

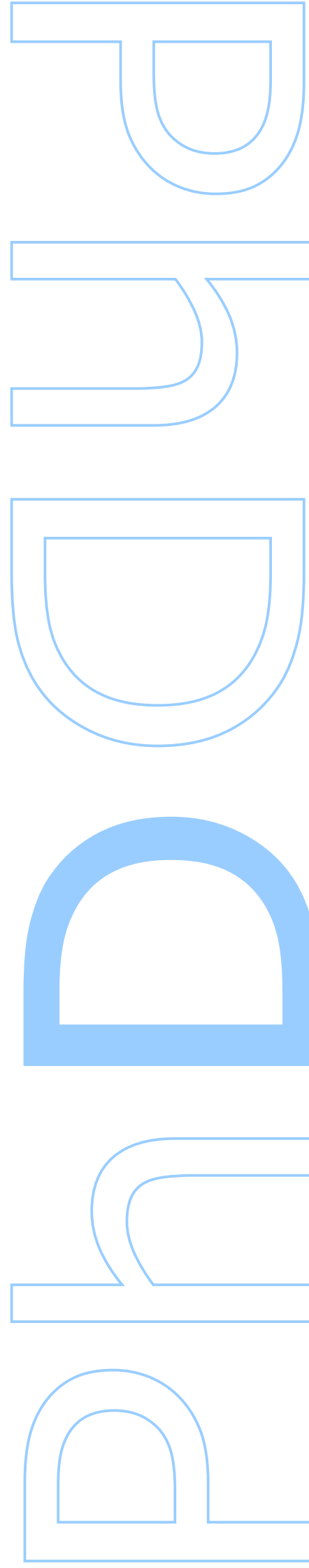
NeuralLog: A Neural Logic System for Parameter and Structure Learning

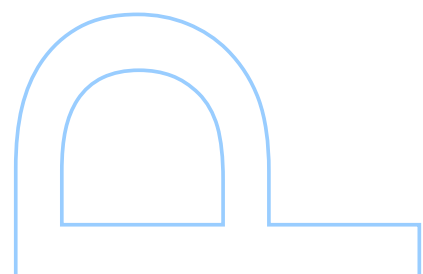
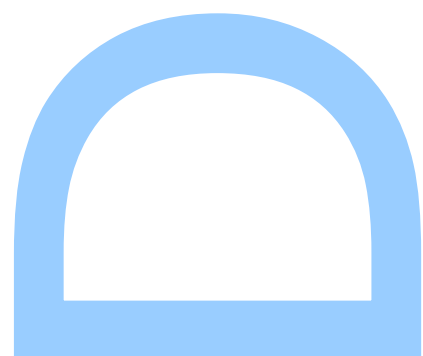
Victor Augusto Lopes Guimarães

PhD in Computer Science
Department of Computer Science
2022

Supervisor

Vitor Manuel de Moraes Santos Costa, Associated Professor, Faculty of Sciences



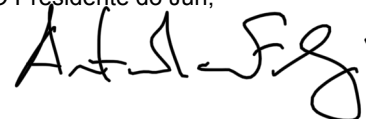


U. PORTO

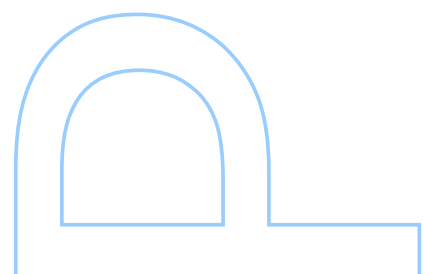
FC FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas
pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,



Porto, 29 / 8 / 2022



Sworn Statement

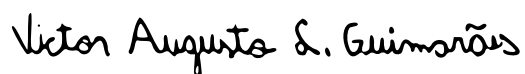
I, Victor Augusto Lopes Guimarães, born in Brazil, resident in Portugal, of Portuguese nationality, bearer of Identification Card No. 31710677, enrolled in the Doctor's Degree in Computer Science at the Faculty of Sciences of the University of Porto hereby declare, in accordance with the provisions of paragraph a) of Article 14 of the Code of Ethical Conduct of the University of Porto, that the content of this thesis reflects perspectives, research work and my own interpretations at the time of its submission.

By submitting this thesis, I also declare that it contains the results of my own research work and contributions that have not been previously submitted to this or any other institution.

I further declare that all references to other authors fully comply with the rules of attribution and are referenced in the text by citation and identified in the bibliographic references section. This thesis does not include any content whose reproduction is protected by copyright laws.

I am aware that the practice of plagiarism and self-plagiarism constitute a form of academic offense.

Victor Augusto Lopes Guimarães



05/09/2022

Agradecimentos

Agradeço primeiramente aos meus pais, Augusto e Lourdes, por todo o suporte e apoio, além de todo o investimento feito no meu desenvolvimento.

Agradeço a Livia, que acompanhou de perto todas as conquistas e dificuldades na realização deste trabalho, e que me apoiou, para que os objetivos tenham sido atingidos.

Agradeço a todos os professores que participaram da minha formação, cada um deles contribuindo com um pouco do seu conhecimento, para que eu pudesse chegar onde estou hoje. Em especial ao Professor Vítor Santos Costas, que me orientou durante este trabalho, sem o qual, este trabalho não seria o mesmo. Agradeço em especial também ao Professor Gerson Zaverucha e a Professora Aline Paes, por anos de orientação durante a minha vida académica.

Agradeço aos Professores Alípio Mário Guedes Jorge, Artur d'Avila Garcez, Ashwin Srinivasan, Pedro Manuel Pinto Ribeiro e António Mário da Silva Marcos Florido por aceitarem participar do júri desta tese, cujos comentários contribuíram para a melhoria deste trabalho.

A equipa administrativa e de manutenção da Universidade do Porto, da Faculdade de Ciências da Universidade do Porto e do INESC-TEC, pelo acolhimento e suporte necessário para a conclusão do programa de doutoramento.

Aos meus familiares e amigos, pelo suporte e compreensão durante esta jornada.

A toda comunidade científica, principalmente aqueles cujos trabalhos estão relacionados a esta tese. Espero que a mesma também contribua positivamente para o avanço da fronteira do conhecimento.

Por fim, mas não menos importante, à agência portuguesa de apoio à investigação em ciência, tecnologia e inovação, FCT - Fundação para a Ciência e a Tecnologia, pelo suporte financeiro prestado, através da bolsa de doutoramento de número 2020.05718.BD; e ao INESC-TEC por ser a instituição de acolhimento.

Abstract

Application domains that require considering relationships among objects which have real-valued attributes are becoming even more important. While neural networks have been remarkably successful on a wide range of tasks, most of the neural network models focus on propositional tasks, where there are no relationships among the examples.

In this thesis, we propose NeuralLog, a first-order logic language that is compiled to a neural network. The main goal of NeuralLog is to provide a language to define neural networks, based on logic, in order to bridge logic programming and deep learning, allowing advances in both fields to be combined to obtain better machine learning models. The main advantages of NeuralLog are to allow neural networks to be described as logic programs; and to be able to handle numeric attributes and functions.

In our experiments, we show that NeuralLog is capable of representing neural networks for relational tasks, as well as Multilayer Perceptrons, and even call neural network models as logic predicates. Since the neural network structure is described as logic, in NeuralLog, we applied structure learning algorithms from Inductive Logic Programming in order to find the structure of neural network models from data, both in batch and in online learning environments.

We compared NeuralLog, for parameter learning, with two distinct systems that use first-order logic to build neural network models. These experiments show that NeuralLog can learn link prediction and classification tasks, using the same theories as the compared systems, and achieving better results for both the area under the ROC curve and the average precision in four datasets: Cora and UWCSE, for link predictions; Yelp and PAKDD15, for classification; and comparable results for link prediction in the WordNet dataset.

In addition, we implemented a Multilayer Perceptron, fully in the NeuralLog language, and applied it to the traditional propositional task, in the well-known Iris dataset. Also, we show how to use a state-of-the-art neural network model for the Named Entity Recognition task, using it as a logic predicate in NeuralLog.

When learning the structure of NeuralLog programs on batch environments, we used a

Meta-Interpretive Learning (MIL) approach, NeuralLog+MIL. NeuralLog+MIL obtained competitive results for the link prediction task when compared with Neural-LP in three different datasets. Achieving comparable values for the hit at top 10 entities and the mean rank metric in the UMLS and WordNet datasets; and outperforming Neural-LP in the UWCSE dataset in those metrics.

On the online learning environment, we applied techniques of theory revision from examples, which starts from a, possibly empty, logic theory and changes it to cope with new examples. We ported the Online Structure Learner by Revision (OSLR) algorithm to NeuralLog, which we called NeuralLog+OSLR. Furthermore, we implemented a theory revision mechanism combining the mechanism from OSLR with the search strategy from MIL, which we called NeuralLog+OMIL. To the best of our knowledge, it is the first time a MIL system is used for online learning.

We compared NeuralLog+OSLR and NeuralLog+OMIL with the original OSLR approach and RDN-Boost, a system that learns Relational Dependency Networks (RDNs) models. Our experiments showed that NeuralLog+OMIL outperformed OSLR and RDN-Boost, for link prediction, on three out of the four target relations from the Cora dataset, and in the UMLS dataset. While both systems, NeuralLog+OSLR and NeuralLog+OMIL, outperformed OSLR and RDN-Boost on the UWCSE, assuming a good initial theory is provided.

Keywords Neural-Symbolic Integration, Inductive Logic Programming, Neural Network, Deep Learning, Relational Learning, Meta-Interpretive Learning, Online Learning and Theory Revision from Examples

Resumo

Domínios de aplicações que requerem considerar relações entre objetos, que possuem atributos de valores reais, estão se tornando cada vez mais importantes. Enquanto redes neuronais têm obtido sucessos notáveis num amplo espectro de tarefas, a maioria desses modelos de redes neuronais focam em tarefas proposicionais, onde não existem relações entre os exemplos.

Nesta tese, nós propomos NeuralLog, uma linguagem lógica de primeira ordem que é compilada para uma rede neuronal. O principal objetivo de NeuralLog é prover uma linguagem, baseada em lógica, que define uma rede neuronal, de modo a conectar programação lógica com aprendizado profundo, permitindo que avanços em ambos os campos possam ser combinados para obtenção de melhores modelos de aprendizado. As principais vantagens de NeuralLog são permitir que redes neuronais sejam descritas como programas lógicos e ser capaz de lidar com atributos numéricos e funções.

Nas nossas experiências, nós mostramos que NeuralLog é capaz de representar redes neuronais para tarefas relacionais, assim como Perceptron multicamadas, e até chamar modelos de redes neuronais a partir de predicados lógicos. Dado que estruturas de redes neuronais são descritas como programas lógicos, em NeuralLog, nós aplicamos algoritmos de Programação em Lógica Indutiva (ILP) de modo a encontrar estruturas de modelos de redes neuronais a partir de dados, tanto em ambientes de aprendizados por batelada, quanto em ambientes de aprendizado *online*.

Nós comparamos NeuralLog, para aprendizado de parâmetros, com dois sistemas distintos que usam lógica de primeira ordem para construir modelos de redes neuronais. Estas experiências mostram que NeuralLog pode aprender tarefas de predição de ligação e classificação, usando as mesmas teorias que os sistemas de comparação, e obtendo resultados melhores, tanto para a área sob a curva ROC, quanto para a precisão média, em quatro conjuntos de dados: Cora e UWCSE, para predição de ligação; Yelp e PAKDD15, para classificação; e obtendo resultados comparáveis para predição de ligação no conjunto de dados do WordNet.

No mais, nós implementamos um Perceptron multicamadas, completamente na linguagem

NeuralLog, e aplicamos ele a uma tarefa proposicional tradicional, no conhecido conjunto de dados do Iris. Também mostramos como usar um modelo de rede neuronal, do estado da arte, como um predicado lógico, para a tarefa de Reconhecimento de Entidade Mencionada (NER).

Quanto a aprender a estrutura de programas lógicos, em NeuralLog, em ambiente de batelada, nós usamos uma abordagem de Aprendizado Meta-Interpretativo (MIL), NeuralLog+MIL. NeuralLog+MIL obteve resultados competitivos para as tarefas de predição de ligação, quando comparados com Neural-LP, em três conjuntos de dados diferentes. Obtendo valores comparáveis para o acerto entre as 10 entidades do topo do *ranking* e para a métrica do *ranking* médio no conjunto de dados do UMLS e do WordNet; e obteve um melhor desempenho que Neural-LP, nestas métricas, no conjunto de dados do UWCSE.

No ambiente de aprendizado *online*, nós aplicamos técnicas de revisão de teoria a partir de exemplos, o que começa a partir de uma teoria lógica, possivelmente vazia, e a modifica para lidar com novos exemplos. Nós portamos o algoritmo do *Online Structure Learner by Revision (OSLR)* para NeuralLog, o que nós chamamos de NeuralLog+OSLR. Não o bastante, nós implementamos um mecanismo de revisão de teoria combinando o mecanismo do OSLR com a estratégia de busca do MIL, o que nós chamamos de NeuralLog+OMIL. No melhor de nosso conhecimento, esta é a primeira vez que um sistema MIL é usado para aprendizado *online*.

Nós comparamos NeuralLog+OSLR e NeuralLog+OMIL com a abordagem original do OSLR e com o RDN-Boost, um sistema que aprende modelos de Redes de Dependência Relacional (RDNs). As nossas experiências mostraram que NeuralLog+OMIL obteve melhor desempenho do que OSLR e RDN-Boost, para predicação de ligação, em três das quatro relações alvo do conjunto de dados do Cora, e para o conjunto de dados do UMLS. Enquanto ambos os sistemas, NeuralLog+OSLR e NeuralLog+OMIL, desempenharam melhor do que OSLR e RDN-Boost no conjunto de dados do UWCSE, assumindo que uma boa teoria inicial é fornecida.

Palavras-chave Integração Neuro-Simbólica, Programação Lógica Indutiva, Rede Neuronal, Aprendizado Profundo, Aprendizado Relacional, Aprendizado Meta-Interpretativo, Aprendizado *Online* e Revisão de Teoria a partir de Exemplos

Contents

List of Tables	xiii
List of Figures	xv
List of Algorithms	xvii
List of Acronyms	xix
1 Introduction	1
2 Background Knowledge	9
2.1 Logic Fundamentals	9
2.1.1 SLD-Resolution	11
2.1.2 Inductive Logic Programming	12
2.1.2.1 Meta-Interpretive Learning	13
2.1.2.2 Theory Revision from Examples	14
2.2 Neural Networks Fundamentals	15
2.3 Online vs Offline Learning	17
2.4 Related Work	18
3 The NeuralLog System	23
3.1 NeuralLog: a Bridge from Logic Programming to Neural Networks	23
3.1.1 Fact Representation	24

3.1.1.1	Function Predicates	24
3.1.1.2	Real-valued Data	25
3.1.2	Rule Representation	26
3.1.3	Network Construction	33
3.2	NeuralLog Structure Learning Algorithms	37
3.2.1	Meta-Interpretive Learning	38
3.2.2	Online Structure Learner by Revision	40
3.2.2.1	Data Representation	41
3.2.2.2	Theory Revision	42
3.2.2.3	Accepting the Revision	43
3.2.2.4	Clause Modifiers	43
3.2.3	Online Meta-Interpretive Learning	44
3.3	Discussion	47
4	Experiments & Results	51
4.1	Parameter Learning	51
4.1.1	NeuralLog in Comparison with Other Logic-based Systems	52
4.1.1.1	Methodology	53
4.1.1.2	Results	55
4.1.2	NeuralLog with Numeric Values	59
4.1.2.1	Iris Dataset	59
4.1.2.2	NCBI Disease Dataset	61
4.2	Structure Learning	62
4.2.1	Batch Structure Learning	63
4.2.1.1	Methodology	63
4.2.1.2	Results	65
4.2.1.3	Comparison with Embedding Systems	66

<i>CONTENTS</i>	xi
4.2.2 Online Structure Learning	67
4.2.2.1 Simulating the Online Environment	68
4.2.2.2 Results	69
5 Conclusions	75
5.1 Future Works	77
References	79
A The NeuralLog Language	87
A.1 The Language	87
A.1.1 Special Predicates	88
A.1.1.1 Learn	89
A.1.1.2 Example	89
A.1.1.3 Mega Example	90
A.1.1.4 Set Parameter	90
A.1.1.5 Set Predicate Parameter	91
A.1.2 Special Terms	91
A.1.3 Syntax Sugars	92
A.1.3.1 For-each Loop	92
A.1.3.2 Wildcard Syntax	93
A.1.4 Comments	94
A.2 Compilation Process	94
A.2.1 Facts Representation	95
A.2.2 Functional Predicates	96
A.2.3 Rules Representation	97
A.2.4 NeuralLog Parameters	97
A.2.4.1 Model Parameters	98

A.2.4.2 Training Parameters 100

List of Tables

2.1	Kinship Example	10
2.2	Metagol Higher-order Theory	13
3.1	A Set of Facts in NeuralLog	24
3.2	Higher-order Logic Theory	40
3.3	Theory Example for the UWCSE Dataset	41
4.1	Maximum Relation Depth	54
4.2	Size of the Datasets	54
4.3	Average Precision for Cora, UWCSE and WordNet Datasets	55
4.4	Area Under the ROC curve for Cora, UWCSE and WordNet Datasets	56
4.5	Average Precision for Yelp and PAKDD15 Datasets	57
4.6	Area Under the ROC curve for Yelp and PAKDD15 Datasets	57
4.7	Example of Learned Weights for WordNet and Cora Datasets	58
4.8	Results for the Named Entity Recognition on the NCBI Disease Dataset	62
4.9	The Meta-Theory Used by NeuralLog+MIL	64
4.10	Results for the Filtered Rank Metric	65
4.11	Example of Learned Rules and Weights for WordNet Dataset	66
4.12	Comparison with Embedding Systems	66
4.13	Size of the Datasets	68

A.1	Mega Examples	90
A.2	A Set of Facts	95
A.3	Functional Predicate Definition	97

List of Figures

1.1	Overview of the Components of NeuralLog	3
2.1	SLD-Resolution Tree	12
2.2	Multilayer Perceptron Example	16
3.1	The Tensors from the NeuralLog Facts	25
3.2	Example of the DAG representation (on the left-hand side) and the found paths (on right-hand side) of the rule	27
3.3	NeuralLog Network Example	35
3.4	Meta SLD-Resolution Tree	40
3.5	Tree Structure Representation of the UWCSE Theory Example	41
4.1	The Evaluation of the Cora Dataset	70
4.2	The Evaluation of the UMLS and UWCSE Datasets	71
A.1	The Tensors from the NeuralLog Program	96

List of Algorithms

1	Find paths: algorithm to find the paths between the source and destination terms	29
2	Find clause paths: algorithm to find the paths between the sources and the destination terms of a clause and the disconnected literals	30
3	Theory to construct a Multilayer Perceptron for the Iris dataset	60
4	Special NeuralLog Terms	92
5	For-each Loop Syntax	93
6	For-each Loop Example	93

List of Acronyms

∂ ILP Differentiable Inductive Logic Programming

ANN Artificial Neural Network

BCP Bottom Clause Propositionalization

BERT Bidirectional Encoder Representations from Transformers

BK Background Knowledge

CWA Closed World Assumption

DAG Directed Acyclic Graph

ILP Inductive Logic Programming

ISG Iterated Structural Gradient

KB Knowledge Base

LCWA Local Closed World Assumption

LNN Logic Neural Network

LP Logic Programming

LRNN Lifted Relational Neural Network

LTN Logic Tensor Network

MIL Meta-Interpretive Learning

MLN Markov Logic Network

MLP Multilayer Perceptron

MRR Mean Reciprocal Rank

NEN Named Entity Normalization

NER Named Entity Recognition

NLP Natural Language Processing

NN Neural Network

OMIL Online Meta-Interpretive Learning

OSLR Online Structure Learner by Revision

RDN Relational Dependency Network

SLD Selective Linear Definite

SLP Stochastic Logic Programming

SRL Statistical Relational Learning

StarAI Statistical Relational Artificial Intelligence

UMLS Unified Medical Language System

Chapter 1

Introduction

Neural Networks (NNs) have achieved a great success on a wide range of tasks, given the advance of deep learning techniques [1]. However, traditional neural network models cannot take advantage of Background Knowledge (BK), which may contain additional information about the examples as well as expert knowledge. On the other hand, Logic Programming (LP) [2] uses logic programs to describe and to reason about structured and multi-relational data, which pose as Background Knowledge [3]. However, LP struggles to deal with numeric features, uncertainty and noise; which are inherent characteristics of real world applications.

The field of Neural-Symbolic Learning and Reasoning tries to combine the strengths of both neural networks and logic programming, in order to obtain models that are both capable of dealing with numeric features, uncertainty and noise and can also take advantage of existing Background Knowledge [4].

In this work, we propose *NeuralLog*, a first-order logic language that is compiled to a neural network. The main goal of NeuralLog is to bridge logic programming and deep learning in order to exploit the advantages of these fields in both discrete and continuous data.

Another advantage of NeuralLog is that it allows the user to abstract the design of deep neural networks by the use of a logic language, which would facilitate the use of deep learning models by people that are not familiarized with common programming language.

In addition, NeuralLog also supports numeric attributes and the use of numeric functions, which makes NeuralLog a very flexible language, and thus, provides the user the ability to create generic neural networks; in contrast with some previous works in this field [5, 6], that are restricted at combining logic with neural networks on discrete domains composed of relations among entities. Furthermore, the logic program is mapped to a neural network in such a way that the logic of the program can be easily identified in the structure of the network and vice versa, making the model easier to interpret.

NeuralLog was designed to transform a first-order logic program into a neural network. It receives as input a set of first-order rules that are used to define the neural network structure, and a set of facts that become weights in the neural network. Then, those weights are fine-tuned given a set of examples.

A great advantage of a system based on first-order logic is that we can rely on a wide range of Inductive Logic Programming (ILP) algorithms in order to find logic theories from data. ILP is a field of study concerned with developing algorithms to learn first-order logic theories from a set of examples, given Background Knowledge [3].

In addition to the inference system of NeuralLog, we also propose three structure learning algorithms to learn first-order logic programs from data in NeuralLog language. The first structure learning algorithm presented is NeuralLog+MIL, a structure learning algorithm based on Metagol [7].

Metagol [7] is a novel Meta-Interpretive Learning (MIL) system that uses higher-order logic theory to define the possible clauses that shall appear in the first-order program. In addition to defining the hypotheses space of the first-order program, the higher-order theory is also used to navigate in this hypotheses space. We believe that the use of higher-order logic to define the possible first-order program is well-suited to integrate with NeuralLog, since it allows the user to precisely define the structure of the clauses that will be used in the program.

Traditional structure learning systems such as [7–9] are designed to learn in batch environment, where the examples are expected to be available at the beginning of the learning process. However, given the availability of data sources where the data changes over time, the need for online learning algorithms has been increasing. An online learning algorithm, different from the batch learners, receives as input a stream of examples and adapts its model to cope with the new examples [10].

In order to give NeuralLog the ability to learn in online environments, we implemented NeuralLog+OSLR, a ported version of Online Structure Learner by Revision (OSLR) [11,12]. OSLR is an online-learning algorithm that applies theory revision to adapt an existing theory to new arriving examples. We opted to use OSLR, since it is a novel theory revision approach for Stochastic Logic Programming (SLP) [13], which achieved good results in comparison with other methods [11,12]. In addition, OSLR has a clear separation of the revision theory algorithm and the underlying inference engine, which makes it easier to port it to different inference engines.

Inspired by the revision algorithm of Online Structure Learner by Revision (OSLR) and the MIL approach of NeuralLog+MIL, we also propose NeuralLog+OMIL, an online structure learning algorithm that uses the same revision mechanism of NeuralLog+OSLR, but it uses a MIL operator to propose revisions in the logic theory, based on a higher-order theory. To

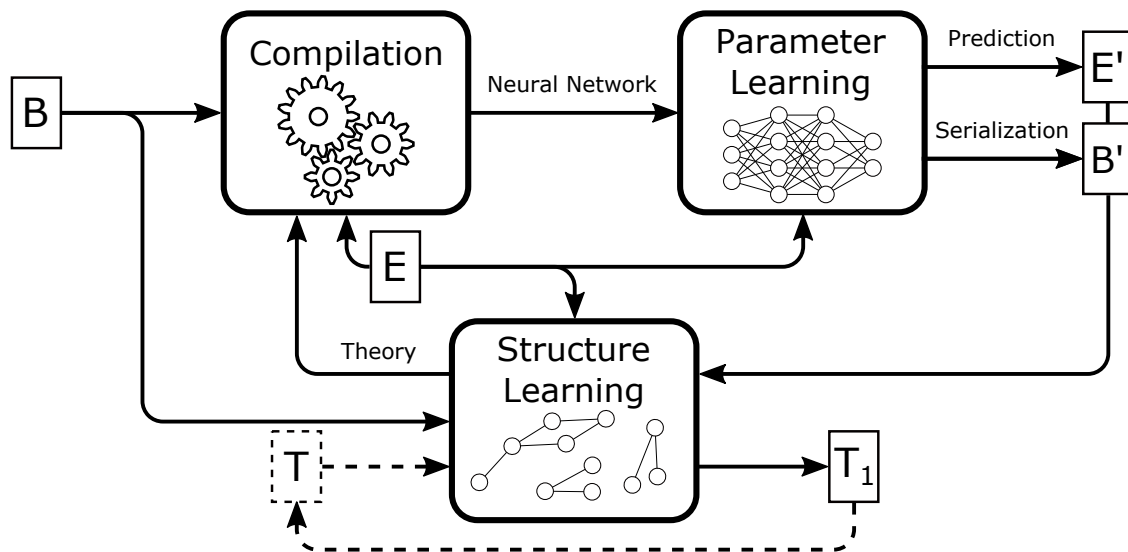


Figure 1.1: Overview of the Components of NeuralLog

the best of our knowledge, it is the first time that Meta-Interpretive Learning is applied to online learning.

In order to better integrate the structure learning algorithms with the neural network aspects of NeuralLog, we implemented the concept of *clause modifiers*, which takes the proposed clauses from the structure learning algorithms and modify it to append attributes of neural networks, such as activation functions or weights.

Figure 1.1 gives an overview of the components of NeuralLog. On the top part of the figure, we have the parameter learning components, which will be described in details in Section 3.1. On the bottom part, we have the structure learning component, that will be described in details in Section 3.2.

The *parameter learning* part is composed of two main components, the *compilation* component, which takes a set of examples E and Background Knowledge B , composed of a set of (weighted) facts and rules, as input and turns it into a neural network. While the *parameter learning* component takes the neural network and a set of examples E and learns the parameters, represented as the weights of the facts, in order to fit the examples. After the parameter learning phase, we can use the neural network to compute the prediction E' of a set of examples; or we can store it back in logic form B' , by updating B with the adjusted weights that were learned during the parameter learning phase.

In addition, NeuralLog also has a *structure learning* component. There are two paradigms for structure learning with NeuralLog: batch (offline) learning and online learning. For batch learning, the component receives as input a Background Knowledge B , and the examples E ;

and it appends a set of new rules to B which is then compiled into a neural network where parameter learning can be performed as usual.

For online learning, in addition to the Background Knowledge and examples, the component also receives a theory T as input, that can be empty, and is represented as the dashed square in the figure. A theory is a set of logic rules, which differs from the rules in the Background Knowledge in the way that the rules in the theory can be changed by the structure learning component, while the ones in the Background Knowledge are fixed.

Different from batch learning, online learning receives a stream of examples, where sets of examples are arriving over time. When a set of examples arrives, the component uses these examples to, possibly, propose a theory T_1 , by modifying the input theory T , which are then compiled into a neural network; the parameters are trained on the examples, and the theory and Background Knowledge with adjusted weights are stored. When another set of examples arrives, this process is repeated, starting from the last proposed theory, with the last learned weights, until no more examples are given.

