Scalable Neuro-Symbolic Reasoning and Learning

Yaniv Aspis

Department of Computing
Imperial College London

Supervisor: Alessandra Russo
Co-Supervisor: Krysia Broda
Co-Supervisor: Jorge Lobo

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
May 2026

Statement of Originality

I, Yaniv Aspis, declare that the work in this thesis is my own. The work of others has been appropriately referenced. A full list of references is given in the bibliography.

The copyright of this thesis rests with the author. Unless otherwise indicated, its contents are licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International Licence (CC BY NC-SA). Under this licence, you may copy and redistribute the material in any medium or format. You may also create and distribute modified versions of the work. This is on the condition that: you credit the author and do not use it, or any derivative works, for a commercial purpose. When reusing or sharing this work, ensure you make the licence terms clear to others by naming the licence and linking to the licence text. Where a work has been adapted, you should indicate that the work has been changed and describe those changes. Please seek permission from the copyright holder for uses of this work that are not included in this licence or permitted under UK Copyright Law.

Abstract

Neuro-Symbolic AI bridges the gap between highly performant connectionist learning methods and the formal reasoning capabilities of symbolic methods. While neural networks can learn tasks with minimal engineering, their black-box nature prevents their inference from being examined, disallowing application in mission-critical settings. Symbolic systems provide interpretable inference with guarantees, but cannot be applied directly to fuzzy data and require some manual engineering. The complementary nature of the approaches motivates developing hybrid systems.

Bridging the continuous nature of neural networks and the discrete nature of symbolic AI is challenging. Approaches generally come in two flavours: either logical operations are approximated using continuous operations, or a loss function is computed using a symbolic solver. The former is more scalable but comes at the expense of exact inference, and offers limited representation capabilities and transferability. The latter gives the full power of a symbolic solver, but lacks scalability. In this work, we address how to bring about inference with guarantees in neuro-symbolic AI while maintaining scalability. We consider both neuro-symbolic reasoning: performing logical deduction from facts observed by neural networks; and neuro-symbolic learning: inducing rules from such observed data.

We begin by introducing a method for embedding a given normal logic program in continuous space using real matrices, allowing differentiable logical deduction while maintaining exact inference. We then show how this matrix representation enables end-to-end training of perception networks for neuro-symbolic reasoning tasks, and study the advantages and limitations of this approach. Building on these observations, we develop a novel approach to neuro-symbolic reasoning and learning, by combining end-to-end pretraining of a neural network, clustering of latent space embeddings, and exact symbolic inference and learning. We achieve a perception training speedup of several orders of magnitude over existing methods, even in the presence of limited labelling of latent features.

Acknowledgements

I would like to thank my supervisors, Alessandra Russo, Krysia Broda and Jorge Lobo, for their amazing support and advice throughout the duration of this work. In particular, Alessandra never gave up on finding a way for me to pursue my PhD studies, without which this work would not exist. Besides providing research direction guidance, Alessandra always found the time to provide support in any way she could, for which I am forever grateful. Krysia was the one who originally took me on as a student during my MSc days, and introduced me to Neuro-Symbolic AI. Much of the research presented here can trace its roots to the work we accomplished back in those days. As such, it would not be an understatement to say this work would not exist without her. Krysia could always be counted on to provide detailed, thoughtful feedback, which has brought this work to far higher standards. I would also like to thank Jorge Lobo. I first met Jorge when he visited Imperial around the time I began my PhD studies. Jorge would join our weekly meetings to provide invaluable advice throughout much of my studies, far beyond his visiting period. Jorge's feedback contributed greatly to this work, as well as his support in theoretical contributions and the writing process. In addition, his challenging of my assumptions would always push me towards higher standards.

I would like to extend my gratitude to Mark Law, who provided great support on sections of this work that employ symbolic learning. Besides his help in setting up experimental environments, Mark was always happy to join meetings to provide advice on the best use of ILASP and FastLAS, and gave helpful feedback on some of the neuro-symbolic learning frameworks I was proposing. Several times, he would even introduce modifications to FastLAS to support my work, for which I am thankful.

I would also like to thank the SPIKE group, for many insightful discussions, ideas, and just general fun throughout the years. In particular, I would like to mention Dan Cunnington, who did his studies alongside me on Neuro-Symbolic AI. Our constant exchange of ideas and his sharing of experimental results helped me refine my own work and improve it. I would also like to thank David Tuckey, Harry Stormfelt and Daniel Furelos Blanco for their advice. I am particularly grateful to Mohammad Albinhassan. Mohammad joined the group during the last stages of my research, and we worked closely together on the final piece of work. Besides being generally fun to work with, Mohammad's hard work helped provide many of the experimental results, improved on others, resulting in higher-quality research. I am grateful to Theo Charalambous and Krishan Sritharar, two MEng students whom I had the great honour of supervising. Theo's work on Neuro-Symbolic Learning and Krishan's on Constrained Generative AI were both extremely insightful, and I learned much from both of them.

