KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Neural Probabilistic Logic Programming

Robin Manhaeve

Supervisor:
Prof. dr. Luc De Raedt

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

December 2021

Neural Probabilistic Logic Programming

Robin MANHAEVE

Examination committee:
Prof. dr. ir. Hendrik Van Brussel, chair
Prof. dr. Luc De Raedt, supervisor
Prof. dr. ir. Hendrik Blockeel
Prof. dr. ir. Tinne Tuytelaars
Prof. dr. Tias Guns
Prof. dr. Artur d'Avila Garcez
(City, University of London)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

December 2021

© 2021 KU Leuven – Faculty of Engineering Science
Uitgegeven in eigen beheer, Robin Manhaeve, Celestijnenlaan 200A box 2402, B-3001 Leuven (Belgium)

Alle rechten voorbehouden. Niets uit deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotokopie, microfilm, elektronisch of op welke andere wijze ook zonder voorafgaande schriftelijke toestemming van de uitgever.

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm, electronic or any other means without written permission from the publisher.

Preface

These four years of my PhD have been quite a journey. However, this PhD would have been impossible without all the help and guidance I have received during these years. The first person I would like to thank is Luc. I still remember the day he invited me into his office to discuss the possibility of doing a PhD and leaving that office with a "gentleman's agreement": he would reserve a spot for me on the SYNTH project, and I promised to start a PhD in September later that year. In the four years since that moment, his mentorship throughout the PhD has been vital to me as a researcher. It was also by his suggestion that I switched from the SYNTH project to the work that is present in this thesis.

Next, I would like to thank all members of the jury. Your time and feedback were vital for the quality of the thesis, and for the PhD to come to a nice conclusion. Hendrik and Tinne, I would like to also thank you as members of the doctoral committee that made sure that my PhD stayed on rails throughout the years. Tias, thank you for your insightful questions during the defense. Artur, thank you for inviting me to come talk at your group, your questions during the defense, your excellent feedback on the thesis and all of your contributions to the field of neural-symbolic AI. I would also like to thank Prof. Van Brussel for chairing the defense. Finally, I would like to thank the FWO for making this research possible by awarding me an SB fellowship.

I would also like to thank my colleagues for making the workplace such a nice place to be. I always looked forward to our coffee breaks and lunches, and the occasional visit to the Oude Markt. I would also like to thank my office mates for the interesting discussions on research, and the irrelevant talks that kept each day interesting. I hope all of you will stick around at our research group for as long as possible.

Outside the people directly related to my research, there are some very special people I want to thank. Mom, dad, I want to thank you for making all of this possible, not only by being the reason I could do the studies that I wanted to,

but also by supporting me in my interests in science and technology growing up, which has led the way to where I am now. Sarah, Jeroen, thank you for being the amazing sister and brother that you are, which has led to me being the person I am today. Quinten, thank you for being an excellent friend, co-houser, provider of the best coffee on campus, and for pushing me through many rough spots (mostly on our grueling, way-too-long trail runs).

Finally, the person closest to my heart, Hanne. Thank you for being there for me at every step of the way, supporting me when things got hard, tolerating me when I was too busy working and celebrating with me when things went right. Thank you for giving me all that I need.

Abstract

In the past decade, deep learning has revolutionized many domains in artificial intelligence (AI), ranging from image classification to natural language processing. Despite these successes, more and more researchers are encountering the limitations of deep learning. Whereas deep learning excels at pattern recognition tasks on high-dimensional data, it struggles with reasoning and generalization. Its strengths and weaknesses are complementary to those of symbolic AI. An integration of symbolic (logic) and sub-symbolic AI (deep learning) could thus bring the best of both worlds. This is the domain of neural-symbolic AI.

This thesis discusses three main contributions. First, we provide a systematic overview of the field of neural-symbolic AI, which is moving very quickly and in which a multitude of different approaches exist. More specifically, we propose a categorization of neural-symbolic AI techniques based on a comparison to the related field of statistical relational AI, which also focuses on integrating reasoning and learning. It turns out that many questions and many well-known concepts in statistical relational AI and neural-symbolic AI can be related to each other.

Second, we introduce our neural-symbolic framework DeepProbLog. DeepProbLog integrates logic and neural networks by introducing the concept of a neural predicate in the probabilistic logic programming language ProbLog. It distinguishes itself from other neural-symbolic methods as it features both probabilistic logic, a fully expressive logic programming language, and neural networks. We evaluate this framework on four sets of experiments that showcase DeepProbLog's ability to: 1) integrate logical reasoning and deep learning, 2) integrate probabilistic reasoning and deep learning, 3) perform program induction through parameter learning, and 4) manipulate embeddings and perform natural language reasoning. The results for these experiments show that DeepProbLog is able to outperform neural networks on tasks requiring

reasoning and learning, and that it is also able to outperform other neural-symbolic frameworks.

Our third contribution is an approximate inference technique for neural probabilistic logic programming, called DPLA*. It addresses one of the main drawbacks of DeepProbLog, i.e. that inference does not scale well because reasoning probabilistically over the entire output of the neural networks can be prohibitively expensive. DPLA* replaces the standard search for proofs by an informed A*-based search that makes inference scalable by only considering a subset of all proofs. However, combining approximate inference with learning brings its own set of challenges. To solve these, we consider the curriculum learning setting and introduce exploration based on the UCB algorithm. We evaluate DPLA* on a set of experiments that showcase that it scales better than DeepProbLog and other neural-symbolic frameworks, and that it can also be applied to a larger set of tasks.

We conclude this thesis by discussing possible future directions for DeepProbLog and by discussing open challenges for the field of neural-symbolic AI.

Beknopte samenvatting

In het afgelopen decennium heeft deep learning gezorgd voor een revolutie in vele toepassing van artificiële intelligentie (AI), gaande van classificatie tot natuurlijke taalverwerking. Na dit eerste succes botsen meer en meer onderzoekers op de beperkingen van deep learning. Het blinkt uit in patroonherkenning op hoog-dimensionale data, maar heeft echter moeite met redeneren en generalisatie. Deze sterktes en zwaktes zijn complementair aan die van symbolische AI. Een integratie van symbolische en sub-symbolische AI (deep learning) zou dus de beste eigenschappen van beide kunnen combineren. Dit is het domein van neuraal-symbolische AI. Deze thesis introduceert drie bijdragen. Eerst geven we een overzicht van het veld van neuraal-symbolische AI door middel van een vergelijking met het gerelateerde veld van statistisch-relationele AI. Vervolgens introduceren we ons neuraal-symbolisch systeem DeepProbLog. Ten slotte introduceren we een techniek voor approximatieve inferentie in neuraal probabilistisch logisch programmeren.

Het veld van neuraal-symbolische AI evolueert zeer snel. Er ontbreekt echter een goed overzicht van de vele verschillende aanpakken. Deze thesis begint met een categorisatie van het veld en een vergelijking met het veld van statistisch-relationele AI dat zich ook focust op het integreren van redeneren en leren.

DeepProbLog integreert logica en neurale netwerken door het concept van het neuraal predicaat te introduceren in de probabilistisch logische programmeertaal ProbLog. DeepProbLog onderscheidt zichzelf van andere neuraal-symbolische methodes doordat het zowel een probabilistisch, logisch en een neuraal aspect bevat. We evalueren DeepProbLog op vier sets van experimenten die aantonen dat het: 1) logisch redeneren en deep learning kan integreren, 2) probabilistisch redeneren en deep learning kan integeren, 3) programma's kan induceren door middel van het leren van parameters, en 4) embeddings kan gebruiken en kan redeneren met natuurlijke taal. De resultaten van deze experimenten tonen aan dat DeepProbLog beter presteert dan neurale netwerken en andere neuraal-symbolische methodes.

Een van de grootste beperkingen van DeepProbLog is dat de inferentie niet goed schaalt. Probabilistisch redeneren over de hele uitvoer van een neuraal netwerk wordt snel onhaalbaar. Daarom introduceren we een approximatieve inferentie methode voor DeepProbLog, genaamd DPLA*. Het vervangt de standaard zoekprocedure naar bewijzen met een op A* gebaseerde zoekprocedure dat inferentie schaalbaar maakt door enkel een subset van alle bewijzen te gebruiken. Het combineren van leren en approximatieve inferentie zorgt echter voor verdere complicaties. Om deze op te lossen maken we gebruiken van curriculum leren en exploratie gebaseerd op het UCB-algoritme. We evalueren DPLA* op een set van experimenten dat aantoont dat het beter schaalt dan DeepProbLog en andere neuraal-symbolische methodes, en dat het toegepast kan worden op een bredere set van taken.

Als conclusie bespreken we mogelijke verdere richting van onderzoek voor DeepProbLog en halen we enkele open problemen aan voor het veld van neuraal-symbolische AI in het algemeen.

List of Abbreviations

$H(E)$ The heuristic over partial proof E. 87

$P(q)$ The success probability of query q. 16

$P_x(q)$ The explanation probability of query q. 16

$T_P(I)$ The immediate consequence operator. 13

μ Geometric Mean. 88

AC Arithmetic Circuit. 20

AD Annotated Disjunction. 18

d-DNNF Decomposable Deterministic Negative Normal Form. 20

DNF Disjunctive Normal Form. 85

DPLA* DeepProbLog with A* proving. 83

GM Geometric-mean heuristic. 89

MCMC Markov Chain Monte Carlo. 85

nAD Neural annotated disjunction. 42

NeSy Neural-symbolic AI. 2

PLP Probabilistic Logic Programming. 15

RBF Radial Basis Function. 48

SDD Sentential Decision Diagram. 20

SLD Selective Linear Definite. 14

StarAI Statistical Relational Artificial Intelligence. 2

UCB Upper Confidence Bound. 91

UCS Uniform-cost search. 88

WMC Weighted model count. 16

Contents

Abstract iii

Beknopte samenvatting v

List of Abbreviations viii

List of Symbols ix

Contents ix

List of Figures xiii

List of Tables xv

1 Introduction 1
- 1.1 Artificial Intelligence and Deep Learning . . . 1
- 1.2 Learning and reasoning . . . 2
  - 1.2.1 Statistical Relational Artificial Intelligence . . . 2
  - 1.2.2 Neural-Symbolic Artificial Intelligence . . . 3
- 1.3 Contributions . . . 5
- 1.4 Structure of the thesis . . . 7

2 Background 9
- 2.1 Logic Programming . . . 10
  - 2.1.1 Logic programs . . . 10
  - 2.1.2 Semantics . . . 11
  - 2.1.3 Inference . . . 13
- 2.2 Probabilistic Logic Programming . . . 15
  - 2.2.1 ProbLog Semantics . . . 15
  - 2.2.2 ProbLog inference . . . 18
  - 2.2.3 Algebraic Model Counting . . . 21

2.2.4 Parameter learning . . . 23
2.3 Deep Learning . . . 24

3 Neural-symbolic AI 27
3.1 Directed vs Undirected . . . 28
3.2 Model-based vs Proof-based Inference . . . 29
3.3 Logic vs Probability vs Neural . . . 30
3.4 Semantics . . . 31
3.5 Structure vs Parameter Learning . . . 32
3.6 Symbols vs Sub-symbols . . . 33
3.6.1 Comparing entities . . . 33
3.6.2 Translating between representations . . . 34
3.7 Type of Logic . . . 35
3.8 Conclusion . . . 39

4 DeepProbLog 41
4.1 The DeepProbLog Language . . . 42
4.1.1 Tensors . . . 42
4.1.2 The Neural Predicate . . . 42
4.1.3 DeepProbLog program . . . 44
4.1.4 Semantics of DeepProbLog . . . 45
4.1.5 Embeddings in DeepProbLog . . . 48
4.2 Inference in DeepProbLog . . . 53
4.3 Learning in DeepProbLog . . . 53
4.3.1 Gradient semiring . . . 55
4.3.2 Gradient-based learning with aProbLog . . . 55
4.3.3 Gradient-based learning for DeepProbLog . . . 56
4.4 DeepProbLog's position along the NeSy dimensions . . . 60
4.5 Experiments . . . 62
4.5.1 **Q1**: Logical reasoning and deep learning . . . 63
4.5.2 **Q2:** Program Induction . . . 68
4.5.3 **Q3:** Probabilistic programming and deep learning . . . 72
4.5.4 **Q4**: Natural Language Reasoning . . . 75
4.5.5 Inference speed . . . 80
4.6 Conclusions . . . 81

5 Approximate inference for Neural-symbolic Logic Programming 83
5.1 Approximate inference in ProbLog . . . 84
5.1.1 Approximate Proving . . . 84
5.1.2 Approximate Weighted Model Counting . . . 85
5.1.3 Other Approaches . . . 85
5.2 Proving as A*-search . . . 85
5.2.1 Partial proofs . . . 86

5.2.2 A* Search . . . 86
5.3 Heuristics . . . 88
5.3.1 Constant Heuristic . . . 88
5.3.2 Geometric Mean Heuristic . . . 88
5.3.3 Parametric Heuristics . . . 89
5.4 Learning with approximate inference . . . 90
5.4.1 Curriculum learning . . . 90
5.4.2 Exploration . . . 90
5.4.3 Training the Parametric Heuristic . . . 91
5.5 Experiments . . . 93
5.5.1 Tasks . . . 94
5.5.2 Baselines . . . 95
5.5.3 Results . . . 95
5.6 Conclusion . . . 104

6 Related work **105**
6.1 Neural-symbolic programming languages . . . 105
6.1.1 Differentiable interpreters . . . 105
6.1.2 Abductive reasoning . . . 106
6.1.3 Interface . . . 107
6.2 Regularization . . . 108
6.2.1 Regularizing embeddings . . . 108
6.2.2 Regularizing neural networks . . . 109
6.3 Templating approaches . . . 110

7 Conclusion **111**
7.1 Results . . . 112
7.2 Future work and Open Challenges . . . 114
7.2.1 Structure learning . . . 114
7.2.2 Tractable inference . . . 114
7.2.3 Undesired solutions . . . 115
7.2.4 Curriculum learning . . . 116
7.2.5 Applications . . . 116

A Experimental details **117**
A.1 Details for the experiments in the DeepProbLog chapter . . . 117
A.2 Details for the experiments in the DPLA* chapter . . . 119

B DeepProbLog Programs **121**
B.1 Programs used in the DeepProbLog chapter . . . 121
B.2 Programs used in the DPLA* Chapter . . . 130

Bibliography **135**

Curriculum Vitae **145**

List of publications **147**

List of Figures

2.1 The SLD tree for the query `calls(mary)` for Example 2.1. . . . 14
2.2 The SLD tree for Example 2.6 showing the derivations for the query `calls(mary)`. . . . 22
2.3 The knowledge compilation and evaluation steps for Example 2.6. For the SDD, A, B, E represent at_home(mary), burglary and earthquake respectively. . . . 22

4.1 The one-dimensional embedding space for MNIST images learned in Example 4.6. Individual MNIST image embeddings are shown in crosses, with the mean for each label as a solid dot. The successor relation is marked by the arrow. . . . 50
4.2 The SLD tree from Example 4.8 54
4.3 The arithmetic circuit from Example 4.8 54
4.4 The AC of the burglary alarm example evaluated using the gradient semiring. Each node is annotated with their value in the gradient semiring. . . . 56
4.5 The AC for query `addition`(, , 1). . . . 61
4.6 MNIST Single-Digit Addition (**T1**). The graphs show the accuracy on the validation set during training. DeepProbLog converges quicker and to a higher accuracy than the baseline, especially with fewer examples. . . . 64
4.7 MNIST Multi-Digit Addition (**T2**). The graphs show the accuracy on the validation set during training for different training set sizes. DeepProbLog converges quicker and to a higher accuracy than the baseline, especially with fewer examples. 65
4.8 The accuracy on the MNIST test set for individual digits while training on (**T3**). The convergence becomes more reliable as the value for the λ parameter increases. . . . 67
4.9 An example input image for the Coins (**T8**) experiment. 73
4.10 An example inference task from the CLUTRR dataset. 75

4.11 The implementation of the neural predicate for relation extraction, illustrated on the relation between Alice and Bob. 76
4.12 The accuracy on the systematic generalization task of CLUTRR for different training sets. k is the relation depth in the training data. DeepProbLog does not lose accuracy as the relation depth increases. 79

5.1 **Q1:** Time to answer queries on the MNIST addition task. . . . 97
5.2 **Q2**: The neural predicate accuracy during training curves for DeepProbLog and DPLA* on the MNIST addition task, with and without pre-training and exploration. The number between brackets is the number of examples used to pre-train the neural predicates. DPLA* converges slower and to a lower accuracy than DeepProbLog, unless pre-training or exploration is used. . 100
5.3 **Q2**: The results on the systematic generalization of the CLUTRR dataset. k indicates the relation lengths present in the training data. Except for $k = 2$, DPLA* is relatively unaffected by the relation depth. 101
5.4 **Q3:** The MIL Addition task with bag size 16 using the parametric heuristic. Pre-training the heuristic and exploration are both necessary to solve this problem well. 103

B.1 Examples of cards without perturbations from the poker experiment (**T9**). 125

List of Tables

3.1 Taxonomy of a (non-exhaustive) list of NeSy models according to the first 3 dimensions. 37
3.2 Taxonomy of a (non-exhaustive) list of NeSy models according to the last 4 dimensions. 38

4.1 The accuracy on the test set for **T1**. DeepProbLog outperforms the baseline, especially when the number of training examples decreases. 63
4.2 The accuracy on the test set for **T2**. DeepProbLog outperforms the baseline, especially when the number of training examples decreases. 66
4.3 The accuracy on the test set for **T4**. Explicitly encoding noise creates a robust model that is able to learn the fraction of noisy labels in the dataset. 68
4.4 Accuracy on the addition (**T5**) problem. The results for $\partial 4$ are as reported by Bošnjak et al. (2017). Training length and test length indicate the length of the sequences for respectively the train and test sets. 69
4.5 Accuracy on the sorting (**T6**) problem. The results for $\partial 4$ are as reported by Bošnjak et al. (2017). Training length and test length indicate the length of the sequences for respectively the train and test sets. $\partial 4$ fails to converge for training lengths ≥ 4 70
4.6 Running time until the models reach 100% accuracy on test length 8 for the sorting (**T6**) problem. 70
4.7 The fraction of runs that converged to different outcomes for the Coins experiment (**T8**). The model converges consistently to the correct solution when given at least 10 labeled examples. 73
4.8 The estimated fraction of the cards in the dataset for the Poker experiment (**T9**). 74

4.9 Robust reasoning on the CLUTRR task for different types of noise. DeepProbLog remains relatively unaffected by the introduction of noisy facts. 79
4.10 Mean and standard deviation of the duration in seconds of different inference steps for 100 queries of the MNIST addition example for increasing number lengths (k). 80

5.1 **Q2** The accuracy on the MNIST addition test set for N=1..3 when trained on datasets of the same length. Although DPLA* can scale to longer numbers, its accuracy is lower, unless pre-training or exploration is used. 99
5.2 **Q2.1** Accuracy on the HWF dataset for expressions of length 1 to 7. Comparison of the proposed approach (DPLA*) with exact DeepProbLog inference and NGS. DPLA* and NGS perform similarly, but NGS fails to converge for some runes for expressions of length 7. 101

A.1 Overview of the hyperparameters used in the experiments. . . . 118
A.2 Overview of the sizes of the datasets used in the experiments. . 118
A.3 Overview of the neural network architectures used in the experiments. 120

Chapter 1

Introduction

1.1 Artificial Intelligence and Deep Learning

As of late, machine learning has dominated the field of artificial intelligence (AI). Especially the field of deep learning has been extremely successful on a wide variety of applications ranging from image classification to natural language processing to reinforcement learning (LeCun et al., 2015; Goodfellow, Bengio, et al., 2016). The reason for this success is that deep learning is able to learn to construct the key features needed to solve the task it is presented with (cf. representation learning), whereas other approaches have to rely on manual feature construction. The success of deep learning has led to the optimism among researchers that deep learning can be used to solve all problems in AI. However, researchers are becoming more and more aware of the limitations of deep learning (Marcus, 2018).

Most successful applications require vast amounts of high-quality, labeled data. Modern computer vision relies on datasets such as the ImageNet dataset (Deng et al., 2009), which contains 14 million hand-labeled images. Natural language processing models are often trained on datasets containing tens to hundreds of gigabytes of textual data (Devlin et al., 2019). In addition to being data-hungry, training such large models also requires large amounts of computational power. Deep learning has also been criticized from the field of explainable AI (Došilović et al., 2018). Deep learning methods (i.e. deep neural networks) are generally considered black-box, as there are no explanations given for the predictions. In fact, research that attempts to open the black box has revealed that deep learning methods often use wrong features to make their predictions, such as

using the snow in the background to predict that the image is of a husky (Ribeiro et al., 2016). Furthermore, neural networks are generally not well suited for reasoning tasks, especially when several reasoning steps are required (Bahdanau et al., 2019; Minervini, Riedel, et al., 2020), or when the task is NP-hard (K. Xu et al., 2020). Often, hand-crafted architectures are proposed to overcome such issues, but these approaches are then no longer generally applicable.

1.2 Learning and reasoning

Artificial intelligence is more than just machine learning. Another facet is that of logic-based reasoning. This approach was itself at the root of an initial wave of optimism surrounding artificial intelligence in the sixties and seventies (Russell and Norvig, 2010). Researchers even went as far as to say that symbolic AI was necessary and sufficient for general AI, cf. the physical symbol system hypothesis (Newell and Simon, 1976). This optimism was struck down however (Lighthill, 1973; Dreyfus, 1976), and an AI winter ensued.

It is interesting to note that the strengths of both learning and reasoning approaches are complementary. Logic based methods allow you to specify expert knowledge that these methods can reason on and deduce new facts from. Learning based approaches are able to discover unknown patterns and missing knowledge from data. That is why the integration of these two approaches is becoming a focus for many researchers. In AI, there are two paradigms that consider the integration of learning and reasoning. First, there's the field of statistical relational AI (StarAI), which integrates logical and statistical (i.e. probabilistic) approaches (De Raedt, Kersting, et al., 2016). The second field is that of neural-symbolic AI (NeSy), which integrates logic based approaches with deep learning.

1.2.1 Statistical Relational Artificial Intelligence

Statistical relational AI (StarAI) seeks to lift logic to a probabilistic level by integrating logic and probabilistic graphical models. Whereas in traditional logic, certain statements are either true or false (or undefined in three-valued logic), in StarAI such statements have a certain probability of being true. StarAI systems can thus naturally deal with uncertainty. This ability to deal with uncertainty is essential when one considers a learning setting. Although a learning setting can also be considered in traditional logic, such systems tend to struggle with noisy or incomplete data, which makes them very brittle. The

capability of StarAI to deal with uncertainty allows it to circumvent such issues, which is why StarAI is able to effectively integrate learning and reasoning.

One example of such a StarAI framework is ProbLog (De Raedt, Kimmig, and Toivonen, 2007), which lifts the logic programming language Prolog to a probabilistic logic programming language through the introduction of the probabilistic fact. More details on ProbLog are given in Chapter 2. We will now exemplify it with the following example:

Example 1.1 (Burglary alarm network)
The following ProbLog program models a variant of the well-known alarm Bayesian network (Pearl, 1988). The program below specifies a probabilistic model. Your alarm is triggered by either a burglar or an earthquake, which both happen with a certain probability. If your neighbor is at home, they will call you if the alarm rings.

```
%Probabilistic facts
0.1::burglary.
0.2::earthquake.
0.5::neighbour_at_home.
%Rules
alarm :- earthquake.
alarm :- burglary.
neighbour_calls :- alarm, neighbour_at_home.
```

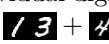
1.2.2 Neural-Symbolic Artificial Intelligence

Neural-symbolic AI (NeSy) seeks to combine the strong function-approximating capabilities of deep learning and the reasoning capabilities of logic-based approaches to create a robust system that can reason and learn. The need for both types of AI has been widely recognized. One example from outside the field of AI is Nobel Prize winner Daniel Kahneman. In his book, *"Thinking, Fast and Slow"* (Kahneman, 2011), he describes two types of thinking in humans: fast (type 1) and slow (type 2). Type 1 thinking here is analogous to neural networks, while type 2 is analogous to logic-based reasoning. LeCun et al. (2015) mention that the next major advancement in AI will be a system that can perform both (complex) reasoning and (representation) learning.

The goal of neural-symbolic computation has already been established for a long time (Garcez, Broda, et al., 2002), namely, the creation of a system that can learn to recognize sub-symbolic data, induce the structure of the task and reason on it in an explainable way. Earlier methods, such as KBANN (Towell

and Shavlik, 1994), focused on translating logic into neural networks, training these networks, and then extracting new knowledge from it. However, since the rise of deep learning, and the insight into its shortcomings, NeSy has gained a lot of popularity and many new methods have been developed since (Sarker et al., 2021; De Raedt, Dumančić, et al., 2020; Garcez and Lamb, 2020; Bekkum et al., 2021). However, a truly comprehensive integration that fulfills all the goals set by the NeSy community has not yet been achieved.

We exemplify neural-symbolic AI by introducing an example that will be used throughout the thesis. It demonstrates a common setting where the system has to reason over sub-symbolic inputs such as images.

Example 1.2 (Hand-written digit addition)
In this example, we are given a pair of images of handwritten digits from the MNIST dataset (Lecun et al., 1998). The goal is to infer the sum of the numbers represented by the images, e.g  $= 8$. It involves a minimal perception task (classifying handwritten digits) that is best handled by a neural network, and a minimal reasoning task (performing the addition) that is best handled in logic. Ideally, the neural network is able to recognize individual digits so that the logic can trivially generalize to larger examples, e.g.  $= 58$.

1.3 Contributions

NeSy and StarAI are both fields that integrate reasoning and learning. However, it is not clear how they exactly compare, which leads to our first research question:

1) *What is the relation between NeSy and StarAI? What are the similarities and differences?*

Our first contribution is an answer to this question in the form of an overview and comparison of the fields of NeSy and StarAI. More specifically, we categorize both NeSy and StarAI methods along seven dimensions. This allows not only to show that these two fields have a lot in common, but also allows us to identify novel research directions for NeSy.

This contribution was published in:

- L. De Raedt, R. Manhaeve, S. Dumančić, T. Demeester, and A. Kimmig (2019). "Neuro-Symbolic = Neural + Logical + Probabilistic". In: *Proceedings of the 2019 International Workshop on Neural-Symbolic Learning and Reasoning*.
- L. De Raedt, S. Dumančić, R. Manhaeve, and G. Marra (2020). "From Statistical Relational to Neuro-Symbolic Artificial Intelligence". In: *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, pp. 4943–4950.

The similarities between these two fields motivated us to postulate that neural-symbolic integrations should have two important properties (De Raedt et al., 2019). The first desirable property is that a neural-symbolic integration should not only consider a neural and a logical component, but also a probabilistic component. This effectively leads to the integration of StarAI and NeSy, which can open up new possibilities. The natural ability of StarAI to deal with uncertainty also makes such systems a great candidate to be included in a neural-symbolic integration, as the predictions made by neural networks are also inherently uncertain. That is why the integration of StarAI and NeSy is the main focus of this thesis. Most state-of-the-art NeSy approaches do not consider a probabilistic component, and rather rely on fuzzy logic or scores that make logic differentiable.

This leads to our second research question:

2) *Are probabilistic logics, and in particular probabilistic logic programming, suitable frameworks for being integrated with neural networks? Does the inclusion of probabilistic reasoning allow for a straightforward integration? Does it open up additional possibilities as compared to neural-symbolic integrations without a probabilistic component?*

The second desirable property is that the base paradigms of the integration need to be preserved as special cases. The rationale behind this is that if these base paradigms cannot be recovered, some abilities have been lost in the integration, and it is thus not a true unification. A correct choice for the mechanism of integration will be vital to achieve this property. In this thesis, we will create an interface between a probabilistic logic programming language and the neural network by introducing the neural predicate. The neural predicate is a general concept that allows for probabilistic logic programming languages to be lifted to the neural-symbolic level, as it allows for part of the program to be parameterized by neural networks.

This leads to our final research question:

3) *Is the neural predicate as an interface between the neural and probabilistic logic programming sides powerful enough to create a widely applicable neural-symbolic system that is able to utilize the neural, logical and probabilistic components fully?*

The second main contribution is an answer to the second and third research questions. We introduce DeepProbLog, a neural-symbolic language and framework that integrates neural networks and probabilistic logic programming. It is the first of such frameworks that has a logic programming, a probabilistic and a neural component. We demonstrate that probabilistic logic programming is a suitable framework to base a neural-symbolic integration on. The probabilistic reasoning aspect makes the integration natural and opens up additional possibilities.

This contribution was published in:

- R. Manhaeve, S. Dumančić, A. Kimmig, T. Demeester, and L. De Raedt (2018). "DeepProbLog: Neural probabilistic logic programming". In: *Advances in Neural Information Processing Systems 31*, pp. 3749–3759.
- R. Manhaeve, S. Dumančić, A. Kimmig, T. Demeester, and L. De Raedt (2021). "Neural probabilistic logic programming in DeepProbLog". In: *Artificial Intelligence* 298, pp. 103–504.

The final contribution strengthens our answers to the second and third research questions by introducing an approximate inference method for neural-symbolic logic programming, called DPLA*. It uses A* to efficiently search for the best subset of all proofs. We show that, with approximate inference, our neural-symbolic framework can be applied to a wider range of tasks.

This contribution was published in:

- R. Manhaeve, G. Marra, and L. De Raedt (2021). "Approximate Inference for Neural Probabilistic Logic Programming". In: *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning*, pp. 475–486.

1.4 Structure of the thesis

Chapter 2 introduces the necessary background for the remainder of this thesis. This includes logic programming, probabilistic logic programming and deep learning. In Chapter 3 we give an analysis of neural-symbolic AI along several dimensions through the comparison with the field of statistical relational AI. Chapter 4 introduces DeepProbLog. It first introduces the neural predicate and the DeepProbLog language. It then gives a description of the semantics and inference of DeepProbLog. The chapter concludes with an experimental evaluation of the DeepProbLog framework. Chapter 5 introduces DPLA*, an approximate inference technique for DeepProbLog based on using a subset of all proofs. It discusses the issues of using approximate inference during learning, and suggest two remedies: curriculum learning and exploration. The chapter finishes with an experimental evaluation of DPLA*. In Chapter 6 we give an overview of the related work in which we give a categorization of neural-symbolic methods based on the mechanism of integration they utilize and compare them to DeepProbLog. We finish the thesis with concluding statements in Chapter 7. We also discuss further directions for DeepProbLog to take and consider open challenges for the field of neural-symbolic AI in general.

Chapter 2

Background

This chapter is largely based on the following paper:

- R. Manhaeve, S. Dumančić, A. Kimmig, T. Demeester, and L. De Raedt (2021). "Neural probabilistic logic programming in DeepProbLog". In: *Artificial Intelligence* 298, pp. 103–504.

In this chapter, we will cover the necessary background for the remainder of this thesis. First, Section 2.1 will cover an introduction to logic programming. Next, Section 2.2 will extend this discussion into probabilistic logic programming, and the ProbLog language in particular. Lastly, Section 2.3 will introduce the notation and necessary concepts of deep learning.

2.1 Logic Programming

In this section, we briefly summarize the basic of logic programming concepts and their syntax. For a more detailed discussion, we refer to (Lloyd, 1989).

2.1.1 Logic programs

A term t is either a constant c, a variable V, or a structured term of the form $f(u_1, ..., u_k)$ where f is a functor and the u_i are terms. Atoms are expressions of the form $q(t_1, ..., t_n)$ where q is a predicate (of arity n, or q/n in shorthand notation) and the t_i are terms. A literal is an atom or the negation of an atom, denoted $\neg q(t_1, ..., t_n)$. A rule is an expression of the form

$$h :- \ b_1, ..., b_n.$$

where h is an atom and the b_i are literals. All variables are universally quantified. Informally, the meaning of such a rule is that h holds whenever the conjunction of the b_i holds. Thus :– represents logical implication ($\leftarrow$), and the comma (,) represents conjunction ($\wedge$). Rules with an empty body $n = 0$ are called facts. A logic program is a finite set of rules. We follow the Prolog syntax convention: constants, functors and predicates start with a lower case character and variables with an upper case. A substitution $\theta = \{V_1 = t_1, ..., V_n = t_n\}$ is a mapping from variables V_i to terms t_i. Applying a substitution θ to an expression e replaces all occurrences of V_i by t_i. The resulting expression is denoted $e\theta$. Expressions that do not contain any variables are called ground.

Example 2.1 (Logic program)
Here, we give an example of a logic program modelling a deterministic variant on the well known burglary-alarm network (Pearl, 1988).

```
%Facts
at_home(mary).
at_home(john).
earthquake.
%Rules
alarm :- earthquake.
alarm :- burglary.
calls(X) :- alarm, at_home(X).
```

It has two constants: *mary* and *john*. There are 5 predicates: *earthquake*/0, *burglary*/0, *alarm*/0, *at_home*/1 and *calls*/1.

2.1.2 Semantics

The semantics of a logic program P defines which atoms a are entailed by the program, denoted $P \models a$.

Least Herbrand base

For logic programs without negation, the semantics can be defined in terms of the *Herbrand base*. The Herbrand universe is the set of all ground terms that can be constructed using the functors and constants occurring in the program. The Herbrand base of a logic program is the set of ground atoms that can be constructed using the predicates and the elements of the Herbrand universe. A Herbrand instantiation of a program is the set of all rules that can be created by substituting variables of the rules in the program by elements of the Herbrand universe. The subsets of a Herbrand base are called Herbrand interpretations. A Herbrand interpretation is a *model* of a rule

$$h :\!- b_1, \ldots, b_n.$$

if, for every substitution θ such that the conjunction $(b_1, \ldots, b_n)\theta$ holds in the interpretation, $h\theta$ is also in the interpretation. A Herbrand interpretation is a model of a logic program if it is a model of all rules in the program. The set of all atoms entailed by a program without negation is the minimal Herbrand model (with respect to set inclusion), known as the least Herbrand model. This least Herbrand model is unique and is considered the only correct model (i.e. the *canonical model*).

Example 2.2 (Least Herbrand model)
The logic program from Example 2.1 has the following Herbrand base:

$$\{earthquake, burglary, alarm,$$

$$at_home(mary), at_home(john), calls(mary), calls(john)\}$$

The logic program has two Herbrand models: the Herbrand base itself and

$$\{earthquake, alarm,$$

$$at_home(mary), at_home(john), calls(mary), calls(john)\}$$

The latter is the minimal Herbrand model and defines the semantics of the logic program, i.e. the set of all atoms entailed by the logic program.