It is important to notice that the structure learning component can make use of the predictions of the neural network, on the set of examples, in order to propose changes to the theory.

We compared NeuralLog, for parameter learning, with two state-of-the-art systems in two very different domains: link prediction with TensorLog [14] and classification with RelNN [15]. Our results show that NeuralLog outperforms TensorLog in three out of four target relations in the Cora dataset [16], and in the UWCSE dataset [17]; while achieving comparable results in the WordNet dataset [18]. In the classification tasks, NeuralLog outperformed RelNN in both experimented datasets: Yelp and PAKDD15 [15]. NeuralLog outperformed these two other systems on tasks for which they were designed, which supports our claim that NeuralLog is a significant contribution towards a flexible neural relational system.

Furthermore, we applied NeuralLog to a propositional classification task on the classic Iris dataset [19,20], showing how one can define a Multilayer Perceptron (MLP) on the NeuralLog language. In addition, in order to demonstrate the capacity of NeuralLog to integrate with other neural network models, we applied NeuralLog to a Named Entity Recognition (NER) task, using a state-of-the-art neural network model called Bidirectional Encoder Representations from Transformers (BERT) [21], which is called as a logic predicate.

In our experiments to learn logic structure in batch environments, we compared NeuralLog+MIL with Neural-LP [8], a neural network system that is based on TensorLog [14], a system closely related to NeuralLog. We performed link prediction on three datasets: UMLS [22], WordNet [18] and UWCSE [17]. NeuralLog+MIL achieved results comparable to Neural-LP, for the hit at top 10 and mean rank metrics [23] for the UMLS and the WordNet

datasets; and outperformed Neural-LP in the UWCSE dataset.

Finally, in the online structure learning environment, we compared NeuralLog+OSLR and NeuralLog+OMIL with the original OSLR approach [11, 12] and RDN-Boost [24], a system that learn Relational Dependency Networks (RDNs) [25], which was also used in [11, 12]. Our experiments showed that NeuralLog+OMIL outperformed OSLR and RDN-Boost, for link prediction, on three out of the four target relations from the Cora dataset and in the UMLS dataset. While NeuralLog+OSLR and NeuralLog+OMIL can outperform OSLR and RDN-Boost on the UWCSE, assuming a good initial theory is provided.

The main contribution of this thesis is NeuralLog, a framework for relational learning that combines first-order logic with neural networks. We can summarize the contribution of NeuralLog as follows:

- It is a flexible first-order language to describe neural networks for relational tasks;
- It allows the use of numeric attributes in the logic program, which allows the definition of some neural network structures, such as Multilayer Perceptrons (MLPs), directly in the logic language;
- It allows the user to call complex neural network structures as logic predicates;
- It has a structure learning algorithm to learn batch tasks, that can learn first-order programs using MIL, a novel ILP framework;
- It has two structure learning algorithms to learn online tasks, which are based on theory revision, one of which uses MIL to revise theories.

In order to access the value of this work, we performed experiments to answer five research questions:

- Q1** Can NeuralLog represent *link prediction* models?
- Q2** Can NeuralLog represent *classification* models?
- Q3** Can NeuralLog+MIL learn the structure representation of NeuralLog models for link prediction tasks?
- Q4** Can NeuralLog+OSLR and NeuralLog+OMIL learn the structure of NeuralLog models online, by using theory revision, for link prediction tasks?
- Q5** Can NeuralLog+OSLR and NeuralLog+OMIL benefit from a previous existing theory, when learning the structure of NeuralLog models online, for link prediction tasks?

In order to answer each of the first two questions, we compared NeuralLog with TensorLog [14] and ReLNN [15], respectively. The results of our experiments showed that NeuralLog outperformed the other systems in most of the datasets, affirmatively answering the questions.

In order to answer the third question, we compared NeuralLog+MIL with Neural-LP [8]. NeuralLog+MIL achieved results comparable to Neural-LP in two of the three experimented datasets, and outperformed Neural-LP in one of the experimented datasets, affirmatively answering the question.

Furthermore, to answer the last two questions, we compared NeuralLog+OSLR and NeuralLog+OMIL with OSLR [11, 12] and RDN-Boost [24] in three datasets, where NeuralLog+OSLR and NeuralLog+OMIL outperformed the other systems in two datasets, starting from empty theories; and outperformed the other system in the third dataset, whenever a good initial theory was available. Thus, we can affirmatively answer the last two questions.

We chose to compare NeuralLog with the systems mentioned above, because they are closely related to NeuralLog in the tasks for which they were tested. Finally, this work directly resulted in the following publications, which include the experiments presented here:

- Victor Guimarães and Vítor Santos Costa. Neurallog: a neural logic language. *CoRR*, abs/2105.01442, 2021, which presents the NeuralLog language; the compilation process of the first-order logic program into a neural network; and its inference and parameter training mechanisms. This work is shown in this thesis in Sections 3.1 and 4.1;
- Victor Guimarães and Vítor Santos Costa. Meta-interpretive learning meets neural networks. *The Semantic Data Mining Workshop, SEDAMI 2021*, 08 2021, which presents our batch structure learning algorithm NeuralLog+MIL. This work is shown in the Subsections 3.2.1 and 4.2.1; and
- Victor Guimarães and Vítor Santos Costa. Online learning of logic based neural network structures. In *Inductive Logic Programming*, Athens, Greece, 2021. Springer International Publishing, which presents our online structure learning algorithms, NeuralLog+OSLR and NeuralLog+OMIL. This work is shown in Subsections 3.2.2, 3.2.3 and 4.2.2.

The motivation of this work was to develop a logic language capable of better integrating neural networks with first-order logic, where the inference of the neural network is related with the logic structure. The parameter learning experiments show that the NeuralLog language is capable of representing neural networks for (relational) classification and link prediction. In addition, the experiments with the Iris dataset [19, 20] and for the NER task

show how NeuralLog can be better integrated with traditional neural networks, due to its capability of handling attribute predicates, which have numeric values.

Nonetheless, the structure learning experiments show that traditional ILP systems can be easily ported to NeuralLog, contributing to advance the integration between first-order logic and neural networks. In addition, it indicates that NeuralLog inference is related to the logic theory, which makes it easier to interpret the meaning of NeuralLog models, by looking at the logic theories and the learned weights.

The remainder of this thesis is organized as follows: we start Chapter 2 by giving the reader Background Knowledge in order to better understand this work, and we finish this chapter by presenting the works related to ours in Section 2.4; then, we present NeuralLog in Chapter 3; followed by the performed experiments in Chapter 4; we conclude this work in Chapter 5, also presenting some directions for future works. Finally, Appendix A gives a detailed description of the syntax of the NeuralLog language.

Chapter 2

Background Knowledge

In this chapter, we give the fundamental knowledge and the notation required in order to better understand this work.

We start by presenting the fundamentals of Logic Programming (LP), which is the base of this work. In Section 2.1, we explain the fundamentals of LP, passing through first-order logic, higher-order logic and finishing with theory revision. Then, in Section 2.2, we explain the fundamentals behind neural networks. In Section 2.3, we compare online vs offline (batch) learning. Finally, in Section 2.4, we give an overview of works related to this thesis.

2.1 Logic Fundamentals

In this work we use first-order logic to represent and learn neural network models in multi-relational tasks. First-order logic is a paradigm that uses a formal language in order to represent knowledge and to reason about this knowledge [2]. The knowledge is composed of logic entities, which may have relations among each other; while the reason is made through logic rules. A set of Horn clauses will be referred hereafter as logic program or Knowledge Base (KB).

In order to represent both the knowledge and to reason about it, we use Horn clauses [29]. A Horn clause has the form of:

$$h(.) \leftarrow b_0(.) \wedge \dots \wedge b_n(.).$$

Where h and b_i are *predicates* and $(.)$ are (possibly empty) lists of *terms*. A *term* can be either a *constant* or a *variable*. A constant is represented by a string starting with a lower case letter, while a variable starts with an upper case letter. The predicate, followed by its terms, is called *atom*. A *literal* is either an atom or the negation of an atom, which is

Table 2.1: Kinship Example

(f_1)	$father(andrew, james).$
(f_2)	$father(andrew, jennifer).$
(f_3)	$father(james, charlotte).$
(r_1)	$parent(X, Y) \leftarrow father(X, Y).$
(r_2)	$grandparent(X, Y) \leftarrow parent(X, Z) \wedge parent(Z, Y).$

represented with the *not* keyword in front of it. The atom $h(\cdot)$ is called the head of the clause, while the conjunction of literals $b_i(\cdot)$ is collectively called the body of the clause. Although we do not consider negation in this work, we will still refer to the atoms in the body of a clause as literals, in order to distinguish them from the atom in the head. If an atom has no variables in it, we say that the atom is grounded. A clause that has an empty body and no variables is called a *fact* and is used to define the knowledge; otherwise, it is called a rule and is used to reason about the knowledge.

The arity of a predicate p is the number of terms n an atom of this predicate accepts, also represented as p/n . An atom of arity 0 is called a propositional atom and is represented by the name of the predicate, without parenthesis. For example, the fact *rain.* states that it rains. In this work, we do not consider first-order functions, only numeric functions, as will be described in more details in Chapter 3. We also limit the facts to have arity no greater than two.

A fact represents the existence of a relation between entities. The relation is represented by the predicate, while the entities are represented by the logic constants. An atom is proved (considered to be true) whenever it exists as a fact in the KB, or it can be proved by rules and facts from the KB. A variable is a term that can be replaced by a constant, in order to make an atom equal to a fact in the KB.

A rule proves the atom in its head whenever there is a substitution of its variables that proves *all* the literals in its body, possibly by using other rules. A negated literal *not A* is proved whenever we *fail* to prove its non-negated form *A*.

Table 2.1 shows an example of the kinship dataset [30]. The first three lines are facts that relate a fatherhood; and the remaining are rules that define the concept of *parent* and *grandparent*, based on the predicate *father/2*.

We base the NeuralLog syntax on DataLog [31], adding the weighted fact syntax from ProbLog [32]. DataLog is a first-order logic language without first-order functions.

2.1.1 SLD-Resolution

A common approach to perform inference of logic program containing a set of Horn clauses is to use the Selective Linear Definite (SLD) clause resolution, also known as SLD-Resolution.

The SLD-Resolution starts from a goal we would like to answer. This goal can be either an atom or a conjunction of atoms, often represented as a headless Horn clause. The result of the SLD-Resolution is the substitution of the variables of the goal which proves it, with respect to the knowledge base, if such substitution exists. One could also continue the resolution in order to find all the possible substitutions. If the goal has no variables, it returns whether it is proved or not.

The resolution starts by creating a SLD-Resolution tree, where the root node is the initial goal and an empty variable substitution. Then, it tries to solve the first goal in the list of goals of the root node by finding a Horn clause whose head can be unified with the goal. We say that two atoms A and B can be unified if there is a substitution θ of the variables of the atoms such as $A\theta = B\theta$. In this case, the *most general unifier* is used, which is the simplest variable substitution that makes both atoms equals, with respect to the knowledge base.

If a clause is used to solve a goal, a child node is created by replacing the goal, from the list of the parent node, by the body of the applied clause with the substitution applied to the clause, and the substitution of the child node becomes the used substitution appended to the substitution of the parent node.

If the body of the clause is empty, in the case of a fact, the list of goals gets smaller. Whenever the list of goals of a node becomes empty, we call this node a solution. Otherwise, if there is no Horn clause that can be applied to a node, the path fails; in this case, the algorithm backtracks to another path, if it exists.

This algorithm is applied recursively to the tree, usually in a depth-first order, until a solution is found or no other path is possible. If a solution is found, it can either quit or backtrack to find other solutions. Each solution node has an associated substitution that, when applied to the original query, would produce a valid answer, given the KB.

Figure 2.1 shows a subset of the SLD-Resolution tree used to resolve the goal $grandparent(X, charlotte)$, given the KB in Table 2.1. Each edge contains the clause used to create it alongside the substitution θ used to unify the goal from the parent node with the head of the used clause. The underlined atoms in the nodes are the current goals. The solution node is marked with a \square , while \times nodes represent the end of a failed path. We omitted some renaming substitution of variables for clarity.

The goal is to find all the grandparents of *charlotte*. In other words, all the valid substitutions of the variable X . Finally, we can see that the only solution is the substitutions $X/andrew$,

meaning that *andrew* is the only grandparent of *charlotte*, given this specific KB.

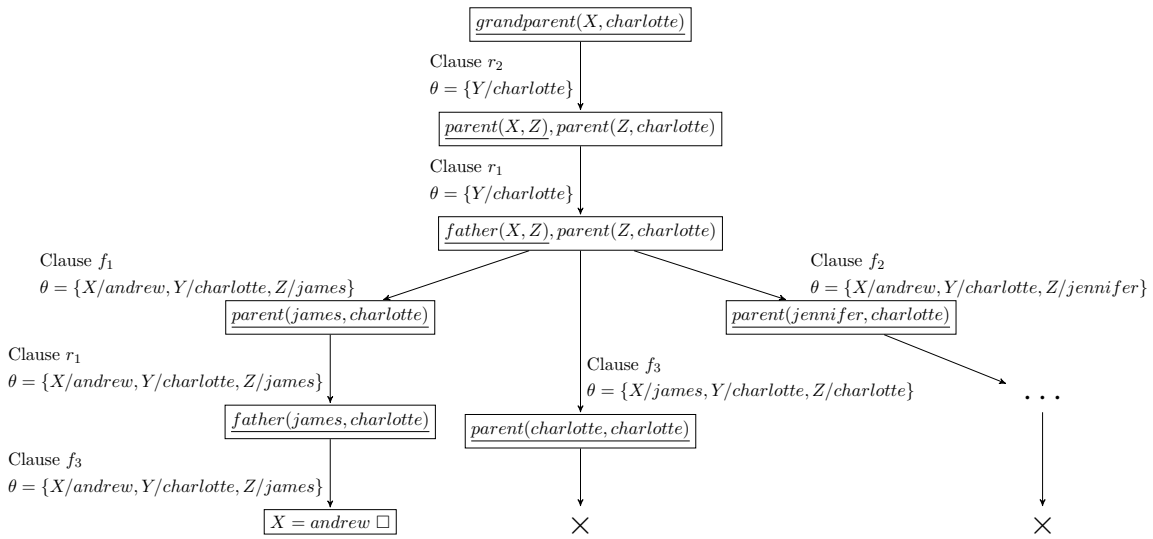


Figure 2.1: SLD-Resolution Tree

The SLD-Resolution is the proof mechanism used by the Prolog interpreter [33], which is proved to be sound and complete for sets of Horn clauses. We refer the reader to [34] for a broader overview on first-order logic and SLD-Resolution.

2.1.2 Inductive Logic Programming

Inductive Logic Programming (ILP) is a subfield of machine learning which is concerned with finding logic theories from examples [34]. Formally, given Background Knowledge (BK) consisting of a set of logic rules and facts; and a set of examples $E = E^+ \cup E^-$, where E^+ is the set of positive examples and E^- is the set of negative examples; ILP algorithms would try to induce a hypothesis H , composed of a set of rules, which conforms with the constraints below:

1. $BK \wedge H \models E^+$, this is, H is *complete*; and
2. $BK \wedge H \not\models E^-$, this is, H is *consistent*.

A hypothesis that conforms with both constraints is said to be *correct* [34].

In real world applications, usually it is not possible to find a correct hypothesis. Thus, these constraints are often relaxed, and the algorithms try to find a hypothesis that optimizes a given metric.

There are several systems to learn logic programs, such as FOIL [35], Progol [36], Aleph [37], among others. In the remaining of this subsection, we show two different ILP approaches

Table 2.2: Metagol Higher-order Theory

$$\begin{aligned}
P(X, Y) &\leftarrow Q(X, Y). \\
P(X, Y) &\leftarrow Q(X, Z), R(Z, Y). \\
P(X, Y) &\leftarrow Q(X, Z), P(Z, Y).
\end{aligned}$$

used in this work: Metagol, which is a Meta-Interpretive Learning algorithm; and Online Structure Learner by Revision (OSLR), which is an online algorithm based on *theory revision from examples*.

2.1.2.1 Meta-Interpretive Learning

Meta-Interpretive Learning is a method that learns first-order logic theories by the use of a higher-order logic theory that will guide the search through the hypothesis space [7].

In first-order logic, the predicate names in the rules, which represent the relations between the logic entities, are constant. In higher-order logic, those predicate names might also be variables, and the logic inference system should find the substitution of the name in order to prove the rule.

Metagol is a MIL system that uses a higher-order theory in order to construct a first-order hypothesis consistent with BK and a set of examples [7]. Metagol achieves its goal by using a modified Prolog meta-interpreter that transverses the higher-order theory in a similar way a conventional first-order SLD-Resolution algorithm would do [33]. However, in the higher-order resolution, the higher-order predicate symbols must also be substituted to first-order predicates found in the BK.

Table 2.2 shows the higher-order theory used by Metagol. In this higher-order theory, P , Q and R are higher-order variables, which shall be replaced by first-order predicate names; and X , Y and Z are first-order variables, which work in the same way as in first-order theory.

These rules are applied to the examples in order to find the substitutions of both the higher-order and first-order variables that prove the positive examples without proving the negative ones. Then, the proof path of the meta SLD-Resolution tree will hold a set of instantiated rules, where the first-order terms are transformed into variables in order to generate a first-order theory.

In subsection 3.2.1, we show an example on how the construction of the first-order theory is performed in our implementation of a MIL structure learning algorithm.

2.1.2.2 Theory Revision from Examples

Theory revision from examples is a subfield of ILP that focuses on modifying an initial (partially correct) theory in order to improve it with respect to a new set of examples [38,39]. This characteristic of starting from a (possibly empty) initial theory in order to adapt it to new examples makes theory revision a suitable candidate to be applied to online environment, where new examples are arriving over time.

The top-level algorithm of a theory revision system can be described as four main steps: (1) finding examples that are incorrectly classified by the current theory (model); (2) finding the *revision points*, the points of the theory responsible for the incorrect classification of the examples; (3) propose the application of the *revision operators* to these points, which will propose changes in the corresponding part of the theory; and, finally, (4) deciding whether the proposed changes must be applied to the theory.

Revision Points. The revision points are points in the theory responsible for misclassifying some example. Those points can either be an entire rule or a specific literal in the body of a rule, and they are usually categorized as follows:

- **Specialization revision point:** representing rules in the theory that participate in the proof of negative examples. This kind of revision point indicates that the theory is too *generic* and, thus, needs to be *specialized* in order to avoid such proves;
- **Generalization revision point:** representing literals in rules that prevent positive examples to be proved. This kind of revision point indicates that the theory is too *specific* and, thus, needs to be *generalized* in order to allow such proves.

Revision Operators. The revision operators are operators that can be applied to the revision points in order to modify it to better describe the new examples. They are usually classified in four categories:

1. **Add rule:** it appends a new rule to the existing theory in order to prove positive examples. It can create a new rule based on an existing one or from scratch;
2. **Delete antecedent:** it erases literals from the body of a rule in order to make the rule prove positive examples;
3. **Delete rule:** it erases a rule in order to avoid the proof of negative examples;
4. **Add antecedent:** it appends new literals to the body of an existing rule in order to avoid the proof of negative examples by the rule.

The former two operators are called generalizing operators, while the last ones are called specializing operators. Those are the four main revision operators. There are more revision operators, but they are usually a combination of these ones.

It is important to notice that a revision operator proposes a revision to the theory. However, such revision might not be implemented by the revising system, since it can hurt the performance of the theory on current or previous examples. Deciding whether to apply a revision to the theory is also part of the theory revision system. In addition, not all operators are suitable to all revision points.

Normally, all suitable operators are applied to all revision points and the one that brings the most improvement is implemented. Then, the algorithm can proceed to find new revision points and to apply the revisions on them, following a greedy approach, until no further improvement of the theory is possible [40].

2.2 Neural Networks Fundamentals

Artificial Neural Networks (ANNs), in this work, simply referred as Neural Networks (NNs), are structures inspired by the human brain which are able to learn mathematical functions to describe a set of examples [41].

In this work, it is sufficient to say that a Neural Network is a directed graph of layers, where each layer receives as inputs the results of its income layers and computes its output based on its inputs and its internal parameters. In this graph, there is a set of layers that are called input layers, which represents the input of the NN and whose inputs are given from the features of the examples. In addition, there is a set of output layers, whose output represents the output of the NN.

Figure 2.2 shows an example of a Multilayer Perceptron (MLP), a simple neural network which is composed of a sequence of fully connected layers [42]. A MLP network consists of a single input layer, a sequence of hidden layers and a single output layer. Each layer is internally composed of a number of neurons. The input layer has a neuron to represent each feature of the example, while the output layer has a neuron to represent each label of the example, this is, the result of the function we would like to learn. In this case, the input layer has three neurons and the output layer has two neurons. This means that the function we would like to learn receives, as input, three values and returns, as output, two values.

In a MLP network, all the layers are fully connected, this means that each neuron in a layer receives as input the output of all the neurons from the previous layer. For each connection, there is a weight, and the output of a neuron in a fully connected layer is given by the formula $f(\sum_{i=0}^n w_i x_i + b)$, where f is a differentiable function, x_i are the inputs of the neuron, w_i

are the weights corresponding to the inputs and b is a bias. The weights and biases are called the parameters of the neural network. In the internal layers, the function f is called activation function; and in the output layer, it is called output function, and gives the result of the neural network for the given example.

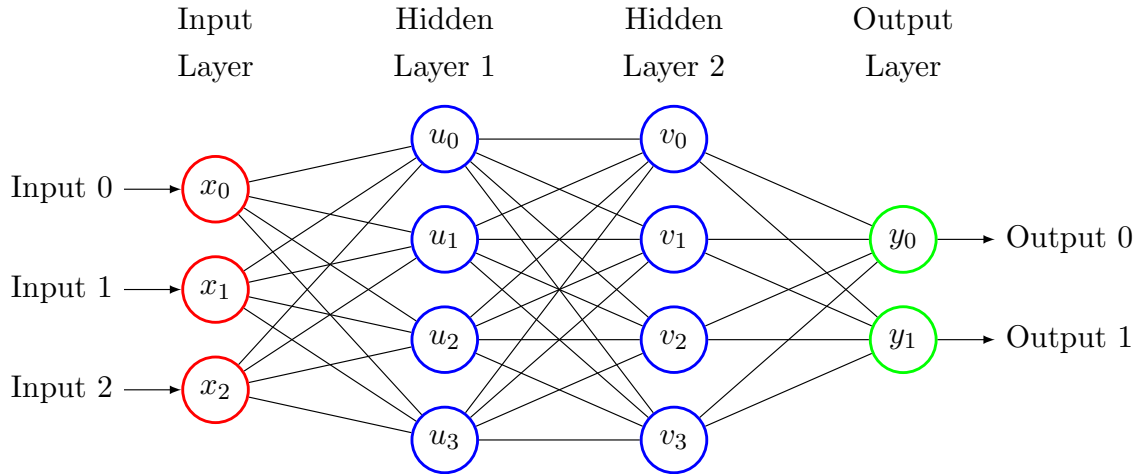


Figure 2.2: Multilayer Perceptron Example

The goal of a neural network is to map the features of each example to a desired output. To achieve this goal, the features of the examples are fed to the input layers, which pass them to the inner layers connected to it, until it reaches an output layer. The final value of the output layers is then compared with the desired values of the examples by a differentiable function called loss function. The closer the output of the neural network is to the expected value of the example, the smaller the result of the loss function must be. Finally, the parameters of the neural network are adjusted in order to minimize the loss function, which is called the learning phase.

The usual way to adjust the loss of the NN is to use a gradient descent algorithm, which differentiates the loss function according to the parameters of the neural networks and adjusts the parameters in order to reduce the loss of the network, possibly applying some sort of regularization in order to prevent overfitting [43].

Recent neural network advances came in the field of deep learning, where several layers are stacked to form deep neural networks. These advances come in the form of new network and layer structures and better learning algorithms. A notable example is Bidirectional Encoder Representations from Transformers (BERT) [21], a model that uses attention mechanism (a mechanism that learns which part of the input is more relevant at a given moment) and is trained on a large set of unlabelled text in order to achieve a good performance on text processing tasks.

BERT has achieved state-of-the-art performance in several text processing tasks, including

in our previous work, where we extracted diseases from biomedical scientific literature [44]. Although powerful, those models are hard to interpret. In this work, we propose a logic language that is: on one hand, able to define neural network structures based on logic, thus, easier to interpret; while, on the other hand, it can use complex neural logic structures which have state-of-the-art performance. In addition, it can also combine both types of structures.

Since the input and output of the neural network are n-tuples of numeric values, they are commonly represented as vectors. Consequently, MLP layers are represented as matrices, and the computation of the neural network is performed by multiplying the input vector by the matrix of the layer, summing the bias and applying the activation function, element-wise. Then, the output of a layer is passed as input to next layer, until the output layer is reached. This matrix representation is the foundation of the NeuralLog knowledge representation, as will be shown in Chapter 3.

2.3 Online vs Offline Learning

Conventional machine learning algorithms are designed for batch (offline) learning. This is, they start on a learning phase, where they receive all the available examples, then they train on those examples to create a model that will be used during the test/production phase. If new examples become available, in order to incorporate their knowledge, the algorithm would need to create a new model, from scratch, to replace the current one.

In order to take advantage of already existing models, online learning algorithms were proposed. Different from batch learning algorithms, online learning methods take the current model as a starting point and try to modify it in order to cope with the new examples.

Online learning algorithms are usually applied to stream of data, where examples are arriving over time and the algorithm keep adapting the model to also describe the new examples [10].

Theory revision from examples is a suitable technique to be applied to online learning, since it starts its process from an existing theory and adapts it to new examples. In this case, it would always start using the current model as the initial theory and adapt it, according to the new arriving examples. Then, the new theory will become the current one and the process would continue to the new examples.

Another advantage of using theory revision is that it can start from a pre-existing logic theory, allowing the use of expert knowledge, if present. It is known from other studies that starting from an initial theory, even if it is only partially correct, may improve the quality of the model even on the presence of fewer examples [40, 45, 46].

2.4 Related Work

The combination of ILP with deep learning can leverage deep learning the ability of handling relational data while addressing the problem of uncertainty and noise from Logic Programming. The integration of logic reasoning and neural networks is precisely the goal of Neural-Symbolic Learning and Reasoning [4].

Furthermore, other approaches that try to address the logic problem of uncertainty and noise without using neural networks exist. Most of them achieve this goal by combining logic with probabilist and/or statistic frameworks. Those approaches are known by different names such as Stochastic Logic Programming (SLP) [13] and Statistical Relational Learning (SRL) [47], among others; which have recently been generically called Statistical Relational Artificial Intelligence (StarAI) [48].

There are many ways to combine first-order logic with neural networks, from using logic features as input to other machine learning models [5, 49] to the use of logic rules to define the structure of the neural network [6, 50–54].

Our approach is more closely related to the latter, where all the logic knowledge is embedded on the neural network weights and structure. In addition to NeuralLog language, which describes the structure and weights of a neural network through a logic program, we also propose structure learning algorithms to learn NeuralLog program from examples.

The field of Neural-Symbolic Learning and Reasoning is receiving a lot of attention in the recent years. In this section, we will focus at works that combine first-order logic with neural networks and are closely related to NeuralLog.

These works can be divided into two types: (1) the ones that represent logic as neural networks; and (2) the ones that learn a neural network structure based on logic. Notice that NeuralLog is related to both types, since it is capable of representing logic as neural networks and also of learning the neural network structure.

CILP++ [50] is an extension of C-IL²P [6] that deals with first-order logic. C-IL²P is a system that generates a neural network from a set of propositional Horn clauses. The author of [50] proposes a Bottom Clause Propositionalization (BCP) to extend the network creating mechanism from C-IL²P to first-order logic. BCP extends the Progol’s bottom clause mechanism [55] to deal with a set of examples while keeping track of the variable substitution of the terms among the examples. The bottom clauses are then used to build a neural network.