I would like to thank my friends, especially my close friend and colleague, Sim Virdi, who supported me as I took time to complete this work, and was always there for me even in the most difficult of times. I would especially like to thank my family, for their unwavering help and advice throughout the years, who first supported me in my move to England to pursue my studies and encouraged me to fulfil my dream of attaining a PhD.

Lastly, and most importantly, I would like to thank my wife Laura. Laura has been by my side throughout the years, supporting me every step of the way, cheering me on as I worked tireless hours to achieve my goals. She also graciously allowed me to borrow her gaming PC, on which several of the experiments presented here were run. During my studies, I had the honour of marrying her, for which I feel extremely lucky. The countless hours we spend together, and with our cat Lily, continue to be the greatest source of joy in my life.

Contents

List of Tables **xiii**

Acronyms **xv**

Notation **xvii**

1 Introduction **1**
1.1 Logic-Based Representation & Inference . . . 2
1.2 Neural Representation & Inference . . . 4
1.3 Neuro-Symbolic Representation & Inference . . . 5
1.4 Contributions . . . 7
1.4.1 List of Publications . . . 9
1.4.2 Structure of This Work . . . 9

2 Background **11**
2.1 Logic Programming . . . 11
2.1.1 Language & Syntax . . . 11
2.1.2 Semantics of Propositional Logic . . . 14
2.1.3 Semantics of First-Order Programs . . . 21
2.1.4 Answer Set Programming . . . 24
2.2 Machine Learning . . . 27
2.2.1 Inductive Logic Programming . . . 27
2.2.2 Clustering . . . 29
2.2.3 Neural Networks . . . 30
2.2.4 Training, Loss & Backpropagation . . . 31
2.2.5 Convolutional Neural Networks . . . 32
2.2.6 ResNets . . . 34
2.2.7 Attention & Transformers . . . 35
2.2.8 Vision-Language Models . . . 36

3 Stable and Supported Semantics in Continuous Vector Spaces **37**
3.1 Multiple Definitions Condition . . . 38
3.1.1 Other Assumptions . . . 38
3.1.2 Definition of the MD condition . . . 39
3.1.3 Transformation . . . 40
3.2 Matrix Representation of Normal Programs . . . 43
3.2.1 Embedding Definite Programs . . . 45
3.2.2 Embedding Program Reduct . . . 49

3.3 Differentiable Immediate Consequences . . . 53
3.3.1 Constraints for Exact Inference . . . 53
3.3.2 Deduction over Continuous Vectors . . . 57
3.3.3 A Solution for the Hyperparameters . . . 59
3.4 Gradient-Based Computation of Supported Models . . . 61
3.5 Evaluation . . . 63
3.6 Summary . . . 68

4 Logic Program Layer 69
4.1 Neuro-Symbolic Theoretical Framework . . . 71
4.2 Reasoning with Logic Program Layers . . . 74
4.2.1 Hierarchical Logic Programs . . . 74
4.2.2 Matrix-Vector Representation . . . 76
4.2.3 Evaluation . . . 81
4.3 Learning Logic Program Layers . . . 90
4.3.1 Learning Framework . . . 91
4.3.2 Model & Loss Function . . . 92
4.3.3 Evaluation . . . 95
4.3.4 Discussion . . . 99
4.4 Summary . . . 100

5 Embed2Sym - Scalable Neuro-Symbolic Reasoning via Clustered Embeddings 101
5.1 Fully Neural Model . . . 102
5.2 Identifying Latent Concepts . . . 104
5.2.1 Cluster Discovery . . . 105
5.2.2 Cluster Labelling . . . 106
5.3 Training & Inference Algorithm . . . 109
5.4 Evaluation . . . 112
5.4.1 MNIST Addition . . . 113
5.4.2 CIFAR-10 Addition . . . 115
5.4.3 Member . . . 116
5.4.4 Forth Sort . . . 118
5.4.5 MNIST Addition Mod 10 . . . 120
5.4.6 Choice of Clustering Algorithm . . . 122
5.5 Discussion . . . 123
5.6 Summary . . . 125