Well-founded semantics

General logic programs also have the concept of negation as failure, denoted $\backslash+ q(t_1, \ldots, t_n)$. The negation of an atom is true exactly if the atom cannot be derived from the program. These programs are not guaranteed to have a unique minimal Herbrand model, and several ways to define a canonical model have been studied. ProbLog follows the well-founded semantics (Van Gelder et al., 1991).

The well-founded semantics requires the concept of a partial interpretation. A *partial interpretation* F is a tuple (F^1, F^0) such that $F^1 \cup F^0$ is a subset of the Herbrand base and $F^1 \cap F^0 = \emptyset$. F^1 and F^0 are the sets of atoms respectively considered *true* and *false*.

The set of atoms that cannot be derived from the program is defined in terms of *unfounded sets*. An unfounded set A for a logic program P with respect to a partial interpretation F is defined as a subset of the Herbrand base such that for all atoms $p \in A$, the following holds:
For each rule in the Herbrand instantiation of P with head p, either:

1. A positive literal in the body of that rule is in F^0.
2. A negative literal in the body of that rule is in F^1.
3. A positive literal in the body of that rule is in A.

The greatest unfounded set of a program with respect to an interpretation is the union of all unfounded sets of a program with respect to that interpretation.

The well-founded model of a logic program can be constructed by iterating the following steps on an initially empty partial interpretation F until convergence.

1. T is the set of all atoms p for which there is a rule in the Herbrand instantiation for P with head p, such that all positive literals in its body are in F^1 and all negative literals in the body are in F^0.
2. U is the greatest unfounded set of P with respect to F.
3. $F = (T, U)$

After convergence, the set of all atoms entailed by the program is F^1 and the set of all negated atoms entailed by the program is F^0.

2.1.3 Inference

The main inference task in logic programming is to determine whether a given atom q (the *query* or *goal*), is true in the canonical model of a logic program P, denoted by $P \models q$. If it is true in this model, the query *succeeds*, otherwise it *fails*. In the case of well-founded semantics, this canonical model is also referred to as the well-founded model. If the query is not ground (i.e. contains variables), the inference searches for *answer substitutions*. An answer substitution is a substitution that grounds the query into a query that succeeds. There are two techniques commonly used for inference in logic programming: forward and backward chaining.

Forward chaining makes use of the immediate consequence operator, denoted $T_P(I)$, where I is a set of ground atoms. As the name implies, the T_P operator calculates the consequence of applying the rules in P where the atoms in I are considered true. Performing inference for a logic program then starts by repeatedly applying the T_P operator, starting with $I = \emptyset$ until $T_P(I) = I$, i.e. a fix-point is reached.

Example 2.3 (Forward chaining)
We now perform forward inference on the logic program from Example 2.1 using the T_P operator.

$$T_P(\emptyset) = \{at_home(mary), at_home(john), earthquake\} = I_1$$

$$T_P(I_1) = I_1 \cup \{alarm\} = I_2$$

$$T_P(I_2) = I_2 \cup \{calls(mary), calls(john)\} = I_3$$

$$T_P(I_3) = I_3$$

The last application shows that the fix-point has been reached.

The benefit of forward chaining is that it deals well with cyclical programs. A downside is that the inference is not guided towards the query. This means that many facts that are irrelevant with respect to the query can be deduced, potentially leading to a lot of wasted computation.

Conversely, backward chaining starts from the query and works backwards towards a proof. Backward chaining keeps track of a goal, which is a conjunction of literals

$$?\text{–}\, l_1, \ldots, l_j$$

It then selects an atom l_i from the goal and applies a rule from the program.

$$h \text{ :– } b_1, \ldots, b_n.$$

such that h unifies with l_i with substitution θ. This is done using Selective Linear Definite Resolution (SLD). Generally, the first literal l_1 is selected. After the application of the rule, the new goal becomes

$$?\text{–}(b_1, \ldots b_n, l_2, \ldots, l_j)\theta$$

This is repeated until an empty goal is deduced or until no more rules can be applied. If the empty goal is deduced, then the query succeeds, otherwise, it fails. Note that the rule selection is not deterministic. Rather, this procedure defines a search tree over possible derivations that can be traversed in several ways, called the SLD tree.

The benefit of backward chaining is that the reasoning is focused on answering the query (i.e. irrelevant parts of the program are not used). However, the search tree defined by the procedure might have infinite branches in the case of a cyclical program, so special care needs to be taken to deal with this.

Example 2.4 (Backward chaining)
The backward chaining algorithm can be summarized in an SLD-tree. The different goals encountered throughout the procedure are represented as nodes. The top node in such a tree is the query. Successful proofs are represented by paths in the tree from the query node to an empty goal leaf node (represented by `true`).

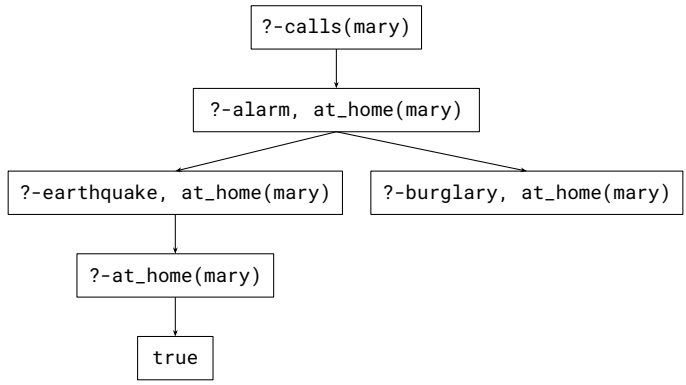


Figure 2.1: The SLD tree for the query `calls(mary)` for Example 2.1.

2.2 Probabilistic Logic Programming

Probabilistic Logic Programming (PLP) is the integration of logic programming and probabilistic (graphical) models. Although it has a rich history and many PLP methods have been developed (Ng and Subrahmanian, 1992; De Raedt and Kimmig, 2015), we will focus on ProbLog (De Raedt, Kimmig, and Toivonen, 2007).

2.2.1 ProbLog Semantics

ProbLog (De Raedt, Kimmig, and Toivonen, 2007) integrates Prolog and probabilities through the introduction of a single unifying concept: the *probabilistic fact*.

Definition 2.1 (Probabilistic fact)
A probabilistic fact is an expression of the form

$$p :: f$$

where $P(f) = p$ is a probability and f is an atom.

This is the only new concept needed to define ProbLog programs.

Definition 2.2 (ProbLog program)
A ProbLog program T consists of a set of probabilistic facts $T_{\mathcal{F}}$ and a set of rules $T_{\mathcal{R}}$.

Each ground instance $f\theta$ of a probabilistic fact f corresponds to an *independent Boolean random variable* that is true with probability p and false with probability $1 - p$. Let us denote the set of all ground instances of probabilistic facts in $T_{\mathcal{F}}$ as $T_{\mathcal{F}}\Theta$.

For ProbLog, the definition of a partial interpretation is slightly different, as F^1 and F^0 are subsets of $T_{\mathcal{F}}\Theta$ rather than the Herbrand base.

Definition 2.3 (Partial Interpretation)
A *partial interpretation* F of a program T is a tuple (F^1, F^0) such that $F^1 \cup F^0 \subseteq T_{\mathcal{F}}\Theta$ and $F^1 \cap F^0 = \emptyset$. F^1 and F^0 are the sets of probabilistic facts respectively assigned the values *true* and *false*.

The probability of a partial interpretation F is given by the product of the probabilities of the random variables corresponding to the ground probabilistic

facts in F:

$$P(F) = \prod_{f_i \in F^1} P(f_i) \prod_{f_i \in F^0} 1 - P(f_i) \tag{2.1}$$

A *complete interpretation* is an interpretation that assigns a value to every probabilistic fact, i.e.

$$T_{\mathcal{F}}\Theta \setminus (F^1 \cup F^0) = \emptyset$$

The set of completions of a partial interpretation I in a program T is defined as:

$$Compl^T(I) = \{(J^1, J^0) \mid I \subseteq J \text{ and } J \text{ a complete interpretation of T}\}$$

Every complete interpretation F of a program T defines a possible world

$$w_F = F^1 \cup \{h\theta \mid T_{\mathcal{R}} \cup F^1 \models h\theta \text{ and } h\theta \text{ is ground}\}$$

The possible world w_F is the model obtained by adding the facts F^1 to the set of rules $T_{\mathcal{R}}$. The probability $P(w_F)$ of such a possible world w_F is equal to $P(F)$.

A query q is supported by a partial interpretation I (denoted $I \models q$) if and only if for all complete interpretations $J \in Compl^T(I)$ there exists a substitution θ such that $q\theta \in w_J$. To keep the presentation simple, we focus on the case of finitely many ground probabilistic facts, but note that the semantics is also well-defined for the countably infinite case.

The set of proofs (or explanations) of a query q is the set of all minimal interpretations E such that $E \models q$.

$$\text{Proofs}(q) = \{E \mid E \models q \wedge \neg\exists I : I \subset E \wedge I \models q\}$$

The *success probability* $P(q)$ of a ground query q is defined as the sum of the probabilities of all possible worlds containing q, i.e.,

$$P(q) = \sum_{q \in w_F} P(w_F) \tag{2.2}$$

The success probability of a query is also equal to the weighted model count (WMC) of the worlds where this query is true. See (Kimmig, Van den Broeck, et al., 2017) for more details

The *explanation probability* $P_x(q)$ of a ground query q is defined as the maximal probability of any proof of q

$$P_x(q) = \max_{E \in \text{Proofs}(q)} P(E) \tag{2.3}$$

Example 2.5 (Bayesian network)
We refer back to Example 1.1. The ProbLog program below models a variant of the well-known alarm Bayesian network (Pearl, 1988):

```
%Probabilistic facts
0.1::burglary.
0.2::earthquake.
0.5::at_home(mary).
%Rules
alarm :- earthquake.
alarm :- burglary.
calls(X) :- alarm, at_home(X).
```

The set of all possible ground probabilistic facts of this program is

$$\mathcal{F}\Theta = \{\texttt{burglary}, \texttt{earthquake}, \texttt{at_home(mary)}\}$$

An example of a complete interpretation is

$$F = (F^1, F^0)$$

$$F^1 = \{\texttt{burglary}, \texttt{at_home(mary)}\}$$

$$F^0 = \{\texttt{earthquake}\}$$

The possible world defined by this interpretation is

$$w_F = \{\texttt{burglary}, \texttt{at_home(mary)}\} \cup \{\texttt{alarm}, \texttt{calls(mary)}\}$$

The table below shows the possible worlds of this ProbLog program.

Possible world	$P(w_F)$
{}	0.36
{at_home(mary)}	0.36
{earthquake, alarm}	0.09
{earthquake, alarm, at_home(mary),calls(mary)}	**0.09**
{burglary, alarm}	0.04
{burglary, alarm, at_home(mary),calls(mary)}	**0.04**
{burglary, earthquake, alarm}	0.01
{burglary, earthquake, alarm, at_home(mary),calls(mary)}	**0.01**

The success probability of the query `calls(mary)` is

$$P(\texttt{calls(mary)}) = 0.09 + 0.04 + 0.01 = 0.14$$

The explanation probability is

$$P_x(\texttt{calls(mary)}) = 0.09$$

For ease of modeling, ProbLog supports non-ground probabilistic facts as a shortcut for introducing a set of ground probabilistic facts, as well as annotated disjunctions (ADs), which are expressions of the form

$$p_1 :: h_1 \ ; \ ... \ ; \ p_n :: h_n \ :- \ b_1, ..., b_m.$$

where the p_i are probabilities that sum to at most one, the h_i are atoms, and the b_j are literals. The meaning of an AD is that whenever all b_i hold, the AD causes one of the heads h_j to be true, or none of them with probability $1 - \sum p_i$. Note that several of the h_i may be true at the same time if they also appear as heads of other rules or ADs. This is convenient to model choices between different categorical variables, e.g. different severities of the earthquake:

$0.4 :: \texttt{no_earthquake}$; $0.4 :: \texttt{mild_earthquake}$; $0.2 :: \texttt{severe_earthquake}$.

or without explicitly representing the event of no earthquake:

$$0.4 :: \texttt{mild_earthquake} \ ; \ 0.2 :: \texttt{severe_earthquake}.$$

in which neither `mild_earthquake` nor `severe_earthquake` is true with probability 0.4. Annotated disjunctions do not change the expressivity of ProbLog, as they can alternatively be modeled through independent facts and logical rules. We refer to De Raedt and Kimmig (2015) for technical details.

2.2.2 ProbLog inference

Inference in ProbLog is the process of calculating the (success) probability of a query q given evidence e

$$P(q|e) = \frac{P(q \wedge e)}{P(e)} \tag{2.4}$$

For this thesis, we will consider the evaluation of $P(q|e)$ as two separate evaluations of $P(q \wedge e)$ and $P(e)$, which simplifies the discussion on inference through this thesis. However, such queries can be executed more efficiently (Fierens et al., 2015). Computing the probability $P(q)$ happens in several steps.

Proving/Grounding

The first step is the logical inference. We discern two different but related approaches. One approach is proving. During the proving procedure, SLD resolution is used to calculate the set of all proofs for a query q.

Another approach is to construct the relevant ground program $G(q)$. Grounding replaces each rule c containing variables $\{V_1, ..., V_k\}$ by all instances $c\theta$ where θ is a substitution $\{V_1 = c_1, ...V_k = c_k\}$ and the c_i are constants or other ground terms appearing in the domain. If q is not ground, the grounding will compute all possible answer substitutions $q\theta$. To keep inference tractable, it is key to only consider the part of the ground program that is relevant to the query (i.e. the grounded facts and rules are used in the derivations of the query). Again, SLD resolution is used to find the relevant grounding.

Logical formula construction

The construction of the logical formula is dependent on how the previous step was performed. In the proving approach, the proofs for a query q are combined into a logical formula

$$q \leftrightarrow \bigvee_{E \in \text{Proofs}(q)} \bigwedge_{f_i \in E^1} f_i \bigwedge_{f_i \in E^0} \neg f_i$$

In the grounding approach, Clark's completion can be used for cycle-free programs. Clark's completion constructs a formula

$$h \leftrightarrow \bigvee_{(h :- b_1, \ldots, b_n) \in G(q)} b_1 \wedge \cdots \wedge b_n$$

for each set of rules with the same ground head h, whose bodies are $b_1, \ldots, b_n$. Cyclical programs first need to be turned into equivalent acyclic programs through cycle breaking. For this, we refer to (Fierens et al., 2015).

Knowledge compilation

Computing $P(q)$ directly on this formula is not efficient. To help alleviate this, knowledge compilation (Darwiche and Marquis, 2002) is used. Knowledge compilation works by splitting the process of calculating $P(q)$ into two parts: an expensive offline step and a cheap online step. This has the benefit that the re-evaluation of the same query but with different parameters is faster,

which is particularly relevant in a learning setting. Knowledge compilation achieves this by transforming the logical formula into a target language for which a certain set of queries is efficient (i.e. polytime evaluation in the size of the new representation). An example of such a target representation is the decomposable, deterministic negative normal form (d-DNNF) (Darwiche, 2001). Decomposable means that, for every conjunction, the conjuncts do not share any variables. Deterministic means that, for every disjunction, the disjuncts are mutually exclusive. This translation effectively solves the disjoint-sum problem (Kimmig, Van den Broeck, et al., 2017) so that the query probability can be calculated efficiently. In this work, we use the sentential decision diagram (SDD) as the target representation (Darwiche, 2011). An example of knowledge compilation with SDDs is given in Example 2.6.

Evaluation

The compiled representation can be trivially turned into an arithmetic circuit (AC). The facts, which are located at the leaves, are replaced by their respective probabilities. An AND is replaced by a multiplication, and an OR is replaced by a summation. $P(q)$ is then computed by a bottom-up evaluation of this arithmetic circuit.

Example 2.6 (ProbLog inference)
We calculate the probability of the query `calls(mary)` on the program from Example 2.5. In the first step, we can consider both the proving and the grounding approach. The proving approach is best summarized by the SLD tree shown in Figure 2.2. The procedure finds two proofs, one where the alarm is triggered by an `earthquake` and one where the alarm is triggered by a `burglary`.

The relevant ground program for the query `calls(mary)` is given below:

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

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

Clark's completion of this ground program is

$$calls(mary) \leftrightarrow alarm \wedge at_home(mary)$$

$$alarm \leftrightarrow earthquake \vee burglary$$

or simplified

$$calls(mary) \leftrightarrow (burglary \vee earthquake) \wedge at_home(mary)$$

The resulting logical formula from the proving procedure is

$$calls(mary) \leftrightarrow (burglary \wedge at_home(mary)) \vee (earthquake \wedge at_home(mary))$$

Note that the formula resulting from Clark's completion can be simplified to the formula from the proving procedure. The two disjuncts in this formula are not mutually exclusive. The probability of the disjunction can thus not be calculated as the sum of the probabilities of the disjuncts, but rather requires solving the disjoint-sum problem. We now use knowledge compilation to turn the formula into a d-DNNF. The resulting formula is:

$$(\neg earthquake \wedge burglary \wedge at_home(mary)) \vee (earthquake \wedge at_home(mary))$$

Through the introduction of $\neg earthquake$, the two disjuncts are now mutually exclusive, and the formula is now a d-DNNF. An alternative to d-DNNFs are sentential decision diagrams (SDDs) (Darwiche, 2011). SDDs are a subset of d-DNNFs, but they are more succinct in general. In this case, the d-DNNF and SDD are equivalent. The SDD for this formula is shown in Figure 2.3a. The round nodes represent a disjunction, and the rectangles denote the conjunction.

We now turn this d-DNNFs/SDD into an arithmetic circuit as shown in Figure 2.3b. Evaluating this arithmetic circuit results in $P(calls(mary)) = 0.14$

2.2.3 Algebraic Model Counting

Algebraic model counting (Kimmig, Van den Broeck, et al., 2017) generalizes weighted model counting to arbitrary commutative semirings. This idea was used to introduce aProbLog (Kimmig, Van den Broeck, et al., 2011), a generalization of the ProbLog language and inference to arbitrary commutative semirings, including the gradient semiring (Eisner, 2002) which allows for automatic differentiation with respect to the parameters in the program (cf. Section 4.3).

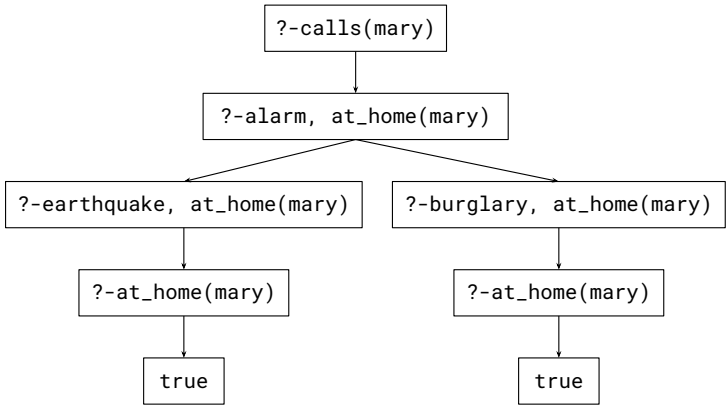
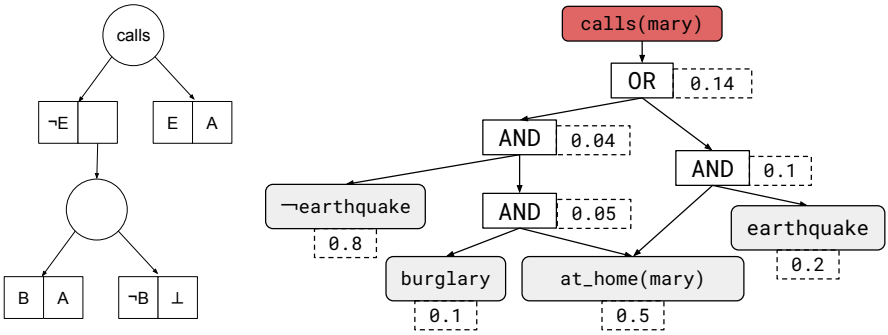


Figure 2.2: The SLD tree for Example 2.6 showing the derivations for the query `calls(mary)`.



(a) The SDD. (b) The resulting arithmetic circuit.

Figure 2.3: The knowledge compilation and evaluation steps for Example 2.6. For the SDD, A, B, E represent at_home(mary), burglary and earthquake respectively.

aProbLog

ProbLog annotates each probabilistic fact f with the probability p that f is true, which implicitly also defines the probability $1 - p$ that its negation $\neg f$ is true. It then uses the probability semiring with regular addition and multiplication as operators to compute the probability of a query on the AC constructed for

this query. The probability semiring is defined as:

$$a \oplus b = a + b$$

$$a \otimes b = ab$$

$$e^{\oplus} = 0$$

$$e^{\otimes} = 1$$

And the accompanying labeling function as:

$$L(f) = p \qquad \text{for } p :: f$$

$$L(\neg f) = 1 - p \qquad \text{with } L(f) = p$$

This idea is generalized in aProbLog to compute such values based on arbitrary commutative semirings. Instead of probability labels on facts, aProbLog uses a labeling function that associates values from the chosen semiring with both facts and their negations, and combines these using semiring addition $\oplus$ and multiplication $\otimes$ on the AC.

Whereas ProbLog is confined to only calculating probabilities, the use of other semirings can change the task that is being solved, without changing any other parts of the inference. Examples include SAT, MPE, sensitivity analysis and differentiation (Kimmig, Van den Broeck, et al., 2011).

2.2.4 Parameter learning

Some probabilities in the ProbLog program might not be known, but should rather be learned from data. These are called the parameters of the ProbLog program. ProbLog, just like any other statistical relational AI model such as Markov Logic (De Raedt, Kersting, et al., 2016; P. Domingos and Lowd, 2009), can be trained discriminatively as well as generatively. In the generative setting, the examples are (partial) possible worlds or interpretations, while in the discriminative setting they correspond to facts for a specific target predicate with the evidence residing in the background theory.

To distinguish between fixed probabilities and parameters, the following notation is used for the latter:

$$t(p) :: f$$

here p is the initial value for the parameter. The probability of a query is thus dependent on a set of parameters $\mathcal{W}$. When necessary to avoid confusion, this is denoted $P(q|\mathcal{W})$.

In this thesis, we focus on the discriminative training setting, which is called *learning from entailment* (Frazier and Pitt, 1993) in ProbLog.

Definition 2.4 (Learning from entailment)
Given a ProbLog program with parameters $\mathcal{W}$, a set $\mathcal{Q}$ of tuples (q, p) with q a query, p its desired success probability, and a loss function $\mathcal{L}$, compute:

$$\arg\min_{\mathcal{W}} \frac{1}{|\mathcal{Q}|} \sum_{(q,p)\in\mathcal{Q}} \mathcal{L}(P(q|\mathcal{W}), p)$$

For example, when using cross-entropy loss, the above formula becomes

$$\arg\min_{\mathcal{W}} \frac{1}{|\mathcal{Q}|} \sum_{(q,p)\in\mathcal{Q}} -\left(p \log(P(q|\mathcal{W})) + (1-p)\log(1-P(q|\mathcal{W}))\right)$$

2.3 Deep Learning

In this section, we touch upon the necessary details to understand the deep learning techniques used in the next chapters. For a more in-depth discussion, we refer to works such as (Goodfellow, Bengio, et al., 2016).

An artificial neural network is a highly parameterized and therefore very flexible non-linear mathematical function that can be trained towards a particular desired behavior by suitably adjusting its parameters. During training, the model learns to capture the most informative features from the input data for the task at hand. The need for feature engineering in non-neural machine learning methods has therefore been replaced by architecture engineering, since a wide variety of neural network components are available to be composed into a suitable model.

Deep neural networks are often designed and trained in an end-to-end fashion, whereby only the raw input and the final target are known during training, and all components of the model are jointly trained. For example, for the task of handwritten digit recognition, an input instance consists of an image of a handwritten digit, and its target is the actual digit.

Consider a supervised learning problem, with a training set $\{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^{N}$ containing N independent and identically distributed input instances $\boldsymbol{x}_i$ with corresponding outputs $\boldsymbol{y}_i$. A model, represented by a mapping function $\mathcal{M}$ with parameters $\mathcal{W}$, maps an input item $\boldsymbol{x}$ to the corresponding predicted output $P(Y|X=\boldsymbol{x}) = \mathcal{M}(\boldsymbol{x}|\mathcal{W})$.

To quantify how strongly the predicted output $P(Y|X = \boldsymbol{x})$ deviates from the target output $\boldsymbol{y}$, a loss function $\mathcal{L}(P(Y|X = \boldsymbol{x}), \boldsymbol{y})$ is defined. Training the model then comes down to minimizing the expected loss over the training set:

$$\frac{1}{N} \sum_i \mathcal{L}\big(\mathcal{M}(\boldsymbol{x}_i|\mathcal{W}), \boldsymbol{y}_i\big)$$

In the specific setting of multiclass classification, each input instance corresponds to one out of a fixed set of M output categories. The target vectors $\boldsymbol{y}$ are typically represented as one-hot vectors: all components are zero, except at index m of the corresponding category. The predicted counterpart $P(Y|X = \boldsymbol{x})$ at the model's output is often obtained by applying a softmax output function to intermediate real-valued scores $\mathbf{s}$ obtained at the output of the neural network. The i-th component of the softmax is defined as

$$\text{softmax}(\mathbf{s})_i = \hat{y}_i = \frac{e^{s_i}}{\sum_j e^{s_j}}$$

The softmax outputs are well-suited to model a probability distribution (i.e., $0 < \hat{y}_i < 1$ and $\sum_i \hat{y}_i = 1$). The standard corresponding loss function is the cross-entropy loss, which quantifies the deviation between the empirical output distribution $\hat{\boldsymbol{y}}$ (i.e., the softmax outputs) and the ground truth distribution (i.e., the one-hot target vector $\boldsymbol{y}$) defined as

$$\mathcal{L} = -\sum_j y_j \log \hat{y}_j$$

The neural network models we will focus on in this work, are *discriminative* classifiers, which directly model the probability distribution $P(Y|X = \boldsymbol{x}_i)$ of the target classes, given the input feature representation $\boldsymbol{x}_i$ of the i-th example. We currently do not consider *generative* classification models. These would also allow modeling the output labels given the inputs, by applying Bayes' law on the prior class probabilities and the inputs distribution given the labels. Yet, we argue that taking into account background information such as prior class probabilities is more naturally handled by the reasoning component in the case of a neural-symbolic integration, where the key idea remains that each part of the integration should focus on what it is best at. Although there has been research on deep generative models Kingma and Welling, 2014, they are unnecessarily complex for the research considered in this thesis.

The most widely used optimization approaches for neural networks are variations of the gradient descent algorithm, in which the parameters $\mathcal{W}$ are iteratively updated by taking small steps along the negative gradient of the loss. An estimate $\mathcal{W}_n$ at iteration n is updated as $\mathcal{W}_{n+1} = \mathcal{W}_n - \lambda \nabla_{\mathcal{W}} \bar{\mathcal{L}}$, in which the

step size is controlled by the learning rate λ. Typically, training is not performed by averaging over the entire dataset per iteration, but instead over a smaller mini-batch of instances. This is computationally more efficient and allows for a better exploration of parameter space. Importantly, the loss gradient can only be calculated if all components of the neural network are differentiable.

A deep neural network typically has a layer-wise architecture: the different layers correspond to nested differentiable functions in the overall mapping function $\mathcal{M}$. The forward pass through the network corresponds to consecutively applying these layer functions to a given input to the network. The intermediate representations obtained by evaluating these layer functions are called hidden states. After a forward pass, the gradient with respect to all parameters can then be calculated by applying the chain rule. This happens during the backward pass, where the gradients are calculated from the output back to the first layer. As an illustration of how the chain rule is applied, consider the network function $\mathcal{M}(\boldsymbol{x}|\mathcal{W}) = \mathbf{g}\big(\mathbf{f}(\boldsymbol{x}, w_f), w_g\big)$, which contains a first layer represented by the vector function $\mathbf{f}$, and a second layer $\mathbf{g}$. For simplicity, each layer has one trainable parameter, respectively written as w_f and w_g. The derivative with respect to these parameters of a scalar loss function applied to the network output, becomes

$$\nabla_{\mathcal{W}}\mathcal{L}\big(\mathcal{M}(\boldsymbol{x}|\mathcal{W})\big) = \left[\frac{d\mathcal{L}}{dw_f},\ \frac{d\mathcal{L}}{dw_g}\right] = \left[\sum_i \frac{\partial \mathcal{L}}{\partial g_i}\sum_j \frac{\partial g_i}{\partial f_j}\frac{\partial f_j}{\partial w_f},\ \sum_i \frac{\partial \mathcal{L}}{\partial g_i}\frac{\partial g_i}{\partial w_g}\right]$$

in which the individual derivatives are evaluated based on the considered input $\boldsymbol{x}$ and current value of the parameters. The entire procedure to calculate the gradients is called the backpropagation algorithm. It requires a forward pass to calculate all intermediate representations up to the value of the loss. After that, in the backward pass, the gradients corresponding to all operations applied during the forward pass, are iteratively calculated, starting at the loss (i.e., with $\partial\mathcal{L}/\partial g_i$ in the example). As such, the gradients with respect to parameters at a given layer can be calculated as soon as the gradients due to all operations further in the network are known, as governed by the chain rule.

Chapter 3

Neural-symbolic AI

This chapter is largely based on:

- L. De Raedt, S. Dumančić, R. Manhaeve, and G. Marra (2020). "From Statistical Relational to Neuro-Symbolic Artificial Intelligence". In: *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, pp. 4943–4950.

Neural-symbolic AI (NeSy) is the field that tries to integrate neural networks and symbolic approaches in a way that brings the best of both worlds. In essence, it tries to integrate learning and reasoning. This goal however is not unique to NeSy. Another subfield of AI that integrates learning and reasoning is the well established field of statistical relational AI (StarAI). For an in-depth discussion of StarAI, see (De Raedt, Kersting, et al., 2016). By drawing parallels, we discuss methods from both fields and analyze them along seven dimensions:

1. Directed vs undirected
2. Model-based vs proof-based
3. Logic vs probability vs neural
4. Semantics
5. Structure vs parameter learning
6. Symbolic vs sub-symbolic
7. Types of logic

3.1 Directed vs Undirected

Within the graphical model community, there is a distinction between *directed* and *undirected* graphical models (Koller and Friedman, 2009), which has led to two distinct types of StarAI systems. The first generalizes directed models, and resembles Bayesian networks; the second generalizes undirected models like Markov networks or random fields. The key difference between the two is that the first class of models indicates a natural direction between the different random variables, while the second one does not.

In StarAI, the first category includes well-known representations such as plate notation (Koller and Friedman, 2009), probabilistic relational models (PRMs) (Friedman et al., 1999), probabilistic logic programs (PLPs) (De Raedt and Kimmig, 2015), and Bayesian logic programs (BLPs) (Kersting and De Raedt, 2007). Today, the most typical and popular representatives of this category are the probabilistic (logic) programs. The second category includes Markov Logic Networks (MLNs) (Richardson and P. M. Domingos, 2006) and Probabilistic Soft Logic (PSL) (Bach et al., 2017). They specify a set of weighted constraints, clauses or formulae.

From a logical perspective, the difference amounts to using a form of definite clauses (cf. Section 2.1) versus the use of full clausal logic or even first order logic. Full clausal logic allows for formulae of the form $h_1 \vee ... \vee h_m \leftarrow b_1, ..., b_n$. Definite clauses can be interpreted as if-then rules. A general clause on the other hand specifies a more general relationship between the atoms in the condition and the conclusion. They can be viewed as constraints that relate these two sets of atoms, as is common in Answer Set Programming (Gebser et al., 2012).

This same distinction can also be observed in NeSy systems. The first category includes systems which retain the directed nature of logical inference. The most prominent members of this category are NeSy systems based on Prolog or Datalog, such as neural theorem provers (NTPs) (Rocktäschel and Riedel, 2017), NLProlog (Weber et al., 2019), DiffLog (Si et al., 2019) and TensorLog (Cohen et al., 2018).

The undirected NeSy approaches use logical formulas as a soft constraint over the neural network output. During training, an additional regularization term encourages solutions consistent with this knowledge. Semantic Based regularization (SBR) (Badreddine et al., 2021), Logic Tensor Networks (LTNs) (Donadello et al., 2017) and Semantic Loss (SL) (J. Xu et al., 2018) are examples of such methods. SBR and LTN implement predicates as differentiable functions and translate the provided logical formulas into a real valued regularization by means of fuzzy logic, while SL uses marginal probabilities of the target atoms to define the regularization term and relies on SDDs for efficient evaluation.

Another form of undirected NeSy approaches are Deep Logic Networks (DLNs) (Tran and Garcez, 2018). Here, knowledge can be inserted and extracted from restricted Boltzmann machines, which are a type of Markov random field.

3.2 Model-based vs Proof-based Inference

The distinction between directed and undirected models is closely related to the distinction between a model-theoretic and a proof-theoretic approach to inference. This is clear when looking at the difference between Answer Set Programming and the programming language Prolog. In the model theoretic perspective, one first grounds out the clauses in the theory and then calls a SAT solver (possibly after breaking cycles), while in a proof-theoretic perspective one performs a sequence of inference steps in order to obtain a proof, cf. Section 2.1.3.

Many StarAI systems use logic as a kind of template to ground out the relational model in order to obtain a grounded model and perform inference. This grounded model can be a graphical model, or alternatively, a ground weighted logical theory on which traditional inference methods apply, such as belief propagation or weighted model counting. This is used in well known systems such as Markov logic networks, Probabilistic soft logic, Bayesian logic programs, and probabilistic relational models.

Alternatively, one can follow a proof based approach. This is akin to what happens in probabilistic programming (Russell, 2015), in StarAI frameworks such as PLPs, probabilistic databases (Van den Broeck, Suciu, et al., 2017) and probabilistic unification based grammars such as Stochastic Logic Programs (SLPs) (Muggleton, 1996). Just like pure logic supports the model-theoretic and proof-theoretic perspectives, both perspectives have been explored in parallel for some probabilistic logic programming languages such as ICL (Poole, 2008) and ProbLog (De Raedt, Kimmig, and Toivonen, 2007).