Lifted Relational Neural Network (LRNN) [52] consists of a set of weighted, non-recursive and function-free Horn clauses and facts that are used to instantiate a *ground* neural network for each example to be predicted. It uses the first-order rules to *ground* a different neural

network model for each example, that can be optimized by using gradient descent in order to tune the weights of the facts and the rules. In addition, the weight of equal facts and rules are shared among the different grounded neural networks.

The drawback of propositionalization, which is used by [50, 52], is that neural network structure depends on the example to be predicted, which may cause the system to change the structure of the neural network based on the example [50]; or to create a different neural network for each example [52], which might be computationally expensive.

Riegel et al. [56] proposes a method which converts a logic program composed of weighed formulas, propositions and facts into a Logic Neural Network (LNN), which is used to predict lower and upper bound values for logic formulas. It is trained using a two-passing approach, where an upward passing adjusts the bounds of the formulas given the propositions and facts, then a downward passing adjusts the weights of the proposition and facts, given the bounds of the formulas. At each passing, the bound of the formulas are tightened, and this process is repeated until convergence.

LNN requires a set of real-value activation functions to return suitable values for logic conjunctions, disjunctions, and \forall and \exists first-order qualifiers. Since the inference of the network returns a lower and upper bound, twice the number of functions is required to properly compute both the lower and upper bounds.

TensorLog [14] is a system that performs inference of first-order logic by using numeric operations. It represents the logic facts as matrices and performs logic inference through mathematical operations on those matrices. It defines a belief propagation method to compute the value of variable in a rule through differentiable operation on the matrices. In this way, it can use an out-of-the-shelf gradient descent algorithm in order to train the weights on a set of examples.

This transformation of logic into a differentiable operation allows a closer integration between the logic method and neural network structures. However, TensorLog is limited in the set of logic rules it supports, for instance, it does not support logic rules with free variables in its body (this is, variables that appear only once in the rule). This limitation might be prejudicial in some tasks, as we show later on our experiments.

Similarly to TensorLog, RelNN uses matrices and vector operations to perform relational learning, using convolution neural networks [15]. They use the number of logic proves of the rules as features, that can also be combined into other rules, multiplied by weights, added to bias, and passed as input to activation functions. A considerable limitation of RelNN, in the version used in our experiments, is that the structure of the neural network is hard-coded into the system, despite it being based on a logic program. This makes it hard to apply RelNN to different tasks, since one would have to know the internal API defined by the system in order to define different neural networks.

Another well-known work that relies on matrix representation is Logic Tensor Network (LTN) [57], which uses a tensor network [58] to predict the confidence of logic formulas. LTN defines logic entities as numeric vector in \mathbb{R}^n , where n is defined by the user. Then, they apply matrices operations on those vectors in order to fit a set of examples. The set of operations is defined by a logic program and the goal is to find the matrices and/or the vector representation of the logic entities that satisfies the set of example.

A negative aspect of LTNs [57] is that they represent logic entities as vectors that might either be handcrafted by human, which might be cumbersome; or be latent features learned from data, whose meaning is hard to interpret.

DeepProbLog [59] combines neural networks with logic by adding *neural predicates* to ProbLog [32]. ProbLog is a probabilistic logic system that follows Sato’s semantics [60] of possible worlds [32]. DeepProbLog extends ProbLog by adding neural predicates, which are predicates whose value is given by the output of a neural network. It integrates the learning mechanism of ProbLog with neural networks, and uses gradient descent to jointly learn the parameter of ProbLog and the neural networks representing the neural predicate. A downside of possible worlds semantics is that its inference can be computationally expensive.

Differentiable Inductive Logic Programming (∂ ILP) [61] defines a way of inferring logic rules through differentiable operations, so it can use a template to generate a set of logic rules and learn the weights of those rules by applying a gradient descent algorithm to fit a set of examples. However, ∂ ILP [61] has several constraints on the logic programs it supports, such as the number of clauses for the same predicate in the head and the number of literals on the body of the clause. Shindo et al. [62] improves upon ∂ ILP by relaxing some of its constraints. In addition, it proposes another structure learning method based on the refinement of general clauses.

Similarly, Neural-LP [8] proposes an end-to-end differentiable framework, based on TensorLog [14], to learn both the Horn clauses and its parameters to infer new facts from a knowledge base. In order to achieve this, the authors restricted themselves to a very restrict subset of Horn clauses, which forms a linear path between the variables in its body and has a maximum length of T , defined by the user. Then, Neural-LP uses a recurrent neural network [63] to decide how to combine the rules from this restrict hypotheses space.

The idea of applying MIL to other inference mechanisms based on first-order is not novel. Iterated Structural Gradient (ISG) [9] proposes to apply MIL to learn theories for ProPPR [64], a Stochastic Logic Programming (SLP) system [13]. However, ProPPR uses a different inference mechanism that cannot be easily integrated with deep learning.

Meta-Interpretive Learning is well suited to integrate with NeuralLog, since the higher-order theory allows the user to define a template in order to create the relational part of the logic theory. This template can be used to append the relational part to an existing theory, which

might include the definition of an existing neural network. Furthermore, the higher-order theory may also be used to find the logic part that integrates with the neural network part, by specifying a constant predicate that will pose as a connection point between the logic part and the neural network.

We implemented the Online Structure Learner by Revision (OSLR) theory revision algorithm as our online learning mechanism [11, 12]. We opted for this algorithm because it has a clear separation between the structure learning algorithm and the underneath inference mechanism, which allowed us to easily port it to work with NeuralLog. Finally, the flexibility of OSLR allowed us to implement a new MIL revision operator to apply Meta-Interpretive Learning online. To the best of our knowledge, it is the first time that MIL is applied to online learning tasks.

Chapter 3

The NeuralLog System

In this chapter we present NeuralLog. NeuralLog is a system based on a first-order logic language of same name, which compiles programs written in the NeuralLog language into neural networks, in order to perform learning tasks. In addition, the NeuralLog system offers three structure learning algorithms to find first-order logic theories based on examples.

First, we present the NeuralLog language and how it is compiled into a neural network. Then, we present the algorithms we use, in order to learn first-order logic theories from examples, in this language; thus, learning structure of the neural networks generated by NeuralLog. Finally, we conclude this chapter by discussing the position of NeuralLog among related works.

We will refer to both the system and the logic language as NeuralLog. Whenever it is not clear, from the context, to which one we are referring, we will explicitly indicate it.

3.1 NeuralLog: a Bridge from Logic Programming to Neural Networks

The NeuralLog language is a first-order logic language designed to be compiled into a neural network, in order to perform relational tasks. Its syntax is based on DataLog [31], a well known logic language, that is often used as query language in databases.

Since neural networks require numeric inputs, we have to transform the logic KB into a numeric form. In order to achieve this goal, we represent each set of facts from a predicate as a (sparse) numeric tensor with n dimensions, where n is the arity of the predicate. This transformation will allow us to compute the inference of the rules through differentiable algebraic operations on those matrices, similar to how it is done by [14].

Table 3.1: A Set of Facts in NeuralLog

<i>parent/2</i>	<i>male/1</i>	Index
<i>parent(a, c).</i>	<i>male(a).</i>	0: a
0.5 :: <i>parent(b, c).</i>	<i>male(d).</i>	1: b
0.3 :: <i>parent(c, d).</i>		2: c
		3: d

3.1.1 Fact Representation

Let us consider that a NeuralLog program is composed of a set of facts F , a set of entities E and a set of rules R ; where each fact may have at most two arguments and always has an associated weight, which will be 1, if omitted.

We first construct an index for E by assigning a distinct integer value in $[0, n)$ to each entity $e \in E$, where $n = |E|$. Then, for each binary (arity 2) predicate $p \in F$, we create a matrix $P \in \mathbb{R}^{n \times n}$ where $P_{ij} = w$ if there is a fact $p(e_i, e_j)$, with weight w , in F ; where i and j correspond to the indices of the entities e_i and e_j , respectively. All the remaining entries of the matrix are set to 0.

We make an analogous process to the unary (arity 1) and propositional (arity 0) predicates, where the unary predicates are represented by vectors in \mathbb{R}^n and propositional predicates are scalar in \mathbb{R} . Similarly, a logic constant is represented by an one-hot vector, which have the value of 1 for the entry correspondent to the index of the entity in E , and the value of 0 for the other entries.

Table 3.1 shows a set of weighted facts, in NeuralLog language, for the predicates *parent/2* and *male/1*. The last column of the table shows the corresponding index of each logic entity. In addition, Figure 3.1 shows the tensor representation for the facts on Table 3.1, where the P matrix represents the facts of predicate *parent/2*, the M vector represents the facts for predicate *male/1*, and the a and b vectors represent the a and b logic constants, respectively.

3.1.1.1 Function Predicates

In addition to the logic predicates, NeuralLog also supports the definition of function predicates. Instead of relying on a logic definition, a function predicate has an associated differentiable function that is applied to its input vector. This function must be differentiable, because it will become part of the constructed neural network, and the gradient descent algorithms used to tune the weights of the neural network require differentiable functions.

The user can provide the implementation of the desired function and associate it to a

	a	b	c	d			a	b	c	d	
P =	0	0	1	0	a	M =	1	0	0	1	
	0	0	0.5	0	b		a =	1	0	0	0
	0	0	0	0.3	c			b =	0	1	0
	0	0	0	0	d						

Figure 3.1: The Tensors from the NeuralLog Facts

predicate or simply use a predicate whose name matches a function provided by TensorFlow’s Keras API in order to use that function. This differentiable function may be simple functions, other neural network layers, or even a whole neural network model.

3.1.1.2 Real-valued Data

One of the goals in the design of NeuralLog was for it to support numeric value data in a seamless way with the logic representation. In order to achieve this goal we allow the definition of numeric terms.

These numeric terms must be associated with logic constants by the use of binary predicates in the form of $p(e, a)$, where $e \in E$ and $a \in \mathbb{R}$. We call this form of predicate as an *attribute predicate*, since it defines a numeric attribute value for an entity in the KB.

NeuralLog uses two vectors, $\mathbf{p}_w \in \mathbb{R}^n$ and $\mathbf{p}_v \in \mathbb{R}^n$, in order to describe an attribute predicate p . For each fact $p(e, a)$, with weight w , of the predicate p in F , the vector \mathbf{p}_w stores the weight w in the position corresponding to the entity e , while the vector \mathbf{p}_v stores the value a , in the same position. The positions whose facts are not presented in the KB are set to 0.

This definition of numeric values in the terms of the logic predicate allows a more flexible use of those values in the NeuralLog program, since the value of the numeric attributes can be captured by logic variables in the definition of rules. This logic variables can, then, be passed to function predicates in a clearer way.

3.1.2 Rule Representation

We can now proceed to the rule representation, since we have already defined the representation of the facts. The key idea behind NeuralLog is to transform a logic program into a neural network, in such way that the rule inference can be performed by the neural network through matrix operations. In such neural network, the structure is defined by the rules in the NeuralLog program, while the weights are defined by the facts.

The goal of NeuralLog is to find all possible entities that might replace the variable Y , associated with the entity $X = \mathbf{a}$, with relation p , for a given query of the form $? - p(X, Y)$; with respect to the knowledge base and the clauses.

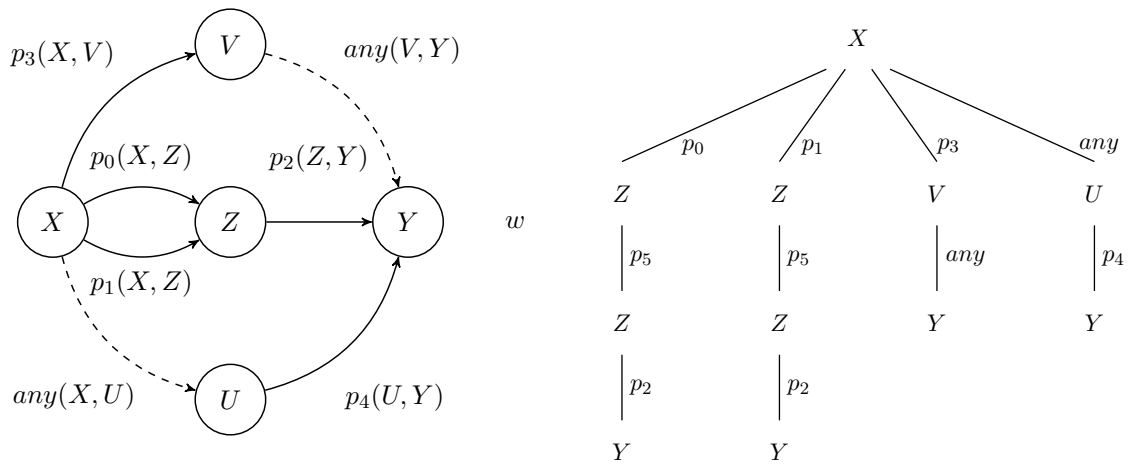
In order to compute the value of Y for the query, consider first the most simple case: a KB composed of facts for a single binary predicate $p/2$. We start by representing this KB in numeric form, as explained above. This will result in a matrix P , representing the facts and a one-hot row vector \mathbf{a} , representing the entity, which contains the value 1 for the entry corresponding to the entity, and the value 0 anywhere else. In this representation, we compute $Y = \mathbf{a}P$, representing the matrix multiplication between \mathbf{a} and P . Finally, the vector Y will contain the weights, at the corresponding entries, for each logic entity related to a , given the relation p . In the other direction, if we would like to compute the vector for the entities X related to a where $Y = \mathbf{a}$, we would transpose matrix P such as $X = \mathbf{a}P^\top = (P\mathbf{a}^\top)^\top$. This is equivalent to assuming that, for each relation $p \in KB$, there is an inverse relation p^{-1} such that $\forall a, b \in E, p(a, b) = p^{-1}(b, a)$.

In addition, this method is extended to unary and attribute predicates by using the element-wise multiplication, also known as the Hadamard product, instead of the matrix multiplication; and to propositional facts, by multiplying the vector by the scalar weight of the propositional fact.

This approach can be extended to queries that form a path, of the form $? - p(X, Y) \wedge q(Y, Z)$, where we want to compute the values of Z given X , by first computing Y as above and then computing Z such as $Z = YQ$, since $Y = \mathbf{a}P$, we have that $Z = \mathbf{a}PQ$, where Q is the matrix representation of the predicate $q/2$. It is also the base of TensorLog [14]. However, NeuralLog extends this approach in a different way, in order to compute the output values of any arbitrary rule.

Given an arbitrary rule, we first create an undirected graph representation of the rule, where the nodes are the terms in the rule; and there is an edge between two nodes if they both appear together in the same literal of the body of the rule, only considering literals whose arity is greater than 1. Then, we define the last term in the head of the rule as the output (destination) node and the remaining terms in the head as input (source) nodes.

We perform a breadth-first search in this graph, in order to find all paths from the source



(a) The DAG representation of the rule (without loop predicates)

(b) All paths from X to Y represented as a tree

Figure 3.2: Example of the DAG representation (on the left-hand side) and the found paths (on right-hand side) of the rule

nodes to the destination node. We constrain ourselves with rules whose arity of the head are greater than 0, and we use a topological order to disallow back-edges and loops, making it a Directed Acyclic Graph (DAG).

Figure 3.2 shows an example of the DAG (3.2a); and the found paths (3.2b), as a tree, of the rule below:

$$target(X, Y) \leftarrow p_0(X, Z) \wedge p_1(X, Z) \wedge p_2(Z, Y) \wedge p_3(X, V) \wedge p_4(U, Y) \wedge p_5(Z) \wedge w. \quad (3.1)$$

Since we do not allow cycles, some nodes may not appear in any path between the source and the destination. Which is the case of V and U , in this example. However, from a logic perspective, such nodes might influence the output value of Y . In order to account for the influence of these type of nodes we use a special $any/2$ predicate, inspired by TensorLog [14], that is essentially *true* for any pair of terms.

In NeuralLog, there are two possible cases where this predicate can be used:

1. Nodes that have a path from the source, but the path does not reach the destination: in this case, we add the $any/2$ predicate to connect the node to the destination, as is the case of the edge $V - Y$ shown in Figure 3.2a;
2. Nodes that have a path connected to the destination, but the path does not reach the source: in this case, we add the $any/2$ predicate to connect the node to the source, as is the case of the edge $X - U$ shown in Figure 3.2a.

The addition of the *any/2* predicates allows the neural network to account for the influence of those nodes.

To illustrate how the influence of such nodes is important in the inference of logic rules, consider the following rules:

$$\begin{aligned} \text{advisedBy}(X, Y) &\leftarrow \text{student}(X). \\ \text{advisedBy}(X, Y) &\leftarrow \text{professor}(Y). \end{aligned}$$

In the first rule, X (source) is not connected to Y (destination), however, the student predicate must be considered, since only entities that are student must be proved by the rule. The *any/2* predicate takes this into account by connecting X to Y . Analogously for the second rule, where the output must be a professor.

In order to find the *any/2* predicates from the second case, we restart the path searching process from the destination to the source, if there are nodes in the rule that are not included in any path. Then, we add the reverse of the found paths to the set of paths of the clause. By calling the searching algorithm, inverting the source and the destination, the second case of the *any/2* predicate is treated in the same way as the first case.

This process is repeated until no new edges are added to the graph. The search always terminates, since the number of steps is bounded by the number of nodes in the graph. Algorithm 1 shows how to find the paths between two terms in a clause, and the list of nodes that are included on these paths; while Algorithm 2, which uses Algorithm 1 as subroutine, describes the process of finding all paths in the clause.

The function *get_non_loop_literals(clause)* returns all the literals from the clause that do not represent a loop in the graph representation, this is, literals with arity greater than 1, whose input terms are different from the output term; the remaining literals, if any, are *loop literals*. The function *get_literal_with_term(literals, terms)* returns all the literals $l \in \text{literals}$ that contain the *term* in it. The function *compute_end_term(path, literal)* computes the term that represents the end of *path*, when the new *literal* is appended to it. Finally, the function *append_loops_to_paths(clause, paths)* appends the *loop literals* for each path $p \in \text{paths}$.

Given the found paths, which are the result of Algorithm 2 applied to the rule, we construct a DAG representation of the rule. Equal edges in different paths are collapsed into a single edge; and different incoming *any/2* edges to the destination are represented as a single *any/n* literal, where n is the number of distinct terms appearing in all these edges, where the destination is the last term. All unary predicates (which would represent a loop in the graph) are then added to their correspondent node. The *any/n* predicate represents a set of *any/2* predicates that has the same destination. The order of the input terms in the *any/n* predicate is not specified and does not affect its result, since the inputs are computed independently and are then combined through an element-wise multiplication, which does

Algorithm 1 Find paths: algorithm to find the paths between the source and destination terms

Input: The clause (*clause*); the source term (*source*); the destination term (*destination*); the visited nodes (*visited_nodes*); and all source terms (*sources_set*);

Output: The paths from the source terms to destination; and the visited nodes

```

1: function find_paths(clause, source, destination, visited_nodes, sources_set)
2:   completed_paths  $\leftarrow$  []
3:   partial_paths  $\leftarrow$  [[source]]
4:   edge_literals  $\leftarrow$  get_non_loop_literals(clause)
5:   while |partial_paths| > 0 do
6:     size  $\leftarrow$  |partial_paths|
7:     for  $i \leftarrow 0; i < size; i++$  do
8:       path  $\leftarrow$  pop_left(partial_paths)
9:       if end(path) = destination then
10:        completed_paths  $\leftarrow$  completed_paths + [path]
11:        continue to next for iteration
12:        not_added_path  $\leftarrow$  True
13:        for each literal  $\in$  get_literal_with_term(edge_literals, end(path)) do
14:          new_end  $\leftarrow$  compute_end_term(path, literal)
15:          if new_end  $\in$  path or new_end  $\in$  sources_set then
16:            continue to the next iteration of the for-each loop (line 13)
17:             $\triangleright$  path comes back to itself or to another input
18:            new_path  $\leftarrow$  path + [literal, new_end]
19:            if new_end = destination then
20:              completed_paths  $\leftarrow$  completed_paths + [new_path]
21:            else
22:              partial_paths  $\leftarrow$  partial_paths + [new_path]
23:              visited_nodes  $\leftarrow$  visited_nodes  $\cup$  literal
24:              not_added_path  $\leftarrow$  False
25:            if not_added_path then
26:              completed_paths  $\leftarrow$  completed_paths + [path + [ANY, destination]]
27:   completed_paths  $\leftarrow$  append_loops_to_paths(clause, completed_paths)
28: return completed_paths, visited_nodes

```

Algorithm 2 Find clause paths: algorithm to find the paths between the sources and the destination terms of a clause and the disconnected literals

Input: The clause (*clause*); and the destination term index (*dest_index*);

Output: The paths from the source terms to destination; and the disconnected grounded literals;

```

1: function find_clause_paths(clause, dest_index)
2:   sources  $\leftarrow$  clause.head.terms
3:   compute_reverse  $\leftarrow$  True
4:   destination  $\leftarrow$  source[dest_index]
5:   if |sources| > 1 then
6:     sources.remove[dest_index]
7:   else
8:     compute_reverse  $\leftarrow$  False
9:   all_paths  $\leftarrow$  {}
10:  all_visited_nodes  $\leftarrow$  {}
11:  sources_set  $\leftarrow$  set(sources)

12:  for each source  $\in$  sources do
13:    visited_nodes  $\leftarrow$  {}
14:    paths, visited_nodes  $\leftarrow$ 
        find_paths(clause, source, destination, visited_nodes, sources_set)
15:    all_paths  $\leftarrow$  all_paths  $\cup$  paths
16:    all_visited_nodes  $\leftarrow$  all_visited_nodes  $\cup$  visited_nodes

17:  if compute_reverse and clause.body  $\not\subseteq$  all_visited_nodes then
18:    destination  $\leftarrow$  source
19:    for each source  $\in$  sources do
20:      sources_set  $\leftarrow$  set(clause.head.terms) - {destination}
21:      backwards_paths, all_visited_nodes  $\leftarrow$ 
          find_paths(clause, source, destination, all_visited_nodes, sources_set)
22:      for each backwards_path  $\in$  backwards_paths do
23:        path  $\leftarrow$  reverse(backwards_path)
24:        all_paths  $\leftarrow$  all_paths  $\cup$  {path}

25:  disconnected_literals  $\leftarrow$  get_disconnected_literals(clause, all_visited_nodes)
         $\triangleright$  gets the grounded literals that does not belong to any path
26:  return all_paths, disconnected_literals

```

not depend on the order of the factors.

The inference of the rule, given its inputs, is represented by the result of the destination node in the DAG, multiplied by any disconnected grounded literal in the body of the rule. Those are grounded literals that do not appear in any path, which include the propositional literals. Since they do not have variables, they are represented as scalars. It is important to notice that, in the example above, w is a propositional predicate, whose weight is represented by a scalar, and not a term.

In order to compute the result of a node, we combine the values of its incoming edges by an element-wise multiplication (representing a logic **AND**); then we combine the values of the unary predicates of the node, if any, in the same order as they appear in the rule.

In order to compute the value of an edge (which represents a literal) we multiply the vector representing the input term of the edge by the matrix representation of the edge's predicate. If the predicate is represented as a vector (in the case of unary predicates), we perform an element-wise multiplication, instead. If it is an attribute predicate, we perform the element-wise multiplication between the input term, the weights and the attribute vectors. Since we need the values of the incoming nodes to compute the value of the current node, we recursively traverse the DAG, starting from the destination node, until we reach a source node, whose value is given by the input.

The *any/n* predicate has a special treatment. First, it is important to notice that the *any/n* predicate can only appear in two specific situations, as enumerated above: (1) connecting the end of a path to the destination; or (2) connecting an input term to the beginning of a path that will lead to the destination (which was computed from the destination to the input and then reversed).

In order to compute the first case of the *any/n* predicate, for each term of the predicate, except the last one, we compute the vector resulting from the term and sum it to get a scalar. For the last term (which is always the destination term of the rule), we compute the vector resulting from this term, based on its unary predicates, by passing a **1** vector (a vector where every entry has value 1) as input value for the predicates; then, we multiply it by the scalar representation of the previous terms.

Intuitively, the result represents the multiplication of the sum of the results of each (any) term by the results of each (any) entity of the last term. The final result is combined with other possible edges, arriving at the destination term. In this case, the unary predicate is not combined with the final result, since it has already been combined by the *any/n* predicate.

The second case of the *any/n* predicate occurs when we have edges connected to the destination that do not lead to any input. As such, the first $n - 1$ terms are input terms and the last term T is the beginning of the path that leads to the destination. In this case,

we pass a $\mathbf{1}$ vector (a vector where every entry has value 1), representing the value of the term T , and continue the computation of the path as usual. In numeric terms, the $\mathbf{1}$ vector represents the union of all (any) logic constants in the KB. In logic terms, the node represents a free variable in the rule, thus, the result of the rule does not depend on it, meaning that it is valid for any constant substitution of this variable.

Finally, the result of the rule is the resulting vector of the output node, multiplied by the weights of all grounded literals in the rule. In the example, the weight of w is multiplied by the final result of the rule, acting as the weight of the rule, since rules do not have associated weights. We restricted the weight to only facts, in order to make it easier to define which weights should be learned by the neural network. This approach does not reduce the expressiveness of NeuralLog, since we can include the weights to the rules in the form of literals. Furthermore, it allows the user to share the same weight among different rules, if desired.

In addition, NeuralLog allows the use of numeric functions. The functions are applied, in order of appearance in the rule, to the computed vector of its inputs. There is no explicit difference between functions and logic predicates in the NeuralLog language. The difference is that logic predicates have facts defined in the KB and the computations are given by multiplying the input terms with the matrix (or vector) representation of the facts of the predicate; while the result of function predicates is the application of the function to the input terms. These functions can be any differentiable function, including other network layers or even whole network models.

Moreover, if a relation involves a constant, we multiply the input (output) variable with the vector representation of the input (output) constant, which represents a logic **AND** between the input (output) variable and the constant.

In order to better illustrate this process, allow us to consider one more time the facts described on Table 3.1 and shown on Figure 3.1 alongside the rule described on Equation 3.2, which define the grandfather relationship using the available predicates. This is a simple example that illustrates the idea behind NeuralLog inference mechanism.

$$grandfather(X, Y) \leftarrow male(X) \wedge parent(X, Z) \wedge parent(Z, Y). \quad (3.2)$$

In order to represent this rule, and then perform the inference of it, we first apply Algorithm 2 to find all the paths between X and Y , the variables in the head of the rule. Then, we can construct a DAG from the single existent path $X \rightarrow parent \rightarrow Z \rightarrow parent \rightarrow Y$, which will be used to compute the rule. Finally, we also have to include the “loop” literal $male(X)$. Given this DAG and the values from the Table 3.1, we can compute the value of Y for a given constant given as input X . If we would like to compute the rule for $X = a$, we would

have that $X = a \odot M$ (element wise), than $Z = XP$, and, finally, $Y = ZP = [0, 0, 0.3, 0]$, which indicates that a is grandparent of c (the third entity from the logic base) with value 0.3. If we would like to repeat this process for b , we would get an all zeros vector for Y , since b is not a male and would be filtered out on the first step.

Consider a logic program composed of a set of rules, where which rule describes a single path between the input and the output variables, possibly containing unary predicates on the existing variables in the body of the rule; no recursions; and a set of facts containing only positive weight. In this case, we can compute the intermediary variable in the rule, by multiplying the input variable by the matrix representation of the relation, similar to the way the neighbours of a node can be computed using the adjacent matrix of a graph; and we can filter the variables that appears in unary relation by doing an element-wise multiplication between the vector of the variable and the vector representation of the relation. Assuming that the non-zero values of the output vector represent proved entities, we can retrieve the same logic semantics for this subset of NeuralLog programs.