6 Embed2Rule - Scalable Neuro-Symbolic Learning via Latent Space Weak-Labelling 127
6.1 Method Overview . . . 128
6.1.1 Joint Optimisation via Program Induction . . . 129
6.2 Weak & Cluster Labelling . . . 130
6.3 Symbolic Learning . . . 133
6.4 Evaluation . . . 134
6.4.1 Hitting Sets . . . 136
6.4.2 Visual Sudoku Classification . . . 138
6.4.3 Follow Suit Winner . . . 142
6.4.4 Discussion . . . 144

6.5 Summary . . . 145

7 Related Work **147**
7.1 Tensor-Based Semantics for Logic Programs . . . 147
7.2 Neuro-Symbolic Reasoning . . . 148
7.2.1 Neural approximation of logical operations . . . 149
7.2.2 Fuzzy Approximations of Logic Operators . . . 149
7.2.3 Other Approximations of Differentiable Operations . . . 150
7.2.4 Solver-based Neuro-Symbolic Reasoning . . . 151
7.2.5 Neural Approximation in Training with Solver-Based Inference . . . 153
7.3 Neuro-Symbolic Learning . . . 154
7.3.1 Neuro-Symbolic Learning: Propositional . . . 154
7.3.2 Differentiable ILP . . . 156
7.3.3 Neuro-Symbolic Learning: First-Order . . . 158
7.4 Additional Related Work . . . 159

8 Conclusion **163**
8.1 Future Work . . . 165

Bibliography **169**

List of Figures

2.1 Architecture of a feedforward network. 30
2.2 Architecture of a CNN applied to images. 33
2.3 An example of a ResNet block. 34

3.1 Graphs of $\sigma_{\gamma,\tau}(x)$ for various values of γ and τ. Left: varying γ. Right: varying τ. 53
3.2 Success rate of converging to a supported model with negative loops. (a) A single negative loop for varying temperatures. (b) N negative loops for $\tau = 0.087$. (c) N negative loops for $\tau = 0.087$ and when given as many attempts as possible within 1 second. 64
3.3 Success rate for N choices under various settings. Short stands for the representation of choice as a set of short clauses. 67
3.4 The binary tree corresponding to the encoding of $N = 6$ choices using short clauses only. 67

4.1 General neural-symbolic architecture. 70
4.2 Architecture for neural-symbolic MNIST Addition task. 82
4.3 Input to the Syllogism task. There are two input images, each an Euler diagram describing a relation between two sets. 84
4.4 PCA visualisation of the latent space of the Embedding model. Four clusters appear to form, corresponding to the four latent concept values. 90

5.1 Embed2Sym architecture. Left: An example of a fully neural architecture for the MNIST Addition task. Middle: After training the fully neural network, clustering is applied to embedding vectors computed by the perception network. Symbolic optimisation is used to label the clusters together with the rules P_g. Right: An example inference on MNIST digits. The cluster labels are used as symbolic inputs to the program P (which may differ from P_g). 103
5.2 t-SNE visualisation of f_{digit} embedding space post-training, clearly showing clusters representing 10 different digits. 105
5.3 High-level diagram of the fully neural architecture for Forth Sort. 118
5.4 PCA visualisation of the embedding space of Swap. 120

6.1 Architecture of the Embed2Rule system. 130

List of Tables

3.1 Success rate with uniform sampling for programs with positive and negative loops. 66

4.1 Dataset information and accuracy for the MNIST Addition experiment. 83
4.2 Accuracy comparison and training time for the MNIST Addition experiment. . . 84
4.3 Consistent conclusions about red-blue pairs given red-green and green-blue premises. 86
4.4 Validation accuracy results for red-blue relation (final label). 87
4.5 Latent accuracy results for the Syllogism experiment. 87
4.6 Validation accuracy results for red-blue relation (final label). 89
4.7 Results for the Syllogism Experiments with more informative target labels. . . . 89
4.8 Accuracy in prediction for the Zoo task 99

5.1 Average test sum accuracy (%) on the MNIST Addition task for varying number of digits. T/O stands for "timeout". 114
5.2 Average test digit accuracy (%) on the MNIST Addition task for varying number of digits. T/O stands for "timeout". 114
5.3 Average training time (sec) on the MNIST Addition task for varying number of digits. Training time of "Embed2Sym - Neuro-Symbolic" includes the training time of "Embed2Sym - Fully-Neural". T/O stands for "timeout". Results reported here for DeepProbLog are after 1 epoch of training. 114
5.4 Results for the CIFAR-10 Single Digit Addition experiment. Task % = Test accuracy in predicting the sum correctly. Image % = Test accuracy in predicting the digit (object) correctly. The fully neural model produces embeddings for the latent concepts and therefore cannot be assessed on Image Accuracy. Training time is in seconds. 116
5.5 Average test membership accuracy (%) on the Member task for a varying size of the set. T/O stands for "timeout". 117
5.6 Average test digit accuracy (%) on the Member task for varying number of digits. T/O stands for "timeout". 117
5.7 Average training time (sec) on the Member task for varying number of digits. Training time of "E2S - NS" includes the training time of "E2S - FN". T/O stands for "timeout". 117
5.8 Average training time (sec) for the Forth Sort task on various list lengths. Time to reach 100% accuracy on the test set is reported. T/O stands for "timeout". B/Q = Bubble/Quick Sort. 119