These two perspectives carry over to NeSy. Approaches like NTPs, DeepProblog, ∂ILP (Evans and Grefenstette, 2018) and DiffLog are proof-based. In contrast, the approaches of lifted relational neural networks (LRNNs) (Šourek et al., 2018), logic tensor networks (LTNs) (Badreddine et al., 2021), relational neural machines (RNMs) (Marra, Diligenti, et al., 2020), neural Markov logic networks (NMLNs) (Marra and Kuželka, 2019), neural logic machines (NLM) (Dong et al., 2019) and NeuralLP (F. Yang et al., 2017) are all based on the model-theoretic perspective.

3.3 Logic vs Probability vs Neural

StarAI methods lie on a spectrum of the two base paradigms: logic and probability. The traditional knowledge-based model construction approach is to use the logic only to generate a probabilistic graphical model. This probabilistic graphical model is then used to define the semantics of the model and also to perform inference. Such methods thus lie more towards the probabilistic side. Conversely, PLP and its variants lie more towards the logical side.

The position on the spectrum between logic and probability has a profound influence on the properties of the underlying model. For methods that lie more towards the probabilistic side, the effect of changing the logic can be more difficult to understand. For instance, in MLNs, the addition of the resolvent of two weighted rules, makes it hard to predict the effect on the distribution. On the other hand, for PLPs, it is clear what the effect of a logical operation is. However, it is often harder to directly identify and exploit properties such as conditional or contextual independencies, which are needed for efficient probabilistic inference.

For NeSy, the spectrum involves logic, probability and neural networks. The majority of NeSy approaches focus on the neural aspect. They are generally a neural method to which logical components have been added. Some of these approaches like LTNs and TensorLog (Cohen et al., 2018) pursue a kind of knowledge-based model construction approach in which the logic is compiled away into the neural network architecture. A different family of NeSy approaches, which includes semantic loss (J. Xu et al., 2018) and semantic based regularization (Diligenti et al., 2017), turns the logic into a regularization function to provide a penalty whenever the desired logical theory or constraints are violated. This leads to the logic being compiled into the weights of the trained neural network. Such an approach has the downside that adding additional rules involves re-training the neural network.

However, there are NeSy methods that do retain the focus on logic. Some of these methods start from existing logic (programming) frameworks and extend them with primitives that allow them to interface with neural networks and allow for differentiable operations. Examples of such methods include DeepProbLog, NeurASP (Z. Yang et al., 2020) and DiffLog (Si et al., 2019). Other methods instead take an existing framework and turn it into a differentiable version. The key inference concepts are mapped onto analogous concepts that behave identically for the edge cases, but are continuous and differentiable in non-deterministic cases. Such methods include ∂ILP, $\partial 4$ (Bošnjak et al., 2017) and NTPs. Many of these methods struggle with scalability and require special techniques such as approximate inference in order to be applied to larger tasks.

3.4 Semantics

Traditionally, StarAI combines two semantics: a logical and a probabilistic one. In the *logical* semantics, atoms are assigned a truth value in the $\{true, false\}$ set. In a *probabilistic* semantics, probability is defined as a measure over sets of possible worlds, where each possible world is an assignment of values to the random variables. This implies that probabilistic logic semantics define probability distributions over ground logical interpretations, that is, over sets of ground facts. Prominent examples in StarAI are ProbLog (from the directed side) and MLNs (from the undirected one). Incorporating probabilistic semantics into a logical one is natural when one wants to perform logical reasoning under uncertainty. Moreover, it extends Boolean logic with probabilities and it preserves the original logical semantics. However, this extension comes at the price of more complex inference, which can make probabilistic StarAI models intractable in large scale domains.

Another approach is to turn the logical operators into real-valued functions, and in doing so, relax the Boolean truth values to the continuous $[0, 1]$ interval. This introduces the semantics of *fuzzy logic* (or soft logic), which is mathematically grounded in t-norm theory. The fuzzy semantics can be used alone or in conjunction with the probabilistic one, e.g. PSL (Bach et al., 2017). The algebraic and geometric properties of t-norms (including especially convexity and differentiability) results in a reduction in the complexity of inference. However, differently from the probabilistic case, the semantics of the original Boolean theory is not always preserved. Indeed, the *fuzzification* procedure can introduce undesirable effects. In particular, improper choices in the fuzzification could lead to behaviors that are different from the ones in the original theory, (Giannini et al., 2019) and particular attention should be paid to assessing that any desired property is preserved. For example, when translating the implication $A \rightarrow B$ with its fuzzy material implication (i.e. $\neg A \vee B$), one may lose the transitivity property (e.g. the material implication in the product t-norm logic). An analysis of such issues, also in connection with differentiability, is performed in (Van Krieken et al., 2020).

Neural-symbolic approaches can be categorized along the same lines. Neural enhancements of the *logic* semantics either use neural networks to turn sub-symbolic inputs into logical atoms, or relax the logical reasoning through tensor calculus. An instance of the former is ABL (Dai et al., 2019), which uses logical abduction to provide the feedback for a neural model processing the sub-symbolic input. Tensor calculus approaches, such as NLM and NeuralLP, interpret predicates as tensors grounded over all constants in a domain and interpret clauses as a product of those matrices.

Neural enhancements of the *probabilistic* semantics usually parameterize the underlying distribution in terms of neural components. NMLN and RNM use neural potentials in order to implement factors (or their weights) as neural networks. Rocktäschel, Singh, et al. (2015) compute marginal probabilities as logistic functions over similarity measures between embeddings of entities and relations.

In neural enhancements of the *fuzzy* semantics, neural networks produce continuously-valued truth assignments. The differentiability of the t-norms allows to easily integrate neural frameworks. In particular, SBR and LTN turn atoms into neural networks taking as inputs the feature representation of the constants and returning the corresponding truth value. Similarly, in LRNN, ∂ILP and DiffLog, the scores of the proofs are computed by using fuzzy logic connectives.

Finally, a large class of methods (Minervini, Demeester, et al., 2017; Demeester et al., 2016; Cohen et al., 2018; Weber et al., 2019) use numerical relaxations of logical statements without giving specific semantics. Here, atoms are assigned scores computed by a neural scoring function over embeddings. Numerical approximations are then applied to either combine these scores according to logical formulas or to aggregate proofs scores. The resulting neural architecture is usually differentiable and, thus, trained end-to-end. It is however hard to interpret the results generated by such approaches.

3.5 Structure vs Parameter Learning

StarAI distinguishes between two types of learning: structure learning, which corresponds to learning the logical clauses of the model (Kok and P. M. Domingos, 2005), and parameter learning, in which the probabilities or weights of the clauses have to be estimated (Gutmann et al., 2008; Lowd and P. M. Domingos, 2007). The former is typically achieved by means of a combinatorial search over the space of possible clauses, while in the latter one an expert user provides a set of informative clauses for which only the probabilities have to be estimated.

In NeSy, learning now also includes the structure and parameters of the neural networks. In earlier work (Towell and Shavlik, 1994; Garcez, Lamb, and Gabbay, 2007), many systems focused on defining (part of) the structure through the logic, cf. the templating approaches discussed in Chapter 6. Nowadays, the focus lies more on learning the parameters of a pre-defined neural network architecture. Learning the structure of the neural network is not yet well understood and is generally not considered in the field of NeSy. Usually, learning the structure of the program in NeSy approaches does not happen through a combinatorial

search. Instead, it is learned though parameter learning. This blurs the distinction between both types of learning. The set of clauses is enumerated from user-provided rule templates. The enumerated rules contain noisy and erroneous patterns that are corrected by learning the corresponding probabilities or weights. While it is certainly possible to extract the most important clauses, the inference is still performed considering all the enumerated clauses. Examples of such systems include NTPs, ∂ILP, NeuralLP and DiffLog.

3.6 Symbols vs Sub-symbols

In artificial intelligence, one of the main properties characterizing an approach is how it represents entities (and relations). The approaches can be categorized into two classes: symbolic methods, where entities are represented using symbols (e.g. strings or natural numbers), and sub-symbolic methods, where entities have numerical or distributed representations.

Symbolic representations include atoms (an, bob), numbers (4, -3.5), variables (X, Y) and structured terms $f(t_1, ..., t_n)$ where f is a functor and the t_i are constants, variables or structured terms. These structured terms are a powerful way of representing entities as they can represent relations, lists, ... Sub-symbolic AI systems, such as neural networks, require that entities are represented numerically. This includes vectors, matrices and tensors. We will call these sub-symbolic representations or sub-symbols. Sub-symbolic AI systems tend to require that the representations have a fixed size and dimensionality. Exceptions to this require special architectures and are still the subject of active research (e.g. Recurrent Neural Networks for list-like inputs or Graph Neural Networks (Scarselli et al., 2009; Kipf and Welling, 2017) for graph-type inputs).

3.6.1 Comparing entities

How entities can be compared depends on their representation. A powerful and elegant mechanism for reasoning with symbols in logic is *unification*. Essentially, it calculates the most general substitution that makes two symbols equal, if it exists. For instance, the atomic expressions $p(a, Y)$ and $p(X, b)$ can be unified using the substitution $\{X = a, Y = b\}$. Although it is a powerful mechanism, unification does not say anything about the similarity of two terms. Conversely, due to their numerical nature, calculating the similarity between sub-symbols is straightforward. Functions such as the radial-basis function or distance metrics such as the L1 and L2 norm can be used. However, it is not straight-forward to decide when two sub-symbolically represented entities are equal or not.

3.6.2 Translating between representations

Many systems need to translate back and forth between symbolic and sub-symbolic representations. In fact, a lot of research on deep learning is on how to efficiently represent symbols so that neural networks can properly leverage them. A straight-forward example is translating words (symbols) to a sub-symbolic representation to serve as an input for an RNN. Generally, these words are replaced by a one-hot encoding or by learned embeddings. A more interesting example is encoding relations in sub-symbolic space. The wide variety of methods (Bordes et al., 2013; Trouillon et al., 2016; B. Yang et al., 2015) developed to do this indicates that this is far from a solved problem, and different ways of encoding have different benefits. For example, TransE (Bordes et al., 2013) encodes relations as vector translations from subject to object embeddings. This has the downside that any symmetric relation is represented by the null vector, and entities in symmetric relations are pushed towards each other. More complex structures are even harder to represent. For example, there is currently a lot of research on utilizing graph-structured data in neural networks (Scarselli et al., 2009; Kipf and Welling, 2017; Schlichtkrull et al., 2018), but there are still plenty of open challenges (Zhang et al., 2019).

Translating from a sub-symbolic back to a symbolic representation happens for the example at the end of a neural network classifier. Here, a sub-symbolic vector needs to be translated to discrete classes. Generally, this happens through the use of a final layer with a soft-max activation function, which then models the confidence scores of these classes as a probability distribution. However, other options are possible. For example, some methods are only interested in the most likely class, and will use an arg-max instead. Alternatively, a Gumbel-softmax activation can be used as a differentiable approximation of a sampling on the probability distribution represented by the soft-max.

In StarAI systems, the input, intermediate and output representations are all the same symbolic representation. In neural systems, the input and intermediate representations are sub-symbolic. The output representations can be either symbolic (e.g. classifiers) or sub-symbolic (e.g. auto-encoders (Kingma and Welling, 2014) or GANs (Goodfellow, Pouget-Abadie, et al., 2014)). The most important aspect of neural-symbolic systems is the fact that they combine symbolic and sub-symbolic representations.

NeSy systems can be categorized by how these are combined. We discern several approaches. In the first approach, the inputs are symbolic, but they are translated to sub-symbols in a single translation step. Afterwards, the intermediate representations used during reasoning are only sub-symbolic. This approach is followed by the majority of NeSy methods. Some examples

include Logic Tensor Networks (Donadello et al., 2017), Semantic-based Regularization (Badreddine et al., 2021), Neural Logic Machines (Dong et al., 2019) and TensorLog (Cohen et al., 2018). However, the symbolical nature is lost after the translation step. Generally, this translation allows the system to map logical inference onto a form for which inference is a lot more efficient, which makes such systems more scalable than the approaches discussed below. However, through this translation, the symbolic nature is lost, which has the risk that the original logic cannot be retrieved.

In the second approach, intermediate representations are both symbolic and sub-symbolic, but not simultaneously. By this we mean that there some parts of the reasoning work on the sub-symbolic representation, and other parts deal with the symbolic representation, but no single part considers both. This is indicative of NeSy methods that implement an interface between the logic and neural aspect. This approach is more natural for systems that originate from a logical framework (e.g. DeepProbLog, NeurASP (Z. Yang et al., 2020)), ABL (Dai et al., 2019) and NeuroLog (Tsamoura et al., 2021). Such methods tend to be quite expressive, but the combination of a (potentially) computationally expensive logic framework and repeated neural network evaluations can cause scalability issues.

In the final approach, intermediate representations are simultaneously symbolic and sub-symbolic. This means that a reasoning mechanism considers both simultaneously. This is implemented in only a few methods, such as the neural theorem prover (Rocktäschel and Riedel, 2017) and the conditional theorem prover (Minervini, Riedel, et al., 2020). In essence, this is the most general approach. However, such methods also suffer from scalability issues. Since the reasoning mechanism has to be adapted to accommodate for both representations, such an integration generally considers only a subset of the base logic framework.

3.7 Type of Logic

StarAI approaches have explored various types of logical representations, following a natural ordering (De Raedt, 2008; Flach, 1994) starting with propositional logic (symbols without arguments), to relational logic (with only constants and variables as terms, without any structured terms) which forms the basis for the Datalog database language, to general first order logic, and then to logic programs (e.g. Prolog). Logic programs are usually restricted to definite clauses. The semantics of a definite clause program is given by its (unique) least Herbrand model, the set of all ground facts that are logically entailed by the

program (cf. Section 2.1). This contrasts with the standard semantics of first order logic that would also allow for other models. This difference carries over to StarAI, where probabilistic logic programs and Markov Logic inherit their semantics from logic programming and first order logic, respectively. On the other hand, logic programs are also the basis for the programming language Prolog, which implies that they can be used to specify traditional programs such as sorting and data structures such as lists through structured terms. This is relevant especially for methods that implement structure learning.

NeSy approaches follow the same natural expressivity order of the underlying logic. For instance, SL focuses only on the propositional setting. ∂ILP, NTPs and DiffLog are based on Datalog, which belongs to relational logic segment. Logic tensor networks and semantic based regularization use fuzzy logic to translate a general first-order logic theory into a training objective, either isolated or in conjunction with a supervised criterion. Just like Markov Logic, relational neural machines (Marra, Diligenti, et al., 2020) and neural Markov logic networks (Marra and Kuželka, 2019) also use first-order logic. NLProlog (Weber et al., 2019) and lifted relational neural networks (Šourek et al., 2018) are examples of neural-symbolic logic programming frameworks.

The chosen type of logic has a significant impact on inference and learning. The more expressive the logic is, the harder the inference and learning become. For instance, for structure learning, the space of possible clauses for structure learning typically becomes exponentially larger as a more expressive class of logic is used. Conversely, a more expressive language generally allows for a more concise problem representation (Flach, 1994).

	Dimension 1 (D)irected (U)ndirected	**Dimension 2** (M)odel-based (P)roof -based	**Dimension 3** (L)ogic (P)robability (N)eural
DeepProbLog	D	P	L+P+N
∂ILP (Evans and Grefenstette, 2018)	D	P	L+N
DiffLog (Si et al., 2019)	D	P	L+N
LRNN (Šourek et al., 2018)	D	P	L+N
LTN (Badreddine et al., 2021)	U	M	L+N
NeuralLP (F. Yang et al., 2017)	D	M	L+N
NLM (Dong et al., 2019)	D	M	L+N
NLProlog (Weber et al., 2019)	D	P	L+P+N
NMLN (Marra and Kuželka, 2019)	U	M	L+P+N
NTP (Rocktäschel and Riedel, 2017)	D	P	L+N
RNM (Marra, Diligenti, et al., 2020)	U	M	L+P+N
SL (J. Xu et al., 2018)	U	M	L+P+N
SBR (Diligenti et al., 2017)	U	M	L+N
Tensorlog (Cohen et al., 2018)	D	P	L+N
DLN (Tran and Garcez, 2018)	U	M	L+N
NeurASP (Z. Yang et al., 2020)	D	M	L+P+N
ABL (Dai et al., 2019)	D	P	L+N

Table 3.1: Taxonomy of a (non-exhaustive) list of NeSy models according to the first 3 dimensions.

	Dimension 4 (L)ogic (P)robability (F)uzzy	**Dimension 5** (P)arameter (S)tructure	**Dimension 6** (S)ymbols (Sub)symbols	**Dimension 7** (P)ropositional (R)elational (FOL) (LP)
DeepProbLog	P	P	S+Sub	LP
∂ILP	F	P+S	S	R
DiffLog	F	P+S	S	R
LRNN	F	P+S	S+Sub	LP
LTN	F	P	Sub	FOL
NeuralLP	L	P	S	R
NLM	L	P+S	S	R
NLProlog	P	P+S	S+Sub	LP
NMLN	P	P+S	S+Sub	FOL
NTP	L	P+S	S+Sub	R
RNM	P	P	S+Sub	FOL
SL	P	P	S+Sub	P
SBR	F	P	Sub	FOL
Tensorlog	P	P	S+Sub	R
DLN	L	P+S	S+Sub	P
NeurASP	P	P	S+Sub	LP
ABL	L	P+S	S+Sub	LP

Table 3.2: Taxonomy of a (non-exhaustive) list of NeSy models according to the last 4 dimensions.

3.8 Conclusion

We analyzed the field of neural-symbolic AI by drawing parallels to the well-established field of StarAI. We discussed 7 dimensions that can be used to describe both NeSy and StarAI systems. In the directed vs undirected dimension, we showed that this choice has an impact on how background knowledge is represented: either as clausal, if-then rules or as constraints over neural network outputs. We then discussed the difference between model-based and proof-based approaches. Next, we argued that, whereas StarAI methods lie on a spectrum between logic and probability, NeSy methods lie on a spectrum between logic, probability and neural. We also discussed how lying close to one end of the spectra has impacts on the strengths and weaknesses of the system. In the parameter vs structure learning dimension, we briefly touched on the difference between these two types of learning for StarAI methods, and how the distinction is less obvious in many NeSy systems, as structure learning is performed using parameter learning. Next, we discussed that NeSy methods have to deal with both symbolic and sub-symbolic representations and how exactly how this integration happens has a big impact on the characteristics of the method. Finally, we discussed the different types of logic used in StarAI systems w.r.t. to their expressivity and tractability of inference and how NeSy systems follow this same pattern.

Chapter 4

DeepProbLog

This chapter is largely based on the following papers:

- R. Manhaeve, S. Dumančić, A. Kimmig, T. Demeester, and L. De Raedt (2018). "DeepProbLog: Neural probabilistic logic programming". In: *Advances in Neural Information Processing Systems 31*, pp. 3749–3759.
- R. Manhaeve, S. Dumančić, A. Kimmig, T. Demeester, and L. De Raedt (2021). "Neural probabilistic logic programming in DeepProbLog". In: *Artificial Intelligence* 298, pp. 103–504.

DeepProbLog is a neural probabilistic logic programming language. It is a neural-symbolic extension of the probabilistic logic programming language ProbLog (De Raedt, Kimmig, and Toivonen, 2007). This extension is realized through the addition of a single concept: the neural predicate. We theoretically and experimentally demonstrate that DeepProbLog supports 1) both symbolic and sub-symbolic representations and inference, 2) program induction, 3) probabilistic (logic) programming, and 4) (deep) learning from examples.

In Section 4.1 we discuss the DeepProbLog language, the neural predicate and its semantics in more detail. In Section 4.2 we will discuss the details of exact inference in DeepProbLog. Section 4.3 explains how learning happens in DeepProbLog. Finally, in Section 4.5 we experimentally evaluate the capabilities of DeepProbLog.

4.1 The DeepProbLog Language

In this section we introduce the DeepProbLog language (Manhaeve, Dumančić, et al., 2018; Manhaeve, Dumančić, et al., 2021). In essence, DeepProbLog is ProbLog extended with the neural predicate. First, we introduce tensors as first class citizens. Next, we introduce the concept of the neural predicate and the different forms it takes in DeepProbLog. Finally, we discuss the semantics of the different forms of the neural predicate.

4.1.1 Tensors

Neural networks work with tensors as their input and intermediate representation. In some cases, their output is a tensor as well. To accommodate for this, we extend the definition of the term. In DeepProbLog, a term is either a constant c, a variable V, a structured term of the form $f(u_1, \ldots, u_k)$ where f is a functor and the u_i are terms or a tensor term of the form $tensor(id)$ where $tensor$ is a reserved functor for tensor terms and id is a unique identifier. We define a function $\mathcal{T}$ that maps the tensor terms onto the actual tensors in neural side.

4.1.2 The Neural Predicate

The neural predicate is a general concept that allows for probabilistic logic programming languages to be lifted to the neural-symbolic level. The neural predicate itself is an interface between the logical and neural components. The main idea is that it allows for part of the program to be parameterized using neural networks, conditioned on high-dimensional inputs such as images.

Concretely, in DeepProbLog, the neural predicate is instantiated into three forms: the neural annotated disjunction, the neural fact and the neural function.

Definition 4.1 (Neural annotated disjunction)
A *neural annotated disjunction* (nAD) is an expression of the form

$$nn(m_r, [X_1, \ldots, X_k], O, [y_1, \ldots, y_n]) :: r(X_1, \ldots, X_k, O)$$

where nn is a reserved functor, m_r uniquely identifies a neural network model (i.e., its architecture as well as its trainable parameters) that defines a probability distribution $p_{m_r}(O|X = \boldsymbol{x})$ over the domain $O \in \{y_1, \ldots, y_n\}$ given the input $\boldsymbol{x} = [x_1, .., x_k]$, $X_1, \ldots, X_k$ are variables representing the

inputs to the neural network, O is the output variable, the ground terms $y_1, \ldots, y_n$ define the domain of the output distribution, and r is a predicate symbol. Note that the arguments of the predicate r can be in an arbitrary order.

The neural network could take any shape, e.g., a convolutional network for image encoding, a recurrent network for sequence encoding, etc. However, its output layer, which is used in the corresponding neural predicate, needs to be normalized.

Example 4.1 (Neural annotated disjunction)
A neural network classifier naturally maps onto a neural annotated disjunction. For example, an MNIST (Lecun et al., 1998) digit classifier can be specified as:

$$\texttt{nn}(\texttt{m_digit}, [\texttt{X}], \texttt{Y}, [0, \ldots, 9]) :: \texttt{digit}(\texttt{X}, \texttt{Y}).$$

where `m_digit` is a network that classifies MNIST digits. For input image , the ground nAD is

$$\texttt{p}_{\texttt{m_digit}}(\texttt{Y} = 0 | \texttt{X} = \texttt{3}) :: \texttt{digit}(\texttt{3}, 0) \ ; \ \ldots \ ; \ \texttt{p}_{\texttt{m_digit}}(\texttt{Y} = 9 | \texttt{X} = \texttt{3}) :: \texttt{digit}(\texttt{3}, 9).$$

We consider an output domain size of two as a special case. Instead of the neural network having two probabilities at the output that sum to one, we can simplify this to a single probability, with the second one the complement of that probability. This difference coincides with the difference between a softmax and single-neuron sigmoid activation layer in a neural network. We call such an expression a neural fact.

Definition 4.2 (Neural fact)
A neural fact is an expression of the form

$$nn(m_r, [X_1, \ldots, X_k]) :: r(X_1, \ldots, X_k).$$

where nn is a reserved functor, m_r uniquely identifies a neural network model that defines a probability distribution $p_{m_r}(O|X = \boldsymbol{x})$ over the Boolean domain $\{1, 0\}$, given the input $X_1, \ldots, X_k$ are variables representing the inputs to the neural network, and r is a predicate symbol. Note that the arguments of the predicate r can be in an arbitrary order.

Example 4.2 (Neural fact)
A binary classifier can be viewed as a discriminative classifier that determines the truth value of a fact. A neural network to classify MNIST

digits as odd or even could be modeled as a neural AD:

$$nn(mnist_cnn, [I], O, [odd, even]) :: odd_or_even(I, O).$$

or equivalently as a neural fact:

$$nn(mnist_cnn, [I]) :: even(I).$$

where *odd* is implicitly defined as $\backslash + even$.

To further broaden the interface, we also define the *neural function*. It encodes a neural network that outputs a tensor rather than a probability distribution. This further extends the definition of the neural predicate as previously given in (Manhaeve, Dumančić, et al., 2021).

Neural functions can be used to generate embeddings as intermediate representations that can be used in other neural networks or differentiable functions.

Definition 4.3 (Neural function)
A *neural function* is an expression of the form

$$nn(m_r, [X_1, \ldots, X_k], O) :: r(X_1, \ldots, X_k, O)$$

where nn is a reserved functor, m_r uniquely identifies a neural network model that defines a function $f_{m_r}(\boldsymbol{x}) = y$ given the input $\boldsymbol{x} = [x_1, .., x_k]$ such that O is unified with a tensor $tensor(nn(m_r, \boldsymbol{x}))$ and $\mathcal{T}(tensor(nn(m_r, \boldsymbol{x}))) = y$.

Example 4.3 (Neural function)
Some neural networks can be viewed as embedding generators (i.e. they map the input onto a structured embedding space). For example, a convolutional neural network that encodes an image using a set of convolutional layers can be encoded using the following neural function:

$$nn(mnist_encode, [I], E) :: mnist_embed(I, E)$$

4.1.3 DeepProbLog program

We can now define a DeepProbLog program.

Definition 4.4 (DeepProbLog Program)
A DeepProbLog program consists of a set of probabilistic facts $\mathcal{F}$, a set of neural predicates $\mathcal{N}$ and a set of clauses $\mathcal{R}$.

Example 4.4 (MNIST addition)
We now implement the running MNIST addition example in DeepProbLog. The goal is to infer the sum of the numbers represented by the images, e.g. `addition(``,``,8)`. This can be encoded in two lines of DeepProbLog code:

```
nn(m_digit, [X], Y, [0...9]) :: digit(X,Y).
addition(X,Y,Z) :- digit(X,N1), digit(Y,N2), Z is N1+N2.
```

4.1.4 Semantics of DeepProbLog

Since DeepProbLog is an extension of ProbLog, we will describe the semantics of DeepProbLog with respect to the semantics of ProbLog as discussed in Section 2.2.1.

First, we will describe the semantics of the neural facts and neural annotated disjunctions.

Definition 4.5 (Semantics of a Neural Annotated Disjunction)
Formally, a neural AD of the form

$$nn(m_r, [X_1, \ldots, X_k], O, [y_1, \ldots, y_n]) :: r(X_1, \ldots, X_k, O)$$

represents a set of *ground neural ADs* of the following form, one for every sequence of ground terms $x_1, \ldots, x_k$ representing inputs to the neural network:

$$p_{m_r}(O = y_1 | X_1 = x_1, \ldots, X_k = x_k) :: r(x_1, \ldots, x_k, y_1) \ ;$$

$$\ldots \ ;$$

$$p_{m_r}(O = y_n | X_1 = x_1, \ldots, X_k = x_k) :: r(x_1, \ldots, x_k, y_n)$$

Informally, this can be considered an *evaluated* form of the nAD where the neural predicate functor is replaced by the probability predicted by the neural network.

Definition 4.6 (Semantics of a Neural Fact)
Formally, a neural fact of the form

$$nn(m_r, [X_1, \ldots, X_k]) :: r(X_1, \ldots, X_k).$$

represents a set of *ground neural facts* of the following form, one for every sequence of ground terms $x_1, \ldots, x_k$ representing inputs to the neural network:

$$p_{m_r}(O = 1 | X_1 = x_1, \ldots, X_k = x_k) :: r(x_1, \ldots, x_k).$$

Definition 4.7 (Semantics of a Neural Function)
Formally, a neural function of the form

$$nn(m_r, [X_1, \ldots, X_k], O) :: r(X_1, \ldots, X_k, O).$$

represents a set of *ground facts* of the following form, one for every sequence of ground terms $x_1, \ldots, x_k$ representing inputs to the neural network:

$$r(x_1, \ldots, x_k, tensor(nn(m_r, \boldsymbol{x}))).$$

with $\mathcal{T}(tensor(nn(m_r, \boldsymbol{x})))$ the tensor that is the output of the neural network m_r evaluated on input $\boldsymbol{x}$

Definition 4.8 (Semantics of a DeepProbLog program)
The semantics of a DeepProbLog program is defined with respect to a fixed set of possible neural inputs. The neural inputs here are considered to be any input to a neural fact, nAD or neural function. By replacing each neural fact, neural AD and neural function by the corresponding sets of ground neural facts, neural ADs and facts over these inputs (i.e. by evaluating the neural networks on the given inputs), we obtain a ProbLog program whose semantics defines the semantics of the original DeepProbLog program.

Example 4.5
We illustrate the DeepProbLog semantics on the MNIST addition example (Example 4.4), implemented by the following DeepProbLog program:

```
nn(m_digit, [X], Y, [0...9]) :: digit(X,Y).
addition(X,Y,Z) :- digit(X,N1), digit(Y,N2), Z is N1+N2.
```

We consider the neural inputs  and . Below, we show relevant ground DeepProbLog program for the query `addition(``,``,1)`.

```
nn(m_digit,[ ],0)::digit( ,0);nn(m_digit,[ ],1)::digit( ,1).
nn(m_digit,[ ],0)::digit( ,0);nn(m_digit,[ ],1)::digit( ,1).
addition( , ,1) :- digit( ,0), digit( ,1), 1 is 0 + 1.
addition( , ,1) :- digit( ,1), digit( ,0), 1 is 1 + 0.
```

The semantics of the DeepProbLog program are given by the following ProbLog program. The exact probabilities are dependent on the neural network parameters.

```
0.8 :: digit( ,0); 0.1 :: digit( ,1).
0.2 :: digit( ,0); 0.6 :: digit( ,1).
addition( , ,1) :- digit( ,0), digit( ,1), 1 is 0 + 1.
addition( , ,1) :- digit( ,1), digit( ,0), 1 is 1 + 0.
```

Generative vs discriminative

It is worth elaborating on the fact that this description of the semantics includes the inputs to the neural networks into the Herbrand domain of the program. We discuss two possible ways to view neural inputs and outputs (e.g. MNIST images). Both explicitly introduce neural inputs through type predicates, but they differ on how the inputs are represented. For instance, we could add MNIST images through the type `image`:

$$\texttt{nn}(\texttt{m}, [\texttt{X1}, \texttt{X2}]) :: \texttt{similar}(\texttt{X1}, \texttt{X2}) : -\texttt{image}(\texttt{X1}), \texttt{image}(\texttt{X2}).$$

In the first view, the `image` predicate simply introduces one constant per input. This implies, however, that neural inputs, just like other constants in the Herbrand domain, are part of the program, and have to be specified upfront. Furthermore, the possible worlds are then, so to speak, conditioned on the possible inputs to the neural networks and the possible worlds will only contain these neural inputs and no other ones. This perspective is in line with discriminative training (cf. Section 2.2.4). This excludes the possibility of using

neural networks in a generative fashion, where "new" images or constants could be generated.

In the second view, neural inputs are represented as structured terms that provide an abstract language to define the possible inputs, rather than enumerating them explicitly. For instance, an image is just a matrix of numbers. Under this view, the predicate `image` could simply succeed when its argument is a list containing exactly 28×28 numbers between 0 and 255, which would include all possible images in the Herbrand domain. This view would also allow one to represent generative neural networks as neural predicates, and would consider possible worlds that contain all possible images.

4.1.5 Embeddings in DeepProbLog

The neural functions allow for embeddings to be represented directly in DeepProbLog. This allows us to widen the interface by adding built-in predicates that perform differentiable operations on tensor terms. We now show two examples: one on learning embeddings for constants and images jointly, and one on how to implement the soft unification principle in DeepProbLog.

Example 4.6 (DeepProbLog Embeddings)
In this example, we manipulate embeddings of MNIST images. A straightforward way to obtain image embeddings, is by optimizing a clustering objective. That is, we minimize the distance between the embeddings of images of the same number, while maximizing the distance between the embedding of images of different numbers. The radial basis function $\varphi(x, y)$ (RBF) can be used a as similarity, which is also used in the neural theorem prover (Rocktäschel and Riedel, 2017). It is defined as

$$\varphi(x, y) = e^{-||x-y||_2}$$

This can be included in DeepProbLog as a predicate `rbf/2` that succeeds with probability $p = \varphi(x, y)$ where x and y are embeddings that are the arguments of the predicate. The code is shown below:

```
similar(I1,I2) :-
    %Encode images I1 and I2 into E1 and E2
    cnn_encode(I1,E1), cnn_encode(I2,E2),
    rbf(E1,E2).
```

where `cnn_encode/2` unifies the second argument with the embedding of the first argument by using a convolutional neural network (cf. Example 4.3).

As a proof of concept of the wider possibilities of implicitly training embeddings through DeepProbLog, we extend the clustering objective towards inducing an order relation in embedding space. The successor relationship between two MNIST images is defined as `successor(I1,I2,R)` where `I1` and `I2` are MNIST images, and `R` is the difference of the digits represented by the two images (e.g. `successor(``,``, −2)`). We model the successor relationship as a translation, cf. TransE (Bordes et al., 2013). We also specify that it should not be zero to avoid collapsing on a trivial solution. Note that this constraint on the size of the embedding can be seen as a form of regularization defined in the logic.

```
successor(I1,I2,N) :-
    %Encode images I1 and I2 into E1 and E2
    cnn_encode(I1,E1), cnn_encode(I2,E2),
    %Embed the successor relation into embedding S
    embed(successor, S),
    %The relation should not be trivial (i.e. 0)
    \+rbf(S,0).
    %E = E1 + N*S should be similar to E2
    mul(S,N,S2), add(E1,S2,E),
    rbf(E,E2).
```

We define a predicate `embed/2` that unifies the second argument with a learnable embedding for the first argument. `rbf/2` and `cnn_encode/2` are as described above, `mul/3` is the product between a scalar and a vector and `add/3` an element-wise addition. These create the mathematical definition of the successor relationship. The number 0 is used as a zero vector of the same dimension as the embeddings.

We used the entire MNIST training dataset to create examples of the `successor/3` relationship (e.g. `successor(``,``, −2)`. We then trained the neural network and the value of the successor relationship by optimizing cross-entropy loss between the success probability of these queries and the probability 1.0. Figure 4.1 shows the embeddings after training of the MNIST test set (crosses) and the mean of each of these images, grouped by label (solid dots). It shows that DeepProbLog can learn to embed images into clusters corresponding to the labels of these images, which are positioned sequentially along the embedding dimension. As can be visually inspected, the cluster centers can be (approximately) linearly mapped to the actual image labels.