However, NeuralLog is not restricted to this very limited subset of logic programs, and the equivalence with the logic inference is relaxed in exchange for flexibility to define more complex programs, which is important to improve performance on certain tasks, as we will show in Chapter 4. Nonetheless, we try to follow the logic intuition as much as possible, which may facilitate the interpretation of the model by humans. In the next section, we explain how to construct a full neural network, by generalizing it to any kind of NeuralLog program.

3.1.3 Network Construction

We have explained how we represent the knowledge base facts and how to compute an individual rule defined as a Horn clause. In this subsection, we show how to construct the complete neural network from a NeuralLog program.

In order to build the neural network, we start by creating a *Literal Layer* for each target predicate, which are the predicates found in the examples. This literal layer receives as input all the clauses (rules and facts) that share its predicate in their heads. The output of a literal layer is the sum of the results of the *Fact Layer* (represented by the matrix of facts in the KB) and the *Rule Layers* of the target predicate, given the input terms of the literal, similarly to [52]. The sum of the results represents a logic **OR**.

The output of the *Fact Layer* is the input vector multiplied by the matrix (or vector) of the weights of the facts in the KB. For attribute facts, their weights and value vectors are combined by an element-wise multiplication.

Recursively, for each rule connected to the target predicate node, we create a *Rule Layer*,

and connect the correspondent *Literal Layers* from the rule's body, as its inputs. The result from Algorithm 2 is used to define the internal structure of the rule layers. We represent function predicates in the body of the rules as *Function Layers*, the output of a function layer is the application of the function to the vector representation of its input terms. The computation of the rule layer is performed as explained in the section above, using the computation performed by the literal layers, instead of multiplying the matrices of the facts. We repeat this process until no more layers are created.

In this way, we unfold the network from the structure of the logic rules, similarly to [51]. However, our language allows the use of attribute predicates and functions. Whenever a recursion is found in the rules, it is unfolded until a pre-defined depth. Since the rule inference operation is differentiable, the unfolding of the rules, and thus, the whole neural network, is differentiable.

The *Literal Layers* of the target predicates receive as input the vectors corresponding to their input terms, and represent the inputs of the neural network; while the result of these layers represents the output. Note that if we want to compute an inverted literal, by going from the last term of the literal to the first, we can simply use the transpose matrix of the relation, for the fact layer; and compute the reversed paths, for the rule layers.

The depth of the neural network is defined by the logic program, and the user can create a hierarchy of abstract concepts by defining new predicate through rules, on top of the facts in the knowledge base.

Figure 3.3 shows the neural network created by NeuralLog, from the *target/2* rule in Equation 3.1. In the figure, $P_i \in \mathbb{R}^{n \times n}$ for $i \in [0, 4]$ are the matrices representations of the predicates $p_i/2$ of the rule; $p_5/1$ is a functional predicate whose function is $P_5 : \mathbb{R}^n \rightarrow \mathbb{R}^n$; T represents the matrix representation of the facts for the *target/2* predicate; and n is the number of distinct entities in the NeuralLog program. X is a row vector representing the input of the neural network, while Y is a row vector representing the output. Vector letters followed by ' (prime) sign(s) represent intermediary values of the corresponding logic variable (in the rule), represented by the same letter.

Each rounded rectangle represents a layer, and each layer has its output expressed by the equation in it, and the output of the rule layer is the combination of the outputs from its inner layer. The \otimes represents the element-wise multiplication of its inputs; the \oplus represents the element-wise sum of its inputs; and the operations between matrices and vectors, in the equations, are matrices multiplications.

We also assume that the *target/2* predicate is defined by a set of facts and a single rule, in the background knowledge; and all the remaining logic predicates are defined exclusive through logic facts, and no rules. If there were more rule layers for a given predicate, it would be added to its literal layer, and their results would be summed with the facts of this

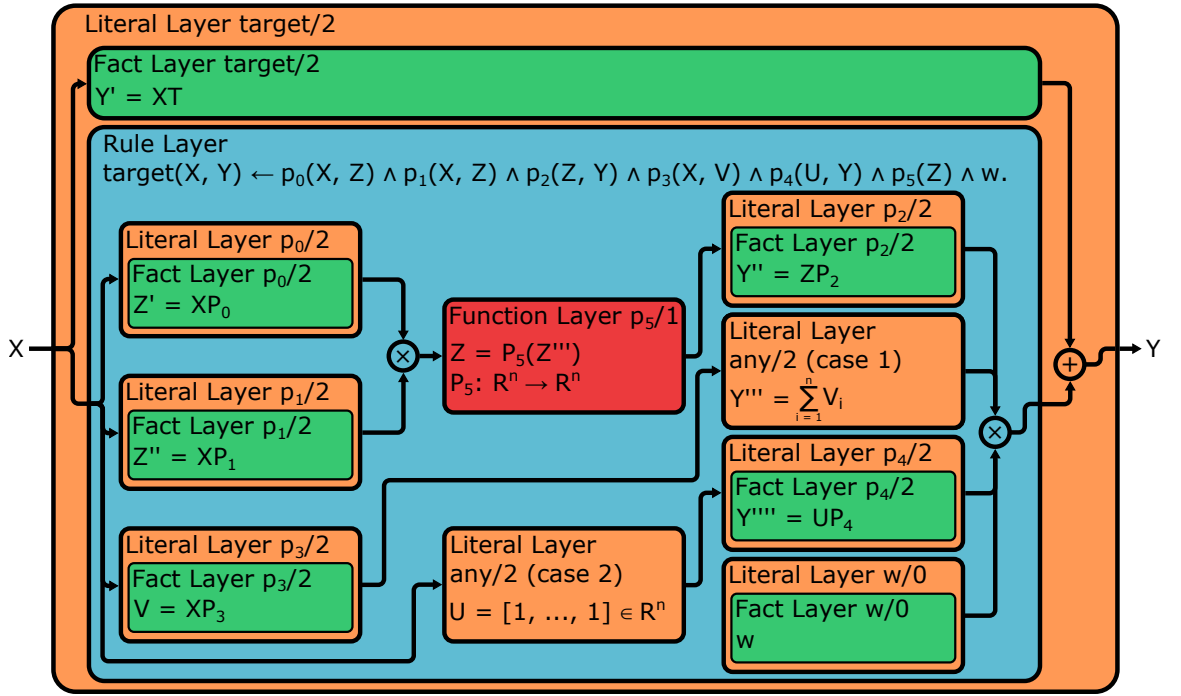


Figure 3.3: NeuralLog Network Example

predicate, as is done for the *target/2* predicate.

In order to construct the whole network described on Figure 3.3, we start by getting the target predicate *target/2* and creating the literal layer for it. Then, we add the fact layer for the *target/2* predicate to its literal layer, in order to account for the facts of the predicate. After that, for each rule containing the *target/2* predicate in the head, we create a rule layer, and we add it to the fact layer. Finally, the input of the network is the input of the fact layer for the *target/2* predicate, while the output of the network is the output of the layer.

In order to create the rule layer, we start from the rule for the *target/2* predicate described in Equation 3.1, the only rule in our example. We apply the Algorithm 2 to the rule, in order to find the paths between the input and output terms of the rule. These paths are then used to construct the DAG of the rule. Then, we use the DAG to construct all the literal in the paths from X to Y , by applying the same procedure to the inner literal on the rule.

In this example, we continue to create the literal layer for the $p_0/2$ predicate, which is composed of the fact layer for the predicate. Analogously, we create the literal layer for the $p_1/2$ predicate and combine the output of both by applying an element-wise multiplication to them, which represents a logic **AND**. After, we construct the function layer for the predicate $p_5/1$, which applies a function to the output of the element-wise multiplication, finally computing the value of the variable Z . Then, we can create the literal layer for $p_2/2$, which will compute one of the components of Y , that is described by this path.

We continue this process until all the layers for the rule are constructed, then we connect all the path to Y , inside the same rule, with an element-wise multiplication, representing the conjunction (**AND**) of the literal in the body of the rule. The results of different rules, for the same predicate in the head, are combined with an element-wise sum, alongside the results of the fact for this predicate, which represents the logic **OR**.

The layer for the case 1 of the *any/2* predicate shows the computation of the $any(V, Y)$ literal by computing $Y''' = \sum_{i=1}^n V_i$. Furthermore, it can be generalized for the *any/n* predicate, to compute the output of the literal $any(X_1, \dots, X_{n-1}, Y)$ by using the formula $Y' = \prod_{i=1}^{n-1} (\sum_{j=1}^m X_{ij})$, where m is the number of distinct entities in the logic program.

In order to learn a task from data, we allow the user to specify predicates whose weights will be adjusted from the examples. In this way, the weights of those predicates will become parameters to be learned in the neural network. Those weights can be adjusted as usual during the learning phase of the neural network and, thereafter, the new values of those weights can be saved back to NeuralLog's logic program.

Saving the weights back into logic form is important for two reasons: (1) it allows us to better interpret the weights learned by the network, by analysing where these weights appear, for example, in which facts and which rules use these facts; and (2) it allows the online learning mechanism to change the learned neural network by changing only the logic theory and still keeping the learned weights that are not involved in the changes, for instance, it can add a new rule with a new fact whose weight will be learned, without changing the weight of facts used by other rules.

NeuralLog is implemented in Python [65] using TensorFlow [66] as backend and its source code is publicly available¹. Using TensorFlow as backend allows NeuralLog to transparently run on different hardware such as CPU and GPU (or other acceleration hardware for neural network) as long as it is supported by TensorFlow.

Although NeuralLog provides a very flexible approach, it has some limitations, for instance, it is unable of handling propositional predicates in the head of the rules, since it would not be possible to specify a source and a destination in order to create the paths. We also limited the use of facts to predicates with arity up to two, since TensorFlow, in its used version (2.0), cannot handle sparse tensors with more than two dimensions, and the use of dense tensors to represent facts with arity greater than two would easily consume too much memory. Although not a negligible limitation, it is also presented in other works such as [7, 14]. On the other hand, predicates with arity greater than two are allowed when used as target predicates, since one can provide more information for each example; and they are also allowed in the rules, to combine different information, facilitating, for instance, the definition of siamese networks [67].

¹<https://github.com/guimaraes13/NeuralLog>

Moreover, different from TensorLog [14], NeuralLog can handle free variables in rules, this is, variables that appear only once in the rule, as described above. This capacity allows NeuralLog to represent more complex logic theories, which may lead to better performance on relational tasks, as we will show in Chapter 4.

We opted to use multiplication to represent the logic **AND** and summation to represent the logic **OR** because, although simple, they provide similar meaning to what we would expect from logic programs.

It is important to point out that our goal is to use a first-order logic language in order to describe a neural network model. For logic programs containing facts with positive weights; rules with binary predicates in their head and bodies that form a path between the input and output variable in the head, and possibly unary predicates on existing variables in the rule; and no recursion; the semantics of the neural network is equivalent to the first-order semantics, considering that the non-zero entries of the resulting vector Y , for the query $? - p(a, Y)$, computed by the neural network, would coincide with the logic entities proved by a logic inference system.

This follows from our design of considering logic **AND** as multiplication and logic **OR** as summation. Given a logic query $? - p(a, Y)$, we want to compute all the entities related to a through relation p . Following our matrix representation, we have that the multiplication of the one-hot row vector representing a by the matrix representing p will result in a row vector with non-zero values in the position of the entities connected to a , assuming all weights are positive and missing facts have zero weight. Analogously, we can compute the result of an arbitrary path rule, by computing the result of the inner variables at each step of the path. Since, in logic, a relation can be proved by any rule or fact, and since we sum the result of the different rules and facts, we retrieve the logic **OR** from logic, given positive weights.

However, NeuralLog is not restricted to this subset of logic programs. Instead, it allows the user to define different types of neural networks, where the logic language is used to define how the entities relate to each other, while the semantics of the network may be different from the semantics of a traditional logic program. This semantics is given by the way the network is constructed, as described in this section, and it is partially learned from the examples and the gradient descent mechanism, which adjusts the weights of the facts.

Finally, we give more details about the NeuralLog language itself in the Appendix A.

3.2 NeuralLog Structure Learning Algorithms

In this section, we present the different algorithms used by NeuralLog in order to learn first-order logic programs from examples.

The syntax of NeuralLog is based on DataLog [31], which is a subset of Prolog [33], a widely used language in the ILP community. There are several structure learning algorithms based on Prolog, or similar languages, such as: FOIL [35], Progol [36], Tilde [68], ALEPH [37], among others. Each of these algorithms implements a different way of navigating through the hypotheses space, in order to find the best logic theory to describe a set of examples, given the background knowledge.

In addition to traditional structure learning algorithms, which find rules to describe examples, given the background knowledge, there are theory revision algorithms, which are able to modify existing rules in the background knowledge, in order to better describe the set of examples. FORTE [40] is a well-known theory revision algorithm that revises theories by identifying the *revision points* and applying *revision operators* to these points. Other works propose to improve FORTE; some examples of these works are: FORTE-MBC [45], which uses the concept of Bottom Clause [36] to define the hypotheses space for FORTE; and YAVFORTE [46], which employs stochastic search for FORTE.

Some of those structure learning algorithms can be easily integrated to NeuralLog, in order to find logic theories in the NeuralLog language, then, these theories are compiled to neural network models. We ported two of those algorithms to NeuralLog, namely: Metagol [7], a system that uses a higher-order logic program in order to limit the hypotheses space and guide the search of the solution; and Online Structure Learner by Revision (OSLR) [12], a system designed to learn a logic theory in an online manner, where the examples are arriving over time.

Moreover, we propose a novel learning algorithm, by combining Metagol and OSLR. It relies on the online learning algorithm of OSLR, but uses Metagol's higher-order logic mechanism in order to search for the possible revisions of the theory.

We give more details about each one of these three structure learning algorithms in the subsections below.

3.2.1 Meta-Interpretive Learning

We first introduce the NeuralLog+MIL, a structure learning algorithm based on Metagol [7], which is a Meta-Interpretive Learning system.

Meta-Interpretive Learning is a method that learns first-order logic theories by the use of a higher-order logic theory that will define the hypotheses space and guide the search of the hypothesis in this space [7].

In first-order logic, the predicate names in the rules, which represent the relations between the logic entities, are constant. In higher-order logic, those predicate names might be variable,

and the logic inference system should find the substitution of these names in order to prove the rule.

Metagol uses a modified Prolog meta-interpreter that generates a first-order logic theory that proves the positive examples, without proving the negative ones, given the background knowledge. It does it by traversing a higher-order theory in a similar way a SLD-Resolution algorithm would do [33].

NeuralLog+MIL takes a slightly different approach: instead of adding rules to the first-order theory as needed, it creates a first-order theory by adding all possible rules that satisfy the higher-order program, given a pre-defined depth. Each rule will then receive an associated weight and an activation function.

This approach is better suited to be integrated with NeuralLog, since a single neural network is created and the task to find the weight of the rules is passed to the neural network optimization process. If we had followed Metagol’s approach, we would have to create and evaluate several intermediary neural networks, which would be more computationally expensive.

Given a higher-order theory and a target predicate p , NeuralLog+MIL creates a “meta” SLD-Resolution tree, starting with a list of goals $[g]$, replacing its terms by distinct variables, as the root node. Then, for each node in the tree, a child node is created by applying a meta-clause that unifies with one of the goals in the node; in the same way a conventional SLD-Resolution method would. The new child node is created with the list of goals from the parent node, with the goal whose rule was applied replaced by the body of the unified meta-clause.

We grow this tree, breadth-first, by applying all possible meta-clauses to all goals in all nodes until we reach the maximum depth. Each edge represents the application of a meta-clause whose head was unified to the used goal in the parent node. For each path from the root to a leaf, we have a meta-program with variable predicates to be instantiated.

Finally, for each meta-program we generate a first-order program by replacing the set of variable predicates for each possible predicate in the knowledge base. The final program is the concatenation of all generated clauses in all the programs.

For instance, consider the higher-order theory in Table 3.2 and a target predicate $p/2$. The tree would start with the root node containing the goal $[p(X, Y)]$, referred here as level 0. By applying each clause to the goal of the root node, we would end up with two nodes at level 1: the node $[Q(X, Y)]$, generated by the unified clauses $p(X, Y) \leftarrow Q(X, Y)$.; and the node $[Q(X, Z), R(Z, Y)]$, generated by the unified clause $p(X, Y) \leftarrow Q(X, Z) \wedge R(Z, Y)$. After the addition of both nodes, the level 1 would be complete.

Table 3.2: Higher-order Logic Theory

- (1) $P(X, Y) \leftarrow Q(X, Y)$.
- (2) $P(X, Y) \leftarrow Q(X, Z) \wedge R(Z, Y)$.

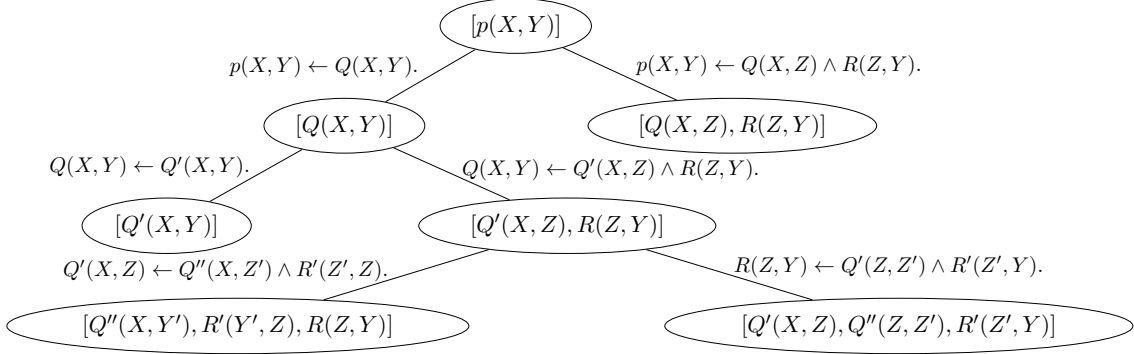


Figure 3.4: Meta SLD-Resolution Tree

If we would like to go further in the resolution, we could apply the meta-clause to each goal in the nodes and keep adding new nodes until a pre-defined depth. Figure 3.4 shows an example of part of a tree until depth 3; where some nodes were omitted for clarity.

3.2.2 Online Structure Learner by Revision

Online Structure Learner by Revision (OSLR) [12] is a system developed to learn logic theories in an online fashion, originally based on ProPPR [64]. It relies on theory revision techniques in order to adapt an existing theory to cope with new arriving examples. In this subsection, we present NeuralLog+OSLR, our implementation of OSLR to learn theories in the NeuralLog language.

The top-level revision algorithm implemented by OSLR is as follows: when new examples arrive, they are placed into a tree structure that represents the logic theory; then, a revision is proposed to the point of the theory that has the biggest potential to bring a gain for the theory; after, the revision is evaluated against an accepting criterion; finally, the revision is either accepted or rejected and the algorithm evaluates the next revision, until no more revision points are changed by the current examples, when the algorithm waits for further examples.

NeuralLog+OSLR follows the OSLR top-level algorithm with minimal changes, in order to make it better suited for neural networks. In the remainder of this subsection, we resumed the OSLR algorithm implemented by NeuralLog+OSLR, in order to keep this work self-contained. In addition, we point out the differences between NeuralLog+OSLR and the original OSLR. However, we refer the reader to [11, 12], for a more detailed explanation

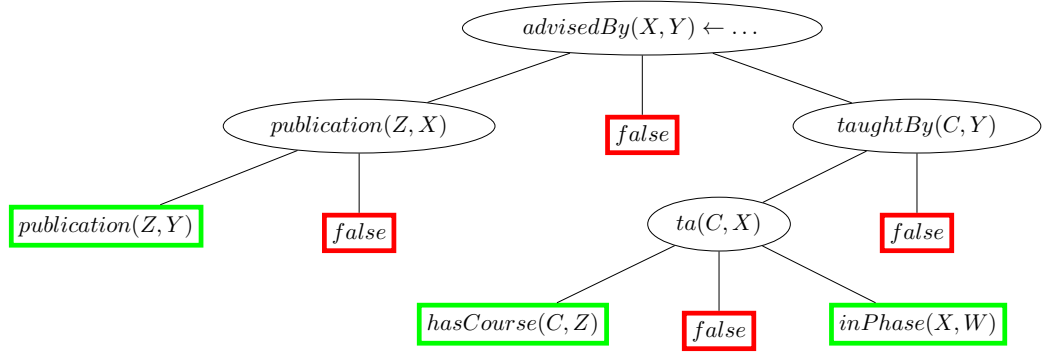


Figure 3.5: Tree Structure Representation of the UWCSE Theory Example

Table 3.3: Theory Example for the UWCSE Dataset

$$\begin{aligned}
 & \textit{advisedBy}(X, Y) \leftarrow \textit{taughtBy}(C, Y) \wedge \textit{ta}(C, X) \wedge \textit{hasCourse}(C, Z). \\
 & \textit{advisedBy}(X, Y) \leftarrow \textit{taughtBy}(C, Y) \wedge \textit{ta}(C, X) \wedge \textit{inPhase}(X, W). \\
 & \textit{advisedBy}(X, Y) \leftarrow \textit{publication}(Z, X) \wedge \textit{publication}(Z, Y).
 \end{aligned}$$

about OSLR.

3.2.2.1 Data Representation

Online Structure Learner by Revision (OSLR) starts by constructing a tree representation of a (possibly empty) theory for each target predicate. This tree structure represents all the rules in the theory, whose head predicate is equal to the target predicate. The root of the tree represents the head of the rules for the target predicate, while the other nodes represent the literals in the body of the rules. The level immediately after the head represents the literals in the first position in the body of the rules, and there will be a node for each different literal in the first position. The next level represents the literal in the second position and so on. For each internal node in the tree (nodes that are not leaves), a default *false* node is appended as child. The *false* nodes are always leaves.

This tree structure plays two roles: the first one is to identify revision points on the theory; the second, is to store the examples that shall be used to revise those points. Figure 3.5 shows an example of the tree representation of the theory shown in Table 3.3.

Each path from the root of the tree to a non-*false* leaf represents a rule in the theory. Rules that are a subset of another rule are not considered. The leaves in the tree represent the possible revision points and are shown as squares in Figure 3.5.

When a new example, for the target predicate, arrives, we pass it through the tree to decide where it will be placed. The example starts in the root, then it is recursively passed through

the nodes in the tree as describes: for each node u in the children nodes of the current node v , if the (partial) rule from the root to u proves the example, we pass the example down to u . If the example is not proved by any of the children nodes of v , it is placed in the false node connected to v . This process is repeated until the example reaches the leaves of the tree. If the example is proved by more than one child node, it goes to all the nodes that prove it.

3.2.2.2 Theory Revision

After placing the arrived examples in the correspondent leaves, all the leaves that received examples are candidates to be revised. OSLR uses a heuristic function in order to sort the revision points to prioritize the ones that may have the bigger impact in the evaluation of the theory. This heuristic is simply the number of misclassified examples in the leaf. The number of misclassified examples is the number of positive examples, in the case of a false leaf; or the number of negative examples in the case of a non-false leaf.

In order to propose the revision to the theory, it uses revision operators that are applied to the revision points. There are two possible operations to revise the tree: adding new nodes to the tree; or removing nodes from the tree. OSLR applies all the possible operators to each revision point and uses the examples contained in the point in order to evaluate the revision on the theory.

Adding Node. It can be applied to both false and non-false leaves. When applied to a false leaf, the new nodes are used in order to generate a new rule that starts from the root until the parent of the false leaf. This new rule will be added to the theory in order to make the theory more generic and it is an attempt to prove positive examples that fell in the false leaf. On the other hand, adding node to a non-false leaf extends the path from the root to the new leaf, thus, extending the rule and making the theory more specific, which is an attempt to avoid proving false examples in the non-false leaf. There are two algorithms to select the nodes to be added, both relying on the concept of the bottom clause [36]: the hill-climbing, which tries to add a candidate literal at a time, until certain stop criteria is met; and the relational path-finding [69], which tries to find a path between the variables of the example.

Deleting Node. It can be applied to non-false leaves or to false leaves whose parent has a single non-false child and this non-false child is also a leaf; this approach is described as *Alternative 1* in [11]. When applied to the false leaf, it deletes the literal represented by the sibling node, in an attempt to make the theory more generic, in order to prove the positive examples in the false leaf. When applied to a non-false leaf, it deletes the rule represented

by the leaf, in an attempt to make the theory more specific and to avoid proving negative examples.

After applying all the possible revision operators to a given revision point, the operator that better improves the performance of the theory, given the examples in the revision point, is selected to be evaluated against the acceptance criterion, which will be discussed in the following subsection.

The adding node operator is actually two distinct revision operators, one for the hill-climbing and another for the relational path-finding. Thus, alongside the deletion operator, they all compete among each other and the one that achieves the best result for the revision point is selected.

3.2.2.3 Accepting the Revision

Once the operator that has the biggest metric in a given revision point is selected, OSLR uses a threshold to decide if the improvement of the revised theory over the current theory is significant.

This threshold is based on the Hoeffding's bound [70] and is given by the equation:

$$\epsilon = \sqrt{\frac{R^2 \ln 1/\delta}{2n}} \quad (3.3)$$

where R is the size of the range of the given metric, n is the number of examples used in the evaluation and δ is a parameter defined by the user. An improvement larger than ϵ means that the probability of the revised theory to be actually better than the current theory is $1 - \delta$.

The examples used to evaluate the theories are the ones in the revision points. If the improvement of the revised theory over the current one is greater than ϵ , the revision is accepted and the examples used to evaluate it are discarded. Otherwise, the revision is discarded and the examples are unchanged. After either case, the algorithm continues to try to revise the remaining revision points, if there are any revision points left to be revised.

3.2.2.4 Clause Modifiers

In OSLR, after a revision is accepted, a feature, in the ProPPR language [64], is generated for the rule that was modified by the revision. This is the point that the implementation of NeuralLog+OSLR and OSLR differs the most.

NeuralLog language does not support the ProPPR features, although, in some cases, they might be similar to the addition of a literal to the rule whose weight should be learned by

the neural network.

As such, instead of computing the features in the same way OSLR would do, which would select a subset of terms in the rule to have associated weights to be learned, we create a unique weight for each rule, which is learned by the neural network and is independent of the instantiation of the terms in the rule. Our experiments, using OSLR, showed that the difference between this approach and the original one is minimal, for the used datasets.

The addition of the weight is done by a clause modifier, which appends the weight to the body of the revised rules. These weights are represented in the form of a literal, with a unique constant for each rule; and this literal is marked as learnable in the NeuralLog language. More details of how to mark a predicate to be learned by the neural network can be found in Appendix A.

In addition, to append a literal to the rule with a unique constant, we have two more clause modifiers that are useful for the neural network construction.

The first one is a clause modifier that appends a literal to the rule with a term from the head of the rule. This modifier is used to append an activation function for the rule, whose term must be the last variable in the head of the rule.

The other modifier changes the predicate name of the head of the rule to another name, by appending a suffix at the end of the name. This modifier is useful to learn a set of rules that indirectly proves the examples.

For instance, suppose one wants to learn examples from the predicate $p/2$ without changing rules that have the $p/2$ predicate in the head. One could add the rule $p(X, Y) \leftarrow p_1(X, Y)$. to the background knowledge and use a clause modifier to change the head of the rules, learned from the examples, from $p/2$ to $p_1/2$.

This is specially useful in the definition of neural networks, because it allows the user to isolate the learned part of the theory and to add neural network components around it. The use of those clause modifiers will become clearer in our experiments in Chapter 4.

3.2.3 Online Meta-Interpretive Learning

In this work, we propose a novel approach by combining the MIL hypotheses search strategy with the online learning mechanism from OSLR. We call this approach **Online Meta-Interpretive Learning (OMIL)** and its NeuralLog variation, NeuralLog+OMIL.