5.9 Results for the MNIST Addition Mod 10 experiment. Task % = Test accuracy in predicting the sum mod 10 correctly. Digit % = Test accuracy in predicting the digit correctly. The fully neural model produces embeddings for the latent concepts and therefore cannot be assessed on Image Accuracy. 121
5.10 Results for the Clustering Ablation study on the MNIST Addition experiment. Fully Neural % = Test accuracy in predicting the sum correctly for the fully neural model. Neuro-Symbolic % = Test accuracy in predicting the sum correctly for the neuro-symbolic model. Digit % = Test accuracy in predicting the digit correctly. 123

6.1 Accuracy of the classifier implemented using BLIP-2's Image-Text Matching head. 131
6.2 Average downstream test accuracy (%) on the Hitting Sets task. 5-4 means 5 elements in U and 4 images. T/O = "timeout", OOM = "Out of Memory". . . . 137
6.3 Average latent test accuracy (%) on the Hitting Sets task. 138
6.4 Average training time (sec) on the Hitting Sets task. 138
6.5 Average downstream test accuracy (%) on the Visual Sudoku Classification task. 140
6.6 Average latent test accuracy (%) on the Visual Sudoku Classification task. . . . 141
6.7 Average training time (sec) on the Visual Sudoku Classification task. 142
6.8 Results for Follow Suit Winner. Training time is in seconds. 144

Acronyms

AI Artificial Intelligence

ASP Answer Set Programming

BLIP Bootstrapping Language-Image Pre-Training

CBM Concept Bottleneck Model

CDNL Conflict Driven Nogood Learning

CDPI Context-Dependent Partial Interpretation

CLIP Contrastive Language-Image Pre-Training

CNF Conjunctive Normal Form

CNN Convolutional Neural Network

DNF Disjunctive Normal Form

GELU Gaussian Error Linear Unit

GPT Generative Pre-Trained Transformer

GRU Gated Recurrent Unit

ILASP Inductive Learning of Answer Set Programs

ILP Inductive Logic Programming

LAS Learning from Answer Sets

LFIT Learning from Interpretation Transition

LFIVT Learning from Interpretation Vector Transition

LHM Least Herbrand Model

LLM Large Language Model

LSTM Long Short-Term Memory

MAP Maximum A-Posteriori

MILP Mixed Integer Linear Programming

MLP Multi-layer Perceptron

MRF Markov Random Field

NSAI Neuro-Symbolic Artificial Intelligence

PCA Principle Component Analysis

ReLU Rectified Linear Unit

SGD Stochastic Gradient Descent

SMT Satisfiability Modulo Theories

t-SNE t-distributed Stochastic Neighbour Embedding

VLM Vision Language Model

Notation

The following list includes notations that are recurrently used throughout the thesis.

General Mathematical Symbols

S	a set
$\subseteq$	subset
$\subsetneq$	proper subset
(A, B)	a (two) tuple
$A \otimes B$	the cartesian product of A and B
$\bigotimes_i A_i$	the cartesian product of a collection $\{A_i\}$, indexed by i
$\emptyset$	the empty set
$\|S\|$	the cardinality of set S
$f(\cdot)$	a function
$g \circ f$	a function composition of g applied to the results of f
$H(x)$	the Heaviside function: $H(x) = 1$ if $x \geq 0$ else 0
$\sigma(x)$	the Sigmoid function: $\sigma(x) = (1 + e^{-x})^{-1}$
$\sigma_{\gamma,\tau}(x)$	the Sigmoid function parameterized by threshold and temperature: $\sigma_{\gamma,\tau}(x) = \left(1 + e^{-\frac{x-\gamma}{\tau}}\right)^{-1}$
$\pi(x)$	a permutation function
a	a scalar
$\boldsymbol{v}$	a vector
v_i	entry i of vector $\boldsymbol{v}$
$\boldsymbol{v}^\intercal$	the transpose of vector $\boldsymbol{v}$
$\boldsymbol{u}^\intercal \boldsymbol{v}$	the dot product of vectors $\boldsymbol{u}$ and $\boldsymbol{v}$
$\boldsymbol{e}_i$	the i^{th} (unit) vector of the standard basis
M	a matrix
M_{ij}	the entry corresponding to the i^{th} row and j^{th} column of matrix M
$O(f(N))$	asymptotically bounded by $f(N)$