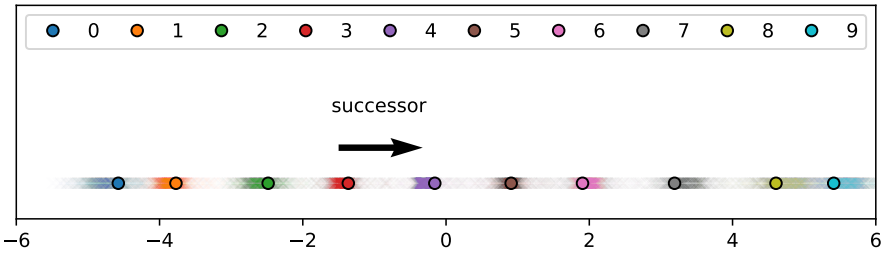


Figure 4.1: The one-dimensional embedding space for MNIST images learned in Example 4.6. Individual MNIST image embeddings are shown in crosses, with the mean for each label as a solid dot. The successor relation is marked by the arrow.

Example 4.7 (Soft unification)
The `rbf/2` predicate provides DeepProbLog with the power of soft unification from the Neural Theorem Prover (NTP) (Rocktäschel and Riedel, 2017). With soft unification, terms can be unified if they have the same structure (e.g. `grandPa(abe,bart)` can be unified with `grandFather(abe,bart)`). Such unifications would be scored based on the similarity between the embeddings of the predicates and constants. This has proven to be a powerful concept in Rocktäschel and Riedel (2017). By implementing it explicitly in DeepProbLog, we showcase the flexibility of our framework, although a native implementation could be more efficient. This explicit implementation has the added benefit that we have full control over the soft unification, as we can decide to only apply it to parts of the logic program, whereas in the NTP, it is applied to the entire program. In Example 4.6 soft unification is only applied to MNIST images, and the remainder of the program uses standard reasoning and unification. Complete soft unification could be implemented as in the following program:

```
soft_unify(Term1,Term2) :-
    Term1 =.. List1,
    Term2 =.. List2,
    soft_unify_list(List1,List2).

soft_unify_list([],[]).
soft_unify_list([H|T1],[H|T2]) :- soft_unify_List(T1,T2).

soft_unify_list([H1|T1],[H2|T2]) :-
```

```
    H1 \= H2,
    embed(H1),
    embed(H2),
    rbf(H1,H2),
    soft_unify_list(T1,T2).
```

In this program, we make use of lists to represent terms. A list is an expression of the form

$$[\mathtt{I}_1, ..., \mathtt{I}_j \mid \mathtt{T}]$$

where the $\mathtt{I}_i$ are the terms in the list. The part after the | is called the tail, and it is the list that follows after the item $\mathtt{I}_j$. The $=..$ predicate succeeds when the argument on the left-hand side is a term $f(u_1, \ldots, u_n)$ and the right hand side argument is the list $[f, u_1, \ldots, u_n]$.

Intermezzo: DeepStochLog

In DeepStochLog (Winters et al., 2022), we applied the concept of the neural predicate to stochastic definite clause grammars, which are a type of stochastic logic program (SLP). Although SLPs are similar to PLPs, they have different semantics. The main difference is that whenever a non-deterministic choice has to be made, i.e. when there are several rules that can be used to prove part of a goal, SLPs require that the sum of all probabilities for such rules is equal to one, and these rules are considered to be mutually exclusive. This mean that the disjoint-sum problem does not have to be solved, which makes inference more efficient.

The difference between the two semantics can be thought of as the difference between random graphs and random walks. For PLPs, the semantics can be thought of as whether a query is entailed by a logic program that is randomly sampled from the probabilistic logic program, cf. a random graph. For SLPs, the semantics define a distribution over individual derivations, cf. random walks.

The neural predicate here takes the form of a neural definite clause grammar rule, which are expressions of the form:

$$nn(m, \mathcal{I}, \mathcal{O}, \mathcal{D}) :: nt \rightarrow g_1, \ldots, g_n$$

where m is a unique identifier for a neural network, $\mathcal{I}$ is a list of variables that will be unified with the input, $\mathcal{O}$ is a list of variables that will be unified with the different possible outputs, $\mathcal{D}$ is a list of predicate names that define the domain of the output values, nt corresponds to the head of the rule and $g_1, \ldots, g_n$ corresponds to the body.

The MNIST addition from Example 4.4 is encoded as:

```
digit(0). digit(1). ...digit(9).
nn(mnist,[I],[D],[digit]) :: digit(D) -> [I].
addition(N) -> number(N1), number(N2), {N is N1+N2}.
```

4.2 Inference in DeepProbLog

We now discuss how inference is performed in DeepProbLog. As with the semantics, inference in DeepProbLog is discussed with respect to exact inference in ProbLog. The only change required for DeepProbLog inference is that we need to instantiate nADs, neural facts and neural functions to regular ADs and (probabilistic) facts. This is done in two steps. During grounding, we obtain ground nADs and ground neural facts with a symbolic representation of the probabilities. In a separate step after grounding, the concrete probabilities are determined by evaluating the neural network with the input as determined by the ground neural facts, ADs and functions.

Example 4.8 (DeepProbLog inference)
We calculate the probability of the query `addition`(, , `1`) on the program from Example 4.4. The first steps of inference are identical to that of ProbLog. In the first step, SLD resolution finds the proofs for this query. We summarize this process with the SLD search tree shown in Figure 4.2.

The procedure finds two proofs, one where the first image is classified as 0 and the second as 1. In the other proof, these two classifications are swapped. The resulting logical formula is

$$(digit(\texttt{}, 0) \wedge digit(\texttt{}, 1)) \vee (digit(\texttt{}, 1) \wedge digit(\texttt{}, 0))$$

Note that these two disjuncts are already mutually exclusive due to mutual exclusivity constraint of the annotated disjunction. This formula can directly be translated into the arithmetic circuit shown in Figure 4.3 Evaluating this arithmetic circuit results in $p(\texttt{addition}(\texttt{}, \texttt{}, \texttt{1})) = 0.14$

4.3 Learning in DeepProbLog

In contrast to the earlier approach for ProbLog parameter learning in this setting by Gutmann et al. (2008), DeepProbLog uses gradient based learning instead of expectation maximization. This allows for seamless integration with neural network training. The key insight here is that we can use the arithmetic circuit both for probabilistic inference and gradient computations. This arithmetic circuit is a differentiable structure, as it is composed of addition and multiplication operations.

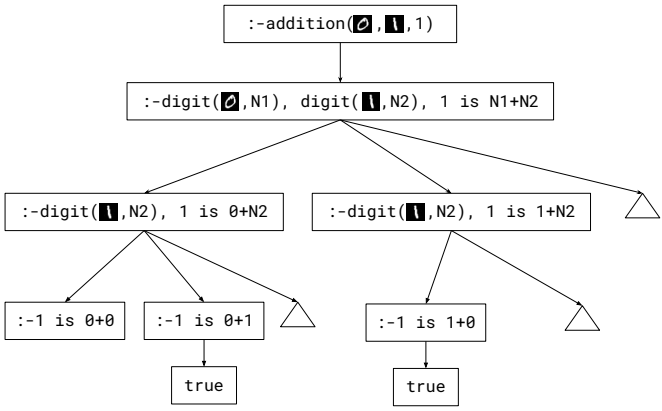


Figure 4.2: The SLD tree from Example 4.8

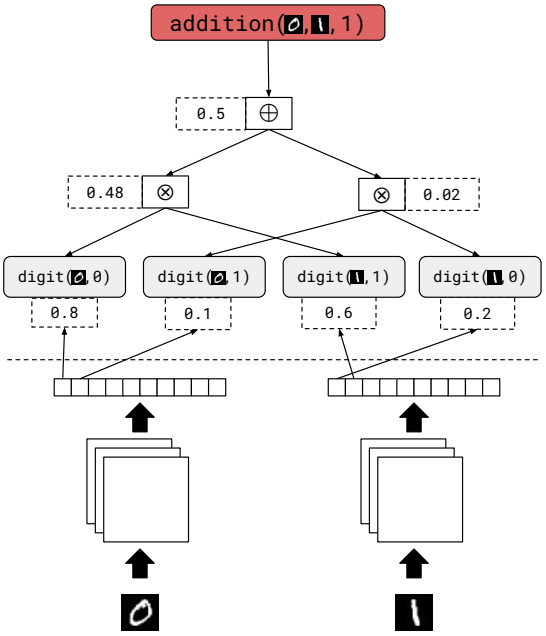


Figure 4.3: The arithmetic circuit from Example 4.8

DeepProbLog relies on the automatic differentiation capabilities already available in ProbLog to derive these gradients. More specifically, to compute the gradient through the probabilistic logic program we rely on aProbLog (Kimmig, Van

den Broeck, et al., 2011) as discussed in Section 2.2.3, and the gradient semiring.

4.3.1 Gradient semiring

To derive gradients, we use the gradient semiring. The elements of this semiring are tuples

$$\left(p, \frac{\partial p}{\partial w}\right)$$

where p is a probability, and $\frac{\partial p}{\partial w}$ is the partial derivative of that probability with respect to a parameter w. This is easily extended to a vector of parameters $\vec{w} = [w_1, \ldots, w_N]^T$, the concatenation of all N parameters in the ground program. The elements of the semiring then become tuples

$$(p, \nabla p)$$

where p is a probability and ∇p the gradient of p with respect to all parameters in $\vec{w}$. Addition $\oplus$, multiplication $\otimes$ and the neutral elements with respect to these operations are defined as follows:

$$(p_1, \nabla p_1) \oplus (p_2, \nabla p_2) = (p_1 + p_1, \nabla p_1 + \nabla p_2) \tag{4.1}$$

$$(p_1, \nabla p_1) \otimes (p_2, \nabla p_2) = (p_1 p_2, p_2 \nabla p_1 + p_1 \nabla p_2) \tag{4.2}$$

$$e^{\oplus} = (0, \vec{0}) \tag{4.3}$$

$$e^{\otimes} = (1, \vec{0}) \tag{4.4}$$

Note that the first element of the tuple performs ProbLog's probability computation, whereas the second element computes the gradient of the first element. Evaluation of the arithmetic circuit using the gradient semiring allows for both $P(q)$ and $\nabla P(q)$ to be calculated.

4.3.2 Gradient-based learning with aProbLog

To use the gradient semiring for parameter learning in ProbLog, we first transform the ProbLog program into an aProbLog program by extending the label of each probabilistic fact $p :: f$ to include the probability p as well as the gradient vector of p with respect to the learnable probabilities of all probabilistic

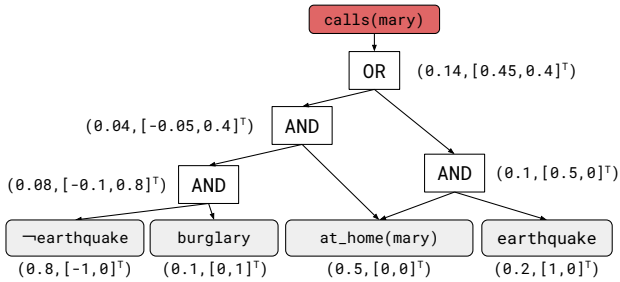


Figure 4.4: The AC of the burglary alarm example evaluated using the gradient semiring. Each node is annotated with their value in the gradient semiring.

facts and ADs in the program.

$$L(f) = (p, \vec{0}) \qquad \text{for } p :: f \text{ with fixed } p \tag{4.5}$$

$$L(f_i) = (p_i, \mathbf{e}_i) \qquad \text{for } t(p_i) :: f_i \text{ with learnable } p_i \tag{4.6}$$

$$L(\neg f) = (1 - p, -\nabla p) \qquad \text{with } L(f) = (p, \nabla p) \tag{4.7}$$

where the vector $\mathbf{e}_i$ has a 1 in the i-th position and 0 in all others. For fixed probabilities, the gradient does not depend on any parameters and thus is 0. Note that after each update step, the probabilistic parameters are clipped to the $[0, 1]$ range, and the parameters of an AD are re-normalized to ensure that they sum to one.

Example 4.9
Assume we want to learn the probabilities of `earthquake` and `burglary` from Example 2.5, while keeping those of the other facts fixed. Figure 4.4 shows the evaluation of the same AC as in Figure 2.3a, but with the gradient semiring. The nodes in the AC now also contain the gradient (the second element of the tuple). The result on the top node shows that the partial derivative of the query is 0.45 and 0.4 with respect to the earthquake and burglary parameters respectively.

4.3.3 Gradient-based learning for DeepProbLog

Just as the only difference between inference in ProbLog and DeepProbLog is the evaluation of the neural facts and nADs, the only difference between gradient-based learning in ProbLog and DeepProbLog is optimizing the neural parameters alongside the probabilistic parameters. As mentioned in the previous

section, the probabilistic parameters p_i in the logic program can be optimized by using the gradient semiring, which allows us to calculate $\partial P(q)/\partial p_i$. This gradient is then used to perform the update by using gradient-based learning. Note that since the outputs of the neural networks are used as probabilistic facts in the logic program, they are leafs in the AC. Conceptually, it might be easiest to think of the AC being connected with the differentiable structure of the neural network at these leaves (cf. Figure 4.5). This creates a single differentiable structure in which standard differentiation techniques can be used to derive gradients.

However, in this work, we do not explicitly join the AC and neural networks. Instead, we rely on the aProbLog method as described above. This makes the joint learning of probabilistic and neural parameters straightforward. Note that the gradients for the internal parameters inside the neural networks are still derived using backpropagation. The outputs of the neural networks can be considered *abstract* parameters. Although we can derive a gradient for these abstract parameters, we cannot optimize them directly, as the logic is unaware of the neural parameters that determine the value of these abstract parameters. Below, we show how we can derive the gradient for these neural parameters of the loss applied to $P(q|\mathcal{W})$ (Definition 2.4), rather than a loss function defined directly on the output of the neural network.

For example, consider the case of a single neural annotated disjunction, with probabilities $\hat{p}_i$ (i.e., the aforementioned abstract parameters), calculated by evaluating a neural network with softmax activation. The success probability of the query, based on the current values of the neural and probabilistic parameters, is written $P(q|\mathcal{W})$. While training, positive examples should yield a predicted success probability close to the expected success probability, which is expressed by means of a loss function $\mathcal{L}$ as introduced in Definition 2.4.

Application of the chain rule leads to

$$\frac{d\mathcal{L}}{dw_k} = \frac{\partial \mathcal{L}}{\partial P(q|\mathcal{W})} \sum_i \frac{\partial P(q|\mathcal{W})}{\partial \hat{p}_i} \frac{\partial \hat{p}_i}{\partial w_k}$$

where the derivative of the loss with respect to any trainable parameter w_k in the neural network is decomposed into the partial derivative of the loss with respect to the predicted output $P(q|\mathcal{W})$, the latter's derivative $\partial P(q|\mathcal{W})/\partial \hat{p}_i$ with respect to each head in the annotated disjunction as obtained with the gradient semiring, and finally $\partial \hat{p}_i/\partial w_k$, the derivative of the neural network's output components with respect to the considered parameter. The latter is obtained by the standard application of the chain rule in the neural network. The backpropagation procedure in the neural network can thus be started by providing $\partial P(q|\mathcal{W})/\partial \hat{p}_i$, to systematically obtain the loss gradients for

all neural parameters. Extending this approach to the situation of multiple neural predicates is straightforward. If the same neural network is used for different neural predicates, the final derivative is obtained by summing over the contributions of each neural predicate.

Then, standard gradient-based optimizers are used to update the parameters of the network, cf. Section 2.3. During gradient computation with aProbLog, the probabilities of neural ADs are kept constant. Furthermore, updates on neural ADs come from the neural network part of the model, where the use of a softmax output layer ensures a normalized distribution, hence not requiring the additional normalization as for non-neural ADs.

To extend the gradient semiring to DeepProbLog programs, we define it for nADs and neural facts. To construct the labels, we need to define an ordering for all neural facts and nAD heads in the program. The label for the nAD is defined as:

$$L(f_i) = (\hat{p}_j, \mathbf{e}_n)$$

for each $nn(m, [x_1, .., x_k], y_j) :: r(x_1, .., x_k, y_j)$ a ground nAD head, where y_j is the j-th domain element, $\hat{p}_j$ is the j-th element of the output of the neural network m evaluated on input $[x_1, .., x_k]$ and n is the index of the nAD in the chosen ordering.

The label for a neural fact is defined as:

$$L(f_i) = (\hat{p}, \mathbf{e}_j)$$

for each $nn(m, [x_1, .., x_k]) :: r(x_1, .., x_k)$ a ground neural fact, where $\hat{p}$ is the output of the neural network m evaluated on input $[x_1, .., x_k]$ and n is the index of the nAD in the chosen ordering.

Example 4.10
To demonstrate the learning procedure for neural networks and probabilistic parameters, we will apply it on the MNIST addition example with a small extension. We introduce label noise where some of the labels have been replaced randomly from a uniform distribution over $[0, 18]$. The goal is to both train the neural network and learn the fraction of noisy examples. The line `t(0.0) :: noisy` indicates that the estimated probability of having a noisy label is a parameter initialized at 0.0. This parameter is learned jointly with the neural network parameters. The DeepProbLog program is below:

```
nn(mnist, [X], Y, [0 ...9]) :: digit(X,Y).
t(0.0) :: noisy.
```

```
1/19 :: uniform(X,Y,0) ; ...; 1/19 :: uniform(X,Y,18).

addition(X,Y,Z) :-
    noisy,
    uniform(X,Y,Z).
addition(X,Y,Z) :-
    \+noisy,
    digit(X,N1),
    digit(Y,N2),
    Z is N1+N2.
```

Grounding on the query `addition(``,``,1)` results in following the ground DeepProbLog program:

```
nn(mnist,[],0)::digit(,0); nn(mnist,[],1)::digit(,1).
nn(mnist,[],0)::digit(,0); nn(mnist,[],1)::digit(,1).
t(0.2)::noisy.

1/19::uniform(,,1).
addition(,,1) :-
    noisy,
    uniform(,,1).

addition(,,1) :-
    \+noisy,
    digit(,0),
    digit(,1).
addition(,,1) :-
    \+noisy,
    digit(,1),
    digit(,0).
```

The arithmetic circuit corresponding to the ground program is shown in Figure 4.5. As can be seen, the neural networks already have a confident prediction for both images (`0.8 :: digit(``,0)` and `0.6 :: digit(``,1)`). The top right shows how the different partial derivatives that are calculated: one with respect to the noisy parameter, ten for the evaluation of the neural network on input and ten for the evaluation on input .

4.4 DeepProbLog's position along the NeSy dimensions

When classified along the dimensions introduced in Chapter 3, DeepProbLog is a directed, proof-based framework. It is an extension of ProbLog, and thus lies more towards the logic side in the logic vs probability neural spectrum. As will be shown in Section 4.5.2, DeepProbLog can perform structure learning through parameter learning, and can be used in the program sketching setting. DeepProbLog implements an interface between the logic and neural side, and thus uses both symbolic and sub-symbolic representations, but not simultaneously.

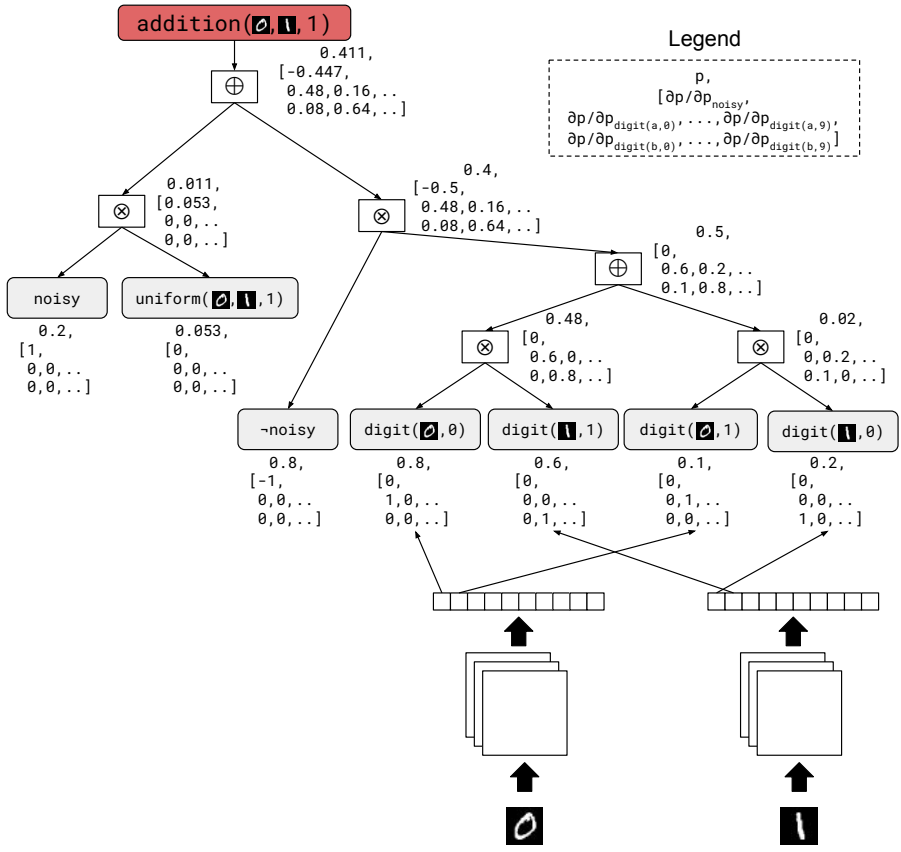


Figure 4.5: The AC for query `addition`(0, 1, 1).

4.5 Experiments

We perform four sets of experiments to demonstrate that DeepProbLog supports 1) logical reasoning and deep learning, 2) program induction, 3) probabilistic inference and combined probabilistic and deep learning, and 4) natural language reasoning and embeddings. All programs are given in full in Appendix B. Further details on the experiments are provided in Appendix A.1. We are looking to answer the following questions:

Q1.1 Can DeepProbLog combine learning and reasoning to learn from distantly supervised examples?

Q1.2 How does DeepProbLog compare to a neural network baseline w.r.t accuracy and data efficiency?

Q1.3 How well do DeepProbLog models generalize?

Q1.4 How noise-tolerant are DeepProbLog models?

Q2.1 Can DeepProbLog perform program induction through the program sketching approach?

Q2.2 How does DeepProbLog compare to other NeSy frameworks?

Q3.1 Can DeepProbLog combine learning and *probabilistic* reasoning?

Q3.2 How often does DeepProbLog converge on undesired solutions? How much additional supervision is needed to rectify this problem?

Q4.1 Can DeepProbLog deal with tasks involving natural language reasoning?

4.5.1 Q1: Logical reasoning and deep learning

To show that DeepProbLog supports both logical reasoning and deep learning, we extend the classic learning task on the MNIST dataset (Lecun et al., 1998) to four tasks that require reasoning. The accuracy is measured on the MNIST addition dataset created from the test images in the MNIST dataset.

T1: `addition`(, , 8)

Instead of using labeled single digits, we train on pairs of images, labeled with the sum of the individual labels. This is the same as Example 4.4. The DeepProbLog program consists of the clause

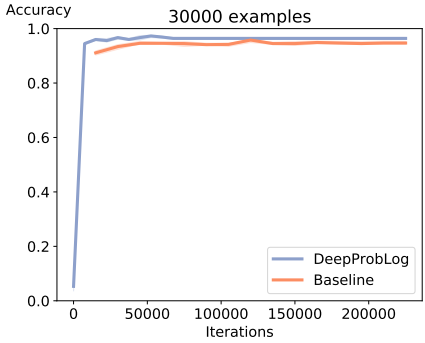
$$\texttt{addition}(\texttt{X},\texttt{Y},\texttt{Z}) :- \texttt{digit}(\texttt{X},\texttt{X2}),\texttt{digit}(\texttt{Y},\texttt{Y2}),\texttt{Z is X2} + \texttt{Y2}$$

and a neural AD for the `digit/2` predicate, which classifies an MNIST image. We compare to a CNN baseline that classifies the two images into the 19 possible sums.

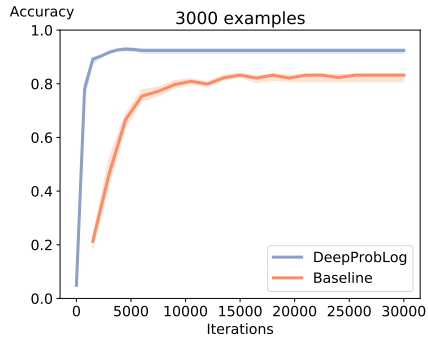
Results Figure 4.6 shows the learning curves for the baseline (orange) and DeepProbLog (blue) on the single-digit addition. To answer **Q1.1** and **Q1.2**, we evaluated on 3 levels of data availability: 30000 examples, 3000 and 300 examples. As can be seen in the figures, DeepProbLog converges faster and achieves a higher accuracy than the baseline. In the case for $N = 30000$ (Figure 4.6a), the difference between the baseline and DeepProbLog is significant, but not immense. However, for $N = 3000$ and especially $N = 300$, the difference becomes more apparent.
The reason behind this disparity is that the baseline needs to learn how to classify the combined input digits (and there are a 100 different sums possible), whereas the neural predicate only needs to recognize individual digits (with only 10 possibilities). Table 4.1 shows the average accuracy on the test set for the different models for different training set sizes.

	Number of training examples		
Model	30 000	3 000	300
Baseline	93.46 ± 0.49	78.32 ± 2.14	23.64 ± 1.75
DeepProbLog	97.20 ± 0.45	92.18 ± 1.57	67.19 ± 25.05

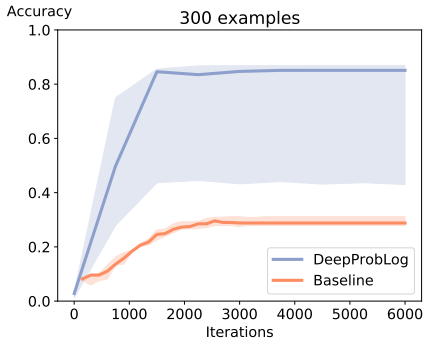
Table 4.1: The accuracy on the test set for **T1**. DeepProbLog outperforms the baseline, especially when the number of training examples decreases.



(a) 30 000 examples



(b) 3 000 examples



(c) 300 examples

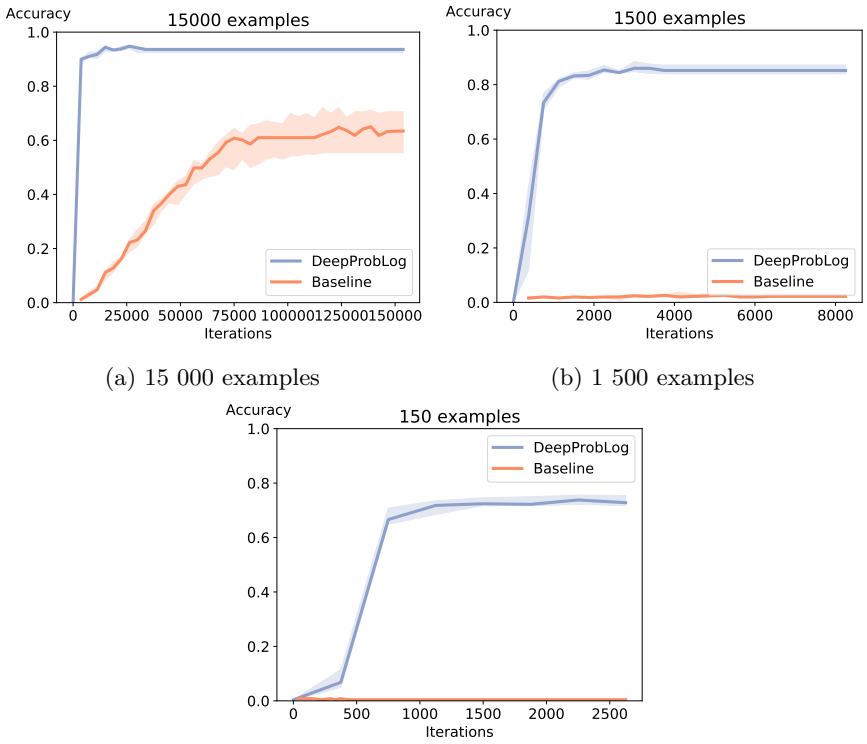
Figure 4.6: MNIST Single-Digit Addition (**T1**). The graphs show the accuracy on the validation set during training. DeepProbLog converges quicker and to a higher accuracy than the baseline, especially with fewer examples.

T2: `addition`([3, 8], [2, 5], 63)

The input consists of two lists of images. Each list represents a multi-digit number. The label is the sum of the two numbers. The neural predicate remains the same. Learning the new predicate requires only a small change in the logic program. Because the CNN baseline cannot handle numbers of varying size, we fixed the size of the input to two-digit numbers.

Results To answer **Q1.3**, we first perform an experiment where we take the neural network trained in **T1** and use it in this model without any further training. Evaluating it on the same test set, we achieve an accuracy that is not

significantly different from training on the full dataset of **T2**. This demonstrates that the approach used in DeepProbLog causes it to generalize well beyond training data. Figure 4.7 shows the learning curves for the baseline (orange) and DeepProbLog (blue) on the multi-digit addition, which further answers questions **Q1** and **Q2**. DeepProbLog achieves a somewhat lower accuracy compared to the single digit problem due to the compounding effect of the classification error on the individual digits, but the model generalizes well. The baseline fails to learn from few examples (150 and 1 500). It is able to learn with 15 000 examples, but converges noticeable slower than DeepProbLog. Table 4.2 shows the average accuracy on the test set for the different models for different training set sizes.



(a) 15 000 examples

(b) 1 500 examples

(c) 150 examples

Figure 4.7: MNIST Multi-Digit Addition (**T2**). The graphs show the accuracy on the validation set during training for different training set sizes. DeepProbLog converges quicker and to a higher accuracy than the baseline, especially with fewer examples.

	Number of training examples			
Model	15 000	1 500	150	**T1** (30 000)
Baseline	60.85 ± 9.77	1.34 ± 0.53	0.80 ± 0.14	–
DeepProbLog	95.16 ± 1.70	87.21 ± 1.92	72.73 ± 3.03	93.36 ± 1.18

Table 4.2: The accuracy on the test set for **T2**. DeepProbLog outperforms the baseline, especially when the number of training examples decreases.

T3: `addition`(, , )

The input consists of 3 MNIST images, where the last is the sum of the first two. This task demonstrates potential pitfalls of only having distant supervision. Namely, without any regularization, the neural network quickly learns to predict 0 for all digits, The model collapses to always predicting $0 + 0 = 0$, as it is a valid logical solution and a trivial solution for the neural network. To avoid this, we add a regularization term based on entropy maximization. It is calculated per network for each batch. It is calculated as

$$1.0 - \sum_{i=1}^{n} P_i^{\mu} \log_n(P_i^{\mu}) \tag{4.8}$$

where

$$P_i^{\mu} = \frac{1}{m} \sum_{j=1}^{m} P_{i,j}$$

$P_{i,j}$ is the i-th output of the j-th element from the batch, n is the total number of output classes and m is the size of the batch. The intuition behind this regularization term is that it penalizes mode collapse (i.e. one class receiving all probability mass for all inputs) by requiring the entropy of the average output distribution per batch to be high. As such, this term encourages exploration, but is only necessary to start the training of the neural networks. If they are sufficiently trained, this term can be dropped. This additional regularization loss is multiplied by a factor λ and added to the cross-entropy loss. We run the experiment for different values of λ.

Results Figure 4.8 shows the accuracy of the neural predicate on classifying single digits for different levels of the regularization parameter. As can be seen, for $\lambda = 2$, the neural predicate converges to the trivial solution. For $\lambda = 4$, the neural predicate sometimes converges on the correct solution, but can also converge on the wrong solution. For $\lambda = 8$, the neural network consistently

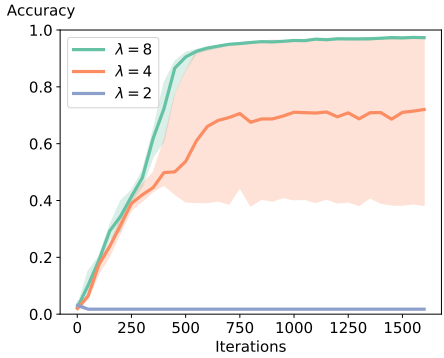


Figure 4.8: The accuracy on the MNIST test set for individual digits while training on (**T3**). The convergence becomes more reliable as the value for the λ parameter increases.

converges on the correct solution. This experiment further answers **Q1.1**. DeepProbLog is not always able to correctly learn from examples using distant supervision, but some of these shortcomings can be overcome through additional regularization.

T4: `addition`(, , 14)

This experiment is the same as Example 4.10. It is the MNIST addition example, but with noise introduced in the labels. Namely, a fraction of the labels is replaced by a uniformly selected number between 0 and 18. We compare three models: the CNN baseline from **T1**, the DeepProbLog model from **T1**, and a DeepProbLog model where the noise is explicitly modeled as shown in Experiment 4.10. To answer **Q1.4**, we train with different levels of noise.

Results Table 4.3 shows the accuracy on the test set which has no noise. The baseline is not tolerant to noisy labels, quickly dropping in accuracy as the fraction of noisy labels increases. The DeepProbLog model from **T1** is more tolerant, but also drops noticeably in accuracy as the fraction of noise goes over 0.5. Explicitly modeling the noise makes the model very noise tolerant, even retaining an accuracy of 73.2% with 80% noisy labels. The last row shows the value of the learned `noisy` parameter, stating the estimated likelihood of a noisy label. As shown in the last row of the table, this value is close to the actual fraction of noisy labels. This shows the power of including probabilistic reasoning, as the model is able to recognize which examples have noisy labels.

	Fraction of noisy labels					
	0.0	0.2	0.4	0.6	0.8	1.0
Baseline	93.46	87.85	82.49	52.67	8.79	5.87
DeepProbLog	97.20	95.78	94.50	92.90	46.42	0.88
DeepProbLog w/ explicit noise	96.64	95.96	95.58	94.12	73.22	2.92
`noisy` parameter	0.000	0.212	0.415	0.618	0.803	0.985

Table 4.3: The accuracy on the test set for **T4**. Explicitly encoding noise creates a robust model that is able to learn the fraction of noisy labels in the dataset.

.