In order to achieve such a goal, we created a new revision operator that uses the MIL search strategy to propose new nodes in the OSLR tree representation. Then, we used the same NeuralLog+OSLR machinery, replacing the three original revision operators by our MIL

revision operator.

In order to reuse NeuralLog+OSLR machinery, we have to slightly adapt the NeuralLog+MIL algorithm, so it can work with the OSLR tree structure.

The first difference between NeuralLog+OMIL and the NeuralLog+MIL is that NeuralLog+OMIL does not return a concatenation of all the programs generated by the higher-order theory, as NeuralLog+MIL does. Instead, it evaluates a modified version of the current theory, with each set of clauses generated by each node from the meta SLD-Resolution tree; and it returns the first modified theory whose evaluation is greater than the Hoeffding's bound threshold, if such theory exists. It uses a breadth-first search to grow the tree and evaluates the sets of clauses in the order they are generated.

The second difference between both systems is that NeuralLog+OMIL is applied to a node in the tree representation structure, and not to the income example itself. In this way, the first clause generated by the MIL operator should be a rule of the form $p(.) \leftarrow \dots$, where $p(.)$ is the node in the tree where the operator was applied.

Instead of adding all the clauses generated by the operator, we get the body of the first rule and use its literals as the proposed node to be appended to the tree, as does the adding nodes operators from NeuralLog+OSLR. In the same way as NeuralLog+OSLR, if the operator is applied to the false leaf, it uses the new nodes to create a new rule whose body starts with the path from the root to the parent of the false leaf node in the tree. On the other hand, if it is applied to a non-false leaf, it uses the new literal to replace the literal represented by the leaf.

In order to give the OMIL operator more information, we change the variables of the target atom, depending on the situation.

Root node. When the operator is applied to the false leaf of the root node, we use the predicate of the example as target atom; in this case, the operator tries to learn a rule to directly predict the examples.

Propositional literal. When the literal has no variables, the target atom is a special predicate name, only used inside this operator, with the same variables as the head of the rule. This will inform the operator the input and output variable of the rule.

Literal connected to the output. When the target literal is connected to the output, it will be the target atom. In this way, the same terms of the target literal will be used by the operator.

Literal disconnected from the output. When the literal is not connected to the output, the target atom will have the special predicate name, and the variables will be the input variable of the disconnected literal and the output variable of the rule. As such, the operator will be able to find a body which connects the input of the literal to the output of the rule, closing this path.

These modifications to the target atom gives the OMIL operator enough information in order to find meaningful rules: (1) having the information about the input and output variables of the rule in the first two cases; (2) the information about the final part of a path in the rule in the third case; and (3) the information about an open path and the output variable of the rule, in the last case.

How this information is used will depend on the higher-order theory. However, it allows the user to define meta-rules that might: (1) create new paths, (2) replace the final part of an existing path, or (3) close an existing open path; respectively.

The remaining clauses generated by the operator, for a given node, are added to the theory, as usual. It only happens if the depth of the application of the meta-clauses is bigger than 1. These additional clauses must be added to the theory, because they might define predicates that were invented by previously generated clauses.

Finally, the third difference between the systems is that we add two special rules to the higher-order theory:

$$\begin{aligned} P(X,Y) &\leftarrow true. \\ P(X,Y) &\leftarrow false. \end{aligned} \tag{3.4}$$

These special rules are used in order to get the deletion behaviour from NeuralLog+OSLR.

The *true* literal is a special literal that is always true. Since the body of a rule in a conjunction of literals, in order words, represents a logic **AND** between the literals, the addition of a literal that is always true does not change the result of the conjunction, as such, it can be removed from the rule. When the true rule is applied to a literal, the literal is replaced by the true literal, that is not added to the rule, since it will have no effect on it, thus, the application of this rule represents a literal removal.

On the other hand, the *false* literal is a special literal that is always false. A rule whose body includes a *false* literal will always fail to be proved, as such, it can be removed from the theory. Thus, the application of the false rule to append a literal in the body of a rule will result in the deletion of the rule from the theory.

Both rules only have effect when applied to a non-false leaf. Applying it to a false leaf would not have any affect. For the true rule, it would generate a rule whose body is a subset of another rule, which is not allowed by the OSLR algorithm. For the false rule, it would

result in an attempt to create a new rule with the *false* literal in the body, which would be discarded and the theory would remain the same.

After the proposal of the modification of the theory, by the operator, the clause modifiers are applied to the modified rule as usual, which will be the rule formed by the existing tree and the literal in the body of the first clause generated by the higher-order theory.

Consider again the tree shown in Figure 3.5 and the rule $P(X, Y) \leftarrow Q(X, Z) \wedge R(Z, Y)$. shown in Table 3.2.

If the rule is applied to the root node, by trying to prove the examples in its false leaf, the target atom would be $advisedBy(X, Y)$ and the operator will propose a new rule in the form $advisedBy(X, Y) \leftarrow Q(X, Z) \wedge R(Z, Y)$., where Q and R will be replaced by two binary predicates from the knowledge base.

If the rule is applied to the leaf $publication(Z, Y)$, the target atom would be the literal itself and the operator will propose a rule of the form $publication(Z, Y) \leftarrow Q(Z, Y') \wedge R(Y', Y)$. In this case, the literal will be replaced by the body of the proposed rule, changing the rule $advisedBy(X, Y) \leftarrow publication(Z, X) \wedge publication(Z, Y)$. to $advisedBy(X, Y) \leftarrow publication(Z, X) \wedge Q(Z, Y') \wedge R(Y', Y)$. Once again, Q and R will be replaced by two binary predicates from the knowledge base.

Finally, if the rule is applied to the leaf $hasCourse(C, Z)$, the target atom would be $new_pred(C, Y)$. The term C in the target atom is because it is the input of the literal, given by the rule $advisedBy(X, Y) \leftarrow taughtBy(C, Y) \wedge ta(C, X) \wedge hasCourse(C, Z)$., and the term Y is because it is the output of the rule, this is, the last variable of the head. The operator will propose a rule of the form $new_pred(C, Y) \leftarrow Q(C, Z') \wedge R(Z', Y)$. where the body would replace the $hasCourse(C, Z)$ in the original rule. This will give the higher-order theory the chance to close the path from Z to Y , which is open in the current rule.

As in the previous cases, the higher-order predicates Q and R are replaced by first-order predicates from the knowledge base. The permutations are evaluated in no particular order, and the first one to exceed the Hoeffding's bound threshold is returned, if any; otherwise, the theory is kept unchanged.

3.3 Discussion

In this section, we show how NeuralLog relates to other systems that use logic language to generate neural networks, and to systems that learn logic structures that are used as neural networks.

Differently from systems such as [50, 52], NeuralLog does not rely on propositionalization in order to create the neural network, instead, the neural network is directly created based on the first-order rules and facts. Furthermore, NeuralLog generate a single neural network for the task, independently of the example at hand. In addition, a single neural network can be used to predict more than one target relation, and weights might be shared among different (target) relations.

NeuralLog uses the same underlying idea of systems like TensorLog [14], by representing logic facts as matrices and performing mathematical operations to compute logic rules. However, it differs from TensorLog in two main aspects: (1) in the way the logic rules are compiled to a neural network; and (2) by treating numeric attributes as logic terms that can be flexibly manipulated and also given as input to numeric functions. This allows NeuralLog to be more flexible than TensorLog.

More precisely, NeuralLog allows the use of rules containing free variables (variables that appear only once in the rule). This might be important for some tasks, as we will show in our experiments on Chapter 4. Moreover, NeuralLog allows the definition of numeric attributes and the use of numeric functions in the logic language itself, as well as defining all the relevant hyperparameters for the training of the neural network, while TensorLog requires the user to configure some of those hyperparameters directly into the Python program [71].

NeuralLog is quite different from LTN [57]; in NeuralLog, the facts are represented as matrices and the entities are one-hot vectors (vectors that have the value of 1 in a single position, and the value 0 in all other positions), where the goal is to find the weights of the facts that better describe the examples. On the other hand, LTN represents the entities as dense vectors and the predicates are represented as functions on those vectors, where the goal is to find the parameters of these functions (and possibly the numeric representation of the entities) that better describe the examples.

DeepProbLog [59] has its own probabilistic logic inference mechanism and integrates it with neural networks. Differently from NeuralLog, which creates a neural network from a logic program and the whole inference is performed by the neural network itself. Thus, NeuralLog does not follow Sato’s semantics of possible worlds used by DeepProbLog.

Differently from the mentioned works, that are based on a logic system and only change the interpretation of the program, NeuralLog brings additional features to the logic language and its meaning, allowing a better integration between the logic and the neural network parts. These features include the capability of handling predicates with numeric values in it and new syntaxes to define similar rules by using a for-loop or a template rule that is instantiated based on other predicates, as we described in Section 3.1. We will show an example of these features on Subsection 4.1.2.

The addition of these features makes the NeuralLog language a superset of DataLog [31],

which means that NeuralLog accepts DataLog programs (as long as the maximum arity of the facts is two), but the users can also take advantage of these special features. This similarity with DataLog, which is a well-known logic language, allows us to port other ILP algorithms to NeuralLog with minimal changes.

Systems such as ∂ ILP [61] and Neural-LP [8] are related to the structure learning approaches from NeuralLog, since they also learn the structure of logic programs which are inferred as neural networks. In particular, ∂ ILP generates a set of rules, defined by a template, and learns the weights of the rules using gradient descent, which is similar to NeuralLog+MIL for NeuralLog, but the semantic of the neural network is different from ∂ ILP, for instance NeuralLog allows a broader set of logic programs, including recursive programs and predicates defined by both rules and facts, which ∂ ILP does not support.

Another strong feature of NeuralLog is that it supports the definition of rules that can combine logic predicates with other neural network structures. In order to achieve this, one can use background knowledge containing such rules. Neither Neural-LP [8] nor ∂ ILP [61] take advantage of this feature, since they use their own generated theory, not accepting a pre-existing one.

Finally, our main goal with NeuralLog was to propose a language to better integrate first-order logic with neural networks, which motivated us to, on one hand, follow the DataLog syntax to better integrate the logic side, but also, on the other hand, to allow the use of numeric attributes and to call numerical functions through the use of logic predicates to better integrate it to the neural network side. This integration is corroborated by our experiments, where we used logic programs to describe neural networks for relational and propositional learning; and the structure learning algorithms also learned logic rules to describe neural networks in NeuralLog.

Chapter 4

Experiments & Results

In this chapter we present the experiments that support our work. We divide the experiments into two main sections: parameter learning; and structure learning. Following the division presented in Chapter 3.

In the first section, we perform different experiments and show some applications of NeuralLog, assuming that a logic theory exists, for each dataset. Only the parameters of the theory will be learned by NeuralLog, through gradient descent optimization of the neural network, in order to better describe the set of examples.

In the next section, we give a step further and apply NeuralLog to learn both the logic theory and the parameters of the theory, to describe the set of examples, given background knowledge. There are two setups for these experiments, a batch (offline) learning scenario, where all the training examples are given at the beginning of the process; and an online learning scenario, where training examples are arriving over time and the model must adjust itself to fit the new examples.

4.1 Parameter Learning

In this section, we present the parameter learning experiments performed with NeuralLog, in order to demonstrate its flexibility to represent different kinds of models. It is worthy to point out that the goal of the experiments is not to find the best possible model for each task, but to show how NeuralLog can represent different types of models.

Nevertheless, one could manually implement the presented models, without using NeuralLog. However, we show that NeuralLog is a powerful and flexible language to define different neural network models, specially for relational learning tasks, abstracting much of the complexity of defining the neural network structure and the data processing pipeline. Furthermore, the

use of a logic language to abstract the design of neural network structures could facilitate the use of such models by people less familiarized with common programming languages.

4.1.1 NeuralLog in Comparison with Other Logic-based Systems

In order to demonstrate the flexibility of NeuralLog, we compared it with two distinct relational systems that use first-order logic to create relational neural networks. In these experiments, we would like to answer two main research questions, in order to address the flexibility of NeuralLog: **(Q1)** can NeuralLog represent *link prediction* models? And **(Q2)** can NeuralLog represent *classification* models?

In order to answer each question, we compared NeuralLog with other similar systems proposed specifically for each of the above type of tasks. The first system is TensorLog [14], which uses belief propagation to compile logic theories into neural networks. The second one is RelNN [15], which learns relational neural networks that use the number of instantiation of logic rules as features to predict the value of a logic atom with a given entity.

TensorLog and RelNN are two distinct systems that use logic to build neural networks. On one hand, TensorLog focus at link prediction, by answering queries of the form $q(a, X)$, where the goal is to find all the entities that are related (as second term) to the entity a (as first term) through relation q . On the other hand, RelNN focus at classification tasks, by predicting the value of relations of the form $q(X)$, for a given entity a , based on the relations of a , defined by a set of logic rules.

We compared NeuralLog with TensorLog and RelNN, in particular, because in those systems the structure of the neural network clearly reflects the logic theory, and it makes them closely related to NeuralLog. All the three systems (NeuralLog, TensorLog and RelNN) use a logic theory in order to define the neural network structure and fine-tune the weights of the network to better predict the set of examples.

First, we compared NeuralLog with TensorLog. Since TensorLog predicts the values of the entities related to a given entity, we ran link prediction experiments in three datasets: the Cora dataset [16], and the UWCSE dataset [17], two popular datasets among the ILP community; and the WordNet dataset [18], a frequently used benchmark dataset for link prediction (the version from [8], since it was already properly divide into *train*, *validation* and *test* sets).

The link prediction task is the task of predicting the entities that relate to a given entity, through a target predicate [72]. To perform this task, we have a set of examples in the form $q(a, x)$, where we give the first entity a to the system in order to predict the last entity x .

Afterwards, we compared NeuralLog with RelNN in classification tasks in two different

datasets: the Yelp dataset; and the PAKDD15 dataset. Both datasets were available with RelNN [15]. For the Yelp dataset, the task is to predict whether a restaurant serves Mexican food through the *mexican/1* predicate; while, in the PAKDD15 dataset, the task is to predict the gender of a person, based on the movies the person watched, through the *male/1* predicate.

4.1.1.1 Methodology

In order to perform a fair comparison between the systems, we used the same logic theory available with the compared systems for the respective datasets. In our system, for the Cora and for the WordNet datasets, we applied an activation function to the output of the rules, and also added a bias to the target predicate; then, applying an output function to the weighted sum of the rules and the bias for the final result. Those functions act as activation and output functions, respectively, from a conventional neural network.

The original theories for the Yelp and PAKDD15 datasets already included the sigmoid function as activation and output functions and already had biases.

In the case of the UWCSE, we have used a manually written theory provided by Alchemy¹. Since Alchemy supports a more complex logic language than both NeuralLog and TensorLog, we removed the rules that use logic features not supported by the compared systems, resulting in a theory with 8 clauses. The UWCSE dataset also contains ternary relations (of arity 3), which are not supported by either the compared systems. We converted them into binary relations by concatenating two terms that always appear together in the theory. We also added two additional predicates to extract either term, given the concatenated form.

A bias rule has the form $target(X, Y) \leftarrow b$, where b is a fact to be learned. Since the variables of the head do not depend on its body, the rule is always true. Then, the rule is summed to other rules with the same head, acting as a bias.

We used adagrad to optimize the mean square error with a learning rate of 0.1 (except for UWCSE, which used a learning rate of 0.01) and no regularization, running for 16 epochs for the Cora and UWCSE datasets; and for 10 epochs for the WordNet, Yelp and PAKDD15 datasets. We set the recursion depth to 1, meaning that recursive rules are applied only once to itself.

The depth of the models generated from NeuralLog may not be directly compared with traditional neural networks. However, we can have an approximated meaning of the depth of a neural network, by defining the depth of a NeuralLog network as follows: the depth of an atom is equal to the maximum depth of the rules which have the predicate of the atom in its head; the depth of a functional atom or an atom without variables is 0, in the case of

¹<http://alchemy.cs.washington.edu/>

Table 4.1: Maximum Relation Depth

Relation	Depth	Relation	Depth
Cora (each target relation)	6	WordNet	2
Advised by	4	Yelp	2
Advised by sim	2	PAKDD15	3

Table 4.2: Size of the Datasets

Dataset	Facts	Train		Test	
		Pos	Neg	Pos	Neg
Cora	44,711	28,180	24,104	7,318	3,660
WordNet	106,088	35,354	53,732	5,000	9,098
UWCSE	3,146	90	49,622	23	3,320
Yelp	45,541	2,160	1,368	540	342
PAKDD15	127,420	5,301	18,702	1,325	4,676

these experiments, where the functional predicates are simple activation or output functions; the depth of an atom only defined by facts is 1; and, finally, the depth of a rule is the sum of the depth of the literals in its body.

Table 4.1 shows the maximum depth of each relation, following our definition of the depth of a NeuralLog network.

We ran each experiment 10 times and reported the average metrics of the runs. We used holdout with the Cora and the WordNet datasets; and 5-folds cross-validation with the UWCSE, Yelp and PAKDD15 datasets; following the way in which each dataset was split. In order to evaluate the results, we used the area under the ROC curve and the average precision² in order to evaluate the models.

For TensorLog and RelNN, we used the parameters reported at their papers [14] and [15], respectively. Since TensorLog had no parameters for the UWCSE, we experimented using the same parameters as Cora, for a different number of epochs, with and without l2 regularization. The best results were obtained by running for 16 epochs without regularization.

In the WordNet dataset, we used 25% of the facts in the train set as training examples

²The average precision is a way of summarizing the Precision-Recall curve, but it can be less optimistic than the area under the curve https://scikit-learn.org/stable/modules/generated/sklearn.metrics.average_precision_score.html

Table 4.3: Average Precision for Cora, UWCSE and WordNet Datasets

Relation	NeuralLog	NeuralLog σ	TensorLog	Baseline
Same Author ind	1.0000 \pm 0.0000	0.9984 \pm 0.0001	0.9914 \pm 0.0000	0.9053
Same Bib ind	<u>0.9556 \pm 0.0158</u>	0.9043 \pm 0.0202	0.9614 \pm 0.0000	0.6827
Same Title ind	0.8907 \pm 0.0003	0.9184 \pm 0.0002	0.0000 \pm 0.0000	0.5099
Same Venue ind	0.8060 \pm 0.0081	0.7866 \pm 0.0056	0.0000 \pm 0.0000	0.5238
Same Author	0.9939 \pm 0.0062	0.9996 \pm 0.0007	0.9854 \pm 0.0004	0.9053
Same Bib	0.9654 \pm 0.0034	0.9476 \pm 0.0066	0.9058 \pm 0.0319	0.6827
Same Title	<u>0.8946 \pm 0.0083</u>	0.9295 \pm 0.0026	<u>0.8779 \pm 0.0176</u>	0.5099
Same Venue	0.7992 \pm 0.0056	0.7512 \pm 0.0133*	<u>0.7256 \pm 0.0072*</u>	0.5238
Advised by	0.2058 \pm 0.0076	0.1699 \pm 0.0067	0.1037 \pm 0.0000	0.0075
Advised by sim	0.2272 \pm 0.0061	<u>0.2232 \pm 0.0061</u>	0.1037 \pm 0.0000	0.0075
WordNet	0.5654 \pm 0.0004	0.4664 \pm 0.0230	0.5661 \pm 0.0001	0.3541
WordNet (un)	<u>0.6048 \pm 0.0005</u>	0.5039 \pm 0.0298	0.6054 \pm 0.0003	0.3749

and the remaining as background knowledge. Since WordNet has no negative examples, we artificially generated approximately 2 negative examples for each entity appearing in the first position of a target relation. We did this by replacing the second entity of each example by another entity appearing in the same relation, which does not appear as positive example for the first entity, following the Local Closed World Assumption (LCWA) [73]. At each run, the negative examples were resampled.

The size of the datasets can be seen in Table 4.2. The UWCSE, Yelp and PAKDD15 lines show the average size of the cross-validation folds.

4.1.1.2 Results

We applied NeuralLog and TensorLog to each relation of the Cora dataset and to the UWCSE dataset. Tables 4.3 and 4.4 show the *average precision* and the *area under the ROC curve*, respectively, averaged for the 10 runs of each dataset.

The *NeuralLog* column shows the results using tanh as both activation and output functions, while the *NeuralLog σ* uses sigmoid and softmax as activation and output functions, respectively. The *baseline* column represents a dumb classifier that predicts 0 for every example. For each line, the highest value is bold-faced; and pairs of underlined values, or values followed by *, means that there is no statistical-significant difference between the marked values, based on the two-tailed paired t-test with $p < 0.05$.

Table 4.4: Area Under the ROC curve for Cora, UWCSE and WordNet Datasets

Relation	NeuralLog	NeuralLog σ	TensorLog	Baseline
Same Author <i>ind</i>	1.0000 \pm 0.0000	0.9838 \pm 0.0008	0.9276 \pm 0.0000	0.5000
Same Bib <i>ind</i>	0.9120 \pm 0.0179	0.8361 \pm 0.0174	0.9339 \pm 0.0000	0.5000
Same Title <i>ind</i>	0.8712 \pm 0.0003	0.9074 \pm 0.0002	0.0000 \pm 0.0000	0.5000
Same Venue <i>ind</i>	0.7699 \pm 0.0081	0.7370 \pm 0.0374	0.0000 \pm 0.0000	0.5000
Same Author	0.9432 \pm 0.0552	0.9957 \pm 0.0073	0.8863 \pm 0.0020	0.5000
Same Bib	0.9261 \pm 0.0056	0.9154 \pm 0.0056	0.8376 \pm 0.0545	0.5000
Same Title	0.8771 \pm 0.0118	0.9296 \pm 0.0032	0.8351 \pm 0.0309	0.5000
Same Venue	0.7637 \pm 0.0069	0.6589 \pm 0.0059*	<u>0.6933 \pm 0.0071*</u>	0.5000
Advised by	0.9527 \pm 0.0013	0.9434 \pm 0.0025	0.7107 \pm 0.0000	0.5000
Advised by <i>sim</i>	0.7115 \pm 0.0002	0.7501 \pm 0.0005	0.7107 \pm 0.0000	0.5000
WordNet	0.6603 \pm 0.0002	0.5516 \pm 0.0301	0.6609 \pm 0.0001	0.5000
WordNet (<i>un</i>)	<u>0.6865 \pm 0.0003</u>	0.5429 \pm 0.0368	<u>0.6868 \pm 0.0001</u>	0.5000

The Cora relations ending with *ind* are the results when running the system for each target relation individually, while the others are the results when jointly learning the four relations of the Cora dataset. When learning each relation individually, TensorLog can only outperform NeuralLog for the *Same Bib* relation, jointly learning the four relations, NeuralLog outperforms TensorLog for all tests but the *Same Venue* relation for the NeuralLog σ model, however, without statistical significance.

It is worthy to point out that TensorLog was not able to individually learn the *Same Title* nor the *Same Venue* relations in any of the 10 runs. When jointly learning the relations, TensorLog was able to learn in only 8 and 2 out of the 10 runs, for the *Same Title* and *Same Venue* relations, respectively. We reported the average of the successful runs.

In the UWCSE dataset, TensorLog was not able to run using the full theory, since it contains some rules with free variables and rules whose output variables do not have inputs, such as $advisedBy(X1, X2) \leftarrow student(X1) \wedge professor(X2)$. Thus, we had to create a simplified theory, by removing those rules from the theory and keeping only simpler rules. The results of this theory are in the lines *Advised by sim*. We repeated the TensorLog results, with the same simplified theory, in the lines *advised by* for easy comparison.

It is interesting to notice that NeuralLog achieved an average precision of 0.2272 ± 0.0061 for the UW-CSE dataset, which is greater than the area under the precision-recall curve of 0.215 ± 0.0172 , reported by [17] for a Markov Logic Network (MLN) model. MLN is method that transform a logic background knowledge, containing weighted clauses, into a

Table 4.5: Average Precision for Yelp and PAKDD15 Datasets

Relation	NeuralLog	NeuralLog σ	RelNN	Baseline
Yelp	0.8139 \pm 0.0061	0.8070 \pm 0.0050	0.7990 \pm 0.0010	0.6122
PAKDD15	<u>0.6652 \pm 0.0016</u>	0.6660 \pm 0.0008	0.6384 \pm 0.0017	0.2208

Table 4.6: Area Under the ROC curve for Yelp and PAKDD15 Datasets

Relation	NeuralLog	NeuralLog σ	RelNN	Baseline
Yelp	0.7649 \pm 0.0055	0.7410 \pm 0.0055	0.7521 \pm 0.0007	0.5000
PAKDD15	<u>0.8106 \pm 0.0010</u>	0.8109 \pm 0.0008	0.7722 \pm 0.0007	0.5000

Markov Network, where the facts are predicted by the Markov networks [74] in such a way that violating a clause with a high weight would make the fact less likely to be true [17]. The MLN model used the full original theory provided by Alchemy for the same dataset. Although our experiments are not exactly comparable with the experiments from [17], it demonstrates the power of NeuralLog, to achieve a similar performance using only 3 rules, which is a fraction of the 72 rules used by the MLN.

Tables 4.5 and 4.6 show the comparison between NeuralLog and RelNN for classification in the Yelp and PAKDD15 datasets. In this case, NeuralLog outperforms RelNN in both datasets, meaning that NeuralLog is also suited to classification tasks.

We can also see from the experiments, that the choice of the activation and output functions might have a great impact in the performance of the NeuralLog models. Since NeuralLog allows the user to choose between different functions directly in the logic theory, it makes it easier to experiment with different options, differently from TensorLog and RelNN, whose functions are fixed.

Despite the better results achieved by tanh in some cases, its use must be done carefully. Since the output of the tanh function is in $[-1, 1]$, the element-wise multiplication of different paths to which the tanh have been applied would possibly lose the logic meaning of an **AND**. It is not the case of these experiments, since all the rules have only a single path, thus, no element-wise multiplication is performed with the results of the tanh.

Furthermore, Table 4.7 shows the weights learned for some rules to predict the *also_see/2* relation from the WordNet dataset, and for the *sameauthor/2* relation from Cora dataset. For clarity, we omitted the activation function from the body of the rule. Remembering that NeuralLog rules do not hold weights, only the facts do, and weights are represented in the rule by adding a literal for the corresponding fact, for instance: *also_see*(X, Y) \leftarrow

Table 4.7: Example of Learned Weights for WordNet and Cora Datasets

Weight	Rule
0.71315	$also_see(X, Y) \leftarrow deriv_related_form(X, Z) \wedge deriv_related_form(Z, Y) \wedge w(w1).$
-0.00081	$also_see(X, Y) \leftarrow deriv_related_form(Y, X) \wedge w(w2).$
-0.02814	$also_see(X, Y) \leftarrow deriv_related_form(X, Z) \wedge hyponym(Z, Y) \wedge w(w3).$
0.92750	$sameauthor(A1, A2) \leftarrow$ $haswordauthor(A1, W) \wedge kaw(W) \wedge haswordauthorinverse(W, A2) \wedge w(authorword).$

$similar_to(X, Y) \wedge act_func(Y) \wedge w(w4).$, where $w(w4)$ represents the weight of the rule, and $act_func(Y)$ represents the activation function.

As can be seen from the first rule in the table, there is a strong relation of type *also_see/2* between entities X and Y , if they both are connected to another entity Z through the *derivationally_related_form/2* relation, and the result of the first rule will also be proportional to the number of possibly entities Z between X and Y . On the other hand, entities X and Y that are connected by a path passing through *derivationally_related_form/2* and *hyponym/2* will have a lower prediction for *also_see/2*, given the negative weight of the last rule in the table. Notwithstanding, the second rule has a low impact on the prediction of the *also_see/2*, given its small weight, in absolute value.