Logic Programming

$\top$	top, always evaluates to true
$\bot$	bottom, always evaluates to false
a	a propositional atom
$p(t_1, t_2, \ldots, t_n)$	n-ary predicate with terms $t_1, t_2, \ldots, t_n$
not	negation-as-failure

V	a variable term
c	a constant term
r	a rule
h^r	the head atom of rule r
B^r	the positive body of rule r
C^r	the negative body of rule r
P	a logic program
B_P	the Herbrand Base of the program P
I	interpretation of a program
M	model of a program
A	answer set of an ASP program
T_P	the immediate consequence (Tarski) operator for program P
$LHM\,(P)$	the least Herbrand model of program P
P^I	the program reduct of interpretation I

Inductive Logic Programming

B	background knowledge of an ILASP task
S_M	the hypothesis space of an ILASP task
e^{inc}	an inclusion set of atoms of a partial interpretation
e^{exc}	an exclusion set of atoms of a partial interpretation
e_{pi}	a partial interpretation $e_{pi} = \left(e^{inc}, e^{exc}\right)$
e_{id}	an identifier for a weighted context-dependent example
e_{ctx}	context for a weighted context-dependent example
e_{pen}	penalty (or weight) for a weighted context-dependent example
E^+	a set of positive examples for an ILASP task
E^-	a set of negative examples for an ILASP task
H	a hypothesis program

Linear Algebraic Characterisation of Logic Programs

$\boldsymbol{v}^I$	the vector representation of interpretation I
D^P	the matrix representation of logic program P
$D^{P,I}$	the matrix representation of program reduct P^I
$\gamma^\perp$	an upper bound on false values
$\gamma^\top$	a lower bound on true values
$\sim^{\gamma^\top}_{\gamma^\perp}$	semantic equivalence between two interpretation vectors, with respect to $\gamma^\perp$ and $\gamma^\top$

Neuro-Symbolic Framework

x_i	raw input (indexed by i)
$\mathcal{X}_i$	the raw input space associated with raw input x_i
$\int_k$	symbolic input (indexed by k)
$\mathcal{S}_k$	the set of values associated with symbolic input s_k
c_j	latent concept (indexed by j)

$\mathcal{V}_j$	the set of values associated with latent concept c_j
$\mathcal{Z}_j$	the (continuous) latent concept space associated with latent concept c_j
C	the set of latent concepts
y_l	target concept (indexed by l)
T_l	the set of values associated with target concept y_l
$\mathcal{D}$	a dataset
f_j	a perception function associated with latent concept c_j
f	the set of perception functions
G_j	a symbol grounding function associated with latent concept c_j
P_j	a classifier function associated with latent concept c_j
g	a reasoning function
P_g	a logic program representation of the reasoning function g
h	a neuro-symbolic function, the composition of perception functions and a reasoning function
$\mathcal{C}_j$	a clustering function associated with latent concept c_j
$\mathcal{C}$	the set of clustering functions
T_j	a description function for latent concept c_j, mapping a latent concept value to text
WP_j	a weak label classification function for latent concept c_j
M^j	a profit matrix for latent concept c_j

Chapter 1

Introduction

Over the last decade and a half, progress in AI has accelerated tremendously. Much of the progress is owed to developments in *Deep Learning*, a machine learning paradigm based on *Neural Networks*. With Deep Learning, an autonomous system can improve its performance in a given task by learning from data, with minimal engineering effort. The success of this approach revolutionised many fields in AI, including Computer Vision, Natural Language Processing and Planning, among others (Pouyanfar et al., 2018). However, neural networks have been criticised due to their black-box nature, which reduces user trust, as well as their limited reasoning capabilities and data inefficiency (Ahmed et al., 2023).

Another paradigm in AI with a rich history and many successes is Logic-Based Symbolic Representation (Calegari et al., 2020). In this paradigm, a system stores its knowledge or beliefs in a declarative knowledge base, such as a First-Order Logic Program. These systems support high-level reasoning, by resolving queries against the knowledge base using logic-based inference. A highly successful Logic Programming language is *Answer Set Programming* (ASP; Falkner et al., 2018). ASP can be used to model many tasks by encoding knowledge, observations, rules, constraints and choice in First-Order formulas. Its support of default negation allows for non-monotonic reasoning, a prerequisite for common-sense reasoning. Over the last few decades highly efficient solvers have been developed for ASP (Gebser et al., 2012), resulting in its application in areas such as planning, verification and recommendation systems, among many. In addition, learning programs from small amounts of data is also supported (Law et al., 2014). While symbolic logic-based systems enjoy the benefit of being data efficient, fully interpretable and exact, they also come with some drawbacks: they tend to require a large amount of engineering effort. They are also relatively inefficient, as the exact solving procedure may be computationally expensive.