4.5.2 Q2: Program Induction

The second set of problems demonstrates that DeepProbLog can perform program induction. We follow the program sketching (Solar-Lezama, 2013) setting of differentiable Forth (Bošnjak et al., 2017), where the holes in programs are filled by neural networks, whose behavior is learned through input-output examples for the entire program. We consider three tasks: addition, sorting (Reed and Freita, 2016) and word algebra problems (Roy and Roth, 2015).

T5: `forth_addition`$([4],[8],1,[1,3])$

The input consists of two numbers, represented as lists of digits, and a carry. The output is the sum of the numbers and the carry. This is implemented in DeepProbLog as:

```
nn(m_result,[D1,D2,C],Y,[0,...,9])::result(D1,D2,C,Y).
nn(m_carry,[D1,D2,C],Y,[0,1])::carry(D1,D2,C,Y).

hole(I1,I2,Carry,NewCarry,Result) :-
    result(I1,I2,Carry,Result),
    carry(I1,I2,Carry,NewCarry).

add([],[],[C],C,[]).
add([H1|T1],[H2|T2],C,Carry,[Digit|Res]) :-
    add(T1,T2,C,NewCarry,Res),
    hole(H1,H2,NewCarry,Carry,Digit).

addition(L1,L2,C,[Carry|Res]) :- add(L1,L2,C,Carry,Res).
```

The program specifies the basic addition algorithm in which we go from right to left over all digits (`add/5`), calculating the sum of two digits and taking the carry over to the next pair. The hole in this program (`hole/5`) corresponds to calculating the resulting digit (`result/4`) and carry (`carry/4`), given two digits and the previous carry.

Results The results are shown in Table 4.4. Both differentiable Forth($\partial 4$) and DeepProbLog achieve 100% on all training sizes, giving a positive answer to **Q2.1**.

		Training length		
	Test length	2	4	8
$\partial 4$ (Bošnjak et al., 2017)	8	100.0	100.0	100.0
	64	100.0	100.0	100.0
DeepProbLog	8	100.0	100.0	100.0
	64	100.0	100.0	100.0

Table 4.4: Accuracy on the addition (**T5**) problem. The results for $\partial 4$ are as reported by Bošnjak et al. (2017). Training length and test length indicate the length of the sequences for respectively the train and test sets.

T6: `forth_sort`$([8, 2, 4], [2, 4, 8])$

The input consists of a list of numbers, and the output is the sorted list. The program implements bubble sort, but leaves open what to do on each step in a bubble (i.e. whether to swap or not: `swap/2`).

```
nn(m_swap, [X,Y]) :: swap(X,Y).

hole(X,Y,X,Y):- \+swap(X,Y).
hole(X,Y,Y,X):- swap(X,Y).

bubble([X],[],X).
bubble([H1,H2|T],[X1|T1],X):-  hole(H1,H2,X1,X2), bubble([X2|T],T1,X).

bubblesort([],L,L).
bubblesort(L,L3,Sorted) :- bubble(L,L2,X), bubblesort(L2,[X|L3],Sorted).

forth_sort(L,L2) :- bubblesort(L,[],L2).
```

Results The results are shown in Table 4.5. Similarly to $\partial 4$, DeepProbLog achieves 100% on training sizes 2 and 3. However, whereas $\partial 4$ fails to converge on training sizes larger than 3, DeepProbLog stills achieves 100% accuracy. As Bošnjak et al. (2017) mention, the failure of $\partial 4$ is due to computational issues arising from the long program trace resulting from sorting long lists. DeepProbLog does not suffer from these issues. Table 4.6 answers **Q2.2** as it shows that DeepProbLog runs faster and scales better with increasing training length compared to $\partial 4$.

		Training length				
	Test length	2	3	4	5	6
$\partial 4$ (Bošnjak et al., 2017)	8	100.0	100.0	49.22	–	–
	64	100.0	100.0	20.65	–	–
DeepProbLog	8	100.0	100.0	100.0	100.0	100.0
	64	100.0	100.0	100.0	100.0	100.0

Table 4.5: Accuracy on the sorting (**T6**) problem. The results for $\partial 4$ are as reported by Bošnjak et al. (2017). Training length and test length indicate the length of the sequences for respectively the train and test sets. $\partial 4$ fails to converge for training lengths ≥ 4

	Training length				
	2	3	4	5	6
$\partial 4$ on GPU	42 s	160 s	–	–	–
$\partial 4$ on CPU	61 s	390 s	–	–	–
DeepProbLog	11 s	14 s	32 s	114 s	245 s

Table 4.6: Running time until the models reach 100% accuracy on test length 8 for the sorting (**T6**) problem.

T7: `wap(`*'Robert has 12 books. He sells 3 ...'*`,12,3,1,10)`

The input to the word algebra problems consists of a natural language sentence describing a simple mathematical problem. These problems always contain three numbers, which are extracted from the string and are given as part of the input. The output is the solution to the question. Every problem can be solved by chaining the following 4 steps: permuting the three numbers (`permute/2`), applying an operation on the first two numbers (addition, subtraction or multiplication: `operation_1/2`), potentially swapping the intermediate result and the last digit (`swap/2`), and performing a last operation (`operation_2/2`). The hole in the program is in deciding which of the alternatives should happen on each step.

Results DeepProbLog reaches an accuracy of up to 96.5%, similar to the results for $\partial 4$ reported by Bošnjak et al. (2017) (96%).

4.5.3 Q3: Probabilistic programming and deep learning

In this section, we introduce two experiments that show the intricacies involved in combining probabilistic logic programming and deep learning.

T8: Coin classification and comparison

In this experiment, we train two neural networks using distant supervision. The input consists of a synthetic image containing two coins (an example is shown in Figure 4.9). They are either heads or tails. The image is labeled either with *same* or *different*. We train a neural network for each coin to predict either *heads* or *tails* from the entire image. Solving this task requires solving two problems. On the one hand, the neural networks have to learn to recognize and separate the two different coins; on the other hand, they also have to each classify a different coin as heads or tails.

The first question we ask is whether the neural networks can recover the expected latent structure imposed by the logic program. We expect the two neural networks to agree on which side of the coin is heads and which is tails, however, this might be the inverse of what is generally considered heads and tails. Furthermore, we expect the two neural networks to each pick one coin to label, but which network classifies which coin will vary between runs. As such, there are four possible solutions that the neural networks can converge on. The second question we ask is how many additionally labeled examples (with both the label for same/different and heads/tails of one of the coins given) we need for the neural network to recover the desired latent representation.

Results To answer **Q3.2**, we ran each experiment 100 times. The fraction of runs that converged on either no solution, the expected solution or a logically equivalent solution is shown in Table 4.7. We see that with no additionally labeled examples, DeepProbLog does not converge on a satisfactory solution in about half of all runs. When it does converge on a solution, it converges on the *expected* solution 25% of the time, and on different solutions 75% of the time, which is to be expected as there are four possible interpretations of which coin is first or second, and which side is heads or tails. We can also see that as the number of additionally labeled examples increases, DeepProbLog converges more reliably, and more on the expected solution. Starting with 10 additionally labeled examples, DeepProbLog reliably converges on the desired solution. Beyond 20 additionally labeled examples, we do not see any further improvements.

Labeled examples	Not solved	Expected solution	Other solution
0	56%	11%	33%
5	39%	40%	21%
10	7%	92%	1%
20	4%	96%	0%
50	3%	97%	0%
100	4%	96%	0%

Table 4.7: The fraction of runs that converged to different outcomes for the Coins experiment (**T8**). The model converges consistently to the correct solution when given at least 10 labeled examples.



Figure 4.9: An example input image for the Coins (**T8**) experiment.

T9: `0.8::poker([Q♡, Q♢, A♢, K♣],loss).`

In this experiment we demonstrate that DeepProbLog can perform combined probabilistic reasoning, probabilistic learning and deep learning.

We do this by playing a simplified poker game: there are two players that are dealt two cards from several decks. There is also one community card. Each player then makes a poker hand (e.g. pair, straight, ...) with their two cards and the community card. The best combination wins.

The task is to learn to correctly predict the probability that the first player wins. To do this, the model also has to learn the distribution of the community card, which is not observed and is considered a latent variable. By modelling the task with knowledge of its structure, we are able to construct an interpretable model in which the distribution of this latent variable is made explicit. This contrasts

with a neural model, which, although it might be able to perform the same regression, would not provide the user with information of this distribution.

For simplicity, we only use the jack, queen, king and ace. We also do not consider the suits of the cards. The input consists of 4 images that show the cards dealt to the two players. Additionally, every example is labeled with the chance that the game is won, lost or ended in a draw, e.g.:

$$0.8 :: \texttt{poker}([\texttt{Q}\heartsuit, \texttt{Q}\diamondsuit, \texttt{A}\diamondsuit, \texttt{K}\clubsuit], \texttt{loss})$$

We expect DeepProbLog to: 1) train the neural network to recognize the four cards, 2) reason probabilistically about the non-observed community card, and 3) learn the distribution of the non-observed community card.

To make DeepProbLog converge more reliably, we add some examples with additional supervision. Namely, in 10% of the examples we additionally specify the community card, i.e.

$$\texttt{poker}([\texttt{Q}\heartsuit, \texttt{Q}\diamondsuit, \texttt{A}\diamondsuit, \texttt{K}\clubsuit], \texttt{A}\diamondsuit, \texttt{loss}).$$

This also showcases one of the strengths of DeepProbLog, namely, it can make use of examples that have different levels of observability. The loss function used in this experiment is the mean squared error between the predicted and target probabilities, as this proved to lead to more accurate estimates for the probabilistic parameters.

Results To answer **Q3.1** and **Q3.2**, we ran the experiment 10 times. Out of these 10 runs, 4 did not converge on the correct solution. The average values of the learned parameters for the remaining 6 runs are shown in Table 4.8. As can be seen, DeepProbLog is able to correctly learn the probabilistic parameters. In these 6 runs, the neural network also correctly learned to classify all card types, achieving a 100% accuracy. The other runs did not converge because some of the classes were permuted (i.e., queens predicted as aces and vice versa) or multiple classes mapped onto the same one (queens and kings were both predicted as kings).

Distribution	Jack	Queen	King	Ace
Actual	0.2	0.4	0.15	0.25
Learned	0.203 ± 0.002	0.396 ± 0.002	0.155 ± 0.003	0.246 ± 0.002

Table 4.8: The estimated fraction of the cards in the dataset for the Poker experiment (**T9**).

4.5.4 Q4: Natural Language Reasoning

T10: Compositional language understanding and reasoning

The final experiment demonstrates DeepProbLog's ability to manipulate distributed representations (embeddings), effectively leading to a wider interface between the neural and logic components. The aim is to illustrate the framework's potential for real-world applications beyond the toy tasks in the previous examples. In particular, this experiment explores to what extent the framework allows reasoning over natural language sentences. For this purpose, we will use the Compositional Language Understanding with Text-based Relational Reasoning (CLUTRR) dataset (Sinha et al., 2019).

The CLUTRR dataset consists of short natural language texts of several sentences containing facts about family relations between people. The goal is to infer the relation between two people mentioned in the text. These relations are not directly mentioned, but have to be inferred from other relations in the text. The task in this experiment is thus to train the neural networks to perform relation extraction. That is, we have to extract logical facts from natural language sentences, as shown in Figure 4.10. In contrast to the original setup of the CLUTRR experiment, we do not solve the structure learning aspect. Instead, we follow the idea that is central to DeepProbLog that we can provide the model with background knowledge when it is available, and only learn those parts that we cannot define easily in logic. In this case, we provide the family relations shown in Listing 4.1, and focus on the relation extraction task from the sentences. Thus, one can expect good generalization capabilities in the model, if the individual relations are extracted correctly.

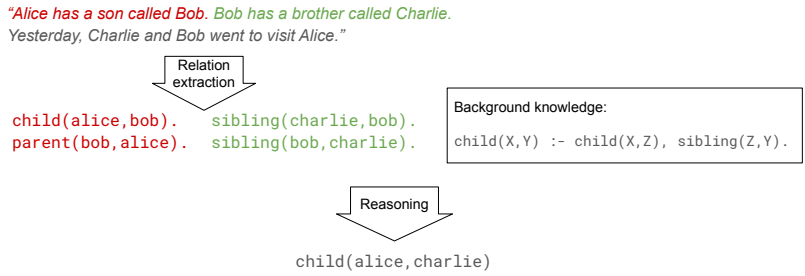


Figure 4.10: An example inference task from the CLUTRR dataset.

A simplified version of the code can be seen in Listing 4.2. There are two ways to prove relations between entities: reasoning using background knowledge, and extracting relations from the text. The relation extraction happens in the

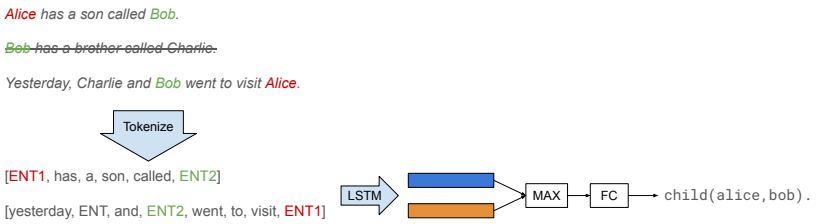


Figure 4.11: The implementation of the neural predicate for relation extraction, illustrated on the relation between Alice and Bob.

`neural_relation/4` predicate. First, we select all relevant sentences for the given entities, that is, the sentences that contain both entities. Then we embed these using the `embed/3` predicate, which embeds a single sentence (e.g., encodes the sequence of tokens into a vector) using a gated recurrent unit encoder (GRU (Cho et al., 2014), a recurrent architecture similar to the LSTM) on top of randomly initialized token embeddings. The final state of the GRU for each sentence is used as the embedding. Afterwards, the `max_tensor/2` predicate performs a max pooling over the sequence of embeddings. The relations are predicted through a logistic regression classifier (a single linear layer with a softmax activation function) on top of the final text representation (`nn_rel/2`). A diagrammatic overview is given in Figure 4.11. A similar process happens for the `neural_gender/3` predicate that predicts the gender for a single entity. The benefit of being able to manipulate embeddings inside the logic itself is that the neural model becomes simpler and the logic becomes more expressive. This shows that DeepProbLog is more than a simple pipeline where logical reasoning follows neural network evaluation. Instead, DeepProbLog is an interface between the logic and neural sides.

The model as it is currently shown relies on the fact that all clauses are present. However, if this were not the case, we could replace the gaps in the knowledge by a neural counterpart (i.e., a neural component that tries to directly predict the missing relation directly from the entire text).

Experiments The first experiment on the CLUTRR dataset investigates the innate ability of DeepProbLog to generalize. There are several facts that connect the two query entities. The number of facts needed to connect the two will be called k (e.g. in the example in Figure 4.10, $k = 2$). In particular, we show how the model generalizes to longer texts and an increasing number of reasoning steps than seen during training. To do this, it is trained for smaller k (e.g. $k < 4$) and evaluated on larger k (e.g. $k < 10$). The model is trained until

```
grandchild(X,Y) :- child(X,Z), child(Z,Y).
grandchild(X,Y) :- so(X,Z), grandchild(Z,Y).
grandchild(X,Y) :- grandchild(X,Z), sibling(Z,Y).
grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
grandparent(X,Y) :- sibling(X,Z), grandparent(Z,Y).
child(X,Y) :- child(X,Z), sibling(Z,Y).
child(X,Y) :- so(X,Z), child(Z,Y).
parent(X,Y) :- sibling(X,Z), parent(Z,Y).
parent(X,Y) :- child(X,Z), grandparent(Z,Y).
sibling(X,Y) :- child(X,Z), uncle(Z,Y).
sibling(X,Y) :- parent(X,Z), child(Z,Y).
sibling(X,Y) :- sibling(X,Z), sibling(Z,Y).
child_in_law(X,Y) :- child(X,Z), so(Z,Y).
parent_in_law(X,Y) :- so(X,Z), parent(Z,Y).
nephew(X,Y) :- sibling(X,Z), child(Z,Y).
uncle(X,Y) :- parent(X,Z), sibling(Z,Y).
```

Listing 4.1: The background knowledge used in **T10**

no further improvements occur for two consecutive steps on the accuracy on a held-out set with $k = 2$. We hypothesize that the neural network mainly needs to learn to recognize elementary relations, whereas the logic component will tackle the reasoning required to deal with larger k values.

The second experiment investigates the robustness of the reasoning component. To test this, the CLUTRR dataset adds unnecessary sentences and facts to the text. These include supporting facts, irrelevant facts and disconnected facts. Supporting facts are facts that provide an alternate explanation for (part of) the relation between the two query entities. Irrelevant facts are facts that mention relevant entities, but cannot be used to explain the query relation. Disconnected facts are about entities and relations that are not related at all to the query or other entities. In this experiment, we train on clean data ($k = 2, 3$) and test how well it performs on the other data ($k = 3$). We also see what the impact of training on the supporting, irrelevant or disconnected data has on the accuracy. We trained until we saw no further improvement for 2 steps on the accuracy on the test set for $k = 2$.

```
%Neural network that predicts relation for the given embedding
nn(net1,[E],R,[child,child_in_law,...]) :: nn_rel(E,R).
%Neural network that predicts the gender for a given embedding
nn(net2,[E],G,[male,female]) :: nn_gender(E,G).

%Predicate that proves the relation R between X and Y
query_rel(T,X,R,Y):-
    %Determine the gender G for entity Y in text T
    neural_gender(T,Y,G),
    %Turn the gendered relation R into the un-gendered relation R2
    gender_rel(R2,G,R),
    %Prove relation R2 between entities X and Y
    call(R2,T,X,Y).

%Predicate that predicts the gender G for entity E
neural_gender(T,E,G) :-
    %E is an entity mentioned in the text T
    entity(T,E),
    %Only include relevant sentences (i.e. E is mentioned)
    include(mentioned(E), T, Relevant),
    %Map each relevant sentence onto its embedding
    maplist(embed([E]),Mentioned,Embedded),
    %Take the element-wise max of all embeddings
    max_tensor(Embedded,Max),
    %Classify the resulting embedding
    nn_gender(Max,G).

%Predicate that predicts the relation R between entities X and Y
neural_relation(Sentences,X,R,Y) :-
    %X and Y are entities mentioned in the sentences
    entity(Sentences,X),entity(Sentences,Y),X \= Y,
    %Only include relevant sentences (i.e. X and Y are mentioned)
    include((mentioned(X),mentioned(Y)),Sentences,Mentioned),
    %Map each relevant sentence onto its embedding
    maplist(embed([Y,X]),Mentioned,Embedded),
    %Take the element-wise max of all embeddings
    max_tensor(Embedded,Max),
    %Classify the resulting embedding
    nn_rel(Max,R).

child(T,X,Y) :- neural(T,X,child,Y).
child_in_law(T,X,Y) :- neural(T,X,child_in_law,Y).
...
grandchild(T,X,Y) :- X\=Y, child(T,X,Z), child(T,Z,Y).
grandchild(T,X,Y) :- X\=Y, so(T,X,Z), grandchild(T,Z,Y).
grandchild(T,X,Y) :- X\=Y, grandchild(T,X,Z), sibling(T,Z,Y).
grandparent(T,X,Y) :- X\=Y, parent(T,X,Z), parent(T,Z,Y).
grandparent(T,X,Y) :- X\=Y, sibling(T,X,Z), grandparent(T,Z,Y).
...
```

Listing 4.2: Part of the code for the CLUTRR experiment (**T10**)

Results Figure 4.12 shows the result for the systematic generalization experiment. The DeepProbLog model shows strong generalization capabilities for all three training datasets, giving a positive answer to questions **Q4.1** and **Q1.3**. This is in contrast with the results as shown in Sinha et al. (2019), where most methods have a steady decline in accuracy. This result shows the benefit of performing logical reasoning with background knowledge. If the relation extraction is correct, DeepProbLog has no problem generalizing to larger values of k. Table 4.9 shows the result for the robust reasoning experiment. It shows that the DeepProbLog model is indeed robust (**Q1.4**). This is in line with its observed generalization capabilities. As long as the individual relations are predicted correctly, the reasoning is also correct.

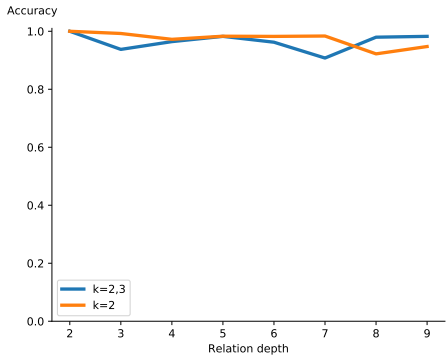


Figure 4.12: The accuracy on the systematic generalization task of CLUTRR for different training sets. k is the relation depth in the training data. DeepProbLog does not lose accuracy as the relation depth increases.

Training	Testing	Accuracy
Clean	Clean	1.0
	Supporting	0.99
	Irrelevant	0.98
	Disconnected	0.99
Supporting	Supporting	1.0
Irrelevant	Irrelevant	1.0
Disconnected	Disconnected	0.94

Table 4.9: Robust reasoning on the CLUTRR task for different types of noise. DeepProbLog remains relatively unaffected by the introduction of noisy facts.

4.5.5 Inference speed

Due to the nature of the exact inference used in DeepProbLog, which it inherits from aProbLog (Kimmig, Van den Broeck, et al., 2011), the grounding and compilation steps can become expensive as the problem size grows. Since training the neural networks requires many iterations over the dataset, this can become prohibitively expensive. Table 4.10 shows the mean time and standard deviation for the different steps in inference for the MNIST addition task for numbers of increasing length. As we can see, the time spent on grounding increases steeply and quickly becomes the dominant factor.

It is important to note that when we evaluate an example a second time, the structure of the AC, which is determined by the grounding and compilation, remains the same. Only the learned probabilities in the nAD change. We make use of this to improve the performance by caching the arithmetic circuits so that we only have to perform the (potentially expensive) grounding and compilation steps once. During evaluation, we only re-evaluate the neural networks and evaluate the AC with the updated probabilities.

Note that this optimization can also be applied to, for example, the queries `addition`(, , `8`) and `addition`(, , `8`), as both have the same structure. To do this, we introduce placeholder constants and change both queries to the single query `addition`(`a`, `b`, `8`), which reduces all queries in **T1** to 19 unique queries, one for each different label. During the evaluation of the neural networks, we replace the constants with the correct input and use the resulting probabilities in the cached ACs. We apply these optimizations to all experiments.

Number length	Grounding (s)	Compilation (s)	Evaluation (s)
1	0.018 ± 0.00	0.004 ± 0.00	0.017 ± 0.01
2	1.39 ± 0.25	0.035 ± 0.02	0.158 ± 0.07
3	114 ± 5.23	0.140 ± 0.04	0.567 ± 0.08

Table 4.10: Mean and standard deviation of the duration in seconds of different inference steps for 100 queries of the MNIST addition example for increasing number lengths (k).

4.6 Conclusions

In this chapter, we introduced the DeepProbLog language. It is the integration of the probabilistic logic programming language ProbLog and neural networks. The unifying concept to perform this integration is the neural predicate, which represents neural networks as probabilistic facts inside the program. We experimentally evaluated the DeepProbLog framework on experiments that demonstrate its capabilities in combined symbolic and sub-symbolic reasoning, program induction, probabilistic logic programming, embeddings and natural language processing.

We showed that with the inclusion of logic, DeepProbLog models can outperform neural networks in accuracy and data efficiency. We also showed that DeepProbLog models are robust and generalize well. We investigated some of the pitfall of combining reasoning with distant supervision, and how to overcome such challenges. Finally, we also demonstrated the power of DeepProbLog on natural language reasoning tasks by evaluating it on the CLUTRR dataset. However, we also note that exact inference in combination with reasoning over the entire output of the neural network can be prohibitively slow.

Chapter 5

Approximate inference for Neural-symbolic Logic Programming

This chapter is largely based on: R. Manhaeve, G. Marra, and L. De Raedt (2021). "Approximate Inference for Neural Probabilistic Logic Programming". In: *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning*, pp. 475–486.

It is well known that probabilistic logic inference is computationally hard. DeepProbLog originally only supported exact inference. However, since all proofs for the query have to be considered, this can be prohibitively expensive. There are however several approximate inference algorithms for ProbLog that we can draw inspiration from.

In this chapter, we introduce an approximate inference technique for DeepProbLog, called *DPLA**. It applies ideas from A*-search to DeepProbLog inference to find a small set of good proofs, which are then compiled into a circuit for learning and inference. Which proofs are best depends on the likelihood of these proofs, which in turn depends on the parameters of the program. An important challenge is to determine the best proofs while learning the parameters themselves, as any change in parameters will have an effect on the likelihood of the proofs. This challenge is particularly important with neural networks as they can perform better when their parameters are suitably initialized, cf. the success of curriculum learning.

It is well-known that for some domains the number of possible ground proofs can explode, and also that knowledge compilation is computationally hard (Darwiche and Marquis, 2002). This has inspired various approximate inference techniques for probabilistic logic programming. Especially relevant are the techniques that do not compute the full set of proofs but focus on a subset of the proofs (Renkens et al., 2012; De Raedt, Kimmig, and Toivonen, 2007), e.g., the k-best proofs. If fewer proofs are used, then subsequent inference steps also become simplified and faster. In this work, we will optimize the parameters in the model with only the most likely proof, cf. Viterbi training (Jelinek, 1976), which is equivalent to optimizing the explanation probability instead of the success probability for our query. The method discussed here will only be used to find the best proof, but it is also capable of finding the k best proofs, or even all proofs. In this work, we will only consider programs without negation.

5.1 Approximate inference in ProbLog

Exact inference in ProbLog is hard as it involves weighted model counting (WMC), which is #P-complete. This is why there has already been a lot of research into approximate inference for probabilistic logic programming and StarAI. Approximate inference for probabilistic logic programming can be divided into two categories: approximate proving and approximate WMC.

5.1.1 Approximate Proving

These methods use only a subset of all proofs for a query. The k-best approach (Gutmann et al., 2008) is a branch-and-bound algorithm that only includes the top-k most likely proofs. This makes it very similar to DPLA* with uniform-cost search without exploration. The idea of k-best was further refined with k-optimal (Renkens et al., 2012). K-best considers the probability of each proof separately. K-optimal, on the other hand, looks at the total probability of the k proofs that it is considering, and subsequently selects the proof that yields the maximum increase in total probability. Neither k-best nor k-optimal consider a learning setting.

A related approach is that of bounded approximation. Here, partial proofs are pruned when they reach a certain depth. The disjunction of the proofs that have been found already are used to compute the lower bound of the query probability, as finding more proofs can only increase the probability. The partial proofs that were not fully resolved are used to compute the upper bound. If

the difference between these two bounds is too large, the procedure is restarted with a deeper depth bound.

5.1.2 Approximate Weighted Model Counting

Some methods rely on only approximating the weighted model count to avoid solving the disjoint sum problem. Everything up to this step is unchanged (i.e. the grounding / proving phase, and logical formula construction). In disjunctive normal form (DNF) sampling (Shterionov et al., 2010), a sampling approach is used to approximate the real WMC from a DNF formulation of the logical formula. Similarly, Markov chain Monte Carlo (MCMC) estimation (Moldovan et al., 2013) estimates conditional probabilities by implementing an MCMC on the and-or tree produced by the grounding phase.

5.1.3 Other Approaches

Some approaches deviate from the standard inference procedure. One such approach is program sampling (Kimmig, Costa, et al., 2008). In this approach, one repeatedly samples a logic program by determining the truth value for all probabilistic facts by sampling according to their specified probability. For each sampled program, we check if the query is entailed. The fraction of samples that entail the query is the probability of the query.

T_p compilation (Vlasselaer et al., 2016) combines forward reasoning with formula construction. At each step of the forward inference, if a rule can be applied, it both adds the head to the partial interpretation, and adds a logical formula representing that rule to the compiled formula.

There has also been work on best-first search for pure Prolog (Lu and Liu, 2013). Since this is not a probabilistic setting, there is no preference of one derivation over another. In this work, a UCB-based heuristic is used to guide the proving process so that it more quickly finds a derivation for the query.

5.2 Proving as A*-search

The core component of DPLA* is an A*-based search in the SLD tree for the best proof. The standard SLD-resolution procedure used in Prolog and ProbLog searches the tree depth-first, left-to-right. DPLA* changes this ordering based on the probability of the proof considered so far, and a heuristic.

5.2.1 Partial proofs

The nodes in the SLD search tree are partial proofs. To be able to unify the concepts of the query, a partial proof and a completed proof, we define partial proofs as follows:

Definition 5.1
A partial proof is a pair $E = (E^f, E^g)$. E^f is the partial interpretation that contains the facts that have already been used in the proof, and E^g denotes the goal that still needs to be proven.

Since we do not consider negation in this work, E^f denotes only the set of probabilistic facts of the partial interpretation that are assigned the value *true*.

If E^g contains no literals, then E represents a complete proof and contains the set of ground probabilistic or neural facts used in the proof. A query q corresponds to a partial proof $E = (\emptyset, q)$. The explanation probability of a partial proof is defined as

$$P_x(E) = P_x(\bigwedge_{f \in E^f} f \bigwedge_{g \in E^g} g) \tag{5.1}$$

Example 5.1 (Partial proof)
One of the partial proofs encountered while proving the query `addition(``, ``, 1)` in the running MNIST addition example (Example 4.4) is

$$E = (\{\texttt{digit(}\texttt{,0)}\}, \{\texttt{digit(}\texttt{,N2), 1 is 0 + N2}\})$$

In this proof, we have derived that the first image represents the number 0. It still needs to find which number `N2` is represented by the second image so that the two numbers sum to 1.

5.2.2 A* Search

The heuristic that we will use is an estimate of the probability of a partial proof E. The explanation probability of a partial proof can be factorized as

$$P_x(E) = \left(\prod_{f \in E^f} P(f) \right) P_x(\bigwedge_{g \in E^g} g \mid \bigwedge_{f \in E^f} f)$$

By taking the negative logarithm of this we get the formulation for the value function as used in A* search, namely

$$\text{f-value}(E) = \text{cost}(E) + h(E)$$

with:

$$\text{cost}(E) = -\log\left(\prod_{f \in E^f} P(f)\right)$$

$$h(E) = -\log H(E) \approx -\log P_x(\bigwedge_{g \in E^g} g \mid \bigwedge_{f \in E^f} f)$$

To have an admissible heuristic, $h(E)$ should never overestimate

$$-\log P_x(\bigwedge_{g \in E^g} g \mid \bigwedge_{f \in E^f} f)$$

or equivalently $H(E)$ should never underestimate

$$P_x(\bigwedge_{g \in E^g} g \mid \bigwedge_{f \in E^f} f)$$

If A* uses an admissible heuristic, it is guaranteed to find the best result, which is the proof with the highest probability. $H(E) = 1$ is an admissible heuristic, as it can never be an underestimation. If the heuristic is not admissible, then we do not have this guarantee, and in general will only be able to approximate the explanation probability. (Pearl, 1984) shows that if the overestimation of the heuristic is bounded

$$h(E) - \left[-\log P_x(\bigwedge_{g \in E^g} g \mid \bigwedge_{f \in E^f} f)\right] \leq \epsilon$$

then the difference between the result and the optimal solution for a query q is also bounded:

$$[-\log P'(E)] - [-\log P_x(q)] \leq \epsilon'$$

with $P_x(q)$ the explanation probability for a query q and $P'(E)$ the probability of the proof found by the search. Rewriting this from negative log probabilities to probabilities, for $\epsilon = e^{-\epsilon'}$, we have:

$$P'(E) \geq \epsilon P_x(q)$$

5.3 Heuristics

We now define several heuristics that can be used to guide the proving process for DeepProbLog. The first two heuristics are uninformed, i.e., they do not take into account the goals of the partial proof. The last heuristic does model the expected probability of the goals.

5.3.1 Constant Heuristic

We define the *constant heuristic* as $H_c(E) = c$. If $c = 1$, then the search becomes uniform-cost search (UCS), and it will always find the optimal solution, but it might traverse more nodes than a better informed heuristic. Uniform-cost search is equivalent to the k-best method as explored in previous work for approximate inference in ProbLog (Renkens et al., 2012; De Raedt, Kimmig, and Toivonen, 2007).

5.3.2 Geometric Mean Heuristic

We consider an improvement over the constant heuristic for a specific case that often occurs in the neural-symbolic setting, namely that when the neural networks are completely correct and confident, the success probability is equal to the explanation probability. This is the case when there is only one correct proof, and all other proofs have a 0 probability. This is the case in many classification tasks, for example in the case of the MNIST addition. There are 9 proofs for the query `addition`(, , `9`), but only the proof `digit`(, `4`) $\wedge$ `digit`(, `5`) is correct and should carry all the probability mass, while all other proofs have a probability of zero. The same reasoning holds for many other neural-symbolic tasks.

To define this heuristic, we re-write $P_x(E)$ in terms of the complete proof E'

$$P_x(E) = \prod_{f \in E^{f*}} P(f) = \sqrt[N]{\prod_{f \in E^{f*}} P(f)}^{N} = {\mu^*}^N$$

where E^{f*} is the set of facts in the complete proof, μ^* the geometric mean of their probabilities and $N = |E^{f*}|$. With $n = |E^f|$, we can write

$$P(E) = {\mu^*}^n {\mu^*}^{N-n}$$

Let us assume that the geometric mean μ^* of the probabilities of the facts in E^{f*} is close to the geometric mean $\mu = \sqrt[n]{\prod_{f \in E^f} P(f)}$ of the facts in E^f.

This assumption holds when $P(f) \approx \mu$ for all $f \in E^{f*}$, or, equivalently, when the variance of the log probabilities of all the facts in the proof E^f is small. The geometric mean is chosen over the arithmetic mean, as the probabilities of individual facts are not summed, but multiplied. Under this assumption, we define the *geometric mean heuristic* (GM) as $H_g(E) = \mu^{N-n}$. Note that the f-value for each node is $\mu^n \mu^{N-n} = \mu^N$. If we assume N is the same for all proofs, we do not have to know N. We can replace the f-value by μ as x^N is monotonically increasing in $[0, 1]$.

5.3.3 Parametric Heuristics

The goal of the parametric heuristic is to predict the probability of certain goals before they are proven. With the right predictions, the proving will be guided more efficiently towards the correct proofs. This will not only speed up inference, but will also make training more efficient as fewer wrong proofs will be considered. This idea is similar to that of neurally-guided theorem proving (Wang et al., 2017; Rawson and Reger, 2019).