Looking at the last rule from Table 4.7, from the Cora dataset, we can see that it is similar to the first rule in the table, in the sense that it will give a higher score to entities $A1$ and $A2$ that are related through an entity W in a path formed by the relations *haswordauthor/2* and *haswordauthorinverse/2*. However, since the predicate *kaw/1*, which is applied to the word W , is marked as a learnable predicate, the neural network is able to learn a, potentially, different weight for each word W that might appear in the rule. Thus, in addition to the weight represented by the literal $w(authorword)$, that is the same for any instantiation of the rule, this rule has a weight that might be different for each instantiation of the word W . The weight for the word W might also decrease the prediction of the rule, if it is negative, which could be learned from words that indicate an inverse correlation with the relation *sameauthor/2*.

As can be seen from the experiments, NeuralLog achieved better performance than the compared systems, in tasks for which they were designed, even if we look for a single variant of NeuralLog (tanh; or sigmoid and softmax). The main goal of these experiments were to show that NeuralLog is flexible enough to perform both classification and link prediction tasks, thus, affirmatively answering both questions **Q1** and **Q2**.

4.1.2 NeuralLog with Numeric Values

In order to show the capabilities of NeuralLog to deal with numeric attributes and to incorporate complex neural network models, we experimented with NeuralLog on the classical Iris dataset [19, 20] and in a Named Entity Recognition (NER) NCBI Disease dataset [75].

4.1.2.1 Iris Dataset

The Iris dataset is a dataset of flowers containing 3 classes (*iris setosa*, *iris versicolor* and *iris virginica*), with 50 examples each, where the task is to predict the class of each example, given 4 numeric attributes: *sepal length*, *sepal width*, *petal length* and *petal width*.

We split the dataset into training and test sets using a 70%-30% ratio and applied a simple Multilayer Perceptron (MLP) neural network with a single hidden layer, using sigmoid for both activation and output functions. MLPs are simple neural networks composed of a sequence of fully-connected layers, where all the neurons from a layer are connected to all the neurons of the next layer [42].

Algorithm 3 shows the theory to construct the neural network. The lines 1-3 specify the parameters whose weights will be learned from the examples, the remaining is the theory to define the neural network itself.

In order to facilitate the definition of rules with similar bodies, we created two syntax sugars. The first one is a simple for-each loop that iterates a variable over a list of elements, inspired by the bash syntax. The list can be defined in two ways: (1) by enumerating the whole list; or (2) by passing a range of the form $\{i..j\}$, $i \leq j$, which will iterate from i to j , with both extremes included. The for-each loop simply repeats everything within it, replacing the iteration variable by a value of the list at a time. The for-each loop defined in line 4 is used to create a hidden layer with 15 neurons, by multiplying each input feature by a weight (defined by $w1/1$) and summing a bias (defined by $b/1$). Notice that the rules from the line 5 to line 8 use the four attribute predicates, where the Y variable represents the value of the numeric predicate.

The second syntax sugar first appears in line 11, which defines the activation of the hidden layer. We call each term surrounded by curly brackets a wildcard. A wildcard can appear as (part of) a predicate name or as (part of) a term. Whenever a wildcard appears in the head of a rule, it must also appear in its body, this restriction is not applied to the variables defined in the for-each loops.

Then, we create a set S of all possible values a wildcard in the body of a clause may assume, as follows:

Algorithm 3 Theory to construct a Multilayer Perceptron for the Iris dataset

```

1: learn(w1).
2: learn(w2).
3: learn(b).

4: for i in {0..14} do
5:   hidden_{i}(X) :- sepal_length(X, Y), w1(h1_{i}_1).
6:   hidden_{i}(X) :- sepal_width(X, Y), w1(h1_{i}_2).
7:   hidden_{i}(X) :- petal_length(X, Y), w1(h1_{i}_3).
8:   hidden_{i}(X) :- petal_width(X, Y), w1(h1_{i}_4).
9:   hidden_{i}(X) :- b(h_{i}).
10: done

11: activation_{i}(X) :- hidden_{i}(X), sigmoid(X).

12: for type in setosa versicolor virginica do
13:   output_{type}(X) :- activation_{i}(X), w2(output_{type}_{i}).
14:   output_{type}(X) :- b(output_{type}).

15:   iris_{type}(X) :- output_{type}(X), sigmoid(X).
16: done

```

- If the wildcard appears as (part of) a predicate name, we add to S all possible values this wildcard could assume, in order to make the name of the predicate equal to the name of another predicate defined in the program;
- If the wildcard appears as (part of) a term name, we add to S all possible values this wildcard could assume, in order to make the name of the term equal to the name of another term defined in the program, as long as the names of other predicates containing this wildcard would still exist in the program.

Finally, for each possible permutation of the values of each wildcard in the clause, we generate a new clause, replacing the wildcard by those values.

We need to filter for existing predicates in order to replace the value of a wildcard in the name of a predicate in the body, because it does not make sense to define new predicates in the body of a rule, that do not appear elsewhere. This restriction does not apply to the head of the rule, since a rule can create new a predicate in its head, while the definition of this predicate is given by the body of the rule. The generation of new terms is also allowed and new entities are added to the program. We refer the reader to Appendix A for more details about the NeuralLog language.

The second for-each loop, in line 12, defines the output layer with 3 neurons, represented by the predicates *iris_setosa*, *iris_versicolor* and *iris_virginica*. It starts by summing the output of the hidden neurons multiplied by a weight (defined by $w2/1$), and then summing a bias (defined by $b/1$). Finally, it creates the output neurons by summing the correspondent combinations of the hidden layer and bias, and applying the output function. As expected, this model can perfectly learn the Iris task, achieving 100% accuracy.

4.1.2.2 NCBI Disease Dataset

In the NCBI Disease dataset, we performed the Named Entity Recognition (NER) task. The NER consist of identifying which words in a sentence belong to an entity. In the case of the NCBI Disease, which words correspond to diseases.

In order to perform this task, we implemented the Bidirectional Encoder Representations from Transformers (BERT) model as a function, a state-of-the-art model to perform Natural Language Processing (NLP) tasks [21]. The BERT model learns how to represent words in a vector space representation. We assigned the *bert/2* predicate to represent the BERT model and the *dense/2* predicate to be a dense layer, which will compute the final label of the word, given the representation learned by BERT. Aside from the configuration, the logic theory to construct the neural network has only a single rule $ner(X, Y) \leftarrow bert(X, W) \wedge dense(W, Y)$. In this experiment, the input X is processed as a sequence of words, representing a sentence; and the output Y is a sequence of labels.

Table 4.8: Results for the Named Entity Recognition on the NCBI Disease Dataset

Metric	No pre-train	5 Epochs	20 Epochs	BioBERT
Precision	27.34	73.53	80.98	88.22
Recall	14.58	74.38	88.23	91.25
F1	19.02	73.95	84.45	89.71

We ran three experiments, one learning the task directly for 5 epochs without pre-training, and two more using the set BioBERT v1.1³ of pre-trained weights for biomedical domain provided by [76]. We fine-tuned the model to the task for 5 and 20 epochs, respectively. All experiments used stochastic gradient descent with learning rate of 0.01, to minimize the mean squared error with 0.01 L2 regularization rate.

Table 4.8 shows the results of each model. In addition, we included, into the table, the results achieved by BioBERT itself, fine-tuned for NER, which was extracted from its paper [76]. As we can see, NeuralLog can represent models to learn NER, although it has not achieved the best possible performance. We believe that the difference in performance is due to different training parameters, which is not easy to find, since this model takes a long time to train. Nonetheless, it corroborates our claims that NeuralLog can represent different neural network architectures.

In our recent survey [77], we summarized several deep learning models to perform NLP tasks on biomedical domains. In addition, in our previous work [44], we used two separated state-of-the-art models to perform Named Entity Recognition (NER) and Named Entity Normalization (NEN) on biomedical text. Named Entity Normalization is the task of assigning a piece of text to a known entity from a dictionary. This piece of text is often the output of a NER model. We believe that a logic layer, that can be described as a logic theory and supported by background knowledge, can improve the result of NLP models in such tasks. As such, NeuralLog would be a suitable tool in order to combine the strengths of deep learning, which reads the raw text; with relational learning, which is based on relations between the entities in the text, represented by first order logic.

4.2 Structure Learning

In this section, we present the results of our experiments of the structure learning algorithms from NeuralLog. We split it into two subsections: batch learning and online learning.

³The BioBERT model was a courtesy of the U.S. National Library of Medicine and can be found at <https://github.com/dmis-lab/biobert>

4.2.1 Batch Structure Learning

In order to demonstrate the capabilities of NeuralLog to batch learn neural network structures using MIL, we compared NeuralLog+MIL with Neural-LP [8] in three datasets: the Unified Medical Language System (UMLS) [22]; the WordNet dataset [18]; and the UWCSE dataset [17]. We chose to compare NeuralLog+MIL with Neural-LP, since Neural-LP is based on TensorLog, which is closely related to NeuralLog.

4.2.1.1 Methodology

We focused at the link prediction task, where we are given a query of the form $? - q(a, X)$, and the goal is to find all the entities that are related (as second term) to the entity a (as first term) through relation q . For each dataset, we selected a set of target relations and applied the system to learn them. Then, we evaluated the system by using the mean rank, the Mean Reciprocal Rank (MRR) and the hit at top 10 entities, based on the filtered rank of the entities, following the procedure described in [23]. In these experiments, we would like to answer the research question (**Q3**) can NeuralLog+MIL learn the structure representation of NeuralLog models for link prediction tasks?

We use these rank metrics to reproduce the experiments from Neural-LP [8]. Since Neural-LP only uses positive examples, to achieve a fair comparison, we give NeuralLog only positive examples as well. Here, we take advantage of the sparse representation of NeuralLog. Since positive examples have an associated value of 1 and the missing examples will have an associated value of 0, the loss function will be able to tune the weights in order to prove only the positive examples. This is equivalent to consider all the missing examples as negatives, which is known as the Closed World Assumption (CWA).

UMLS. It is a dataset consisting of 46 relations between biomedical concepts. We selected the most frequent relation (*Affects*) as target relation and applied the system to predict this relation from the remaining ones. We randomly selected 90% of the facts from the target relation as training examples and the other 10% as test examples, following the procedure and dataset used in [9]. We repeated this process 10 times and reported the average of the runs.

WordNet. It is a dataset consisting of 18 relations between words, and it is already split into train, development and test sets. We ran each system once, holding out one relation at a time and reported the mean results of the relations.

Table 4.9: The Meta-Theory Used by NeuralLog+MIL

$$\begin{aligned}
P(X, Y) &\leftarrow Q(X, Y). \\
P(X, Y) &\leftarrow Q(Y, X). \\
P(X, Y) &\leftarrow Q(X, Z) \wedge R(Z, Y).
\end{aligned}$$

UWCSE. It is already split into 5-folds for the *Advised by* relation. We run the system once for each fold and reported the mean results. We applied the same transformation in order to change the ternary relations in a set of binary relations, as used in the parameter learning experiments.

The used meta-theory was the one shown in Table 4.9, with the depth of 1. It means that each meta-clause is applied directly to the example. For instance, applying the first clause to the example $advisedby(X, Y)$ would generate the clause $advisedby(X, Y) \leftarrow Q(X, Y)$., where Q is replaced for each valid predicate in the knowledge base (KB). If the depth were 2, in addition to replacing Q for each example in the KB, we would also have tried to prove it with another meta-clause; for instance, the last one, which would result in an invented predicate $f(X, Y) \leftarrow Q(X, Z) \wedge R(Z, Y)$., but it would also exponentially increase the size of the theory.

The head of each generated rule is changed from the target predicate $p/2$ to a predicate $p'/2$, and a rule of the form $p'(X, Y) \leftarrow p(X, Y), output_function(Y)$. is added to the theory. Also, each generated rule has two atoms appended to it. The first one is the $activation_function(Y)$ atom, where Y is the output of the rule, and it serves as an activation function to the output result of the rule. The second one is an atom of the form $w(id)$, where id is a unique constant for each rule and w is a predicate whose weight will be learned by the network and will multiply the final result of the rule. Finally, we add a rule of the form $p(X, Y) \leftarrow b$. to the theory. Since this rule does not depend on the input, it will be true for any input and its value will be b , which will be added to the output of the other rules with the predicate $p/2$ in the head, acting as a bias. The weight of the predicate b will also be learned by the network.

We used sigmoid as activation function and softmax as output function; and adagrad to minimize the binary cross-entropy with L2 regularization for 50 epochs in the case of the UMLS and UWCSE datasets, and 10 epochs for the WordNet dataset.

We used the default parameters for Neural-LP, which achieved good results in two of the three datasets.

Table 4.10: Results for the Filtered Rank Metric

Dataset	NeuralLog+MIL			Neural-LP		
	Hit @ 10	Mean Rank	MRR	Hit @ 10	Mean Rank	MRR
UMLS	0.9894	1.6056	0.8856	0.9970	1.2773	0.9303
WordNet	0.8216	10.0647	<u>0.6221</u>	0.9418	9.2479	<u>0.9298</u>
UWCSE	<u>0.9165</u>	<u>5.5818</u>	0.2158	<u>0.4446</u>	<u>36.8060</u>	0.2458

4.2.1.2 Results

Table 4.10 shows the results for each dataset. The lower the mean rank, the better; for the other metrics, the higher, the better. The best value of each metric, for each dataset is bold-faced. Pair of underlined values, of the same metric, in the same dataset, means that the difference between the values has statistical significance according to the two-tailed paired t-test with $p < 0.05$.

As can be seen from the table, NeuralLog+MIL shows a competitive result with Neural-LP in both the UMLS and WordNet datasets; and is considerably better in the UWCSE dataset, for the hit at top 10 entities and the mean rank.

WordNet is composed of 18 relations with a different number of examples for each relation. If we take the average of the metrics, weighted by the number of examples of each relation, the difference between NeuralLog+MIL and Neural-LP is even smaller, being 0.9280 against 0.9544, for the hit at top 10 entities; and 4.1480 against 6.1228, for the mean rank; for the NeuralLog+MIL and Neural-LP, respectively. In this scenario, the mean rank of NeuralLog+MIL is better than the one of Neural-LP.

Table 4.11 shows examples of rules learned by the system, alongside their weights. By looking at the weights of the rules, we can try to interpret the reasoning of the learned model. For instance, we can see that entities are related through the *also-see/2* relations if there is a path between entities X and Y through the pair of relations (*hyponym/2* and *hypernym/2*) or the pair (*synset_dom_usage_of/2* and *memb_of_dom_usage/2*). On the other hand, we can see that the paths composed of the pairs (*memb_meronym/2* and *memb_holonym/2*) and (*memb_of_dom_region/2* and *deriv_related_form/2*) have small impact on the prediction of the *also-see/2* relation, given the small absolute value of the weights of their corresponding rules.

Table 4.11: Example of Learned Rules and Weights for WordNet Dataset

Weight	Rule
0.1357	$also_see(X, Y) \leftarrow hyponym(X, Z) \wedge hypernym(Z, Y) \wedge w(w_{.392})$
0.1161	$also_see(X, Y) \leftarrow synset_dom_usage_of(X, Z) \wedge memb_of_dom_usage(Z, Y) \wedge w(w_{.194})$.
-2.2248×10^{-08}	$also_see(X, Y) \leftarrow memb_meronym(X, Z) \wedge memb_holonym(Z, Y) \wedge w(w_{.214})$.
-2.2248×10^{-08}	$also_see(X, Y) \leftarrow memb_of_dom_region(X, Z) \wedge deriv_related_form(Z, Y) \wedge w(w_{.109})$.

Table 4.12: Comparison with Embedding Systems

Dataset	WordNet	
	Hit @ 10	Mean Rank
TransE	0.892	251
TransH	0.867	303
CTransR	0.923	218
NeuralLog+MIL	0.8216	10.0647
Neural-LP	0.9418	9.2479

4.2.1.3 Comparison with Embedding Systems

Furthermore, on Table 4.12 we compare NeuralLog+MIL and Neural-LP against TransE [23], TransH [78] and CTransR [79]. The table was extracted from the results reported on [79]. TransE is a system that performs link prediction based on an embedding technique. It represents entities and relations as dense low-dimensional vectors that “embeds” the semantic meaning of the entities and relations. Those vectors are learned from a set of examples, then the link prediction is performed based on calculation on the vectors [23]. Although simple, TransE achieved a great result for link prediction at the time. Several systems, such as TransH [78], TransR and CTransR [79] tried to improve upon TransE by adding some complexity to overcome some of its limitations.

Although the experiments are not exactly comparable, due to differences on the setup of the dataset and the evaluation, Table 4.12 shows that logic based approaches are competitive against embeddings based approach, for link prediction, on the Hit@10 metric. In addition, the logic based approaches (NeuralLog+MIL and Neural-LP) achieved a much better result for the mean rank metric. Nonetheless, logic based models are easier to interpret, given that their meaning is given by logic rules; differently from embeddings approaches that embed the knowledge on dense vectors and calculations on these vectors, which are hard to be interpreted by humans.

Finally, from these results, we show that NeuralLog+MIL can learn the structure of NeuralLog models for link prediction tasks, affirmatively answering **Q3**.

4.2.2 Online Structure Learning

In this subsection, we present the two online learning algorithms. In order to show the capabilities of NeuralLog+OSLR and NeuralLog+OMIL, we compared them with OSLR in online learning of logic theories for link prediction in three distinct datasets: the Cora dataset, which is a citation matching dataset [16]; the Unified Medical Language System (UMLS), which is a medical dataset [22]; and the UWCSE dataset, which describes relations between professors and students in the University of Washington [17]. In addition, we also include the comparison with RDN-Boost [24], a system that learns Relational Dependency Networks (RDNs) [25], which was also used in [11, 12].

The online structure learning algorithms for NeuralLog take advantage that NeuralLog is able to store the weights learned by the neural network back into the logic program. In this way, these algorithms can focus on changing the logic theory directly, and it will result in a new neural network that is able to preserve the learned weights from the facts that were not changed on the new theory. Thus, the learning algorithm could add a new rule with a new fact, and the weight of the other facts will be the same that was already learned from the previous network, for the rest of the program. Then, all the weights, the old and the new ones, can be adjusted by the current neural network.

In this set of experiments, we would like to answer the following research questions: **(Q4)** can NeuralLog+OSLR and NeuralLog+OMIL learn the structure of NeuralLog models online, by using theory revision, for link prediction tasks? And **(Q5)** can NeuralLog+OSLR and NeuralLog+OMIL benefit from a previous existing theory, when learning the structure of NeuralLog models online, for link prediction tasks? Following the previous experiments, we receive a query of the form $? - q(a, X)$ as input and would like to find the substitutions of X that match the positive examples, without matching the negative ones.

Cora. The Cora dataset contains four target relations, *Same Author*, *Same Bib*, *Same Title* and *Same Venue*. We ran each of these relations separately.

UMLS. It is a dataset of biomedical entities. We selected the *Affects* as target relation, since it is the most frequent relation in this dataset, as it is done in the batch experiments above and in [9]. Since it has no negative examples, we generated approximately 2 negative examples for each positive example, by selecting a random output entity that appears in the target relation, but is not related to the input entity, following the Local Closed World Assumption (LCWA) [73].

Table 4.13: Size of the Datasets

Relation	# Positives	# Negatives
Same Author	488	66
Same Bib	30,971	21,952
Same Title	661	714
Same Venue	2,887	4,976
Affects	1,022	-
Advised by	113	16,601

UWCSE. This dataset has one target relation *Advised by*, that relates the students with supervisors. Again, we converted the ternary predicates into binary predicates, in the same way it was done in the previous experiments.

Table 4.13 shows the statistics of the datasets, for a total of 6 target relations. Since the number of negative examples in the UWCSE is much bigger than the positive ones, we downsampled the set of negative examples to be twice as much as the number of positives ones, as it is done in [11, 12, 24].

4.2.2.1 Simulating the Online Environment

In order to properly evaluate the online systems, we use these datasets to simulate an online environment by reproducing the procedure used in [11, 12]. We split each target relation into $N + 1$ iterations, where iteration 0 has only the background knowledge and each of the following iterations has approximately $|E|/N$ examples, where $|E|$ is the total number of examples of each relation.

We pass each iteration, in order, to the system. When an iteration arrives, the current model is tested with it, before training. Then, the system trains on this iteration and the evaluation of each iteration is recorded. This evaluation procedure is known as *Prequential* [80]. Since RDN-Boost is designed for batch learning, we transformed each iteration of the online learning environment into a batch learning task by appending all the examples from the previous iteration to it. Then, we applied RDN-Boost to each of those tasks.

Following the procedure from [11, 12], we set the number N of iterations to 30 for all the target relations except the *Advised by* which was set to 91. We ran each experiment 30 times and reported the average of the area under the Precision-Recall curve as evaluation metric. The Hoeffding’s bound δ parameter was set to 10^{-3} and updated to half its value, each time a revision was accepted. The systems only consider the number of unrelated examples to

compute the Hoeffding’s bound [11, 12]. However, we had to relax this restriction for the UMLS dataset, given the reduced number of unrelated examples.

NeuralLog adds a weight for each rule, as well as an activation function. Also, a bias is added to the output of the target relation, which then passes through an output function. The weights and biases are parameters to be learned by the neural network. After each accepted revision, the neural network adjusts its parameters on the same set of examples used by the revision. It uses the adagrad optimization algorithm to reduce the mean squared error loss function with L2-regularization for 10 epochs, with a learning rate of 0.01 and the l2 $\lambda = 0.01$. For OSLR, we replaced its feature generation by another one that creates a single weight for each rule, in order to be closely related to NeuralLog. This change had only a small impact on the final result.

4.2.2.2 Results

We now present the results of the experiments. All pairs of systems were compared for statistical significance using two-tailed paired t-test with $p < 0.05$. There is a statistical significance between the pairs, unless stated otherwise.

Figure 4.1 shows the evaluation of the systems in the Cora dataset over the epochs. As can be seen, NeuralLog+OMIL outperforms both OSLR and RDN-Boost over all iterations for the *Same Author* and the *Same Venue* relations, while it underperforms OSLR and RDN-Boost on all iterations on the *Same Bib* relation. For the *Same Title* relation, NeuralLog+OMIL has a stable behaviour over the iterations, while OSLR have some ups and downs. However, OSLR is able to achieve a better result in the final iteration, where all the examples are used, and also has a better overall result, measured by the area under the curve of iterations. NeuralLog+OSLR performed worse than OSLR in all relations of the Cora dataset, but it is able to outperform RDN-Boost in all relations, except for the *Same Bib*.

There were no statistical difference between NeuralLog+OMIL and OSLR, and NeuralLog+OSLR and RDN-Boost for the area under the curve and the final result, in the *Same Author* relation. There were no statistical difference between OSLR and RDN-Boost for the area under the curve and the final result, in the *Same Bib* relation. Finally, there were no statistical difference between NeuralLog+OMIL and OSLR for the area under the curve and the final result, in the *Same Title* relation.

Figure 4.2 shows the results of the experiments for the UMLS and UWCSE datasets. On Figure 4.2a, we can see that NeuralLog+OMIL, OSLR and RDN-Boost get better as new examples arrive, ending with an evaluation greater than NeuralLog+OSLR. However, NeuralLog+OSLR performs much better than the other systems on the initial iterations, and also achieved a better overall evaluation, given the area under the curve. There

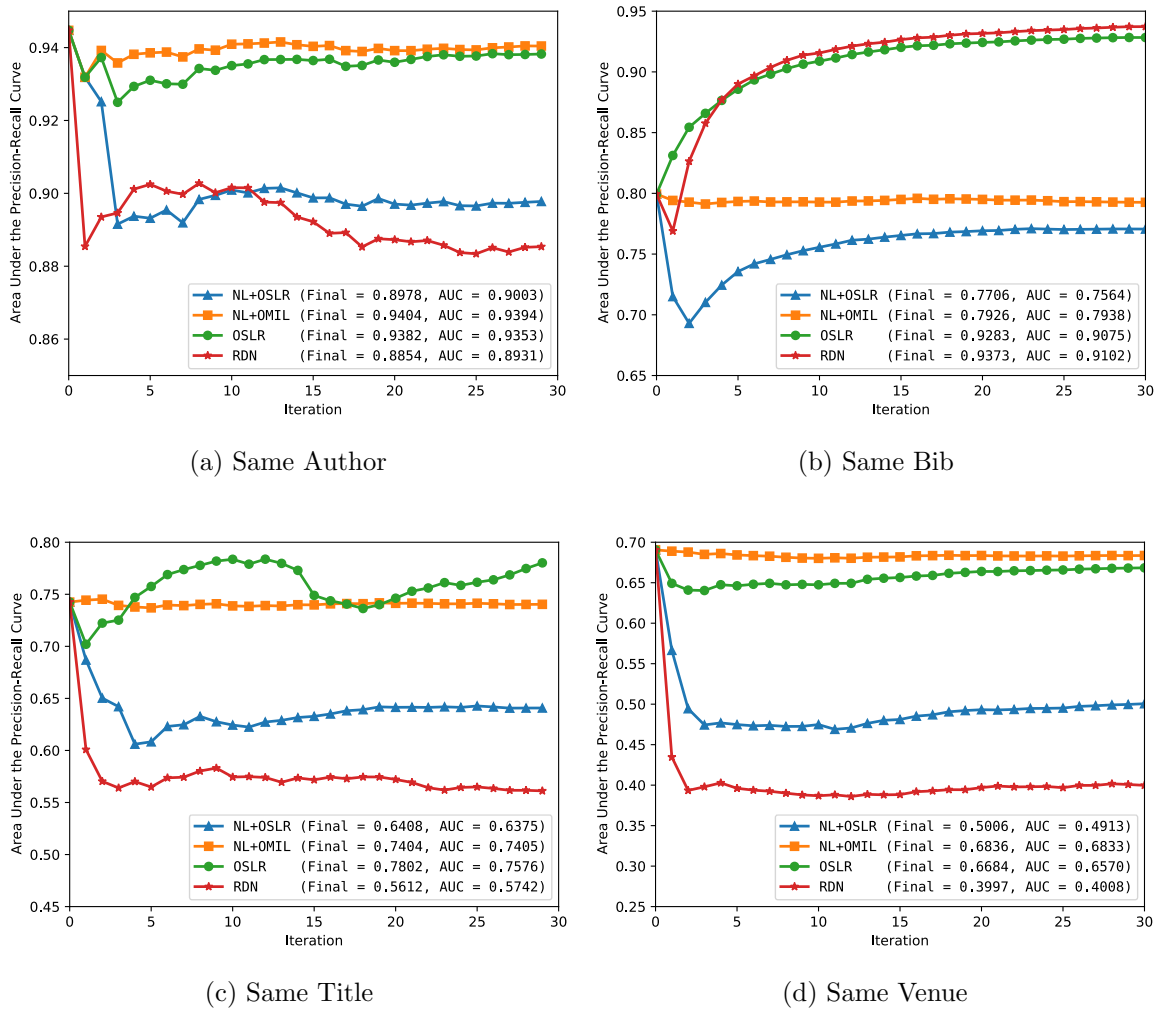


Figure 4.1: The Evaluation of the Cora Dataset

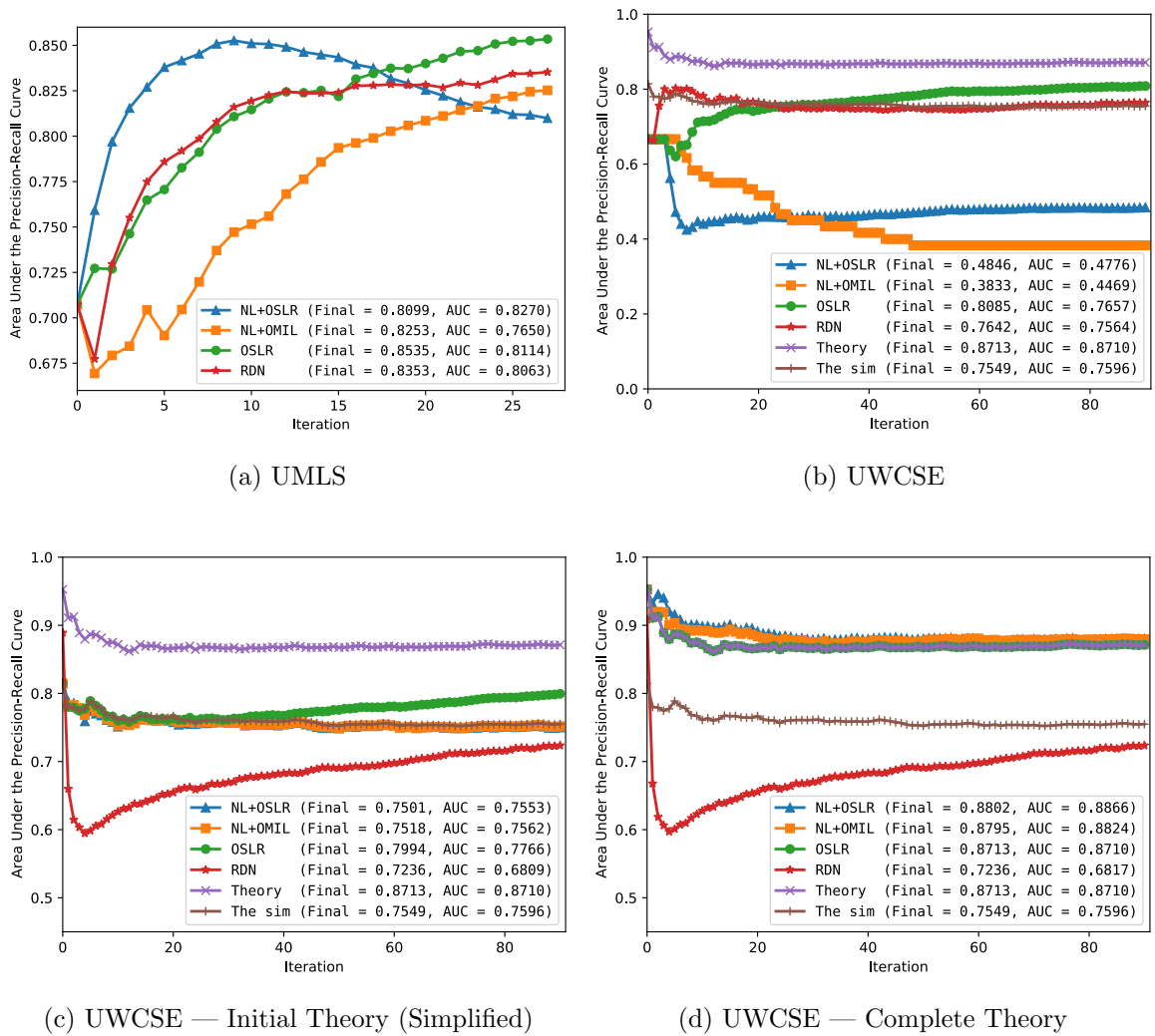


Figure 4.2: The Evaluation of the UMLS and UWCSE Datasets

was no statistical difference between NeuralLog+OSLR and NeuralLog+OMIL nor between NeuralLog+OMIL and RDN-Boost, for the result of the final iteration. Also, there was no statistical difference for the area under the curve between OSLR and RDN-Boost.