Combining these two approaches, neural and symbolic, can therefore be highly beneficial, as the advantages of both may be unlocked. To this end, research into *Neuro-Symbolic AI* has gained increased attention (Bhuyan et al., 2024). Systems that support both neural and logic-based representation & inference have been developed, although the combination has proven non-trivial. Approaches will typically involve approximation of logical inference (Minervini et al., 2018; Badreddine et al., 2022a; Dasaratha et al., 2023), which results in systems that are less opaque with fewer guarantees, or introduce a dependency on an exact symbolic solver (Manhaeve et al., 2021a; Tsamoura et al., 2021), with limited scalability. In particular, approaches involving ASP are typically of the latter type (Shakarian et al., 2023; Cunnington et al., 2023b), with scalable neuro-symbolic reasoning and learning yet to be realised.

In this work, we develop novel approaches for realising neuro-symbolic reasoning and learn-

ing at scale. Our focus is on methods that support non-monotonic logic-based reasoning with guaranteed correctness. We explore two approaches, the first using a new matrix-based representation of Logic Programs in vector spaces. The second is a completely new approach to neuro-symbolic AI, based on neural pre-training and the extraction of symbolic concepts from latent space. We demonstrate that this latter approach is successful in realising many of the goals of neuro-symbolic AI while being far more scalable than existing systems.

1.1 Logic-Based Representation & Inference

Symbolic Logic-Based AI seeks to represent knowledge through human-understandable symbols. This knowledge is often given in declarative form, such as First-Order logic. For example, a statement such as "it rained this morning and the grass is wet" can be expressed as the logical formula:

$$\text{observed_event}\,(\text{rain}, \text{morning}) \wedge \text{observed_state}\,(\text{grass}, \text{wet}) \tag{1.1}$$

One can also express declarative knowledge, such as:

$$\text{capital}\,(\text{France}, \text{Paris}) \wedge \text{capital}\,(\text{United Kingdom}, \text{London}) \tag{1.2}$$

or even formal rules, such as rules for how to compute sums:

$$\begin{aligned} &\text{sum}\,(2) \leftarrow \text{summand_1}\,(1) \wedge \text{summand_2}\,(1) \\ &\text{sum}\,(3) \leftarrow \text{summand_1}\,(1) \wedge \text{summand_2}\,(2) \\ &\qquad\vdots \end{aligned} \tag{1.3}$$

Using formal logic to represent knowledge provides several benefits:

- **Interpretable:** The knowledge the system possesses is written in human-understandable terms and is therefore readily available for inspection and for correction, if needed.
- **Cumulative:** Knowledge can be stored accurately and in full, for example in a database or in a program. It can therefore be added to without losing previous information, supporting lifelong learning.
- **Logic-based:** Inference can be carried out using the semantics of First-Order Logic. As this inference process is guaranteed to be exact, complex reasoning is naturally supported.
- **Verifiable:** As the knowledge is interpretable and expressed in an exact manner, it can be formally verified to ensure the accuracy and consistency of the system's performance.
- **Modular:** The declarative nature of the knowledge base allows it to be easily decomposed, transferred or put together with other knowledge bases, allowing systems to be reused for multiple purposes.
- **Data efficient:** Logic-based systems can make extensive use of background knowledge. Combined with their exact inference process, they support learning mechanisms that are extremely data efficient. Logic-based learning requires very few examples, at times as few as one, making them ideal in domains where data is sparse but background knowledge is available.

Logic-based inference comes in three different types:

- **Deduction:** Deduction is the process of deriving conclusions from premises and rules. For example, if the rule is "If it rained earlier, then the grass is wet" and the premise is "It rained this morning", then we can deduce "The grass is wet". Deduction, therefore, lets us extract knowledge that has only been known implicitly by following a sequence of logical steps, also known as theorem proving. AI systems can include a knowledge base that can be queried by a theorem prover, to deduce new facts.

- **Abduction:** Abduction involves finding underlying explanations to a given observation. For example, suppose we observe that the grass is wet. We know that if it rained this morning, the grass will be wet. We can therefore *abduce* the possibility that it rained this morning, as an explanation for why the grass is wet. This is not the only possible explanation, another might be that a sprinkling system was in use earlier. Abduction is an important aspect of an autonomous system's operation, as it allows it to reason about missing knowledge from observed facts, especially when said knowledge can no longer be observed. An example application is the diagnosis of a patient's illness from observed symptoms.