The parametric heuristic works on the assumption that the probability of a goal is independent of other goals and the probabilistic facts in the partial proof. This allows us to factorize the probability of the partial proof as

$$P_x(E) = \prod_{f \in E^f} P(f) \prod_{g \in E^g} P(g)$$

Under this assumption, we can also factorize the heuristic

$$H(E) = \prod_{g \in E^g} H(g)$$

This factorization allows for $P(g)$ to be estimated for every goal separately. If this assumption does not hold, then

$$\prod_{g \in E^g} P_x(g) < P_x(\bigwedge_{g \in E^g} g \mid \bigwedge_{f \in E^f} f)$$

and the heuristic will be an underestimation. The goals for which we should calculate such a heuristic is problem dependent, which is why the subset $\mathcal{R}$ of all the predicates in the program for which a heuristic should be estimated is specified up front. We define the heuristic as

$$H_n(r(i_1, ..., i_n)) = \begin{cases} f_r(i_1, ..., i_n) & \text{if } r \in \mathcal{R} \\ 0 & \text{otherwise} \end{cases}$$

The function f_r is an external implementation provided to the framework (e.g. a neural network).

5.4 Learning with approximate inference

Learning with approximate inference in the neural-symbolic setting holds an additional set of challenges. The main challenge is that DPLA* only uses the best proof, but the selection of the best proof is based on the probabilities of facts and neural predicates in the program. At the start of learning, these probabilities are not guaranteed to be correct, so the danger is that the selected proof is not correct. This prevents DPLA* from correctly training the neural networks. We call this the *cold-start* issue. There are two ways to overcome this issue: curriculum learning and exploration.

5.4.1 Curriculum learning

One way to solve the cold-start issue is to use curriculum learning to make sure that the accuracy of the neural predicates is better than random. This implies that the correct proofs will be more likely to be selected, which allows the neural predicates to be trained. Due to the flexibility of the DeepProbLog framework, curriculum learning is a very natural setting. Consider the multi-digit MNIST addition experiment. Training on long sequences of digits (e.g.  $+$  $= 832$) is hard, as the number of proofs grows rapidly. This means that exact inference quickly becomes infeasible. Approximate inference without any pre-initialization would also struggle, as it is hard to determine the correct proof for the given sum. One solution is to incorporate a few examples of shorter sequences (e.g. $+$ $= 13$). Alternatively, one can provide a few labeled examples for the neural predicate, which equates to direct supervision for the neural networks that classifies the digits. Because of the flexibility of DeepProbLog as a programming language, no changes to the model have to be made to allow for this form of curriculum learning. For example, in the addition program, one could directly maximize the probability of a few `digit` facts, like `digit(``, 1)` or `digit(``, 6)`. While in general it is not hard to collect a few labeled examples, they are valuable to initialize the neural predicates to guide the approximate inference.

5.4.2 Exploration

The second way to solve the cold-start problem is to incorporate a form of exploration. When neural networks are initialized with a slight preference for one output, the proof containing that output will be selected more often. Due to the optimization, the neural network will be trained towards this output even more. If this proof is not a correct proof, DPLA* can get stuck in a local

optimum. Exploration can make sure that, regardless of the initialization, the proofs are sufficiently diverse so that the neural networks are able to identify the patterns only present in the correct proofs and are able to learn from these. To include exploration in the algorithm, we borrow an idea from reinforcement learning, namely the upper confidence bound (UCB) (Lai and Robbins, 1985), which has been used in techniques such as Monte Carlo tree search. UCB assumes rewards in the interval $[0, 1]$, so it is appropriate for probabilities. Here, the concept of a reward is analogous to the probability of a proof, as a higher probability proof is desirable (cf. a reward of 1). In UCB, actions are selected based on their observed mean reward and an additional exploration term

$$\sqrt{\frac{\log(p)}{2N_a}}$$

where p is an additional parameter controlling the amount of exploration, often taken to be t^{-4}, t is the total number of actions taken and N_a is the number of times an action a was selected. To encourage exploration in the proof selection, we augment $cost(E)$ with an additional UCB term for each ground neural AD :

$$\text{cost}(E) = -\log\left(\prod_{f \in E^f} P(f) + P_{\text{UCB}}(f) - P(f)P_{\text{UCB}}(f)\right)$$

where

$$P_{\text{UCB}}(f) = \begin{cases} min(1, \sqrt{\frac{2\log(N_r)}{N_r^j}}) & \text{if } f = r(i_1, ..., i_n) \text{ is a neural predicate} \\ 0 & \text{otherwise} \end{cases}$$

Here, N_r^j is the number of times output j for neural predicate r was selected during the proving throughout the training process and N_r is the sum of all such N_r^j. As we treat these quantities as probabilities, we combine them as a disjunction $(P(f) + P_{\text{UCB}} - P(f)P_{\text{UCB}})$ rather than the addition.

5.4.3 Training the Parametric Heuristic

The advantage of using a parametric heuristic is that its parameters can be optimized to fit the properties of a specific task. Training parametric heuristics is a common technique in neurally-guided theorem proving (Wang et al., 2017; Rawson and Reger, 2019), where the heuristic is pre-trained on data of known proofs. However, in this setting, the proofs found in the training procedure cannot be used as supervision, as these can be wrong or may not yield the correct probability due to the inaccuracy of neural predicates. To overcome this

issue, we use a different approach. In order to train the parametric heuristic in parallel with the neural predicates, we optimize in parallel an equivalent DeepProbLog program where all definitions of predicate $r(i_1, ..., i_n)$ are replaced by a neural predicate representing the heuristic. The idea here is that we train the heuristic as shortcuts in the proving procedure. While the heuristic needs to solve a harder, higher-level task, it does not have to be perfectly trained as it will be used only as a heuristic during the actual search and doesn't affect the probability of the selected proof.

Example 5.2 (Neural Heuristics)
Consider the following program. It represents a multi-instance setting of the MNIST addition example.

```
nn(m,[I],N,[0,...,9]) :: digit(I,N).
addition(I1,I2,R) :-
    digit(I1,N1), digit(I2,N2),
    R is N1+N2.
mil_addition(Bag,R) :-
    member([I1,I2],Bag),
    addition(I1,I2,R).
```

`mil_addition`([[, ], [, ]], 7) is true if any of the pairs in the outer list sums to 7. We could use a heuristic on the `addition` predicate that can estimate if the sum holds for a given pair in the bag. This will be able to guide the heuristic towards the pair in the bag that are most likely to have the correct sum. Training the neural heuristics is equivalent to optimizing the following DeepProbLog program, where the `addition/3` is replaced by the heuristic neural network evaluation `h_addition/3`.

```
nn(h,[I1,I2],R,[0,...,18]) :: h_addition(I1,I2,R).
mil_addition(Bag,R) :-
    member([I1,I2],Bag),
    h_addition(I1,I2,R).
```

5.5 Experiments

The goal of our experiments is to investigate whether DPLA* can be applied to tasks that are intractable for exact inference in DeepProbLog. We will compare the DPLA* with different heuristics to DeepProbLog with exact inference and other neural-symbolic frameworks. Our questions are grouped in three topics: approximate inference, approximate learning and parametric heuristics.

First, we will look at how approximate inference, the heuristics and neural predicate accuracy interact. We hypothesize that approximate inference will outperform exact inference, but that its performance depends on the heuristic and the neural predicate accuracy. We will verify that by answering the following questions.

Q1.1 How does approximate inference compare to exact inference and related frameworks in terms of speed?

Q1.2 How do the different heuristics compare to each other?

Q1.3 What is the impact of neural predicate accuracy on the inference speed of the approximate inference?

Performing learning with approximate inference has potential complications. The proving process is guided by the probabilities which are the output of the neural networks, which are themselves being trained. This can lead to issues where the correct proofs are not selected and the system converges to a local optimum. We answer the following questions.

Q2.1 What is the influence of approximate inference on the accuracy and convergence of the learning?

Q2.2 How does curriculum learning impact the performance of the different heuristics?

Q2.3 What is the impact of exploration on the learning process?

Finally, we will also investigate how a parametric heuristic influences inference and learning. We answer the following questions.

Q3.1 Can parametric heuristics be used to guide the proving process?

Q3.2 Can parametric heuristics be learned in conjunction with the neural predicates?

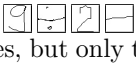
All programs are given in full in Appendix B.

5.5.1 Tasks

In order to answer these questions, we will use four datasets, each of them coupled with a different neural-symbolic task.

MNIST Addition The MNIST addition example was introduced in the original DeepProbLog paper (Manhaeve, Dumančić, et al., 2018). The task is to predict the sum of two integer numbers represented as two sequences of MNIST images, e.g. $[\boxed{3}, \boxed{4}] + [\boxed{1}, \boxed{5}] = 49$. During training, we are not provided with any direct supervision for the images but, only the sum of the numbers they represent. The dataset is composed of 60000 images and $\frac{60000}{2N}$ pairs. Each pair contains two equally long sequences of length N. For this task, N varies from 1 to 5.

Multi-Instance Learning MNIST Addition Here, we introduce a new task that is an extension of the *MNIST Addition* task. The difference is that it is extended to a multi-instance learning task. Several pairs are combined in a bag, and the bag is labeled with a sum randomly selected out of all the pairs in the bag. Example: $(\boxed{3} + \boxed{4} = 7) \vee (\boxed{1} + \boxed{5} = 7)$. The goal of this task is being able to train the neural predicates to recognize MNIST digits using the weak training signal provided by the multi-instance learning setting. This task is interesting as it investigates the multi-instance learning setting, but also because large parts of the proof space are irrelevant, as only a few items from each bag are interesting. A well-informed proving algorithm should be able to leverage this sparsity and avoid unnecessary work on proving and neural network evaluation.

Handwritten Formulas (HWF) The Handwritten Formula (HWF) dataset was introduced in (Li et al., 2020). The task is to predict the result of an expression represented as a sequence of images. Each image represents either a digit or an operator$(+, -, \times, \div)$. An example of an expression is: . As for the addition, no supervision is provided on the single images, but only the result of the expression is provided. We do not consider the curriculum learning setting for this task, i.e., we train and test only on expressions of exactly length N.

CLUTRR The CLUTRR dataset was proposed in (Sinha et al., 2019). Each example is a natural language sentence describing a kinship graph of variable size. The goal is to deduce a family relation between two nodes of the graph. The query relation is not directly mentioned in the text but can be deduced by chaining relations mentioned in the text. The setting we consider here is slightly different from the original. Instead, we consider the setting as used in (Manhaeve, Dumančić, et al., 2021), where the rules describing family relations are specified as background knowledge, and the task is to train neural networks to extract the relations from the natural language sentences.

5.5.2 Baselines

We compare the proposed approach with two neural-symbolic systems.

NeurASP: NeurASP (Z. Yang et al., 2020) inherits neural predicates from DeepProbLog, but they are used to extend Answer Set Programming instead of ProbLog. While it also uses exact inference in probabilistic logic programming, NeurASP showed to be a more efficient implementation than the original DeepProbLog paper for some tasks.

NGS: Neural-Grammar-Symbolic (NGS) (Li et al., 2020) combines grammar parsing with neural networks at the level of terminal symbols, which resembles the interface layer implemented by DeepProbLog using neural predicates. However, the solution to the neural-symbolic integration is quite different. Instead of relying on a probabilistic semantics of the logic, NGS provides a backward (i.e. correction) module for propagating the error through the symbolic grammar. By being tailored to the specific task, the backward module can be very efficient. This method is less general than DeepProbLog, where the correction is automatically computed given any possible logic program.

5.5.3 Results

To concisely describe the results, we will use the following abbreviated terms.

- *Query time*: The time it takes for a method to perform the proving for a single query. For exact methods, this involves all proofs, and for DPLA*, this involves a single proof (unless otherwise specified).
- *Neural predicate accuracy*: The accuracy of the neural predicate measured on the relevant dataset. For tasks based on MNIST images, this is the test set, unless specified otherwise.

For all graphs, a line represents the median, and the shaded area spans between the first and third quartiles. For all tables, we report the mean and standard deviation. All experiments were run 5 times.

Q1: Scaling Inference

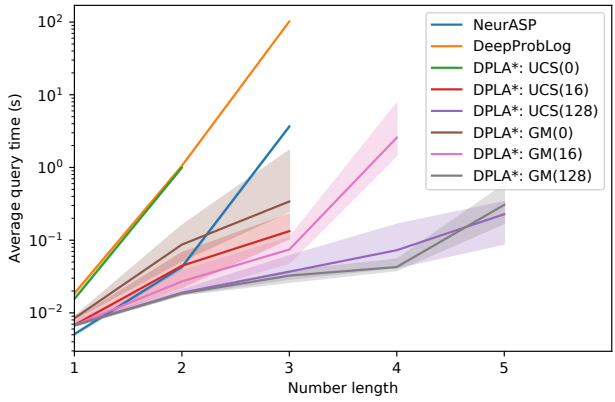
For this first question, we are only interested in the scalability of inference, and we do not consider the learning aspect. The aim of this experiment is to show how approximate inference is capable of answering queries faster than

exact inference, and thus can scale to larger examples. To answer questions **Q1.1** and **Q1.2**, we evaluate DPLA* for both the uniform cost search and geometric mean heuristics, DeepProbLog with exact inference and NeurASP. To investigate the effect of the accuracy of the neural predicates, we use varying numbers of examples to pre-train the neural predicates. As the pre-training trains the neural predicates on an easier task first, it is a form of curriculum learning.

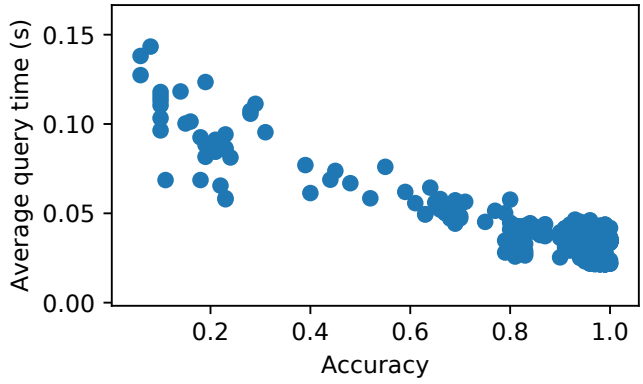
The results are shown in Figure 5.1a. When we compare the methods that use exact inference: NeurASP and DeepProbLog, we see that the average time to answer a query grows rapidly and quickly becomes intractable. This is expected, as these methods consider all proofs for a query, which grows rapidly with the length of the numbers. We can also see that NeurASP is slightly faster than DeepProbLog.

However, DPLA* with both Uniform-cost Search (UCS) and the Geometric Mean (GM) approximate inference with DeepProbLog can answer the queries several order of magnitude faster than the exact methods. The geometric mean heuristic is also considerably faster than the uniform-cost search when the neural predicate accuracy is low, but the difference grows smaller as this accuracy increases. For this reason, *we will use the geometric mean heuristic in the remainder of the DPLA* experiments.* To answer question **Q1.3**, we plot the query time with respect to the accuracy during training. In Figure 5.1b, we see that the speed of inference for DPLA* depends on the accuracy of the neural predicates. As the accuracy goes up, the time it takes to find a proof in DPLA* decreases.

To conclude, DPLA* is able to prove queries orders of magnitude faster than exact methods. The geometric mean heuristic outperforms uniform cost search if the neural predicate accuracy is low, but the difference decreases as the accuracy improves. The speed of DPLA* improves as the neural predicate accuracy increases.



(a) **Q1.1, Q1.2:** The average query time for different methods on the MNIST addition set. The number between brackets is the number of examples used to pre-train the neural predicates. DPLA* outperforms DeepProbLog and NeurASP, and can be applied to longer numbers.



(b) **Q1.3:** The average query time on the MNIST addition task (N=2) for DPLA* with the geometric mean heuristic, with respect to the neural predicate accuracy. As accuracy increases, the average query time decreases.

Figure 5.1: **Q1:** Time to answer queries on the MNIST addition task.

Q2: Scaling Learning

To answer this question, we compare DPLA*, NeurASP and NGS on the MNIST addition, HWF and CLUTRR datasets. **Q2.1** investigates how approximate inference impacts the learning in terms of accuracy, speed and convergence.

We first compare DeepProbLog, NeurASP and DPLA* on MNIST addition for $N = 1..3$ for uninitialized neural networks and neural-networks pre-trained with limited amounts of direct supervision. As can be seen in Figure 5.2, DPLA* performs only slightly worse than DeepProbLog with exact inference. Without pre-training, DPLA* converges slower than exact inference. Due to the wider spread and lower mean, we can also see that the training is less stable and does not converge reliably to the correct solution.

In Table 5.1, we compare the accuracy of the different methods on the test set of the MNIST addition. We can see that the exact methods reach the highest accuracy, but can only scale to $N = 2$. DPLA* with exploration attains a slightly lower performance than the exact methods. Note, however, that DPLA* with exploration does not scale to $N = 3$, since it considers too many proofs at the start of training. DPLA* achieves the lowest accuracy of all considered methods, but it is the only one that scales to $N = 3$. DPLA* without exploration, but pre-trained using 16 MNIST digits, performs similar to DPLA* with exploration, but it can also scale to $N = 3$, where it performs significantly better.

We further explore the same question using the HWF task. We report the accuracy on the test set, averaged over five runs.

In Table 5.2, we see the performance of DeepProbLog, DPLA* and NGS on handwritten formulas of varying length. Exact inference in DeepProbLog can handle only expressions of length up to 3. Using approximate inference with the geometric mean heuristic allows DeepProbLog to scale to all the expressions in the dataset. We also note that DPLA* is able to learn without curriculum learning. The performance of both DeepProbLog and DPLA* for expressions of lengths up to 5 is close to the state-of-the-art NGS, which is quite interesting since DPLA* uses a task-agnostic approach. However, we have found a sharp decrease in mean accuracy for expressions of length 7 for NGS, as for some initializations NGS fails to converge.

For the CLUTRR dataset we used the same setting as in (Manhaeve, Dumančić, et al., 2021) but with forward inference (implemented on a meta-level). Due to the cyclical nature of the rules, forward inference is more efficient than backward reasoning. The results in Figure 5.3 show that DPLA* is able to achieve good performance on an NLP setting as well, and is comparable to that of exact inference as shown in (Manhaeve, Dumančić, et al., 2021).

In question **Q2.2**, we analyze the convergence on the MNIST addition task. The results are shown in Figure 5.2. Without pre-training, DPLA* converges noticeably slower than exact inference. However, by pre-training the neural predicates with a small amount of images (only 16), approximate proving

converges considerably faster. When pre-training the neural predicates, DPLA* shows the same convergence trend and the same performances as the exact algorithm with the same pre-training, with the latter taking a lot more time to answer each query, cf. Figure 5.1a.

For question **Q2.3**, we investigate whether exploration mitigates the drop in performance in absence of pre-training. We run the experiment with exploration for the neural predicates. The result can be seen in Figure 5.2. Here we can see that, when using exploration, the spread in the accuracy with and without pre-training is small, indicating that the convergence is reliable. This is not the case without exploration, indicating that for some runs, the learning does not converge on the correct solution. We can also see that exploration slows convergence.

To conclude, learning with approximate inference has a negative impact on learning in absence of pre-training or exploration. Convergence is slower and the final accuracy is also lower, as some runs do not converge onto a good solution. Using curriculum learning to pre-train the neural predicate makes the method converge as fast as exact inference and with the same accuracy. Moreover, the use of exploration on the neural predicates allows all the runs to converge on a good solution, but it converges slower than without exploration.

	N		
Methods	1	2	3
DeepProbLog	97.2 ± 0.5	95.2 ± 1.7	–
NeurASP	97.3 ± 0.3	93.9 ± 0.7	–
DPLA*	88.9 ± 14.8	83.6 ± 23.7	77.4 ± 33.3
DPLA* w/ pre-training (16)	95.8 ± 0.4	93.7 ± 0.2	88.9 ± 1.6
DPLA* w/ exploration	96.7 ± 0.2	93.4 ± 0.7	–

Table 5.1: **Q2** The accuracy on the MNIST addition test set for N=1..3 when trained on datasets of the same length. Although DPLA* can scale to longer numbers, its accuracy is lower, unless pre-training or exploration is used.

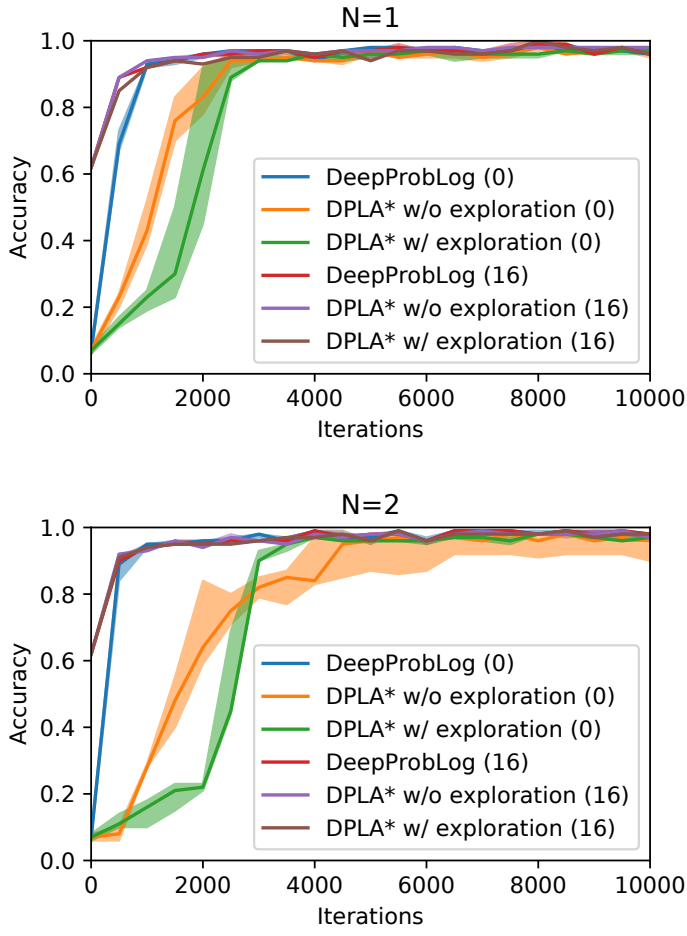


Figure 5.2: **Q2**: The neural predicate accuracy during training curves for DeepProbLog and DPLA* on the MNIST addition task, with and without pre-training and exploration. The number between brackets is the number of examples used to pre-train the neural predicates. DPLA* converges slower and to a lower accuracy than DeepProbLog, unless pre-training or exploration is used.

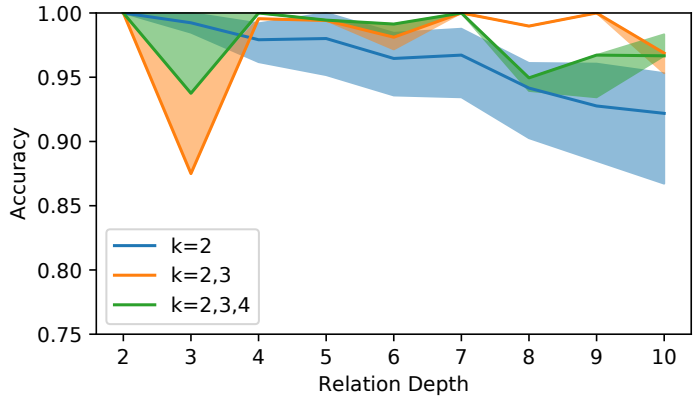


Figure 5.3: **Q2**: The results on the systematic generalization of the CLUTRR dataset. k indicates the relation lengths present in the training data. Except for $k = 2$, DPLA* is relatively unaffected by the relation depth.

	Expression length			
Method	1	3	5	7
NGS	90.2 ± 1.6	85.7 ± 1.0	91.7 ± 1.3	20.4 ± 37.2
DeepProbLog	90.8 ± 1.3	85.6 ± 1.1	–	–
DPLA*	90.9 ± 0.2	81.1 ± 2.4	89.0 ± 0.8	94.8 ± 0.5

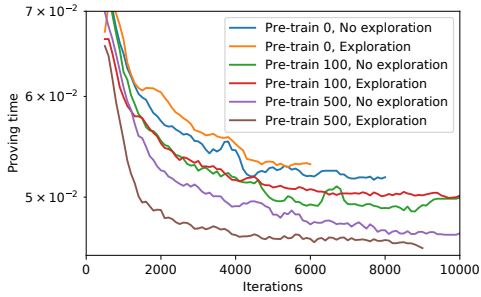
Table 5.2: **Q2.1** Accuracy on the HWF dataset for expressions of length 1 to 7. Comparison of the proposed approach (DPLA*) with exact DeepProbLog inference and NGS. DPLA* and NGS perform similarly, but NGS fails to converge for some runes for expressions of length 7.

Q3: Parametric Heuristic

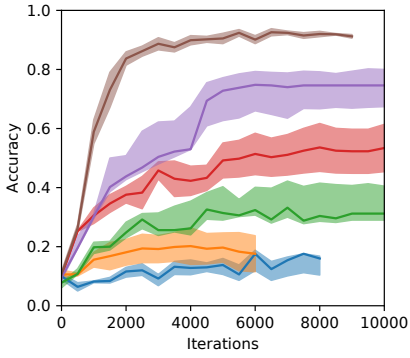
To answer this question, we consider the task of multi-instance MNIST addition. In this task, we investigate the use of defining a heuristic over the `addition/3` goal. The idea is that by being able to estimate the probability of this goal, the correct bags are more likely to be selected and the training is more efficient. We pre-train the heuristic with varying numbers of examples of additions. We also continue training of the heuristic throughout the training of the neural predicates.

Figure 5.4 shows the results for the multi-instance MNIST addition problem on bag size 16. We see that training of the neural predicate is only effective when the proving is guided by the heuristic. A more informed heuristic leads to more efficient training, and a higher final accuracy. We also see that a more informed and accurate heuristic leads to more stable convergence and a higher final accuracy. We can also see that we are able to further fine-tune the heuristic during training, and that exploration has a positive effect on learning the heuristic as well. Interestingly, although exploration usually makes a method slower, here, it makes the method faster instead as the heuristic accuracy is higher, which results in faster inference.

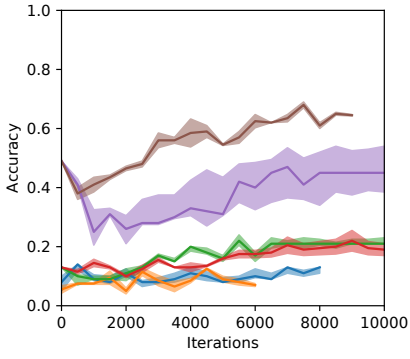
To conclude, we show that parametric heuristic is effective at guiding the proving process when it is sufficiently accurate, reaching a higher accuracy and making the proving process faster. We are also able to fine-tune the heuristic throughout the training process.



(a) Moving average (window size 5) of the proving time for different methods.



(b) Neural predicate accuracy on the first 100 images of the MNIST dataset.



(c) Heuristic accuracy measured on the MNIST addition test dataset.

Figure 5.4: **Q3:** The MIL Addition task with bag size 16 using the parametric heuristic. Pre-training the heuristic and exploration are both necessary to solve this problem well.

5.6 Conclusion

In this chapter, we introduced an approximate inference technique for DeepProbLog called DPLA*. It extends an approximate inference algorithm from the field of statistical-relational AI to the neural-symbolic setting. Instead of considering all possible proofs for a certain query, the system searches for the best proof using an A*-like search. For this A*-like search, we considered several heuristics, including a parametric heuristic that can be trained on additional data to guide the proving process better. We showed on 4 datasets that DPLA* is more scalable than exact inference methods such as DeepProbLog and NeurASP, and can be applied on larger tasks. We also address the convergence issue that arises when the system learns using approximate inference using curriculum learning and UCB-based exploration.

Chapter 6

Related work

In this chapter, we compare DeepProbLog with other neural-symbolic methods. We show a categorization based on the exact mechanism of integration.

6.1 Neural-symbolic programming languages

In this type of integration, an existing (logic) programming language is either modified or extended so that it can leverage the capabilities of a neural network. The exact mechanism of integration still differ. We consider different three types of integrations: differential interpreters, abductive reasoning and interfaces.

6.1.1 Differentiable interpreters

In this type of integration, an interpreter for a programming language is implemented by a neural network, so that the program can be executed in a fully differentiable way. Generally, this is done for low-level programming languages so that the neural architecture remains simple. For every expression in the language, a differentiable counterpart has to be created. The traditional memory is replaced by a differentiable, distributed memory, cf. the work on the differentiable neural computer (Graves et al., 2016). Neural networks are generally integrated as submodules, where their inputs and outputs are part of the (embedded) machine state. The output of the program can be optimized in an end-to-end fashion, so it can simultaneously learn the behavior of the program and train the parameters of the neural network.

A differentiable interpreter has several issues. One downside is that even for simple languages, such an interpreter can be quite complex to implement. Another problem is that since low-level languages are used, writing useful programs is not straightforward. One of the biggest issues, however, is that these approaches are sensitive to the vanishing / exploding gradient issue. As each line of code is performed sequentially by the neural network, there are many operations between input and output. This is further complicated by the fact that low-level languages usually require more lines of code to perform simple tasks.

Example 6.1 (Differentiable Forth)
One example of such an integration is Differentiable Forth (Bošnjak et al., 2017). Forth is a low-level, stack-based programming language. It allows for the behavior of parts of the program to be learned by neural network modules. For example, learning Bubblesort happens as follows (in pseudocode rather than in Forth for simplicity).

```
function sort(l):
    for i in range(len(l)):
        for j in range(len(l)-i-2):
            {OBSERVE l[i] l[i+1] -> CHOOSE swap nop}
```

The last line defines a small neural network that takes two elements of the list as input, and then decides to either swap these elements or not. An experimental comparison between DeepProbLog and this method is made in Section 4.5.2.

6.1.2 Abductive reasoning

Abductive reasoning is a type of reasoning where one tries to find the most likely explanation given a certain set of observations. There are several neural-symbolic methods that use this type of reasoning to abduce the most likely hypothesis, given a certain set of examples and some background knowledge. The hypothesis usually includes the (desired) output of the neural network, and a logical theory that represents missing background knowledge. These systems can thus infer (part of) the logic theory. Generally, these systems make a single choice for the output of the neural network, called a pseudo-label. If no satisfying theory can be abduced using the choices, at least one of the pseudo-labels is wrong. New candidates are selected based on a specific search procedure, and the new pseudo-labels are used to retrain the neural networks.

The benefit of such an approach is that the logic remains simple, as only one pseudo-label is selected. The combinatorial explosion of trying all alternatives together is avoided. The logic also does not need to be made differentiable. The issue is that if the search is not efficient, reasoning could take a long time. It is also possible that the wrong pseudo-labels are selected, and the neural networks receive incorrect updates. This can lead to convergence issues. Examples include ABL (Dai et al., 2019) and NeuroLog (Tsamoura et al., 2021). An experimental comparison with ABL is made in Section 5.4. If DPLA* only uses one proof, it becomes similar to an abductive reasoning system as it (approximately) searches for the most likely, consistent hypothesis and neural network classifications.

Example 6.2 (ABL)
In ABL (Dai et al., 2019) there are three components that function in an alternating fashion: a perception model, a consistency checking component and an abductive reasoning component. Take for example the task where there are handwritten digits such that the last is the sum of the first two (e.g. ). The structure of the expression is given as background knowledge, but the exact operation (addition) needs to be abduced. First, the perception model classifies the images into pseudo-labels, using the most likely prediction. The abductive reasoning component then tries to abduce a logically consistent hypothesis. For example, if the digits are correctly classified as 3, 5 and 8, the only logically consistent hypothesis is that the operation is an addition. If this is not possible, there is an error in the pseudo-labels. A heuristic function is then used to determine which pseudo-labels are wrong. The reasoning module then searches for logically consistent pseudo-labels, which are used to retrain the perception model.

6.1.3 Interface

In this type of integration, an interface is created between the programming language and the neural network. This is done by adding constructs to the language that represent neural network calls as first-class citizens. This means that both the input and the output are represented in the programming language, but the computation in between is considered as black box. To make a system like this differentiable, its computational mechanisms also need to be differentiable. Currently, this type of integration is only seen in fuzzy / probabilistic logic programming languages, which can make the score / probability of the query a differentiable target to be optimized. To optimize the parameters of the neural network, it is essential that the interface also allows for gradients to pass from the logic to the neural network. DeepProbLog accomplished this interface by introducing the neural predicate. Other frameworks have borrowed the concept of the neural predicate (Winters et al., 2022; Z. Yang et al., 2020).

Example 6.3 (NeurASP)
With NeurASP (Z. Yang et al., 2020), the authors show that the idea of the neural predicate can also be applied to answer set programming. The neural predicate in NeurASP takes the form of a *neural atom*, which is defined in (Z. Yang et al., 2020) as an expression of the form:

$$nn(m(e,t),[v_1,\ldots,v_n])$$

where nn is a reserved functor, m uniquely identifies a neural network model, e is a natural number representing the number of random events and t represents an input tensor. The model m then predicts e probability distributions over the domain $v_1,\ldots,v_n$. For example, the same MNIST classifier as defined in Example 4.4 would be defined as:

$$nn(m(1,\boxed{}),[0,\ldots,9])$$

This type of integration has the benefit that both base paradigms are preserved, as it is an extension. No parts of the base language are lost. It is also less sensitive to the convergence issues of the abductive reasoning methods, as all options are considered jointly, which makes it more likely that the neural network is trained in the direction of the correct label. The downside of such an approach is that reasoning jointly about all the possible outputs of the neural networks can be very expensive. These methods can generally only be applied to small problems. To apply them to larger problems, approximate inference has to be used, cf. Chapter 5.

6.2 Regularization

Some methods allow the user to specify a set of logic rules that the neural method should adhere to. In general, these rules act as soft-constraints, as they are not strictly enforced. Rather, they are including during the training phase as a type of regularization. During evaluation, the logic is assumed to be present in a distributed way in the parameters. There are two main types of such methods: regularization applied to embeddings and neural networks, respectively.

6.2.1 Regularizing embeddings

These approaches allow for logical rules to be injected into the embeddings of entities and relations. These logical rules represent background knowledge

about the entities and relations that can be used to complete sparse data about these entities and relations. For example, a rule $\forall x, y : fatherOf(x, y) \Rightarrow parentOf(x, y)$ could be used to help learn embeddings for the $parentOf$ relation, even though only examples of $fatherOf$ are observed in the data. Rocktäschel, Singh, et al. (2015) inject background knowledge into a matrix factorization model for relation extraction, by adding differentiable loss terms for propositionalized first-order rules. Demeester et al. (2016) propose a more efficient alternative by inducing order relations in embedding space, effectively leading to a lifted application of the rules. This is further generalized by Minervini, Demeester, et al. (2017), who investigate injecting rules by minimizing an inconsistency loss on adversarially generated examples.