In order to evaluate the impact of an initial theory, we performed three experiments with the UWCSE dataset, using a hand-crafted theory provided by Alchemy⁴. Since Alchemy supports a more complex logic language, we removed the rules whose logic feature were not supported by both OSLR and NeuralLog. Then, we used two sets of rules: a complete set, containing more rules, which are more specific; and a simplified version of this theory, containing only some generic rules from the complete set. The *Theory* lines in the figures show the result of the initial theory, while the *The sim* shows the results of the simplified theory; both theories were inferred by OSLR, without any training.

⁴<http://alchemy.cs.washington.edu/>

As can be seen in Figures 4.2b, OSLR outperforms both NeuralLog systems and RDN-Boost, when they all start from an empty theory. However, it cannot outperform the complete theory. On the other hand, when the systems start from the simplified theory (Figure 4.2c), OSLR can improve over the simplified theory, but not yet outperforming the complete theory; while NeuralLog systems stay close to the performance of the simplified theory and RDN-Boost performed worse than the theory itself. Finally, when starting from the complete theory (Figure 4.2d), both NeuralLog+OSLR and NeuralLog+OMIL are able to improve over the complete theory, with a slight advantage for NeuralLog+OSLR; while OSLR is only capable of achieving the same performance as the complete theory; and RDN-Boost, again, performed worse than the theory. However, neither NeuralLog+OSLR nor NeuralLog+OMIL were able to change the complete theory, showing that the improvement in this dataset was due to the NeuralLog inference mechanism itself.

This demonstrates the strength of revision theory methods, that are capable of improving the results of initial existing theories, even when the theories are only partially correct, corroborating results already found in other works such as [40, 45, 46]. On the other hand, RDN-Boost is not able to change the initial theory and cannot fix possible mistakes of the theory.

For the empty initial theory, there is no statistical difference between: NeuralLog+OSLR and NeuralLog+OMIL, in either metrics; OSLR and the simplified theory nor OSLR and RDN-Boost, for the area under the curve; and RDN-Boost and the simplified theory, in either metrics. For the initial simplified theory, there is no statistical difference between: NeuralLog+OSLR and NeuralLog+OMIL, in either metrics; and NeuralLog+OSLR/NeuralLog+OMIL and the simplified theory, in either metrics. For the initial complete theory, there is no statistical difference between: NeuralLog+OSLR and NeuralLog+OMIL, in either metrics; NeuralLog+OSLR and OSLR, for the final iteration; NeuralLog+OSLR and the complete theory, for the final iteration; NeuralLog+OMIL and OSLR for the final iteration; NeuralLog+OMIL and the complete theory, for the final iteration; and OSLR and the complete theory, in either metrics.

By analysing the changes in the theories during the iterations, we saw that, in most cases, NeuralLog+OSLR and NeuralLog+MIL learned a few rules during the first iterations, and then, do not change the rules on the next iterations, when starting from an empty theory. This behaviour was also observed on OSLR, however, OSLR often add one or two more rules, during later iterations, after more examples are available.

On one hand, this shows that the system is able to learn to represent the examples when only a few of them are available. On the other hand, it might be missing the opportunity to learn from more examples. However, we notice that, for a particular run of the NeuralLog+OSLR for the Same Venue relation, it kept adding new rules to the theory during the passing of new iterations. This resulted on a more volatile model, where the performance had big changes

from one iteration to the other (increasing and decreasing), and with a worse performance overall, both for the area under the curve and for the performance on the final iteration.

We can see from the experiments that both NeuralLog+OSLR and NeuralLog+OMIL are able to online learn the structure of NeuralLog models for link predictions tasks, thus, we can affirmatively answer **Q4**. When starting from an initial theory, both NeuralLog+OSLR and NeuralLog+OMIL can be at least as good as the initial theory; and, in the case of the complete theory for the UWCSE dataset, both systems were able to outperform the initial theory, showing that starting from an initial theory, when available, can improve the quality of the learned model, positively answering **Q5**.

Chapter 5

Conclusions

In this paper we presented NeuralLog, a first-order logic language that is compiled into a neural network. Our goal in the design of NeuralLog was to bridge the gap between first-order logic and neural networks.

Previous works in the field of Neural-Symbolic Learning and Reasoning [4] are often focused at constructing neural network structures by the use of logic rules, transforming the logic representation into differentiable operations [6, 15, 50]. However, these approaches make it hard to integrate logic with other neural network models. NeuralLog, on the other hand, allows the user to define any neural network structure, which can be easily integrated with the logic part through the definition of *function predicates*.

Furthermore, the definition of *attribute predicates*, which allows NeuralLog to handle numeric values directly in the logic program, also facilitates the integration of logic with other neural network models. Even allowing the definition of Multilayer Perceptrons [42] directly in NeuralLog, as shown in Subsection 4.1.2.

The parameter learning experiments, presented in this work, showed that NeuralLog is a flexible language, capable of describing relational neural network models to perform both link prediction and classification tasks.

In the link prediction task, we compared NeuralLog with TensorLog [14], a system that performs logic inference through numeric operations, representing a neural network, and is focused at link prediction. NeuralLog outperformed TensorLog in both the Cora dataset and in the UWCSE dataset, two commonly used datasets in the ILP community. In addition, NeuralLog achieved results comparable with TensorLog for the WordNet dataset.

In the classification task, we compared NeuralLog with RelNN [15], a system that also generates neural networks based on logic, where the number of logic proves of the logic rules is computed by the neural network and also poses as feature for the neural network

components. NeuralLog outperformed ReLNN in the two tested datasets: the Yelp and the PAKDD15.

Moreover, in order to further show the capabilities of NeuralLog, we presented a NeuralLog program that represents a Multilayer Perceptron for the Iris dataset [19,20]; and a NeuralLog program to perform Named Entity Recognition (NER) that uses a state-of-the-art neural network model called Bidirectional Encoder Representations from Transformers (BERT) [21], which is called as a logic predicate. Nevertheless, differently from TensorLog [14], NeuralLog is capable of handling rules with free variables, which largely improved the results in the UWCSE dataset, for the area under the ROC curve.

A great advantage of using a language based on first-order logic is that we can rely on a wide range of algorithms from Inductive Logic Programming (ILP) that can be used to learn logic programs based on examples. However, some small changes on the ILP algorithm must be done, so it can better integrate with NeuralLog and take advantage of neural network characteristics.

In this work, we presented three structure learning algorithms for NeuralLog. The first structure learning algorithm presented was NeuralLog+MIL, a structure learning algorithm based on Metagol [7]. Metagol is a system that relies on Meta-Interpretive Learning (MIL), a method that learns first-order logic theories by the use of a higher-order theory [7].

In addition to NeuralLog+MIL, which is a system to batch learn NeuralLog models, we also presented two algorithms for online learning, this is, to learn in an environment where the examples are arriving over time. First we presented NeuralLog+OSLR, a ported version of Online Structure Learner by Revision (OSLR), an online learning system initially designed to work with ProPPR [64], a Stochastic Logic Programming (SLP) language [13]. OSLR is based on theory revision, this means that it starts from a, possibly empty, initial theory and changes it in order to cope with the new examples. Taking advantage of the theory revision mechanism from OSLR, we proposed NeuralLog+OMIL, a system that uses MIL searching strategy to online learn first-order logic theories through theory revision.

In order to better integrate the structure learning systems with NeuralLog, we implemented the concept of *clause modifiers*, which receive the proposed clauses from the structure learning systems and modify them in order to append characteristics of neural networks, such as activation functions and weights.

Our experiments showed that NeuralLog+MIL has competitive results when compared with Neural-LP [14], for link prediction in batch environments, for the UMLS [22] and WordNet [18] datasets; and outperformed Neural-LP in the UWCSE dataset [17].

In the online environment, we compared NeuralLog+OSLR and NeuralLog+MIL with OSLR [11, 12] and RDN-Boost [24] for link prediction tasks on three different datasets:

Cora [16], UMLS [22] and UWCSE [17]. NeuralLog+OMIL outperforms OSLR and RDN-Boost on three of the four target relations from the Cora and in the UMLS datasets. While NeuralLog+OSLR and NeuralLog+OMIL outperform OSLR and RDN-Boost on the UWCSE dataset, whenever a good initial theory is provided.

5.1 Future Works

In this work we presented the NeuralLog language, its inference mechanism and its structure learning algorithms. We believe that each of these components may open interesting lines of research. Thus, in this section, we enumerate some directions for future works.

In the NeuralLog language, a future work would be to expand its expressibility, for instance, by adding support for facts with arity greater than two. In addition, the definition of rules with propositional predicates in their head would also be beneficial.

Still in the NeuralLog language and in the inference mechanism, a future work could be to investigate the inference of higher-order rules directly, without transforming it to first-order rules. This could be achieved by assigning a vector for each higher-order predicate in the rule, containing a weight for each possible first-order substitution of the predicate. Then, this vector could be learned from data.

On the structure learning mechanism, differently from inferring higher-order rules directly, an approach would be to learn which higher-order rule would get the best result in the revision of the theory, at each moment, thus, improving the efficiency of the algorithm by applying a better sequence of revision. Furthermore, once a sequence of revision is learned for a given task, it would be interesting to investigate whether it can be transferred to other similar tasks.

Regarding further experiments, a natural path would be to investigate the performance of NeuralLog for both parameter and structure learning on different datasets. Specially on datasets where relational knowledge is required alongside neural network structures to take advantage of raw inputs of some kind, such as: images, text, audio, among others. By using the clause modifiers, one could provide the neural network structure for the raw input and let the structure learning algorithm learn the relational part of the task.

In addition, experiments using the MIL structure learning algorithms with a deeper higher-order resolution tree, which would allow the invention of new predicates, are also worthy of investigation.

Furthermore, it would be interesting to experiment with NeuralLog+OSLR and NeuralLog+OMIL on datasets containing *concept drift*, when the underlying distribution of the examples changes over time and the model must adapt itself to the new distribution [81].

Notwithstanding, NeuralLog has an experimental implementation of negation. However, it imposes some constraints to the logic program. As future work, one could investigate the benefits of using negation in NeuralLog, whether it is necessary or if it can be expressed by rules with negative weights. If negation is beneficial for learning tasks, it would be interesting to investigate if the required constraints can be relaxed and/or ensured by the language and the inference mechanism. Furthermore, considering that negation is beneficial, the structure learning algorithm could be extended to consider it in their hypotheses spaces.

Nevertheless, in this work we presented NeuralLog, a first-order logic language that is compiled to a neural network. In addition, we also presented three structure learning algorithms for NeuralLog: one batch algorithm, and two online algorithms. Our experiments showed that the results from NeuralLog were comparable to or greater than other state-of-the-art systems, in tasks for which they were designed, in several datasets. Finally, we pointed out directions that we find interesting for future works.

References

- [1] Yann LeCun and Yoshua Bengio. The handbook of brain theory and neural networks. chapter Convolutional Networks for Images, Speech, and Time Series, pages 255–258. MIT Press, Cambridge, MA, USA, 1998.
- [2] Ronald Brachman and Hector Levesque. *Knowledge Representation and Reasoning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1 edition, 2004.
- [3] Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. *J. Log. Program.*, 19/20:629–679, 1994.
- [4] A 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 *2015 AAAI Spring Symposium Series*, 2015.
- [5] Huma Lodhi. Deep relational machines. In Minho Lee, Akira Hirose, Zeng-Guang Hou, and Rhee Man Kil, editors, *Neural Information Processing*, pages 212–219, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
- [6] Artur S. Avila Garcez and Gerson Zaverucha. The connectionist inductive learning and logic programming system. *Applied Intelligence*, 11(1):59–77, Jul 1999.
- [7] Stephen H. Muggleton, Dianhuan Lin, and Alireza Tamaddoni-Nezhad. Meta-interpretive learning of higher-order dyadic datalog: predicate invention revisited. *Mach. Learn.*, 100(1):49–73, 2015.
- [8] Fan Yang, Zhilin Yang, and William W Cohen. Differentiable learning of logical rules for knowledge base reasoning. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 2319–2328. Curran Associates, Inc., 2017.
- [9] William Yang Wang, Kathryn Mazaitis, and William W. Cohen. Structure learning via parameter learning. In *Proceedings of the 23rd ACM International Conference on*

- Conference on Information and Knowledge Management, CIKM '14*, pages 1199–1208, New York, NY, USA, 2014. Association for Computing Machinery.
- [10] Pedro Domingos and Geoff Hulten. Mining high-speed data streams. In *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '00, pages 71–80, Boston, Massachusetts, USA, August 2000. ACM.
- [11] Victor Guimarães. Online probabilistic theory revision from examples: A ProPPR approach. Master's thesis, PESC, COPPE, Federal University of Rio de Janeiro, Rio de Janeiro, RJ, Brazil, 2018.
- [12] Victor Guimarães, Aline Paes, and Gerson Zaverucha. Online probabilistic theory revision from examples with ProPPR. *Machine Learning*, 108(7):1165–1189, Jul 2019.
- [13] Stephen Muggleton. Stochastic logic programs. In *New Generation Computing*, volume 32, pages 254–264, Cambridge, Massachusetts, EUA, January 1996. Academic Press.
- [14] William W. Cohen, Fan Yang, and Kathryn Mazaitis. Tensorlog: A probabilistic database implemented using deep-learning infrastructure. *J. Artif. Intell. Res.*, 67:285–325, 2020.
- [15] Seyed Mehran Kazemi and David Poole. RelNN: A deep neural model for relational learning. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 6367–6375, 2018.
- [16] Hoifung Poon and Pedro Domingos. Joint inference in information extraction. In *Proceedings of the 22Nd National Conference on Artificial Intelligence*, volume 1 of *AAAI'07*, pages 913–918, Vancouver, British Columbia, Canada, July 2007. AAAI Press.
- [17] Matthew Richardson and Pedro Domingos. Markov logic networks. *Machine Learning*, 62(1):107–136, 2006.
- [18] George A. Miller. Wordnet: A lexical database for english. *Commun. ACM*, 38(11):39–41, November 1995.
- [19] Edgar Anderson. The species problem in iris. *Annals of the Missouri Botanical Garden*, 23(3):457–509, 1936.
- [20] R. A. Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7(7):179–188, 1936.

- [21] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.
- [22] Stanley Kok and Pedro Domingos. Statistical predicate invention. In *Proceedings of the 24th International Conference on Machine Learning, ICML '07*, pages 433–440, New York, NY, USA, 2007. Association for Computing Machinery.
- [23] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. In C.J.C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 2787–2795. Curran Associates, Inc., 2013.
- [24] Tushar Khot, Sriraam Natarajan, Kristian Kersting, and Jude Shavlik. Gradient-based boosting for statistical relational learning: the markov logic network and missing data cases. *Machine Learning*, 100(1):75–100, 2015.
- [25] Jennifer Neville and David Jensen. Relational dependency networks. *Machine Learning*, 8(Mar):653–692, 2007.
- [26] Victor Guimarães and Vítor Santos Costa. Neurallog: a neural logic language. *CoRR*, abs/2105.01442, 2021.
- [27] Victor Guimarães and Vítor Santos Costa. Meta-interpretive learning meets neural networks. *The Semantic Data Mining Workshop, SEDAMI 2021*, 08 2021.
- [28] Victor Guimarães and Vítor Santos Costa. Online learning of logic based neural network structures. In *Inductive Logic Programming*, Athens, Greece, 2021. Springer International Publishing.
- [29] Alfred Horn. On sentences which are true of direct unions of algebras. *The Journal of Symbolic Logic*, 16(1):14–21, 1951.
- [30] Woodrow W. Denham. *The Detection of Patterns in Alyawarra Nonverbal Behavior*. PhD thesis, University of Washington, 1973.
- [31] Serge Abiteboul, Richard Hull, and Victor Vianu, editors. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.

- [32] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. ProbLog: A probabilistic prolog and its application in link discovery. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence, IJCAI'07*, pages 2468–2473, Hyderabad, India, January 2007. Morgan Kaufmann Publishers Inc.
- [33] David H D Warren, Luis M. Pereira, and Fernando Pereira. Prolog - the language and its implementation compared with lisp. *SIGPLAN Not.*, 12(8):109–115, 1977.
- [34] Roland de Wolf (auth.) Shan-Hwei Nienhuys-Cheng. *Foundations of Inductive Logic Programming*. Lecture Notes in Computer Science 1228 : Lecture Notes in Artificial Intelligence. Springer-Verlag Berlin Heidelberg, Berlin, Heidelberg, 1 edition, 1997.
- [35] J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3):239–266, August 1990.
- [36] Stephen Muggleton. Inverse entailment and prolog. *New Generation Computing*, 13(3):245–286, 1995.
- [37] Ashwin Srinivasan. *The aleph manual*, 2001.
- [38] Luc De Raedt. *Logical and Relational Learning*. Springer-Verlag Berlin Heidelberg, Berlin, Heidelberg, 1 edition, 2008.
- [39] Ehud Y. Shapiro. *Algorithmic program debugging*. The MIT Press, Cambridge, Massachusetts, EUA, 1 edition, 1983.
- [40] Bradley L. Richards and Raymond J. Mooney. Automated refinement of first-order horn-clause domain theories. *Machine Learning*, 19(2):95–131, 1995.
- [41] Tom M. Mitchell. *Machine learning*. McGraw Hill series in computer science. McGraw-Hill, 1997.
- [42] F. Rosenblatt. *Principles of neurodynamics: perceptrons and the theory of brain mechanisms*. Report (Cornell Aeronautical Laboratory). Spartan Books, 1962.
- [43] Simon Haykin. *Neural networks*, volume 2. Prentice hall New York, 1994.
- [44] Nícia Rosário-Ferreira, Victor Guimarães, Vítor S. Costa, and Irina S. Moreira. SicknessMiner: a deep-learning-driven text-mining tool to abridge disease-disease associations. *BMC Bioinformatics*, 22(1):482, 10 2021.
- [45] Ana Luísa Duboc, Aline Paes, and Gerson Zaverucha. Using the bottom clause and modes declarations on FOL theory revision from examples. *Machine Learning*, 76(1):73–107, 2009.

- [46] Aline Paes, Gerson Zaverucha, and Vítor Santos Costa. On the use of stochastic local search techniques to revise first-order logic theories from examples. *Machine Learning*, 106(2):197–241, 2017.
- [47] Lise Getoor and Benjamin Taskar. *Introduction to Statistical Relational Learning*. MIT Press, 1 edition, 2007.
- [48] 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*, 10(2):1–189, 2016.
- [49] Tirtharaj Dash, Ashwin Srinivasan, Lovekesh Vig, Oghenejokpeme I. Orhobor, and Ross D. King. Large-scale assessment of deep relational machines. In Fabrizio Riguzzi, Elena Bellodi, and Riccardo Zese, editors, *Inductive Logic Programming*, pages 22–37, Cham, 2018. Springer International Publishing.
- [50] Manoel V. M. França, Gerson Zaverucha, and Artur S. d’Avila Garcez. Fast relational learning using bottom clause propositionalization with artificial neural networks. *Machine Learning*, 94(1):81–104, Jan 2014.
- [51] Arnaud Nguembang Fadja, Evelina Lamma, and Fabrizio Riguzzi. Deep probabilistic logic programming. *4th International Workshop on Probabilistic logic programming, PLP 2017*, 11 2017.
- [52] Gustav Sourek, Vojtech Aschenbrenner, Filip Zelezný, Steven Schockaert, and Ondrej Kuzelka. Lifted relational neural networks: Efficient learning of latent relational structures. *Journal of Artificial Intelligence Research*, 62:69–100, 2018.
- [53] Honghua Dong, Jiayuan Mao, Tian Lin, Chong Wang, Lihong Li, and Denny Zhou. Neural logic machines. In *International Conference on Learning Representations*, 2019.
- [54] Navdeep Kaur, Gautam Kunapuli, Saket Joshi, Kristian Kersting, and Sriraam Natarajan. Neural networks for relational data, 2019.
- [55] Alireza Tamaddoni-Nezhad and Stephen Muggleton. The lattice structure and refinement operators for the hypothesis space bounded by a bottom clause. *Machine Learning*, 76(1):37–72, Jul 2009.
- [56] Ryan Riegel, Alexander G. Gray, Francois P. S. Luus, Naweed Khan, Ndivhuwo Makondo, Ismail Yunus Akhalwaya, Haifeng Qian, Ronald Fagin, Francisco Barahona, Udit Sharma, Shajith Iqbal, Hima Karanam, Sumit Neelam, Ankita Likhyan, and Santosh K. Srivastava. Logical neural networks. *CoRR*, abs/2006.13155, 2020.
- [57] Luciano Serafini and Artur S. d’Avila Garcez. Learning and reasoning with logic tensor networks. In Giovanni Adorni, Stefano Cagnoni, Marco Gori, and Marco Maratea,

- editors, *AI*IA 2016 Advances in Artificial Intelligence*, pages 334–348, Cham, 2016. Springer International Publishing.
- [58] Richard Socher, Danqi Chen, Christopher D Manning, and Andrew Ng. Reasoning with neural tensor networks for knowledge base completion. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 926–934. Curran Associates, Inc., 2013.
- [59] Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. DeepProbLog: Neural probabilistic logic programming. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 3749–3759. Curran Associates, Inc., 2018.
- [60] Taisuke Sato. A statistical learning method for logic programs with distribution semantics. In *In Proceedings of the 12th International Conference On Logic Programming (ICLP’95)*, pages 715–729, Tokyo, Japan, June 1995. The MIT Press.
- [61] Richard Evans and Edward Grefenstette. Learning explanatory rules from noisy data. *J. Artif. Int. Res.*, 61(1):1–64, jan 2018.
- [62] Hikaru Shindo, Masaaki Nishino, and Akihiro Yamamoto. Differentiable inductive logic programming for structured examples. In *AAAI*, 2021.
- [63] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, 1997.
- [64] William Yang Wang, Kathryn Mazaitis, Ni Lao, and William W Cohen. Efficient inference and learning in a large knowledge base. *Machine Learning*, 100(1):1–26, 2015.
- [65] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.
- [66] Martín Abadi, Ashish Agarwal, Paul Barham, and Eugene Brevdo *et al.* TensorFlow: Large-scale machine learning on heterogeneous distributed systems, 2015.
- [67] Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah. Signature verification using a “siamese” time delay neural network. In *Proceedings of the 6th International Conference on Neural Information Processing Systems, NIPS’93*, pages 737–744, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.
- [68] Hendrik Blockeel and Luc De Raedt. Top-down induction of first-order logical decision trees. *Artificial Intelligence*, 101(1-2):285–297, 1998.

- [69] Bradley L. Richards and Raymond J. Mooney. Learning relations by pathfinding. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pages 50–55, San Jose, CA, July 1992.
- [70] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.
- [71] Python language reference, version 3.7. available at <http://www.python.org>. Python Software Foundation.
- [72] Ben Taskar, Ming fai Wong, Pieter Abbeel, and Daphne Koller. Link prediction in relational data. In S. Thrun, L. K. Saul, and B. Schölkopf, editors, *Advances in Neural Information Processing Systems 16*, pages 659–666. MIT Press, 2004.
- [73] Luis Antonio Galárraga, Christina Teflioudi, Katja Hose, and Fabian Suchanek. Amie: Association rule mining under incomplete evidence in ontological knowledge bases. In *Proceedings of the 22Nd International Conference on World Wide Web, WWW '13*, pages 413–422, New York, NY, USA, 2013. ACM.
- [74] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.
- [75] Rezarta Islamaj Doğan, Robert Leaman, and Zhiyong Lu. Ncbi disease corpus: A resource for disease name recognition and concept normalization. *Journal of Biomedical Informatics*, 47:1–10, 2014.
- [76] Jinhyuk Lee, Wonjin Yoon, Sungdong Kim, Donghyeon Kim, Sunkyu Kim, Chan Ho So, and Jaewoo Kang. BioBERT: a pre-trained biomedical language representation model for biomedical text mining. *Bioinformatics*, 36(4):1234–1240, 09 2019.
- [77] Nícia Rosário-Ferreira, Catarina Marques-Pereira, Manuel Pires, Daniel Ramalhão, Nádia Pereira, Victor Guimarães, Vítor Santos Costa, and Irina Sousa Moreira. The treasury chest of text mining: Piling available resources for powerful biomedical text mining. *BioChem*, 1(2):60–80, 07 2021.
- [78] Zhen Wang, Jianwen Zhang, Jianlin Feng, and Zheng Chen. Knowledge graph embedding by translating on hyperplanes. In Carla E. Brodley and Peter Stone, editors, *AAAI*. AAAI Press, 2014.
- [79] Yankai Lin, Zhiyuan Liu, Maosong Sun, Yang Liu, and Xuan Zhu. Learning entity and relation embeddings for knowledge graph completion. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, AAAI'15*, pages 2181–2187. AAAI Press, 2015.