- **Induction:** Logic-based inference also allows us the induction of rules. For example, if we observe that every time it rained the grass becomes wet, we can induce the rule "When it rains, the grass becomes wet". This type of inference is linked to *learning* - it improves our ability to reason about new facts. Autonomous systems therefore benefit greatly from their ability to induce rules. The field of Inductive Logic Programming (Muggleton, 1991) realises induction through formal logic.

While logic-based representation & inference provide many benefits, it does come with several drawbacks:

- **Manual knowledge engineering:** While symbolic learning approaches such as ILP exist, much of the information in symbolic systems needs to be specified beforehand, either by the user or by the developer. Large knowledge bases have been constructed from various sources and tasks (Fellbaum, 1998; Auer et al., 2007; Speer et al., 2017; Krishna et al., 2017), which are used in many applications, but they are by no means exhaustive. The construction of a symbolic knowledge base may require significant human effort, depending on the complexity of the task.

- **Lack of perception:** Symbolic information is discrete and exact by its nature. In contrast, sensory information is continuous and often noisy. Representing such information in a symbolic system is not trivial. This can limit the applications of logic-based inference, unless sensory information is preprocessed first into symbolic form.

- **Dealing with uncertainty:** Logic-based representation and inference is exact, which means it comes with guarantees. However, information in the real world may be uncertain. An agent may not be sure of its premises or the rules that it has learned. First-order logic does not model such uncertainty, as this is typically the subject of probability theory. Probabilistic variants of logic exist, however (De Raedt et al., 2007a).

- **Scalability**: Symbolic solvers search for proofs to derive their conclusions. Finding these proofs can potentially involve a combinatorial search through a derivation tree, resulting in exponential complexity. This problem is especially pronounced in the abduction and induction cases, hurting the feasibility of knowledge and skill acquisition.

1.2 Neural Representation & Inference

One alternative approach to logic-based processing is *neural inference*. Neural networks are directed graphs, where nodes are referred to as *neurons* connected by edges which are typically weighted. Neurons have values that are transmitted to their neighbours. To update the value of a neuron, the (weighted) incoming values are typically summed and a non-linear activation function is applied. Neural inference is this process of information flow through the graph.

The weights of the graph need to be learned from data. A dataset of pairs (X, y) is typically given, where X is referred to as *features* and y as *labels*. Learning involves using this dataset to perform a search for the optimal weight parameters θ, a process known as *training*. A neural graph structure that has been found to be particularly effective is that of *stacked layers*, where each layer is a group of neurons. The first layer is known as the *input layer*, which is set with the value of the features. The last layer is the *output* layer, which is meant to predict the labels. All layers in between are known as *hidden*. Networks with multiple hidden layers are known as *deep* (giving rise to the term "Deep Learning").

The mappings between one layer to the next are simple, differentiable functions. The simple nature of the f_i functions ensures the inference process remains computationally efficient. In addition, the differentiable nature of these functions allows for gradient-based search algorithms to be performed during training, avoiding the computational complexity of combinatorial search. One can easily model a probability distribution using a neural network, as follows:

$$P_\theta\left(y|X\right) = f_{\theta_n} \circ f_{\theta_{n-1}} \circ \cdots \circ f_{\theta_2} \circ f_{\theta_1}\left(X\right) \tag{1.4}$$

In this manner, uncertainty in the predictions made by the system can be modelled. Neural systems represent their knowledge *implicitly in the model parameters*. This knowledge base can only be queried indirectly by performing inference. For example, to see which statements y may be true given observations X, we must compute $P_\theta\left(y|X\right)$ for the possible candidates, receiving the probability for each. Hence, neural-based AI sees knowledge as embedded in the inference process of autonomous systems, rather than explicitly declared.

Unlike most methods, Deep Learning does not require *feature engineering*. Most statistical methods assume the features X to be independent of one another. In practice, this assumption does not hold, requiring the features to be transformed to an independent set through a manually engineered preprocessing step. The compositional nature of neural networks allows them to learn this feature extraction stage directly from the data, a process known as "end-to-end" training (Goodfellow et al., 2016).

The reduced engineering effort greatly contributed to the popularity and success of Deep Learning. Neural networks have been applied in many domains, including computer vision, natural language processing, audio processing, reinforcement learning, time series modelling, planning, reasoning, and data generation, among others (Pouyanfar et al., 2018).

Deep Learning's success notwithstanding, the approach does suffer from several drawbacks:

- **Black Box:** As mentioned, all knowledge the model possesses is implicitly stored in its weights, and only emerges during inference. As such, our understanding of the inference process is limited, as we cannot translate its internal representations into human-understandable concepts. This severely limits our ability to verify the model or detect any misconceptions or bias.