6.2.2 Regularizing neural networks

These methods apply the same mechanism, but on the output of neural networks instead. First-order logic is used to specify constraints on the output that the neural network should adhere to. This enforces the neural network to make logically consistent predictions and can help with generalization. The logic is thus assumed to be learned into the weights of the neural network. During evaluation, however, the logic is no longer present. Diligenti et al. (2017) use first-order logic to specify constraints on the output of the neural network. They use fuzzy logic to create a differentiable way of measuring how much the output of the neural networks violates these constraints. This term is then added as an additional loss term that acts as a regularizer.

J. Xu et al. (2018) introduce a similar method that uses probabilistic logic instead of fuzzy logic, and is thus more similar to DeepProbLog. They also compile the formulas to an SDD for efficiency. One example from this work made a neural network predict the shortest path between two nodes given an input graph. The neural network trained with the semantic loss (enforcing it to output a valid path) had a higher accuracy, and predicted far fewer disconnected paths.

Logic Tensor Networks (LTNs) (Badreddine et al., 2021) introduce the concept of real logic, where the concepts of first order logic are mapped onto real-valued counterparts (e.g. constants are mapped to vectors, predicates are mapped onto functions, ...). This approach allows for logical formulas to be defined as constraints on the output of neural networks. In this work, the authors show that they can tackle the MNIST addition experiments, achieving similar accuracy to DeepProbLog, but with more efficient inference. They also show that they are able to learn embeddings, perform clustering and perform regression

6.3 Templating approaches

This line of work uses logic as a template for constructing the architecture of neural networks. This is reminiscent of the knowledge base model construction approaches of statistical relational artificial intelligence (De Raedt, Kersting, et al., 2016).

One of the first methods using this approach was KBANN (Towell and Shavlik, 1994). Here, logic programs are turned into neural network architectures. Additional links are introduced that represent unknown rules, and the weights of the neural network are learned through standard gradient descent techniques. Later work (Garcez, Lamb, and Gabbay, 2007) turned modal logic programs into ensembles of neural networks.

More recently, Rocktäschel and Riedel (2017) introduce a differentiable framework for theorem proving called the Neural Theorem Prover (NTP). In the NTP, two entities can be unified if they are similar, and not just if they are identical. As such, this system can interweave both symbols and sub-symbols during inference. For each symbol S, there is a learnable sub-symbol T_S. Soft-unification happens by applying the normal unification procedure where possible. However, if two symbols S_1 and S_2 can not be unified, the comparison is assigned a score based on the similarity between T_{S_1} and T_{S_2}. The similarity is calculated using a radial basis function $\varphi(||x - y||_2)$. Weber et al. (2019) extend the notion of soft unification towards structured textual knowledge, i.e., unification can be performed between sentences, not only symbols.

One problem with soft-unification is that since unification is now less strict, the number of proofs for a certain query can explode. This means that the original Neural Theorem Prover was quite limited in its scope. Since then, there have been several extensions of this original idea. One approach was to not consider all possible unifications, but only the most similar, which is implemented efficiently using an approximate nearest neighbor search (Minervini, Bosnjak, et al., 2018). Another is the Conditional Theorem Prover (Minervini, Riedel, et al., 2020), where the set of rules that can be applied is conditioned on the goal that is being solved, and is learned by a neural network. It can thus learn an efficient search strategy, where not all rules need to be applied on each inference step.

Methods such as lifted relational neural networks (Šourek et al., 2018) treat symbolic rules expressed in first-order logic as a template for constructing a neural network. Cohen et al. (2018) introduce a framework to compile a tractable subset of logic programs into differentiable functions. An idea similar in spirit to ours is that of Andreas et al. (2016), who introduce a neural network for visual question answering composed out of smaller modules responsible for individual tasks, such as object detection.

Chapter 7

Conclusion

In this thesis, we set out to create a neural-symbolic system that has two desirable properties: 1) it should contain a neural, logical and probabilistic aspect, 2) it should preserve all these aspects as special cases. To achieve this goal, we answered the following three research questions:

1. What are the similarities and differences between NeSy and StarAI? Do these similarities allow for the sharing of ideas between the two?
2. Is probabilistic logic programming a suitable framework to be integrated with neural networks? Does the inclusion of probabilistic reasoning allow for a straightforward integration? Does it open up additional possibilities as compared to neural-symbolic integrations without a probabilistic component?
3. Is the neural predicate as an interface between the neural and probabilistic logic programming sides powerful enough to create a widely applicable neural-symbolic system that is able to utilize the neural, logical and probabilistic components fully?

7.1 Results

We now summarize the results from the thesis and discuss how they answer these research questions.

NeSy vs StarAI In Chapter 3 we made the comparison between neural-symbolic AI and the well-established field of StarAI, which also focuses on integrating reasoning and learning. This comparison was made along 7 dimensions: 1) directed vs undirected, 2) model-based vs proof-based, 3) logic vs probability vs neural, 4) semantics, 5) structure vs parameter learning, 6) symbols vs sub-symbols and 7) types of logic. These dimensions also serve as a set of categories to classify neural-symbolic methods.

This comparison answered our first research question. *The two fields are similar and can be compared along a few key dimensions borrowed from the field of StarAI. We were also able to identify research directions that were investigated in StarAI, but have not been considered for NeSy.*

DeepProbLog In Chapter 4, we introduced a neural-symbolic framework called DeepProbLog. It integrates the probabilistic logic programming language ProbLog with neural networks through the novel concept of the neural predicate. It distinguishes itself from other NeSy methods as it features both probabilistic logic, a fully expressive logic programming language and neural networks.

We evaluated this framework on four sets of experiments that showcase DeepProbLog's ability to: 1) integrate logical reasoning and deep learning, 2) perform program induction through parameter learning, 3) integrate probabilistic reasoning and deep learning, and 4) manipulate embeddings and perform natural language reasoning. The results for these experiments show that DeepProbLog is able to outperform neural networks on tasks requiring reasoning, and is also able to outperform other neural-symbolic frameworks.

These results positively answered our second and third research questions. *Creating an interface between probabilistic logic programming and neural networks through the concept of the neural predicate is a good basis for creating a neural-symbolic system.* Further evidence for the value of the neural predicate is that other neural-symbolic systems have borrowed this idea and applied it to other StarAI frameworks.

Moreover, the experiments that combined probabilistic learning with deep learning showed that *the probabilistic reasoning component can bring additional capabilities*, such as easily creating noise-tolerant models.

DPLA* One of the main drawbacks of DeepProbLog is that the exact inference it uses does not scale well. Reasoning probabilistically over the entire output of the neural networks can quickly become prohibitively expensive. This is why, in Chapter 5, we introduced an approximate inference algorithm for DeepProbLog called DPLA*. DPLA* performs an efficient A*-based search for a small set of only the most likely proofs for a query. Considering a subset of all proofs then makes all subsequent steps of inference cheaper.

In DPLA*, the proving is guided by the probability of the proof so far, and a heuristic that estimates the probability of the remaining goals. We considered 3 heuristics: a constant heuristic that leads to a uniform-cost search, a heuristic that estimates the probability of the remaining goals as the geometric mean of the probabilities used so far, and a parametric heuristic. This parametric heuristic uses neural models that estimate the probability of the goals, and can be trained using data from the proving process itself.

However, combining approximate inference with learning can make convergence problematic. To solve this, we considered the curriculum learning setting. We also introduced an exploration term to the proving procedure based on the UCB algorithm to encourage a more diverse set of proofs to be selected. We evaluated DPLA* with a set of experiments that showcase that it scales better than DeepProbLog, but often requires curriculum learning and exploration to perform well. These results further strengthened the answers to our second and third research questions, as the approximate inference allowed DeepProbLog to be applied to a wider range of tasks.

Related work In Chapter 6, we took a wider look at the field of neural-symbolic AI. We classified a wide variety of neural-symbolic methods based on the exact mechanism of the integration they use. We discerned three large groups of methods: 1) neural-symbolic programming languages, 2) regularization-based methods for both embeddings and neural networks, and 3) templating approaches. The neural-symbolic programming languages were then further subdivided in how the languages were executed: through a) a differentiable interpreter, b) abductive reasoning or c) an interface between the programming language and the neural networks. DeepProbLog is an example of the latter.

7.2 Future work and Open Challenges

There are still several interesting directions for further research on DeepProbLog. In its current state, DeepProbLog is missing some features that are present in other neural-symbolic systems. Furthermore, there are some open challenges for the entire field of neural-symbolic AI, some of which we have taken the first step towards with DeepProbLog, and others that have not yet been addressed.

7.2.1 Structure learning

For DeepProbLog, we have assumed that the program (i.e. the structure) is given. We have shown some results on structure learning with the experiments on program induction (Section 4.5.2), but this is limited to the sketching approach. Other NeSy frameworks have further investigated structure learning, but they are generally limited to structure learning as parameter learning. In such approaches, the possible rules are enumerated, and the correct rules are selected by learning their parameters. However, NeSy has not yet caught up with related fields in this regard. Significant progress has been made on learning the structure of purely relational models (without probabilities). Structure learning of StarAI models remains a major challenge due to the complexity of inference and the combinatorial nature of the problem. Incorporating neural networks complicates this problem even more. NeSy methods have certainly shown potential for addressing this problem, but the existing methods are still limited and mostly domain-specific. For instance, the current systems that support structure learning require user effort to specify the clause templates or write a sketch of a model. An interesting direction to look into would be the field of inductive logic programming (Muggleton and De Raedt, 1994; De Raedt, 2008).

7.2.2 Tractable inference

In Chapter 5 we introduced DPLA* and showed how approximate inference can be used to apply DeepProbLog to a wider set of tasks. This is however only one form of approximate inference. Future research could work towards a stronger second approach that requires fewer assumptions and is thus applicable to a larger set of tasks. As described in Section 2.1, backward reasoning and forward reasoning have complementary strengths. For some tasks, mainly those that are very cyclical in nature, forward reasoning is more appropriate. A (possibly approximate) forward inference procedure would allow for DeepProbLog to be applied to a wider variety of tasks.

Training DeepProbLog models also involves training the neural networks that output the probability distributions used by the neural predicates. DeepProbLog uses standard neural network training methods based on backpropagation and gradient descent, which requires many iterations. Efficient evaluation and training of neural networks also relies heavily on batching and the parallelism of the GPU. Part of making DeepProbLog applicable to large-scale tasks is to match the implementation to the learning paradigm of neural networks as much as possible to prevent the models from being infeasibly slow to train.

The issue of tractable inference is a major challenge for StarAI and is further complicated for NeSy with the inclusion of a possible expensive neural network evaluations. Investigating to which extent neural methods can help with this challenge by means of lifted (exploiting symmetries in models) or approximate inference, as well as reasoning from the intermediate representations, (Abboud et al., 2020) are promising future research directions.

7.2.3 Undesired solutions

A problem that DeepProbLog faces is where the desired interpretation of the learned facts does not overlap with what is actually learned. For example, in experiment **T3** shown in Section 4.5, DeepProbLog learns the trivial solution $0 + 0 = 0$, which, although logically correct, is not a desired solution. This was solved by using additional regularization. In **T8** in Section 4.5, we investigate whether the desired interpretation (i.e., that the coins get correctly classified as heads and tails) can be reliably achieved using additional supervision.

Ideally, we would like to develop a more principled approach to solving these issues. There are two possible avenues to look into: regularizing the neural network, and reasoning about similarity between inputs. The current form of regularization works under the assumption that the ground-truth output classes for the neural predicates are uniformly distributed in the dataset. A stronger approach would not need such assumptions. In a second approach, DeepProbLog could reason over the similarity between the neural inputs. For example, if it is able to determine that two images are similar, it could constrain the predictions for the two images to be identical. Although it has not yet been described in other work, it is not unreasonable to assume that the same issue will be present in similar NeSy systems. Ideally, a solution to this problem can be derived that works for NeSy systems in general and across a wide variety of tasks.

7.2.4 Curriculum learning

As mentioned in Chapter 4, as DeepProbLog is a logic programming language, curriculum learning is a very natural setting. For example, in the MNIST multi-digit addition task, examples of any size, or even direct supervision on the neural network can be optimized jointly. A limited form of curriculum learning was also used in Chapter 5 to initialize the neural networks, which was essential to make learning converge.

However, the true potential of curriculum learning has been underused in DeepProbLog for now. The intended future work on structure learning, tractable inference and undesired solution could all benefit greatly from curriculum learning. It would thus be beneficial to look more into the exact effect of curriculum learning on convergence and how it can be best applied in a training strategy.

7.2.5 Applications

So far, we have applied DeepProbLog mostly on small-scale, proof-of-concept tasks. In fact, NeSy systems in general are not yet very mature from an application perspective, and there are no real showcase applications yet. This is due to the limitations mentioned above, mainly the issues with scalability and the lack of powerful structure learning. However, there have been promising proof-of-concepts in various fields.

First, knowledge-base completion is a natural application for NeSy methods, as the knowledge base is inherently symbolic and neural methods can be leveraged to generalize over the facts and predict those that are missing (Rocktäschel and Riedel, 2017; Donadello et al., 2017).

Second, NeSy has contributed to various computer vision tasks by incorporating background knowledge in, for instance, image segmentation and semantic image interpretation (Donadello et al., 2017; Alirezaie et al., 2019).

Lastly, NeSy methods were used to aid natural language tasks such as question answering and visual question answering (Weber et al., 2019; Yi et al., 2018), where the latter is also closely related to the computer vision domain. Some of the methods mentioned show some promising early results in the domain of inductive programming (Evans and Grefenstette, 2018; Si et al., 2019; Rocktäschel and Riedel, 2017), although they are still limited when compared to standard inductive logic programming systems. Developing real-life applications of NeSy is one of the most challenging and pressing open questions for the field.

Appendix A

Experimental details

The code and experimental data for both DeepProbLog and DPLA* is available in the following repository: `https://github.com/ML-KULeuven/deepproblog`. For the implementation we integrate the ProbLog2 framework (Dries et al., 2015) with PyTorch (Paszke et al., 2017). In all experiments, unless specified otherwise, we optimize the cross-entropy loss between the predicted and target query probabilities. We use Adam (Kingma and Ba, 2015) optimization for the neural networks, and SGD for the logic parameters. We performed a grid search to determine the ideal hyperparameters for each experiment.

A.1 Details for the experiments in the DeepProbLog chapter

In experiment **T9** we optimize the MSE between the predicted and target query probabilities as this proved to converge more reliably. This setting was previously used in Gutmann et al. (2008). For **T9**, we add random rotations (max 10 degrees) and shift the colors in the HSV by up to 5% to the images of the cards.
In **T11**, we use negative mining by proving the query for all possible relations and using the correct label as a positive example and all other labels as negative examples. This prevents the model to predict incorrect labels for irrelevant facts. We did not include the templates the authors collected using Amazon Mechanical Turk and have generated the same datasets as in the original paper, using the code provided by the authors.

Task	Batch size	Learning rate	Parameter learning rate	Regularization
T1-T3	2	1e-3		
T4	2	1e-3	1e-3	2, 4, 8
T5	50	0.02		
T6	16	1.0		
T7	100	0.005		
T8	5	1e-4		0.25
T9	50	1e-4	1e-3	0.5
T10	4	1e-3		
T11	2	1e-3		

Table A.1: Overview of the hyperparameters used in the experiments.

Task	Training set	Validation Set	Test set
T1	29 500, 3 000, 300	500	5 000
T2	14750, 1 500, 150	250	2 500
T3	16 000	2 000	3 000
T4	29 500	500	5 000
T5	512	256	1 024
T6	256	32	32
T7	300	100	200
T8	100	–	20
T9	500	–	25
T10	30 000	–	500
T11	1000	–	50

Table A.2: Overview of the sizes of the datasets used in the experiments.

The neural network architectures are summarized in Table A.3. *Conv(o,k)* denotes a convolutional layer with o output channels and kernel size k. *Lin(n)* denotes a fully connected layer of size n. *(Bi)GRU(h)* denotes a single-layer (bidirectional) GRU with a hidden size h. *(layer)*×2 means that there are two identical layers in parallel that are concatenated. A layer in bold means it is followed by a ReLU activation function. All neural networks end with a Softmax layer, unless otherwise specified. The hyperparameters used in the experiments are shown in Table A.1. The sizes of the datasets used are specified in Table A.2.

A.2 Details for the experiments in the DPLA* chapter

For all experiments, DPLA* used only one proof. The experiments on the MNIST addition and CLUTRR dataset used the same architecture and hyperparameters as specified in Appendix A.1. For the parametric heuristic used in the MIL MNIST addition, we used the same architecture as the baseline used for **T1** in the DeepProbLog experiments. For the HWF experiment, we used the same architecture as in Li et al. (2020). For this experiment, we used a learning rate $lr = 3e-3$ and a batch size of 50. The HWF dataset contains 10 000 training formulas and 2 000 test formulas. We split off 10% of the training dataset as the validation set, which we use to select the best model during training. It is important to note that this is different from the testing methodology from the original paper where the test set was used to perform the selection. We use a validation set as this gives a less biased result.

For the pre-training, a different neural network was pre-trained for each amount of fully supervised examples. They were trained using the Adam optimizer, with a learning rate of $1e-3$ and weight decay rate (L2 regularization) of $1e-2$ and a batch size of 4, for 40 epochs.

Task	Network	Architecture
T1-T4	digit/2	MNISTConv, **Lin(120)**, **Lin(84)**, Lin(10)
T1,T3	baseline	MNISTConv×2, **Lin(120)**, **Lin(84)**, Lin(19)
T2	baseline	(MNISTConv×2, **Lin(100)**)×2,**Lin(128)**, Lin(199)
T5	result/4	Lin(50), TanH, Lin(10)
	carry/4	Lin(10), TanH, Lin(2)
T6	swap/3	**Lin(20)**, Lin(10)
T7	RNN	Embedding(256), BiGRU(512), Dropout(0.5)*
	perm/2	Lin(6)
	op1/2	Lin(4)
	swap/2	Lin(2)
	op2/2	Lin(4)
T8	coin1/2	AlexNetConv, **Lin(100)**, Lin(2)
	coin2/2	AlexNetConv, **Lin(100)**, Lin(2)
T9	rank/2	AlexNetConv, **Lin(100)**, Lin(4)
T10	embed/2	Embed(2)
	cnn_embed/2	MNISTConv, **Lin(120)**, **Lin(84)**, Lin(2)*
T11	embed/2	Embed(16), GRU(16)
	nn_rel/2	Lin(11)
	nn_gender/2	Lin(2)

MNISTConv: Conv(6,5), **MP(2,2)**, Conv(16,5), **MP(2,2)***
AlexNetConv: **Conv(64, 11, 2,2)**, MP(3,2), **Conv(192, 5, 2)**, MP(3,2),
Conv(384, 3, 1), **Conv(256, 3, 1)**, **Conv(256, 3, 1)**, MP(3,2)*
* Does not end with a Softmax layer.

Table A.3: Overview of the neural network architectures used in the experiments.

Appendix B

DeepProbLog Programs

This chapter contains the full DeepProbLog as used in the experiments of Chapter 4 and Chapter 5.

B.1 Programs used in the DeepProbLog chapter

```
nn(m_digit,[X],Y,[0,...,9]) :: digit(X,Y).

addition(X,Y,Z) :- digit(X,X2), digit(Y,Y2), Z is X2+Y2.
```

Listing B.1: Single-digit MNIST addition (**T1**)

In Listing B.1, `digit/2` is the neural predicate that classifies an MNIST image into the integers 0 to 9. The `addition/3` predicate's first two arguments are MNIST digits, and the last is the sum. It classifies both images using `digit/2` and calculates the sum of the two results.

In Listing B.2, the only difference with Listing B.1 is that the `multi_addition/3` predicate now uses the `number/2` predicate instead of the `digit/2` predicate. The `number/3` predicate's first argument is a list of MNIST images. It uses the `digit/2` neural predicate on each image in the list, summing and multiplying by ten to calculate the number represented by the list of images (e.g. `number`([, ], 38)).

```
nn(m_digit,[X],Y,[0,...,9]) :: digit(X,Y).

number([],Result,Result).
number([H|T],Acc,Result) :-
    digit(H,Nr),
    Acc2 is Nr+10*Acc,
    number(T,Acc2,Result).
number(X,Y) :- number(X,0,Y).

multi_addition(X,Y,Z) :- number(X,X2), number(Y,Y2), Z is X2+Y2.
```

Listing B.2: Multi-digit MNIST addition (**T2**)

```
nn(m_digit,[X],Y,[0,...,9]) :: digit(X,Y).

addition(X,Y,Z) :- digit(X,X2), digit(Y,Y2), digit(Z,Z2), Z2 is X2+Y2.
```

Listing B.3: All-digit MNIST addition (**T3**)

In Listing B.3, the only difference with Listing B.1 is that all 3 inputs `X,Y,Z` are images. As such, the `digit/2` predicate is also used on the third input. The sum is also redefined as `Z2 is X2+Y2`.

```
nn(classifier, [X], Y, [0 ..9]) :: digit(X,Y).
t(0.2) :: noisy.

1/19 :: uniform(X,Y,0) ; ...; 1/19 :: uniform(X,Y,18).

addition(X,Y,Z) :- noisy, uniform(X,Y,Z).
addition(X,Y,Z) :- \+noisy, digit(X,N1), digit(Y,N2), Z is N1+N2.
```

Listing B.4: Noisy MNIST addition (**T4**)

In Listing B.4, an additional probabilistic fact (`noisy/2`) is added that encodes the chance of an example being noisy. The `addition/3` predicate is split into two cases: when the noisy is true or when noisy is false. The latter is the same as in Listing B.1. If `noisy` is true, Z is considered to be drawn from the uniform distribution (`uniform/3`).

In Listing B.5, there are two neural predicates: `result/4` and `carry/4`. These are used in the `hole/4` predicate that corresponds to the hole in the Forth program. The first three arguments are the two digits and the previous carry to be summed. The next two arguments are the new carry and the new resulting digit. The `add/5` predicate's arguments are: the two list of input digits, the

```
nn(m_result,[D1,D2,Carry],Y,[0,...,9])::result(D1,D2,Carry,Y).

nn(m_carry,[D1,D2,Carry],Y,[0,1])::carry(D1,D2,Carry,Y).

hole(I1,I2,Carry,NewCarry,Result) :-
    result(I1,I2,Carry,Result),
    carry(I1,I2,Carry,NewCarry).

add([],[],[C],C,[]).

add([H1|T1],[H2|T2],C,Carry,[Digit|Res]) :-
    add(T1,T2,C,NewCarry,Res),
    hole(H1,H2,NewCarry,Carry,Digit).

forth_addition(L1,L2,C,[Carry|Res]) :- add(L1,L2,C,Carry,Res).
```

Listing B.5: Forth addition sketch (**T5**)

input carry, the resulting carry and the resulting sum. It recursively calls itself to loop over both lists, calling the `hole/5` predicate on each position, using the carry from the previous step.

```
nn(m_swap, [X]) :: swap(X,Y,).

hole(X,Y,X,Y):- \+swap(X,Y).

hole(X,Y,Y,X):- swap(X,Y).

bubble([X],[],X).
bubble([H1,H2|T],[X1|T1],X):-
    hole(H1,H2,X1,X2),
    bubble([X2|T],T1,X).

bubblesort([],L,L).

bubblesort(L,L3,Sorted) :-
    bubble(L,L2,X),
    bubblesort(L2,[X|L3],Sorted).

forth_sort(L,L2) :- bubblesort(L,[],L2).
```

Listing B.6: Forth sorting sketch (**T6**)

In Listing B.6, there's a single neural predicate: `swap/3`. Its first two arguments are the numbers that are compared, the last argument is an indicator whether to swap or not. The `bubble/3` predicate performs a single step of bubble sort on its first argument using the `hole/4` predicate. The second argument is the

resulting list after the bubble step, but without its last element, which is the third argument. The `bubblesort/3` predicate uses the `bubble/3` predicate, and recursively calls itself on the remaining list, adding the last element on each step to the front of the sorted list.

```
permute(0,A,B,C,A,B,C).
permute(1,A,B,C,A,C,B).
permute(2,A,B,C,B,A,C).
permute(3,A,B,C,B,C,A).
permute(4,A,B,C,C,A,B).
permute(5,A,B,C,C,B,A).

swap(0,X,Y,X,Y).
swap(1,X,Y,Y,X).

operator(0,X,Y,Z) :- Z is X+Y.
operator(1,X,Y,Z) :- Z is X-Y.
operator(2,X,Y,Z) :- Z is X*Y.
operator(3,X,Y,Z) :- Y > 0, 0 =:= X mod Y,Z is X//Y.

nn(m_net1,[Repr],Y,[0,...,5])::net1(Repr,Y).
nn(m_net2,[Repr],Y,[0,...,3])::net2(Repr,Y).
nn(m_net3,[Repr],Y,[0,1])::net3(Repr,Y).
nn(m_net4,[Repr],Y,[0,...,3])::net4(Repr,Y).

wap(Text,X1,X2,X3,Out) :-
    net1(Text,Perm),
    permute(Perm,X1,X2,X3,N1,N2,N3),
    net2(Text,Op1),
    operator(Op1,N1,N2,Res1),
    net3(Text,Swap),
    swap(Swap,Res1,N3,X,Y),
    net4(Text,Op2),
    operator(Op2,X,Y,Out).
```

Listing B.7: Forth WAP sketch (**T7**)

In Listing B.7, there are four neural predicates: `net1/2` to `net4/2`. Their first argument is the input question, and the second argument are indicator variables for the choice of respectively: one of six permutations, one of 4 operations, swapping and one of 4 operations. These are implemented in the `permute/7`, `swap/5` and `operator/4` predicates. The `wap/5` predicate then sequences these steps to calculate the result.

In Listing B.8, there are two neural predicates: `coin1/2` and `coin2/2`. Their input is the image of the two coins (e.g. Figure 4.9). The output is heads or tails. The `coins/2` classifies both coins using these two predicates and then

```
nn(net1, [X], Y, [heads, tails]) :: coin1(X,Y).
nn(net2, [X], Y, [heads, tails]) :: coin2(X,Y).

compare(X,X,same).
compare(X,Y,different) :- \+compare(X,Y,same).

coins(X,Comparison) :-
    coin1(X,C1),
    coin2(X,C2),
    compare(C1,C2,Comparison).
```

Listing B.8: The coins experiment (**T8**)

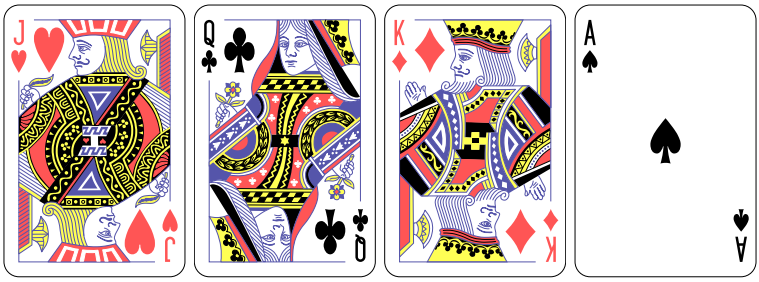


Figure B.1: Examples of cards without perturbations from the poker experiment (**T9**).

performs the comparison of the classes with the `compare/3` predicate.

In Listing B.9, there's a single neural predicate `rank/2` that takes as input the image of a card and classifies it as either a jack, queen, king or ace. There's also an AD with learnable parameters that represents the distribution of the unseen community card (`house_rank/1`). The `hand/2` predicate's first argument is a list of 3 cards. It unifies the output with any of the valid hands that these cards contain. The valid hands are: high card, pair (two cards have the same rank), three of a kind (three cards have the same rank), low straight (jack, queen king) and high straight(queen, king, ace). Each hand is assigned a rank with the `hand_rank/2` predicate. The `best_hand_rank/2` predicate takes as input a list of cards, and unifies the second argument with the highest hand rank that is possible with the three given cards. The `outcome/3` predicate determines the outcome by comparing the two ranks of the best hand. The `game/3` predicate's first argument is a list of the 4 input images. Its second input is the labeled community card. It classifies the cards using the neural predicates, determines the best rank, and then unifies the last argument with

the outcome. The `game/2` determines the community card from the learned distribution `house_rank/1`, and then determines the outcome using the `game/3` predicate. The `member/2` and `select/3` predicates are predicates from the *lists* library. `member/2` is true if its second argument is a list and the first argument appears in that list. `select/3` is true if its second argument is a list and the first argument appears in that list. It also unifies the last argument with the list that is the same as its second argument, but with the first argument removed.

For experiment **T11**, the main code is given in Listing B.11, and the background knowledge is shown in Listing B.12. In Listing B.11, the main predicate is the `query_rel(T,X,R,Y)` predicate that will predict the relationship R that holds between entities X and Y in text T. To simplify reasoning, all relations are reduced to a non-gendered version (e.g., significant other instead of husband). The `gender_rel/3` predicate provides information about all the possible relations, and ties the gendered version to its non-gendered version (e.g., `gender_rel(child,female,daughter)`. The gender is determined separately using the `neural_gender/3` predicate.

After this, a meta-call happens to try to prove the actual relation using the `call` predicate. There are two ways to prove the relations: either directly using the neural predicates, or through the logic that encodes the family relations. Also note that we explicitly encode the depth `D` when proving recursively to avoid cycles. Although DeepProbLog can deal with cycles, this was done to improve performance. Note also that we pre-process the text by splitting it into individual sentences, replacing the entity names with numbers, and keeping track which entities are mentioned in each sentence. Although this could be done inside the logic, we did this in a separate step to keep the logic more readable. In the `neural/4` predicate, we first filter out any sentences that do not mention the relevant entities we want to know the relation for. This is done using the `include/3` predicate. We then map every sentence onto its embedding by using the `maplist/3` predicate to loop over the list and the `embed/3` predicate to map the given sentence. The first argument is a list of entities that determine how the entities should be encoded. The first entity in this list will be encoded with the token ENT1, the second with ENT2, ... Any other entity will be tokenized as ENT. We take the element-wise max of all these embeddings using the `max_tensor/2` predicate. Then the actual prediction is made using the `nn_rel/2` neural predicate.

```
t(1/4)::house_rank(jack);t(1/4)::house_rank(queen);
    t(1/4)::house_rank(king);t(1/4)::house_rank(ace).
nn(net1,[X],Y,[jack,queen,king,ace]):: rank(X,Y).

hand(Cards,straight(low)) :-
    member(card(jack),Cards),
    member(card(queen),Cards),
    member(card(king),Cards).
hand(Cards,straight(high)) :-
    member(card(queen),Cards),
    member(card(king),Cards),
    member(card(ace),Cards).
hand([card(R), card(R), card(R)],threeofakind(R)).
hand(Cards,pair(R)) :-
    select(card(R),Cards,Cards2),
    member(card(R),Cards2).
hand(Cards,high(R)) :-
    member(card(R),Cards).

hand_rank(high(jack),0).
...
hand_rank(straight(high),13).

best_hand_rank(Cards,R) :-
    hand(Cards,H),
    hand_rank(H,R),
    \+(hand(Cards,H2),hand_rank(H2,R2),R2>R).

outcome(R1,R2,win) :- R1 > R2.
outcome(R1,R2,loss) :- R1 < R2.
outcome(R,R,draw).

cards(C1,C2,House,[card(R1), card(R2), House]) :-
    rank(C1,R1),
    rank(C2,R2).

game([C1,C2,C3,C4],House,Outcome) :-
    cards(C1,C2,House,Hand1),
    cards(C3,C4,House,Hand2),
    best_hand_rank(C1,R1),
    best_hand_rank(C2,R2),
    outcome(R1,R2,Outcome).

game(Cards,Outcome) :-
    house_rank(House),
    game(Cards,House,Outcome).
```

Listing B.9: The Poker experiment (**T9**)

```
successor(I1,I2,N) :-
    %Encode images I1 and I2 into E1 and E2
    cnn_encode(I1,E1), cnn_encode(I2,E2),
    %Embed the successor relation into embedding S
    embed(successor, S),
    %E = E1 + N*S
    mul(S,N,S2), add(E1,S2,E),
    rbf(E,E2).
```

Listing B.10: The MNIST successor task (**T10**)

```
:- use_module(library(apply)).

nn(net1,[T],R,[child,child_in_law,...]) :: nn_rel(T,R).
nn(net2,[T],G,[male,female]) :: nn_gender(T,G).

query_rel(T,X,R,Y):- gender_rel(R2,G,R),neural_gender(T,Y,G),call(R2,T,X,Y,3).

mentioned(X,s(E,_)) :- member(X,E).

neural_gender([Entities|Sentences], E, G) :-
    member(E,Entities),
    include(mentioned(E), Sentences, [H|Sentences2]),
    maplist(embed([E]),[H|Sentences2],Embedded),
    max_tensor(Embedded,Max),
    nn_gender(Max,G).

neural([Entities|Sentences], X,R,Y) :-
    member(X,Entities), member(Y,Entities), X \== Y,
    include(mentioned(X),Sentences,Sentences2),
    include(mentioned(Y),Sentences2,[H|Sentences3]),
    maplist(embed([Y,X]),[H|Sentences3],Embedded),
    max_tensor(Embedded,Max),
    nn_rel(Max,R).


child(T,X,Y,_) :- neural(T,X,child,Y).
child_in_law(T,X,Y,_) :- neural(T,X,child_in_law,Y).
grandchild(T,X,Y,_) :- neural(T,X,grandchild,Y).
...
```

Listing B.11: The CLUTRR experiment (**T11**)

```
grandchild(T,X,Y,D) :- D>0,X\=Y, child(T,X,Z,D-1), child(T,Z,Y,D-1).
grandchild(T,X,Y,D) :- D>0,X\=Y, so(T,X,Z,D-1), grandchild(T,Z,Y,D-1).
grandchild(T,X,Y,D) :- D>0,X\=Y, grandchild(T,X,Z,D-1), sibling(T,Z,Y,D-1).
grandparent(T,X,Y,D) :- D>0,X\=Y, parent(T,X,Z,D-1), parent(T,Z,Y,D-1).
grandparent(T,X,Y,D) :- D>0,X\=Y, sibling(T,X,Z,D-1), grandparent(T,Z,Y,D-1).
child(T,X,Y,D) :- D>0,X\=Y, child(T,X,Z,D-1), sibling(T,Z,Y,D-1).
child(T,X,Y,D) :- D>0,X\=Y, so(T,X,Z,D-1), child(T,Z,Y,D-1).
parent(T,X,Y,D) :- D>0,X\=Y, sibling(T,X,Z,D-1), parent(T,Z,Y,D-1).
parent(T,X,Y,D) :- D>0,X\=Y, child(T,X,Z,D-1), grandparent(T,Z,Y,D-1).
sibling(T,X,Y,D) :- D>0,X\=Y, child(T,X,Z,D-1), uncle(T,Z,Y,D-1).
sibling(T,X,Y,D) :- D>0,X\=Y, parent(T,X,Z,D-1), child(T,Z,Y,D-1).
sibling(T,X,Y,D) :- D>0,X\=Y, sibling(T,X,Z,D-1), sibling(T,Z,Y,D-1).
child_in_law(T,X,Y,D) :- D>0,X\=Y, child(T,X,Z,D-1),so(T,Z,Y,D-1).
parent_in_law(T,X,Y,D) :- D>0,X\=Y, so(T,X,Z,D-1), parent(T,Z,Y,D-1).
nephew(T,X,Y,D) :- D>0,X\=Y, sibling(T,X,Z,D-1), child(T,Z,Y,D-1).
uncle(T,X,Y,D) :- D>0,X\=Y, parent(T,X,Z,D-1), sibling(T,Z,Y,D-1)

gender_rel(child,male,son).
gender_rel(child,female,daughter).
gender_rel(parent,male,father).
gender_rel(parent,female,mother).
...
```

Listing B.12: The background knowledge for CLUTRR experiment (**T11**)

B.2 Programs used in the DPLA* Chapter

The programs for the MNIST addition experiment are the same as in the DeepProbLog chapter and are given in Listings B.1 and B.2.

```
nn(m,[I],N,[0,...,9]) :: digit(I,N).
addition(I1,I2,R) :-
    digit(I1,N1), digit(I2,N2),
    R is N1+N2.
mil_addition(Bag,R) :-
    member([I1,I2],Bag),
    addition(I1,I2,R).
```

Listing B.13: The multi-instance learning MNIST addition task

The program for the multi-instance learning MNIST addition task is given in Listing B.13. Here, the `addition/3` is the same predicate as in the normal MNIST addition tasks. The first argument of the `mil_addition/2` predicate is a list of pairs of MNIST images (the bag). The second argument is the result of the summation of (at least) one element of the bag.

