verica Lazova and Lasko Basnarkov. "PageRank Approach to Ranking National Football Teams." In: *arXiv preprint arXiv:1503.01331* (2015).

[308] Tianqi Chen and Carlos Guestrin. "Xgboost: A scalable tree boosting system." In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. Acm. 2016, pp. 785–794.

[309] Edward S Epstein. "A scoring system for probability forecasts of ranked categories." In: *Journal of Applied Meteorology* 8.6 (1969), pp. 985–987.

[310] Nitin Saxena. "Progress on Polynomial Identity Testing." In: *Bulletin of the EATCS* 99 (2009), pp. 49–79.

[311] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. "Collective classification in network data." In: *AI magazine* 29.3 (2008), pp. 93–93.

[312] Daniel Krynský. "Relational Learning with Neural Networks for Machine Translation Evaluation." M.Sc. thesis. Czech Technical University in Prague, 2018.

[313] Jan Studený. "Learning Relevant Reasoning Patterns with Neuro-Logic Programming." B.Sc. thesis. Czech Technical University in Prague, 2017.

[314] Marián Briedoň. "Integration of Relational and Deep Learning Frameworks." M.Sc. thesis. Czech Technical University in Prague, 2018.

PUBLICATIONS OF THE AUTHOR

List of publications presented for the purpose of dissertation defense.
Citations from Web of Science, Scopus, and Google Scholar listed as of April 2nd, 2021.

C.4 PUBLICATIONS RELATED THE TOPIC OF THIS THESIS

C.4.1 *Journal papers*

Gustav Šourek, Vojtěch Aschenbrenner, Filip Železný, Steven Schockaert, and Ondřej Kuželka. "Lifted relational neural networks: Efficient learning of latent relational structures." In: *Journal of Artificial Intelligence Research (JAIR)* 62 (2018), pp. 69–100
WoS: 4, Scopus: 13, Google: 36
Journal IF: 2.4, SJR: 1.0

C.4.2 *Conference papers*

Gustav Šourek, Filip Železný, and Ondřej Kuželka. "Lossless Compression of Structured Convolutional Models via Lifting." In: *International Conference on Learning Representations*. ICLR OpenReview. 2021
WoS: ∅, Scopus: ∅, Google: 2

Gustav Šourek, Martin Svatoš, Filip Železný, Steven Schockaert, and Ondřej Kuželka. "Stacked structure learning for lifted relational neural networks." In: *International Conference on Inductive Logic Programming*. Springer. 2017, 140–151, ⟵ **Best Paper Award**
WoS: 0, Scopus: 2, Google: 6

Gustav Šourek, Suresh Manandhar, Filip Železný, Steven Schockaert, and Ondřej Kuželka. "Learning predictive categories using lifted relational neural networks." In: *International Conference on Inductive Logic Programming*. Springer. 2016, pp. 108–119
WoS: ∅, Scopus: 7, Google: 10

Gustav Šourek, Vojtěch Aschenbrenner, Filip Železný, and Ondřej Kuželka. "Lifted relational neural networks." In: *Proceedings of the Workshop on Cognitive Computation: Integrating Neural and Symbolic Approaches co-located with NeurIPS*. CEUR Workshop Proceedings. 2015
WoS: ∅, Scopus: 4, Google: 57

Gustav Šourek, Filip Železný, and Ondřej Kuželka. "Learning with Molecules beyond Graph Neural Networks." In: *Machine Learning for Molecules worshop at NeurIPS* (2020), paper 24

Martin Svatoš, **Gustav Šourek**, Filip Železný, Steven Schockaert, and Ondřej Kuželka. "Pruning hypothesis spaces using learned domain theories." In: *International Conference on Inductive Logic Programming*. Springer. 2017, pp. 152–168
WoS: 0, Scopus: 2, Google: 1

Gustav Šourek. "Deep learning with relational logic representations." In: *Proceedings of the 28th International Joint Conference on Artificial Intelligence*. AAAI Press. 2019

Ondřej Hubáček, **Gustav Šourek**, and Filip Železný. "Lifted Relational Team Embeddings for Predictive Sports Analytics." In: *ILP Up-and-Coming / Short Papers*. 2018, pp. 84–91

C.4.3 *Others*

Gustav Šourek, Filip Železný, and Ondřej Kuželka. "Beyond Graph Neural Networks with Lifted Relational Neural Networks." In: *arXiv preprint arXiv:2007.06286* (2020)
- currently under journal review

C.5 OTHER PUBLICATIONS OF THE AUTHOR

C.5.1 *Journal papers*

Gustav Šourek and Filip Železný. "Efficient extraction of network event types from NetFlows." In: *Security and Communication Networks* 2019 (2019)
WoS: 1, Scopus: 1, Google: 2
Journal IF: 1.3, SJR: 0.5

Gustav Šourek and Petr Pošík. "Dynamic system modeling of evolutionary algorithms." In: *ACM SIGAPP Applied Computing Review* 15.4 (2016), pp. 19–30

Ondřej Hubáček, **Gustav Šourek**, and Filip Železný. "Learning to predict soccer results from relational data with gradient boosted trees." In: *Machine Learning* 108.1 (2019), pp. 29–47
WoS: 8, Scopus: 13, Google: 22
Journal IF: 2.7, SJR: 1.0

Ondřej Hubáček, **Gustav Šourek**, and Filip Železný. "Exploiting sports-betting market using machine learning." In: *International Journal of Forecasting* 35.2 (2019), pp. 783–796
WoS: 3, Scopus: 4, Google: 12
Journal IF: 2.8, SJR: 1.8

C.5.2 *Conference papers*

Gustav Šourek, Ondřej Kuželka, and Filip Železný. "Learning to detect network intrusion from a few labeled events and background traffic." In: *IFIP International Conference on Autonomous Infrastructure, Management and Security*. Springer. 2015, pp. 73–86
WoS: 3, Scopus: 3, Google: 3

Gustav Šourek and Petr Pošík. "Visual data-flow framework of evolutionary computation." In: *Proceedings of the 2015 Conference on research in adaptive and convergent systems*. 2015, pp. 343–348

Gustav Šourek, Ondřej Kuželka, and Filip Železný. "Predicting top-k trends on twitter using graphlets and time features." In: *International Conference on Inductive Logic Programming 2013, Late Breaking Papers*. CEUR Workshop Proceedings. 2013
WoS: ∅, Scopus: 1, Google: 2

Ondřej Hubáček, **Gustav Šourek**, and Filip Železný. "Deep learning from spatial relations for soccer pass prediction." In: *International Workshop on Machine Learning and Data Mining for Sports Analytics, ECML-PKDD*. Springer. 2018, pp. 159–166
WoS: ∅, Scopus: 1, Google: 5

Matej Uhrín, **Gustav Šourek**, Ondřej Hubáček, and Filip Železný. "Sports betting strategies: an experimental review." In: *MathSport International*. 2019

Ondřej Hubáček, **Gustav Šourek**, and Filip Železný. "Score-based soccer match outcome modeling–an experimental review." In: *MathSport International*. 2019
WoS: ∅, Scopus: ∅, Google: 4

C.5.3 *Patents*

Gustav Šourek, Karel Bartoš, Filip Železný, Tomas Pevný, and Petr Somol. "Events from network flows." In: *United States Patent and Trademark Office*. Pat. no. 20160112442. 2014
Google: 3