- [80] A. P. Dawid. Present position and potential developments: Some personal views: Statistical theory: The prequential approach. *Journal of the Royal Statistical Society. Series A (General)*, 147(2):278–292, 1984.
- [81] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. A survey on concept drift adaptation. *ACM Comput. Surv.*, 46(4):1053–1060, 2014.
- [82] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

Appendix A

The NeuralLog Language

In this appendix, we will describe in details the NeuralLog language. The NeuralLog language is a first-order logic language designed to be compiled into a neural network. The language is based on DataLog [31] and has some additional features to describe neural network structures as well as model and training parameters.

A.1 The Language

A NeuralLog program is composed of a set of Horn clauses [29] of the form:

$$h(.) : - b_1(.), \dots, b_n(.), \text{ not } b_{n+1}(.), \dots, \text{ not } b_m(.).$$

Where $h(.)$ is called the head, and the set of $b_i(.)$ is called the body. Both $h(.)$ and $b_i(.)$ are atoms. An atom is composed of the predicate's name, possibly, followed by a list of terms that the predicate requires. In the case of $h(.)$, h is the predicate and $(.)$ represents the list of terms of the predicate h . The number of terms a predicate supports is called the arity of the predicate, and we can write the predicate h of arity n as h/n . Predicates without terms, this is, predicates of arity 0, are called propositional predicates and have no parentheses. Additionally, we have two special predicates, *true*/0 and *false*/0, which represents the logic values *true* and *false*, respectively.

The body of the Horn clause is composed of literals. A literal is an atom or the negation of an atom (an atom preceded by the *not* keyword). Intuitively, the negation of an atom (*not A*) must be true whenever we cannot prove that the atom (A) is true. In the A.2.2 subsection, we will better detail how negation works in NeuralLog.

A term can be either a string constant, a number or a variable. A string constant represents a logic entity in the knowledge base; a number represents a numeric value that might be

used by the neural network and a variable represents a term that might be replaced by an entity from the logic base or a numeric value, depending on the logic semantics.

A predicate name is represented by a string starting with a lower case letter, possibly, followed by any combination of letters, numbers, dashes and underscores. A constant can be defined by a string following the same rules of the predicate name or by an arbitrary string between single or double quotation marks. A constant cannot be mistaken for a predicate name, since terms appear inside a predicate term definition, surrounded by parentheses; while the predicate name appears outside the parentheses. A number can be defined by any sequence of numbers, using a . (dot) to separate the decimal part; or in scientific notation in the form of a floating number f followed by the letter e and an integer number n , which represents $f \times 10^n$. A variable is defined in the same way as predicates and non-quoted constants, but starting with an upper case letter instead.

Any string after either # or %, until the end of the line; and any block of text in the form /* < block of text > */, surrounded by /**/; are considered comments.

In NeuralLog, we use a set of facts to define the logic entities in the knowledge base and direct relations between those entities. A fact can be seen as a Horn clause with an empty body. For instance, the fact *married(john,marie)*. represents the logic relation *married* between the two entities *john* and *marie*. Which is the same as *married(john,marie) : -true*. NeuralLog only supports grounded facts, this is, facts which does **not** contain variables in its terms. Also, facts must not have arity bigger than two. Similarly, we can have the fact *height(john,1.8)*., which means that the entity *john* has an attribute *height* with value 1.8. All facts have an associated weight that represents the *confidence* in the fact. By default, the associated weight of each defined fact is 1.0, but it can be explicitly defined in the program, using the following syntax $w :: p(.)$, which says that the fact $p(.)$ has weight w . For instance, the fact $0.5 :: rains$. says that the fact *rains* has weight 0.5.

In order to specify more complex relations, we use rules in the form of Horn clauses. An example of a rule is:

$$mother(X,Y) : - parent(X,Y), female(X).$$

Which means that X is mother of Y for every pair of constants x,y that satisfies both requirements: $parent(x,y)$ **AND** $female(x)$. The atom in the head of the rule must have arity of 1 or greater.

A.1.1 Special Predicates

Since NeuralLog was developed to be a language that defines a neural network, which will be later used for learning and prediction of examples, it has five special predicate names to

define the neural network model and the learning parameters, as well as the examples to be used during the training and testing phases.

- *learn*
- *example*
- *mega_example*
- *set_parameter*
- *set_predicate_parameter*

Each one of these special predicate names will be described in details below.

A.1.1.1 Learn

The *learn* predicate name defines two predicates: *learn/1* and *learn/2*.

learn/1 This predicate accepts only one parameter, which is another predicate name. It can be either in the form p/n or p ; where p is the name of the predicate and n is its arity. If the program contains the fact *learn*(" p/n "), the weights of the facts defined by the predicate p with arity n will be marked as learnable and will become variable weights in the neural network, which means that they can be adjusted by the optimization algorithm in order to allow the neural network to better fit the examples. If the arity n is omitted, all the predicates whose name are equal to p will be marked as learnable.

learn/2 This predicate accepts two parameters: the first parameter is the mode, which must be the string *prefix*, and the second parameter is a prefix. If the program contains a fact *learn*(*prefix*, $p_$), the weights of all the predicates whose name **starts with** $p_$ will be marked as learnable, similarly to the *learn/1* predicate.

A.1.1.2 Example

The *example* predicate name defines a family of predicates *example/n*, where $n > 2$. This predicate is used to define the examples that will be used in the learning phase.

For instance, if the program has a fact *example*(*married*, *john*, *marie*), during the learning phase, it will try to adjust its learnable weights in order to predict the fact *married*(*john*, *marie*) with weight 1.0. Alternatively, a fact 0.0 ::

Table A.1: Mega Examples

mega_example(0, *ner*, “The”, “O”).
mega_example(0, *ner*, “sun”, “O”).
mega_example(0, *ner*, “is”, “O”).
mega_example(0, *ner*, “shining”, “O”).
mega_example(0, *ner*, “.”, “O”).

example(*married*, *john*, *alice*). would make the neural network to adjust their weights to predict the fact *married*(*john*, *alice*). with weight 0.0.

A.1.1.3 Mega Example

Similar to the predicate *example*, the predicate *mega_example*/*n*, where $n > 3$, allows the definition of examples to be used for training. The difference is that while the *example* predicate name defines predicates that might be used in the training set in any order and might be even shuffled, the *mega_example* predicate defines a set of examples that might be treated, by the neural network, as a batch of examples.

This predicate is specially useful if the input examples of the network are sequences, for instance, the words in sentences. It guarantees that the words in a sentence will be processed together and in the same order as defined in the program; although the sentences themselves might be shuffled, depending on the configuration. The definition is the same as the *example*, but it starts with an integer number which specifies the group of the example.

For instance, Table A.1 shows a set of facts, which defines a sentence for a text processing task.

A.1.1.4 Set Parameter

The *set_parameter* predicate name defines a family of predicates *set_parameter*/*n*, where $n > 1$. This predicate set the values for different parameters. The first term is the parameter name, the following terms are the names of the parameters inside the previous parameter, and the last term is the value of the parameters.

For instance, the fact *set_parameter*(*optimizer*, *adagrad*). defines that the value of the parameter *optimizer* is *adagrad*. While the pair of facts

set_parameter(*regularizer*, *class_name*, *l2*).
set_parameter(*regularizer*, *config*, *l*, 0.01).

defines that the parameter *regularizer* has two sub-parameters *class_name* and *config*, while *config* is composed of another sub-parameter *l*. Finally, the values of the parameters are *l2* and 0.01, for the *class_name* and *l* parameters, respectively. This value is internally represented as a dictionary, which can be seen, in JSON form, as: `{"regularizer": {"class_name": "l2", "config": {"l": 0.01}}}`.

The final values are converted to the appropriated types, which can be: float, string or boolean.

A.1.1.5 Set Predicate Parameter

Some parameters of the system can be set individually for each different predicate whose parameter may apply. This kind of parameter may have a default value, defined by the *set_parameter* predicate name, but may also have a specific value, for a given predicate, that shall replace the default value when applied to this predicate. In order to allow this behaviour, the parameter name *set_predicate_parameter/n*, where $n > 2$, is used.

This parameter name works similarly to the *set_parameter* predicate name, but its first term is another predicate name. The definition of the predicate parameter argument is the same as the *learn/1* predicate.

For instance, the fact *set_predicate_parameter*("b/1", *initial_value*, *zero*). says that the values of the facts defined by the predicate *b/1* must be initialized to 0.0, when learning the predicate *b/1*.

A.1.2 Special Terms

In order to allow a closer integration between the logic and the neural network parts, we created two special terms that can be used to get information from the logic program in order to be used to create the neural network models. These terms can be used as the last term of either the *set_parameter* or the *set_predicate_parameter* predicates. Both of them will result in an integer value that will be computed at compile time.

The first special term is used to get the number of entities in a given predicate term position and has the form "*\$p/n[i]*". This term will be the number of distinct entities that appears as the *i*-th (starting from 0) term in the predicate of name *p* and arity *n*.

The second special term is used to get the index of an entity in a predicate term position and has the form "*\$p/n[i][e]*". This term will be the index of the entity *e* that appears as the *i*-th (starting from 0) term in the predicate of name *p* and arity *n*.

For instance, consider the NeuralLog program given in Algorithm 4. The lines 1-3 define the

possible labels of the neural network: l_1 , l_2 , l_3 ; while the lines 4-6 define that there will be a predicate $dense/2$ that will be a dense neural network layer with size of “ $\$label/1[0]$ ”, that will result to be the number of possible labels, defined by the predicate $label/1$. On the other hand, the lines 7-9 define the dataset class, whose $empty_word_index$ is equal to the index of the entity ‘ $< EMPTY >$ ’ defined in the position 0 of the predicate $empty_entry/1$.

Algorithm 4 Special NeuralLog Terms

```

1:  $label(l_1)$ .
2:  $label(l_2)$ .
3:  $label(l_3)$ .

4:  $set\_predicate\_parameter($ 
    $“dense/2”, function\_value, class\_name, “Dense”)$ .
5:  $set\_predicate\_parameter($ 
    $“dense/2”, function\_value, config, units, “\$label/1[0]”)$ .
6:  $set\_predicate\_parameter($ 
    $“dense/2”, function\_value, config, activation, “softmax”)$ .

7:  $set\_parameter(dataset\_class, class\_name, sequence\_dataset)$ .
8:  $set\_parameter(dataset\_class, config, expand\_one\_hot, “False”)$ .
9:  $set\_parameter(dataset\_class, config,$ 
    $empty\_word\_index, “\$empty\_entry/1[0][‘ < EMPTY >’]”)$ .

```

A.1.3 Syntax Sugars

NeuralLog also has two syntax sugars in order to facilitate the definition of sets of Horn clauses that have similar form. The first syntax sugar is a for-each loop that iterates a variable over a list of elements. The second syntax sugar is a wildcard that will be replaced accordingly, based on the elements found in the whole NeuralLog program. We further explain those syntax sugars below.

A.1.3.1 For-each Loop

The for-each loop allows the definition of several rules, which share a similar structure, at once. It was inspired by the bash syntax, and has the generic form shown in Algorithm 5. The loop will repeat its content, once for each item in the list, replacing the wildcard of the item in the Horn clauses in it.

The wildcards can appear as a predicate names or terms in the clauses, and is represented by the name of the variable surrounded by curly brackets. The list of items can be a list of

strings separated by space (might include strings containing spaces, surrounded by single or double quotes), or a range of integers in the form $\{i..j\}$, where $i < j$, which will result in a range from i to j with both extremes included.

Algorithm 6 shows an example of each case. In line 1 we have a for-each loop that iterates over a range of integers $[0, 9]$ and in line 4 we have a for-each loop that iterates over the list of items $[setosa, versicolor, virginica]$.

Algorithm 5 For-each Loop Syntax

```

1: for <variable name> in <list of items> do
2:   /* loop content */
3: done

```

Algorithm 6 For-each Loop Example

```

1: for i in {0..9} do
2:   hidden_{i}(X) :- embedding(X, Y), w1(h_{i}).
3: done

4: for type in setosa versicolor virginica do
5:   {type}(X) :- output_{type}(X).
6: done

```

A.1.3.2 Wildcard Syntax

In addition to the for-each loop, NeuralLog has a second syntax sugar which works as a wildcard in the rule. This wildcard appears in the form of a string surrounded by curly brackets, as do the variables in the for-each loop. Those wildcards can appear as (part of) a predicate name or as (part of) a term, either in the head or in the body of a Horn clause. Differently from a variable of a for-each loop, whenever a wildcard appears in the head of a clause, it must also appear in its body.

Then, we create a set S of all possible values a template in the body of a clause may assume, as follows:

- If the wildcard appears as (part of) a predicate name, we add to S all possible values this wildcard could assume, in order to make the name of the predicate equal to the name of another predicate defined in the logic program;
- If the wildcard appears as (part of) a term name, we add to S all possible values this wildcard could assume, in order to make the name of the term equal to the name of another term defined in the program, as long as the names of other predicates containing this wildcard would still exist in the program.

Finally, for each possible permutation of the values of each wildcard in the clause, we generate a new clause, replacing the wildcard by those values.

It is worthy to notice that we must filter the wildcard in the predicate names in the body of a clause, in order not to create new predicates in the body of a clause that do not appear elsewhere in the program, because such clauses will not have a meaningful value. This restriction is not applied to predicate names in the head of a clause, since its definition will be given by the body of the clause. The creation of new terms is also allowed, and new entities are added to the logic program.

The following clause gives an example of the wildcard syntax sugar:

$$activation_{\{i\}}(X) : - hidden_{\{i\}}(X), sigmoid(X).$$

Assuming that there are a set of predicates of the form $hidden_i$, for $i \in [0, 9]$, this clause will generate 10 clauses with the head having the form $activation_i$, one for each $hidden_i$.

A.1.4 Comments

A common feature of programming languages is the ability to add comments in the middle of the code, in order to explain its parts. NeuralLog supports two types of comments: block comments, which may span over multiple lines and are surrounded by an opening and a closing token; and line comments, which start with a specific token and extends until the end of the line.

For block comments, NeuralLog uses the C syntax [82]. A block comment starts with the characters `/*` and ends with the characters `*/`, having the form of:

$$/* COMMENTED BLOCK */$$

For single line comment, NeuralLog supports two distinct tokens to indicate the beginning of the comment: the `%` (percent symbol), which is also the token used by Prolog [33] to define the line comments; and, in order to get a better integration with Python [65], it also supports the `#` (number sign) token to start a line comment.

A.2 Compilation Process

In this section we explain how the compilation process of a NeuralLog program into a neural network works. A NeuralLog program is composed of a set of Horn clauses [29]. These clauses can be divided into two types: (1) facts, which are clauses with empty bodies and no variable terms; and (2) rules, which are the remaining clauses.

Table A.2: A Set of Facts

$p/2$	$height/2$	$w/1$	$rains/0$	Index
$p(a, b).$	$height(a, 1.8).$	$w(a).$	$0.5 : rains.$	0: a
$0.5 :: p(a, c).$	$height(b, 1.5).$	$w(b).$		1: b
$0.3 :: p(b, a).$	$height(c, 3.14).$	$w(d).$		2: c
				3: d

A set of facts of the same predicate is collectively stored in a tensor form, representing the weights of the neural network, while rules are used to define the structure of the neural network. In addition, there are functional predicates that represent differentiable functions in the network.

We give more detail of each one of these elements in the subsections below.

A.2.1 Facts Representation

Let us consider the NeuralLog program composed of the set of facts F in Table A.2, containing a set of entities E ; where each fact has an associated weight, which is 1 when the weight is omitted.

We first construct an index for E by assigning a distinct integer value in $[0, n)$ to each entity $e \in E$, where $n = |E|$. In this case, $n = 4$ and the index is shown in the last column of the Table A.2.

Then, for each predicate in F we create a tensor, where the rank (number of dimensions) of the tensor is the same as the arity of the predicate. For each position in the tensor, we associate the weight corresponding to the fact containing the entities of the indices of the position, if the fact is not in F , we associate the weight of 0.

In the case of the binary predicate $p/2$, we create a matrix $P \in \mathbb{R}^{n \times n}$ where $P_{ij} = w$ if there is a fact $p(e_i, e_j)$, with weight w , in F ; where i and j correspond to the indices of the entities e_i and e_j , respectively. All the remaining entries of the matrix are set to 0. We make an analogous process to the unary (arity 1) and propositional (arity 0) predicates, where the unary predicates are represented by vector in \mathbb{R}^n and propositional predicates are scalar in \mathbb{R} .

In the case of an attribute predicate, we create two vectors, one for the weights and the other for the attribute value. Figure A.1 shows the tensors of the facts in Table A.2.

The rules will define how the tensors are operated. A tensor is only multiplied by another if there is a rule that contains a variable that appears in the predicate of both tensors. In

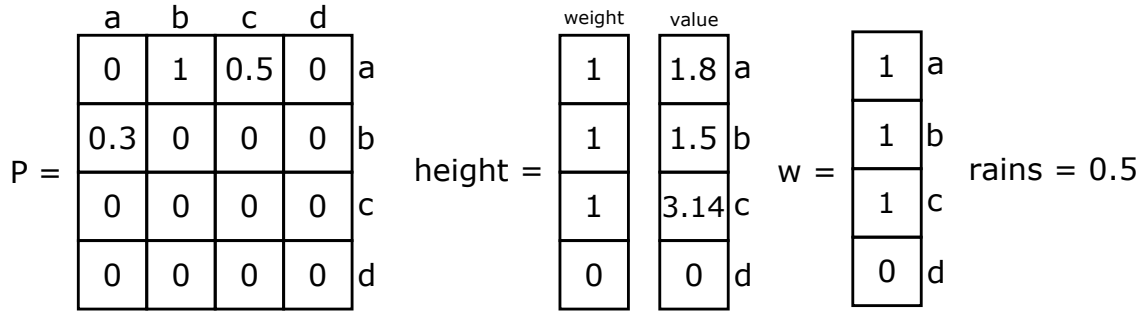


Figure A.1: The Tensors from the NeuralLog Program

order to increase the performance and reduce the amount of used memory, we reduce the dimensions of the tensors to only accommodate the entities needed for that tensor, since the tensors only need to store the entities belonging to itself or to tensors related to it. By analysing the rules in the NeuralLog program, we can identify which tensors are multiplied by each other, and only store, in those tensors, the entities of the related tensors. This might dramatically reduce the dimensions of those tensors. In addition, we use sparse matrices to represent tensors whose weight will not be learned by the neural network and the sparsity of the tensor is above a defined threshold.

A.2.2 Functional Predicates

In addition to the logic predicates, NeuralLog also supports the definition of function predicates. Instead of relying on a logic definition, a function predicate has an associated differentiable function that is applied to its input vector.

There is no explicit difference between functions and logic predicates in the NeuralLog language. The difference is that logic predicates have facts and/or rules defined in the KB and the computations are given by multiplying the input terms with the matrix (or vector) representation of the facts of the predicate and the computation of the rules; while the result of function predicates is the application of the function to the input terms.

There are two ways to use a functional predicate. The first one is to use a predicate whose name matches a defined function. This defined function can be: a function defined in the TensorFlow's Keras API marked as a Keras activation; a Keras layer; or a function defined in NeuralLog, marked with the decorator *neural_log_literal_function*. This way is easier to use, but it is only suitable for parameterless functions, since it does not allow the user to set the function's parameters.

The second way to define a functional predicate is by the use of the *set_predicate_parameter* fact, as explained in the Subsubsection A.1.1.5. In this way, the user can define a

Table A.3: Functional Predicate Definition

```

set_predicate_parameter("dense/2", function_value, class_name, "Dense").
set_predicate_parameter("dense/2", function_value, config, units, 10).
set_predicate_parameter("dense/2", function_value, config, activation, sigmoid).

```

function that will be called on the use of the specified predicate, by setting the parameter *function_value* of this predicate. Table A.3 shows an example of how to define a dense layer with 10 neurons and the sigmoid as activation function; and associate it with the predicate *dense/2*.

As in the first way, the value of the *class_name* must exactly match the name of the desired function, which are the same in both cases. Since the Keras dense layer starts with a capital letter, it could not be used directly as a constant term, since constant terms must start with lower case letters, thus, we have to surround it with quotes.

It is also important to point out that the *dense/2* predicate will be initialized only once. If it is used in different places in the NeuralLog program, the same layer will be applied in all places, meaning that the weights will be shared across different parts of the program. This might be or not be the desired behaviour. In case one wants different weights for each part, different predicates must be defined.

A.2.3 Rules Representation

The rules are Horn clauses with bodies. They define the structure of the neural network and how the predicates relate to each other. A broader explanation of how the rules are compiled to the neural network is given in Section 3.1.

A.2.4 NeuralLog Parameters

In NeuralLog, there are several parameters that can be set in order to customize the compilation of the logic program into the neural networks. Those parameters can be set with the special *set_parameter* and *set_predicate_parameter* predicate names.

In this subsection, we will present all the available parameters, defined in two sets: the model level parameters, with respect to the neural network model; and the training level parameters, with respect to the training phase.

Every available parameter has an associated default value and none of them is required to be defined in order to use NeuralLog; although, by defining other values to those parameters, the user has the flexibility to change several aspects of the constructed model.

A.2.4.1 Model Parameters

In this subsection, we list all the available parameters to define the construction of the neural network model. The parameters marked with a * in the end of name might be individually set to a specific predicate.

If a parameter is set for a specific predicate, this predicate will use the set value for it. Other predicates, that do not have a specific set value, will use the default global value instead.

The *function_value* parameter is the only parameter that has no global value and must be set, individually, for each desired predicate.

- *allow_sparse**: by default, we represent constant facts as sparse matrices whenever possible. Although it reduces the amount of used memory, it limits the multiplication to matrices with rank at most 2. In case one needs to work with higher order matrices, these options must be set to *False*. Default value: *True*.
- *and_combining_function**: function to combine different vectors and get an ‘AND’ behaviour between them. The default is to multiply all the paths, element-wise, by applying the *tf.math.multiply* function. Default value: *tf.math.multiply*.
- *any_aggregation_function*: function to aggregate the input of the *any* predicate. The default function is the *tf.reduce_sum*. Default value: *any_aggregation_function*.
- *attributes_combine_function**: function to combine the numeric terms of a fact. Default value: *tf.math.multiply*.
- *avoid_constant**: Skips adding the constants that appear in the index specified by this parameter, when reading the examples. This may cause entities, that only appear in the examples, not to appear in the knowledge base. This must be handled by the dataset; otherwise, an exception will be raised. Default value: `{}`.
- *dataset_class*: the class to handle the examples. Default value: *default_dataset*.
- *edge_combining_function**: function to extract the value of unary or attribute predicates based on the input. The default is the element-wise multiplication implemented by the *tf.math.multiply* function. Default value: *tf.math.multiply*.
- *edge_combining_function_2d**: function to extract the value of binary predicates based on the input. The default is the matrix multiplication implemented by the *tf.matmul* function. Default value: *tf.matmul*.
- *edge_combining_function_2d:sparse**: function to extract the value of binary predicates based on the input, for sparse matrices. The default is the ma-

trix multiplication implemented by the *tf.matmul* function. Default value: *edge_combining_function_2d:sparse*.

- *edge_neutral_element*: element used to extract the tensor value of grounded literals in a rule. The default edge combining function is the element-wise multiplication. Thus, the neutral element is 1.0, represented by *tf.constant(1.0)*. Default value: `{'class_name' : 'tf.constant', 'config' : {'value' : 1.0}}`.
- *function_value*: the function value of a functional predicate. Default value: `{}`.
- *initial_value**: initializer for trainable predicates. This initializer will be used to initialize facts from trainable predicates that are not in the knowledge base. Default value: `{'class_name' : 'random_normal', 'config' : {'mean' : 0.5, 'stddev' : 0.125}}`.
- *invert_fact_function**: function to extract the inverse of facts. The default is the transpose function implemented by *tf.transpose*. Default value: *tf.transpose*.
- *invert_fact_function:sparse**: function to extract the inverse of facts. The default is the transpose function implemented by *tf.sparse.transpose*. Default value: *tf.sparse.transpose*.
- *literal_combining_function**: function to combine the different proofs of a literal (*Fact Layers* and *Rule Layers*). The default is to sum all the proofs, element-wise, by applying the *tf.math.add* function to reduce the layers and fact outputs. Default value: *tf.math.add*.
- *literal_negation_function**: function to get the value of a negated literal from the non-negated one, the default value is $1 - a$, where a is the result of the non-negated literal. Default value: *literal_negation_function*.
- *literal_negation_function:sparse**: function to get the value of a negated literal from the non-negated one, the default value is $1 - a$, where a is the result of the non-negated literal. Default value: *literal_negation_function:sparse*.
- *output_extract_function**: function to extract the value of an atom with a constant at the last term position. The default function is the *tf.nn.embedding_lookup*. Default value: *tf.nn.embedding_lookup*.
- *path_combining_function**: function to combine different paths from a *Rule Layer*. The default is to multiply all the paths, element-wise, by applying the *tf.math.multiply* function. Default value: *tf.math.multiply*.
- *recursion_depth**: the maximum recursion depth for the predicate. Default value: 1.

- *unary_literal_extraction_function**: function to extract the value of unary predicate. The default is the matrix multiplication, implemented by the *tf.matmul*, applied to the transpose of the literal predicate. Default value: *unary_literal_extraction_function*.
- *value_constraint**: function to be applied to the weights of the trainable predicates in order to constraint its value. This function must take as input the tensor representing the unconstrained weights and return another tensor (of same shape) with the constrained values. Default value: `{}`.
- *weighted_attribute_combining_function**: function to combine the weights and values of the attribute facts. The default function is the *tf.math.multiply*. Default value: *tf.math.multiply*.

A.2.4.2 Training Parameters

In this subsection, we list all the available parameters to define how to train the neural network. Except for the loss function and the metrics, all the parameters are global to the training process and are not associated to any particular predicate.

- *batch_size*: the batch size. Default value: 1.
- *best_model*: a dictionary with keys pointing to *ModelCheckpoints* in the callback dictionary. For each entry, it will save the program and inference files (with the value of the entry as prefix) based on the best model saved by the checkpoint defined by the key. Default value: *None*.
- *callback*: a dictionary of callbacks to be used on training. Default value: *None*.
- *clip_labels*: if *True*, clips the values of the labels to $[0, 1]$. This is useful when one wants to keep the output of the network in the $[0, 1]$ range and also use the *mask_predictions* feature. Default value: *False*.
- *epochs*: the number of epochs. Default value: 10.
- *inverse_relations*: if *True*, also creates the inverted relation for each output predicate. Default value: *False*.
- *loss_function**: the loss function of the neural network and, possibly, its options. It can be individually specified for each output predicate, by just putting another term with the name of the predicate. Default value: *mean_squared_error*.
- *mask_predictions*: if *True*, it masks the output of the network during the training phase. Before the loss function, it sets the predictions of unknown examples to 0

by multiplying the output of the network by the square of the labels. In order for this method to work, the labels must be: 1, for positive examples; -1 , for negative examples; and 0, for unknown examples. Default value: *False*.

- *metrics**: the metric functions to evaluate the neural network and, possibly, its options. The default value is the loss function, which is always appended to the metrics. It can be individually specified for each predicate, by just putting another term with the name of the predicate. Default value: { }.
- *optimizer*: the optimizer for the training and, possibly, its options. Default value: *sgd*.
- *regularizer*: specifies the regularizer, it can be *l1*, *l2* or *l1_l2*. The default value is: *None*.
- *shuffle*: if *True*, it shuffles the examples of the dataset for each iteration. This option is computationally expensive. Default value: *False*.
- *validation_period*: the interval (number of epochs) between the evaluation of the model in the validation set, if set. Default value: 1.