In Listing B.14, the main predicate is `expression/2`. Its first argument is a list of images representing an expression, and the second argument is the result of that expression. It first calls `detect_all/2` to classify all images and then calculates the result with the `parse/2` predicate. The `detect_number/2` and `detect_operator/2` predicates detect the numbers and operators, respectively. The first argument of the `detect_all/2` predicate is a list of images, and the second argument is a list of the classifications of those images. It alternatingly calls `detect_number/2` and `detect_operator/2` to detect expressions of the form $n_1 \odot \cdots \odot n_j$. The `almost_equal/2` is true when the two arguments evaluate to almost the same number.

The main predicate in Listing B.15 is the `clutrr_text(Entities,Text,X,R,Y)` predicate. It predicts the relation `R` between entities `X` and `Y` with information from the sentences in `Text`. The `Entities` argument contains a list of all entities mentioned in the text. The first argument of the `get_edge/2` predicate is a sentence. The second argument is the relation mentioned in this sentence. It predicts this relation by first encoding the sentence with the `encode/4` predicate, and then classifying the embedding with the `extract_relation/2` predicate. Reasoning in this version of the program happens by performing forward inference on the meta-level, which is encoded in the `forward/3` predicate. This predicate function by repeatedly applying the `forward_step/3` until the goal

```
nn(net1,[X],Y,[0,1,2,3,4,5,6,7,8,9]) :: detect_number(X,Y).
nn(net2,[X],Y,[+,-,*,/]) :: detect_operator(X,Y).

detect_all([N],[N2]) :- detect_number(N,N2).
detect_all([N,O|T],[N2,O2|T2]) :- detect_number(N,N2),
    detect_operator(O,O2), detect_all(T,T2).

almost_equal(X,Y) :- ground(Y),abs(X-Y) < 0.0001.
almost_equal(X,Y) :- var(Y), Y is float(X).

expression(Images,Result) :- detect_all(Images,Symbols),parse
    (Symbols,Result).

parse([N],R) :- almost_equal(N,R).
parse([N1,+|T], R) :-
    parse(T,R2),
    almost_equal(N1+R2,R).
parse([N1,-|T], R) :-
    parse([-1,*|T],R2),
    almost_equal(N1+R2,R).
parse([N1,*,N2|T], R) :-
    N3 is N1*N2,
    parse([N3|T],R).
parse([N1,/,N2|T], R) :-
    N2 \== 0,
    N3 is N1/N2,
    parse([N3|T],R).
```

Listing B.14: The handwritten formulas task

is inferred. The `forward_step/3` step implements in the T_P operator. The background knowledge for this version of the program is given in Listing B.16.

```
nn(rel_extract, E, Relation,[child ...so]) :: extract_relation(E,
    Relation).
nn(encoder, [Text,Ent1,Ent2], Embeddings) :: encode(Text,Ent1,Ent2,
    Embeddings).
nn(gender_net,[T,Entity],G,[male,female]) :: gender(T,Entity,G).

clutrr_text(Entities, Text,X,R2,Y) :-
    gender_rel(R,G,R2),
    gender(Text,Y,G),
    maplist(get_edge,Text,Edges),
    clutrr_edges(Entities,Edges,X,R,Y).

get_edge(s(Entities,Text), Edge) :-
    select(E1,Entities,Entities2),
    member(E2,Entities2),
    encode(Text,E1,E2,Encoded),
    extract_relation(Encoded,Relation),
    Edge =.. [Relation,E1,E2].


clutrr_edges(Entities,Edges,X,R,Y) :-
    relation(R),
    Goal =.. [R,X,Y],
    rules(Rules),
    forward(Edges,Rules, Goal).

forward(Facts,Rules, Goal) :- member(Goal,Facts).
forward(Facts,Rules, Goal) :- \+member(Goal,Facts),forward_step(Facts,
    Rules,NextFacts),forward(NextFacts,Rules,Goal).

forward_step(Facts,Rules, AllFacts) :-
    findall(Head,(member(rule(Head, Body),Rules), maplist([X]>>member(X,
        Facts),Body), \+member(Head,Facts)), [New|NewFacts]),
    append([New|NewFacts],Facts,AllFacts).
```

Listing B.15: The forward inference for the CLUTRR task

```
rules([
    rule(grandchild(X,Y), [child(X,Z), child(Z,Y)]),
    rule(grandchild(X,Y), [so(X,Z), grandchild(Z,Y)]),
    ...,
    rule(uncle(X,Y), [parent(X,Z), sibling(Z,Y)])]).

relation(child).
...
relation(so).

gender_rel(child,male,son).
gender_rel(child,female,daughter).
...
gender_rel(so,male,husband).
gender_rel(so,female,wife).
```

Listing B.16: The background knowledge for the CLUTRR task with forward reasoning

Bibliography

Abboud, R., İ. İ. Ceylan, and T. Lukasiewicz (2020). "Learning to Reason: Leveraging Neural Networks for Approximate DNF Counting". In: *Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence, the Thirty-Second Innovative Applications of Artificial Intelligence Conference, the Tenth AAAI Symposium on Educational Advances in Artificial Intelligence*, pp. 3097–3104.

Alirezaie, M., M. Längkvist, M. Sioutis, and A. Loutfi (2019). "Semantic referee: A neural-symbolic framework for enhancing geospatial semantic segmentation". In: *Semantic Web* 10.5, pp. 863–880.

Andreas, J., M. Rohrbach, T. Darrell, and D. Klein (2016). "Neural module networks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 39–48.

Bach, S. H., M. Broecheler, B. Huang, and L. Getoor (2017). "Hinge-Loss Markov Random Fields and Probabilistic Soft Logic". In: *Journal of Machine Learning Research* 18, pp. 1–67.

Badreddine, S., A. d'Avila Garcez, L. Serafini, and M. Spranger (2021). "Logic Tensor Networks". In: *Artificial Intelligence*, p. 103649.

Bahdanau, D., S. Murty, M. Noukhovitch, T. H. Nguyen, H. de Vries, and A. C. Courville (2019). "Systematic Generalization: What Is Required and Can It Be Learned?" In: *Proceedings of the 7th International Conference on Learning Representations*.

Bekkum, M. van, M. de Boer, F. van Harmelen, A. Meyer-Vitali, and A. ten Teije (2021). "Modular design patterns for hybrid learning and reasoning systems". In: *Applied Intelligence* 51.9, pp. 6528–6546.

Bordes, A., N. Usunier, A. García-Durán, J. Weston, and O. Yakhnenko (2013). "Translating Embeddings for Modeling Multi-relational Data". In: *Advances in Neural Information Processing Systems 26*, pp. 2787–2795.

Bošnjak, M., T. Rocktäschel, and S. Riedel (2017). "Programming with a differentiable forth interpreter". In: *Proceedings of the 34th International Conference on Machine Learning*. Vol. 70, pp. 547–556.

Cho, K., B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio (2014). "Learning phrase representations using RNN encoder-decoder for statistical machine translation". In: *arXiv preprint arXiv:1406.1078*.

Cohen, W. W., F. Yang, and K. R. Mazaitis (2018). "TensorLog: Deep Learning Meets Probabilistic Databases". In: *Journal of Artificial Intelligence Research* 1, pp. 1–15.

Dai, W.-Z., Q. Xu, Y. Yu, and Z.-H. Zhou (2019). "Bridging Machine Learning and Logical Reasoning by Abductive Learning". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., pp. 2811–2822.

Darwiche, A. (2001). "On the tractable counting of theory models and its application to truth maintenance and belief revision". In: *Journal of Applied Non-Classical Logics* 11.1-2, pp. 11–34.

– (2011). "SDD: A new canonical representation of propositional knowledge bases". In: *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence*, pp. 819–826.

Darwiche, A. and P. Marquis (2002). "A knowledge compilation map". In: *Journal of Artificial Intelligence Research* 17, pp. 229–264.

De Raedt, L. (2008). *Logical and relational learning*. Cognitive Technologies. Springer.

De Raedt, L., S. Dumančić, R. Manhaeve, and G. Marra (2020). "From Statistical Relational to Neuro-Symbolic Artificial Intelligence". In: *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, pp. 4943–4950.

De Raedt, L., K. Kersting, S. Natarajan, and D. Poole (2016). "Statistical relational artificial intelligence: Logic, probability, and computation". In: *Synthesis Lectures on Artificial Intelligence and Machine Learning* 10.2, pp. 1–189.

De Raedt, L. and A. Kimmig (2015). "Probabilistic (logic) programming concepts". In: *Machine Learning* 100.1, pp. 5–47.

De Raedt, L., A. Kimmig, and H. Toivonen (2007). "ProbLog: A Probabilistic Prolog and Its Application in Link Discovery." In: *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence*. Vol. 7, pp. 2462–2467.

De Raedt, L., R. Manhaeve, S. Dumančić, T. Demeester, and A. Kimmig (2019). "Neuro-Symbolic = Neural + Logical + Probabilistic". In: *Proceedings of the 2019 International Workshop on Neural-Symbolic Learning and Reasoning*.

Demeester, T., T. Rocktäschel, and S. Riedel (2016). "Lifted rule injection for relation embeddings". In: *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pp. 1389–1399.

Deng, J., W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei (2009). "Imagenet: A large-scale hierarchical image database". In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee, pp. 248–255.

Devlin, J., M.-W. Chang, K. Lee, and K. Toutanova (2019). "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, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, pp. 4171–4186.

Diligenti, M., M. Gori, and C. Sacca (2017). "Semantic-based regularization for learning and inference". In: *Artificial Intelligence* 244, pp. 143–165.

Domingos, P. and D. Lowd (2009). "Markov logic: An interface layer for artificial intelligence". In: *Synthesis lectures on artificial intelligence and machine learning* 3.1, pp. 1–155.

Donadello, I., L. Serafini, and A. d'Avila Garcez (2017). "Logic Tensor Networks for Semantic Image Interpretation". In: *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*, pp. 1596–1602.

Dong, H., J. Mao, T. Lin, C. Wang, L. Li, and D. Zhou (2019). "Neural Logic Machines". In: *Proceedings of the 7th International Conference on Learning Representations*.

Došilović, F. K., M. Brčić, and N. Hlupić (2018). "Explainable artificial intelligence: A survey". In: *2018 41st International convention on information and communication technology, electronics and microelectronics (MIPRO)*. IEEE, pp. 0210–0215.

Dreyfus, H. (1976). "What computers can't do". In: *British Journal for the Philosophy of Science* 27.2.

Dries, A., A. Kimmig, W. Meert, J. Renkens, G. Van den Broeck, J. Vlasselaer, and L. De Raedt (2015). "ProbLog2: Probabilistic logic programming". In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, pp. 312–315.

Eisner, J. (2002). "Parameter estimation for probabilistic finite-state transducers". In: *Proceedings of the 40th annual meeting on Association for Computational Linguistics*. Association for Computational Linguistics, pp. 1–8.

Evans, R. and E. Grefenstette (2018). "Learning Explanatory Rules from Noisy Data". In: *Journal Artificial Intelligence Research* 61, pp. 1–64.

Fierens, D., G. V. den Broeck, J. Renkens, D. S. Shterionov, B. Gutmann, I. Thon, G. Janssens, and L. D. Raedt (2015). "Inference and learning in probabilistic logic programs using weighted Boolean formulas". In: *Theory Pract. Log. Program.* 15.3, pp. 358–401. DOI: 10.1017/S1471068414000076. URL: https://doi.org/10.1017/S1471068414000076.

Flach, P. (1994). *Simply Logical: Intelligent Reasoning by Example*. John Wiley & Sons, Inc. ISBN: 0471941522.

Frazier, M. and L. Pitt (1993). "Learning From Entailment: An Application to Propositional Horn Sentences". In: *Machine Learning, Proceedings of the*

Tenth International Conference, University of Massachusetts, Amherst, MA, USA, June 27-29, 1993, pp. 120–127.

Friedman, N., L. Getoor, D. Koller, and A. Pfeffer (1999). "Learning probabilistic relational models". In: *Proceedings of the 16th International Joint Conference on Artificial*. Vol. 99, pp. 1300–1309.

Garcez, A. d'Avila, K. Broda, and D. Gabbay (2002). *Neural-symbolic learning systems - foundations and applications*. Perspectives in neural computing. Springer.

Garcez, A. d'Avila and L. Lamb (2020). "Neurosymbolic AI: The 3rd Wave". In: *CoRR* abs/2012.05876. arXiv: 2012.05876. URL: https://arxiv.org/abs/2012.05876.

Garcez, A. d'Avila, L. Lamb, and D. Gabbay (2007). "Connectionist modal logic: Representing modalities in neural networks". In: *Theor. Comput. Sci.* 371.1-2, pp. 34–53.

Gebser, M., R. Kaminski, B. Kaufmann, and T. Schaub (2012). "Answer set solving in practice". In: *Synthesis lectures on artificial intelligence and machine learning* 6.3, pp. 1–238.

Giannini, F., M. Diligenti, M. Gori, and M. Maggini (2019). "On a Convex Logic Fragment for Learning and Reasoning". In: *IEEE Transactions Fuzzy Systems* 27.7, pp. 1407–1416.

Goodfellow, I., Y. Bengio, and A. Courville (2016). *Deep learning*. MIT press.

Goodfellow, I., J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio (2014). "Generative adversarial nets". In: *Advances in neural information processing systems* 27.

Graves, A., G. Wayne, M. Reynolds, T. Harley, I. Danihelka, A. Grabska-Barwińska, S. G. Colmenarejo, E. Grefenstette, T. Ramalho, J. Agapiou, et al. (2016). "Hybrid computing using a neural network with dynamic external memory". In: *Nature* 538.7626, pp. 471–476.

Gutmann, B., A. Kimmig, K. Kersting, and L. De Raedt (2008). "Parameter learning in probabilistic databases: A least squares approach". In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, pp. 473–488.

Jelinek, F. (1976). "Continuous speech recognition by statistical methods". In: *Proceedings of the IEEE* 64.4, pp. 532–556.

Kahneman, D. (2011). *Thinking, Fast and Slow*. Farrar, Straus and Giroux.

Kersting, K. and L. De Raedt (2007). "Bayesian Logic Programming: Theory and Tool". In: *An introduction to Statistical Relational Learning*. MIT Press, p. 291.

Kimmig, A., V. S. Costa, R. Rocha, B. Demoen, and L. De Raedt (2008). "On the efficient execution of ProbLog programs". In: *International Conference on Logic Programming*. Springer, pp. 175–189.

Kimmig, A., G. Van den Broeck, and L. De Raedt (2011). "An Algebraic Prolog for Reasoning about Possible Worlds." In: *Proceedings of the 25th AAAI conference on Artificial Intelligence*, pp. 209–2014.

– (2017). "Algebraic model counting". In: *Journal of Applied Logic* 22, pp. 46–62.

Kingma, D. P. and J. Ba (2015). "Adam: A Method for Stochastic Optimization". In: *Proceedings of the 3rd International Conference on Learning Representations*.

Kingma, D. P. and M. Welling (2014). "Auto-Encoding Variational Bayes". In: *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*. Ed. by Y. Bengio and Y. LeCun.

Kipf, T. N. and M. Welling (2017). "Semi-Supervised Classification with Graph Convolutional Networks". In: *Proceedings of the 5th International Conference on Learning Representations*.

Kok, S. and P. M. Domingos (2005). "Learning the structure of Markov logic networks". In: *Proceedings of the 22nd International Conference on Machine Learning*. Vol. 119. ACM International Conference Proceeding Series. ACM, pp. 441–448.

Koller, D. and N. Friedman (2009). *Probabilistic Graphical Models - Principles and Techniques*. MIT Press.

van Krieken, E., E. Acar, and F. van Harmelen (2020). "Analyzing Differentiable Fuzzy Implications". In: *Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning*. Ed. by D. Calvanese, E. Erdem, and M. Thielscher, pp. 893–903.

Lai, T. L. and H. Robbins (1985). "Asymptotically efficient adaptive allocation rules". In: *Advances in applied mathematics* 6.1, pp. 4–22.

LeCun, Y., Y. Bengio, and G. Hinton (2015). "Deep learning". In: *Nature* 521.7553, pp. 436–444.

Lecun, Y., L. Bottou, Y. Bengio, and P. Haffner (1998). "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE*, pp. 2278–2324.

Li, Q., S. Huang, Y. Hong, Y. Chen, Y. N. Wu, and S.-C. Zhu (2020). "Closed Loop Neural-Symbolic Learning via Integrating Neural Perception, Grammar Parsing, and Symbolic Reasoning". In: *Proceedings of the 37th International Conference on Machine Learning*.

Lighthill, J. (1973). "Artificial Intelligence: A General Survey". In: *Artificial Intelligence: a paper symposium*.

Lloyd, J. W. (1989). *Foundations of Logic Programming*. 2nd ed. Springer.

Lowd, D. and P. M. Domingos (2007). "Efficient Weight Learning for Markov Logic Networks". In: *Proceedings of the 11th European Conference on Principles and Practice of Knowledge Discovery in Databases*. Vol. 4702. Lecture Notes in Computer Science. Springer, pp. 200–211.

Lu, B. and Z. Liu (2013). "Prolog with best first search". In: *2013 25th Chinese Control and Decision Conference (CCDC)*. IEEE, pp. 917–922.

Manhaeve, R., K. De Grave, L. Antanas, and L. De Raedt (2017). *Realtime road user detection and classification with single pass deep learning*. Abstract of the MA Thesis: Realtime road user detection with single pass deep learning.

Manhaeve, R., L. De Raedt, A. Kimmig, S. Dumančić, and T. Demeester (2019). *DeepProbLog: Integrating logic and learning through algebraic model counting*. Vancouver, Canada.

Manhaeve, R., S. Dumančić, A. Kimmig, T. Demeester, and L. De Raedt (2018). "DeepProbLog: Neural probabilistic logic programming". In: *Advances in Neural Information Processing Systems 31*, pp. 3749–3759.

– (2021). "Neural probabilistic logic programming in DeepProbLog". In: *Artificial Intelligence* 298, pp. 103–504.

Manhaeve, R., G. Marra, and L. De Raedt (2021). "Approximate Inference for Neural Probabilistic Logic Programming". In: *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning*, pp. 475–486.

Manhaeve, R., G. Marra, T. Demeester, S. Dumančić, A. Kimmig, and L. De Raedt (2020). "From Statistical Relational to Neural-Symbolic Artificial Intelligence". In: *Neuro-Symbolic Artificial Intelligence - The State of the Art*. Ed. by P. Hitzler and M. K. Sarker. IOS Press.

Marcus, G. (2018). "Deep learning: A critical appraisal". In: *arXiv preprint arXiv:1801.00631*.

Marra, G., M. Diligenti, F. Giannini, M. Gori, and M. Maggini (2020). "Relational Neural Machines". In: *Proceedings of the 24th European Conference on Artificial Intelligence*. Vol. 325. Frontiers in Artificial Intelligence and Applications. IOS Press, pp. 1340–1347.

Marra, G. and O. Kuželka (2019). "Neural Markov Logic Networks". In: *CoRR* abs/1905.13462.

Minervini, P., M. Bosnjak, T. Rocktäschel, and S. Riedel (2018). "Towards Neural Theorem Proving at Scale". In: *CoRR* abs/1807.08204. URL: `http://arxiv.org/abs/1807.08204`.

Minervini, P., T. Demeester, T. Rocktäschel, and S. Riedel (2017). "Adversarial Sets for Regularised Neural Link Predictors". In: *Proceedings of the 33rd Conference on Uncertainty in Artificial Intelligence*.

Minervini, P., S. Riedel, P. Stenetorp, E. Grefenstette, and T. Rocktäschel (2020). "Learning reasoning strategies in end-to-end differentiable proving". In: *Proceedings of the 37th International Conference on Machine Learning*. PMLR, pp. 6938–6949.

Moldovan, B., I. Thon, J. Davis, and L. D. Raedt (2013). "MCMC Estimation of Conditional Probabilities in Probabilistic Programming Languages". In: *Proceedings of the 12th Europen Conference on Symbolic and Quantitative*

Approaches to Reasoning with Uncertainty. Vol. 7958. Lecture Notes in Computer Science. Springer, pp. 436–448.

Muggleton, S. (1996). "Stochastic logic programs". In: *Advances in Inductive Logic Programming* 32, pp. 254–264.

Muggleton, S. and L. De Raedt (1994). "Inductive Logic Programming: Theory and Methods". In: *Journal of Logic Programming* 19/20, pp. 629–679.

Newell, A. and H. A. Simon (1976). "Computer Science as Empirical Inquiry: Symbols and Search". In: *Communications of the ACM* 19.3, pp. 113–126.

Ng, R. and V. S. Subrahmanian (1992). "Probabilistic logic programming". In: *Information and computation* 101.2, pp. 150–201.

Paszke, A., S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer (2017). "Automatic differentiation in PyTorch". In: *Proceedings of the Workshop on The future of gradient-based machine learning software and techniques, co-located with the 31st Annual Conference on Neural Information Processing Systems (NIPS 2017)*.

Pearl, J. (1984). "Intelligent search strategies for computer problem solving". In: *Addision Wesley*.

– (1988). *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann Publishers Inc. ISBN: 0-934613-73-7.

Poole, D. (2008). "The Independent Choice Logic and Beyond". In: *Probabilistic Inductive Logic Programming - Theory and Applications*. Vol. 4911. LNCS. Springer, pp. 222–243.

Rawson, M. and G. Reger (2019). "A neurally-guided, parallel theorem prover". In: *International Symposium on Frontiers of Combining Systems*. Springer, pp. 40–56.

Reed, S. and N. de Freita (2016). "Neural Programmer-Interpreters". In: *Proceedings of the 4th International Conference on Learning Representations*.

Renkens, J., G. Van den Broeck, and S. Nijssen (2012). "k-optimal: A novel approximate inference algorithm for ProbLog". In: *Machine learning* 89.3, pp. 215–231.

Ribeiro, M. T., S. Singh, and C. Guestrin (2016). "" Why should i trust you?" Explaining the predictions of any classifier". In: *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pp. 1135–1144.

Richardson, M. and P. M. Domingos (2006). "Markov logic networks". In: *Machine Learning* 62.1-2, pp. 107–136.

Rocktäschel, T. and S. Riedel (2017). "End-to-end Differentiable Proving". In: *Advances in Neural Information Processing Systems 30*. Vol. 30, pp. 3788–3800.

Rocktäschel, T., S. Singh, and S. Riedel (2015). "Injecting Logical Background Knowledge into Embeddings for Relation Extraction". In: *NAACL HLT 2015,*

The 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pp. 1119–1129.

Roy, S. and D. Roth (2015). "Solving General Arithmetic Word Problems". In: *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pp. 1743–1752.

Russell, S. (2015). "Unifying logic and probability". In: *Communications of the ACM* 58.7, pp. 88–97.

Russell, S. and P. Norvig (2010). *Artificial Intelligence: A Modern Approach (3rd Edition)*. Pearson. ISBN: 9780132071482.

Sarker, M. K., L. Zhou, A. Eberhart, and P. Hitzler (2021). "Neuro-Symbolic Artificial Intelligence: Current Trends". In: *CoRR* abs/2105.05330. arXiv: 2105.05330. URL: https://arxiv.org/abs/2105.05330.

Scarselli, F., M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini (2009). "The Graph Neural Network Model". In: *IEEE Trans. Neural Networks* 20.1, pp. 61–80. DOI: 10.1109/TNN.2008.2005605. URL: https://doi.org/10.1109/TNN.2008.2005605.

Schlichtkrull, M. S., T. N. Kipf, P. Bloem, R. van den Berg, I. Titov, and M. Welling (2018). "Modeling Relational Data with Graph Convolutional Networks". In: *The Semantic Web - 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, June 3-7, 2018, Proceedings*. Ed. by A. Gangemi, R. Navigli, M.-E. Vidal, P. Hitzler, R. Troncy, L. Hollink, A. Tordai, and M. Alam. Vol. 10843. Lecture Notes in Computer Science. Springer, pp. 593–607.

Shterionov, D., A. Kimmig, T. Mantadelis, and G. Janssens (2010). "DNF sampling for ProbLog inference". In: *Proceedings International Colloquium on Implementation of Constraint and LOgic Programming Systems (CICLOPS)*, p. 15.

Si, X., M. Raghothaman, K. Heo, and M. Naik (2019). "Synthesizing Datalog Programs using Numerical Relaxation". In: *Proceedings of the 28th International Joint Conference on Artificial Intelligence*.

Sinha, K., S. Sodhani, J. Dong, J. Pineau, and W. L. Hamilton (2019). "CLUTRR: A Diagnostic Benchmark for Inductive Reasoning from Text". In: *Empirical Methods of Natural Language Processing (EMNLP)*.

Solar-Lezama, A. (2013). "Program sketching". In: *International Journal on Software Tools for Technology Transfer* 15.5-6, pp. 475–495.

Šourek, G., V. Aschenbrenner, F. Zelezný, S. Schockaert, and O. Kuželka (2018). "Lifted Relational Neural Networks: Efficient Learning of Latent Relational Structures". In: *Journal of Artificial Intelligence Research* 62.

Towell, G. G. and J. W. Shavlik (1994). "Knowledge-based artificial neural networks". In: *Artificial intelligence* 70.1-2, pp. 119–165.

Tran, S. and A. d'Avila Garcez (2018). "Deep Logic Networks: Inserting and Extracting Knowledge From Deep Belief Networks". In: *IEEE Transactions on Neural Networks and Learning Systems* 29.2, pp. 246–258.

Trouillon, T., J. Welbl, S. Riedel, É. Gaussier, and G. Bouchard (2016). "Complex Embeddings for Simple Link Prediction". In: *Proceedings of the 33nd International Conference on Machine Learning*. Vol. 48. JMLR Workshop and Conference Proceedings. JMLR.org, pp. 2071–2080.

Tsamoura, E., T. Hospedales, and L. Michael (2021). "Neural-Symbolic Integration: A Compositional Perspective". In: *Proceedings of the 35th AAAI Conference on Artificial Intelligence*. Vol. 35. 6, pp. 5051–5060.

Van den Broeck, G., D. Suciu, et al. (2017). "Query processing on probabilistic data: A survey". In: *Foundations and Trends in Databases* 7.3-4, pp. 197–341.

Van Gelder, A., K. A. Ross, and J. S. Schlipf (1991). "The Well-Founded Semantics for General Logic Programs". In: *Journal of the ACM* 38.3, pp. 620–650.

Vlasselaer, J., G. Van den Broeck, A. Kimmig, W. Meert, and L. De Raedt (2016). "T_P-Compilation for inference in probabilistic logic programs". In: *International Journal of Approximate Reasoning* 78, pp. 15–32.

Wang, M., Y. Tang, J. Wang, and J. Deng (2017). "Premise Selection for Theorem Proving by Deep Graph Embedding". In: *Advances in Neural Information Processing Systems 30*.

Weber, L., P. Minervini, J. Münchmeyer, U. Leser, and T. Rocktäschel (2019). "NLProlog: Reasoning with Weak Unification for Question Answering in Natural Language". In: *Proceedings of ACL 2018, Tutorial Abstracts*.

Winters, T., G. Marra, R. Manhaeve, and L. D. Raedt (2022). "DeepStochLog: Neural Stochastic Logic Programming". In: *Proceedings of the 35th AAAI Conference on Artificial Intelligence*.

Xu, J., Z. Zhang, T. Friedman, Y. Liang, and G. V. den Broeck (2018). "A Semantic Loss Function for Deep Learning with Symbolic Knowledge". In: *Proceedings of the 35th International Conference on Machine Learning*, pp. 5498–5507.

Xu, K., J. Li, M. Zhang, S. S. Du, K.-i. Kawarabayashi, and S. Jegelka (2020). "What Can Neural Networks Reason About?" In: *8th International Conference on Learning Representations*.

Yang, B., W.-t. Yih, X. He, J. Gao, and L. Deng (2015). "Embedding Entities and Relations for Learning and Inference in Knowledge Bases". In: *Proceedings of the 3rd International Conference on Learning Representations*.

Yang, F., Z. Yang, and W. W. Cohen (2017). "Differentiable Learning of Logical Rules for Knowledge Base Reasoning". In: *Advances in Neural Information Processing Systems 30*, pp. 2319–2328.

Yang, Z., A. Ishay, and J. Lee (2020). "NeurASP: Embracing neural networks into answer set programming". In: *Proceedings of the 29th International Joint Conference on Artificial Intelligence*, pp. 1755–1762.

Yi, K., J. Wu, C. Gan, A. Torralba, P. Kohli, and J. Tenenbaum (2018). "Neural-symbolic vqa: Disentangling reasoning from vision and language understanding". In: *Advances in Neural Information Processing Systems 31*, pp. 1039–1050.

Zhang, S., H. Tong, J. Xu, and R. Maciejewski (2019). "Graph convolutional networks: a comprehensive review". In: *Computational Social Networks* 6.1, pp. 1–23.

Zuidberg Dos Martires, P., V. Derkinderen, R. Manhaeve, W. Meert, A. Kimmig, and L. De Raedt (2019). *Transforming Probabilistic Programs into Algebraic Circuits for Inference and Learning*. Vancouver.

Curriculum Vitae

In 2015, Robin Manhaeve attained a Bachelor of Engineering in Computer Science and Electrical Engineering (cum laude) at the KU Leuven. In 2017, he graduated from the KU Leuven with a Master of Engineering in Computer Science with a specialization in Artificial Intelligence (magna cum laude). His master's thesis was titled "Realtime Road User Detection and Classification with Single Pass Deep Learning". In 2017, he started his PhD in Computer science at the DTAI lab at KU Leuven under the supervision of Prof. Luc De Raedt. In 2018, he received an FWO SB fellowship to work on neural probabilistic logic programming, which the work presented here is about.

List of publications

Journal articles

- R. Manhaeve, S. Dumančić, A. Kimmig, T. Demeester, and L. De Raedt (2021). "Neural probabilistic logic programming in DeepProbLog". In: *Artificial Intelligence* 298, pp. 103–504.

Conference papers

- R. Manhaeve, S. Dumančić, A. Kimmig, T. Demeester, and L. De Raedt (2018). "DeepProbLog: Neural probabilistic logic programming". In: *Advances in Neural Information Processing Systems 31*, pp. 3749–3759.
- R. Manhaeve, G. Marra, and L. De Raedt (2021). "Approximate Inference for Neural Probabilistic Logic Programming". In: *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning*, pp. 475–486.
- L. De Raedt, S. Dumančić, R. Manhaeve, and G. Marra (2020). "From Statistical Relational to Neuro-Symbolic Artificial Intelligence". In: *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, pp. 4943–4950.

Workshop papers

- R. Manhaeve, L. De Raedt, A. Kimmig, S. Dumančić, and T. Demeester (2019). *DeepProbLog: Integrating logic and learning through algebraic model counting*. Vancouver, Canada.
- L. De Raedt, R. Manhaeve, S. Dumančić, T. Demeester, and A. Kimmig (2019). "Neuro-Symbolic = Neural + Logical + Probabilistic". In:

Proceedings of the 2019 International Workshop on Neural-Symbolic Learning and Reasoning.

- P. Zuidberg Dos Martires, V. Derkinderen, R. Manhaeve, W. Meert, A. Kimmig, and L. De Raedt (2019). *Transforming Probabilistic Programs into Algebraic Circuits for Inference and Learning*. Vancouver.

Abstracts

- R. Manhaeve, K. De Grave, L. Antanas, and L. De Raedt (2017). *Realtime road user detection and classification with single pass deep learning*. Abstract of the MA Thesis: Realtime road user detection with single pass deep learning.

Book chapters

- R. Manhaeve, G. Marra, T. Demeester, S. Dumančić, A. Kimmig, and L. De Raedt (2020). "From Statistical Relational to Neural-Symbolic Artificial Intelligence". In: *Neuro-Symbolic Artificial Intelligence - The State of the Art*. Ed. by P. Hitzler and M. K. Sarker. IOS Press, *in preparation*.



FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
DTAI
Celestijnenlaan 200A box 2402
B-3001 Leuven
robin.manhaeve@cs.kuleuven.be
http://www.dtai.cs.kuleuven.be