- **Data inefficient:** Background knowledge cannot easily be incorporated into a neural network, and even when it is included, the model must learn how to use it. The result is

a training process that requires large amounts of data to provide good results. In general, the more parameters a model has, the more representation power it possesses, and the more data it requires to learn. For example, Llama 3, a state-of-the-art neural network, has 405 billion parameters and was trained on over 15 trillion tokens, equivalent to about 44 TB of text data.

- **Fixed:** Post-training, the parameters of the neural network remain fixed, preventing further learning. This limits the capability of the model to adapt to novel situations. In addition, training only new data results in *catastrophic forgetfulness* — as it adapts to new data, performance on the old data degrades. Therefore, the network must either be trained on the full dataset again (a costly process), or specialised to perform well only on the new data (a process known as *fine-tuning*).
- **No extrapolation:** The probability distribution learned by the neural network can model the training data's distribution well. But at inference time the data encountered may not come from this distribution. Or, it may represent rare cases that were not included in the training data. Neural networks generally show lacklustre performance on out-of-distribution data, limiting their application only to cases similar to those seen during training.
- **Approximate:** The statistical nature of the inference process means no strong guarantees can be provided on the correctness of the output. This is in contrast to how software is traditionally developed, where behaviour is enforced through its design and implementation. This is particularly troublesome in mission-critical applications, where random failures may result in catastrophic consequences, hurting user trust in AI.

1.3 Neuro-Symbolic Representation & Inference

The neural and symbolic approaches to knowledge representation & inference both have their advantages and disadvantages. Their strengths and weaknesses, however, are surprisingly complementary. This naturally gives rise to the idea of combining the two. This approach is referred to as *Neuro-Symbolic AI* (also referred to in the literature as Neural-Symbolic AI, NeSy AI or NSAI). It offers to unlock the advantages of both approaches:

- Interpretable symbolic knowledge and logic-based processes can be interleaved with neural representations to reduce or eliminate the black box nature of the latter.
- By using symbolic background knowledge and inference, data efficiency may be improved.
- The neural nature can be used to deal naturally with continuous sensory data, allowing the application of symbolic modules to a greater range of tasks.
- Uncertainty is more naturally modelled in symbolic systems using neural representations.
- Representing knowledge symbolically can help a neural network achieve lifelong learning while avoiding the need for further training on new data.
- Symbolic rules can help a neural network deal with out-of-distribution occurrences, improving the model's performance beyond its training.
- Exact symbolic computations provide guarantees that can increase trust in autonomous systems.

When solving tasks in neuro-symbolic settings, we are particularly interested in two cases:

- **Neuro-Symbolic Reasoning:** In this case, the symbolic knowledge relevant to the task is assumed to be fully given. It may have been manually engineered, retrieved from existing knowledge bases, or previously induced from data. The neural component is therefore the only component that still needs to be learned, where the symbolic knowledge is either beneficial or even crucial to the training's success.
- **Neuro-Symbolic Learning:** In this case, both the neural component and the symbolic knowledge are to be learned. The neural network will generally be in charge of mapping raw sensory data to symbolic concepts, while the induced symbolic knowledge would focus on higher-level reasoning. This will reduce the burden on the neural network, which now only needs to focus on learning the perception task, while the symbolic learner can induce rules from little data. As this case is more challenging than neuro-symbolic reasoning, it is less frequently considered, but it is potentially more useful as designing the system requires less engineering.

Research into neuro-symbolic AI began in the 1990s (Garcez et al., 2002) and has recently gained much traction (Garcez and Lamb, 2023), in large part due to the increased popularity of neural networks. A major challenge in integrating the two approaches is the dichotomy of the nature of the representation. Symbols are inherently discrete, as they form singular atoms of a given language. Neural networks are instead composed of continuous parameters and produce vectors in continuous space. Any attempt to combine the two representations must bridge the gap between the continuous and the discrete, do so in a manner that correctly translates between the two, and allow each type of representation to maintain its distinct advantages.

Approaches to bridging the gap generally fall into two categories. Most approaches attempt to approximate logical representations and reasoning in continuous space, using either fuzzy logic (Daniele and Serafini, 2019; Riegel et al., 2020; Badreddine et al., 2022b), probabilistic logic (Pryor et al., 2023; Dasaratha et al., 2023) or neural approximations (Minervini et al., 2018; Shi et al., 2019; Yang et al., 2023b). As these approaches support gradient-based learning, they are generally more scalable. But their approximate nature reduces both interpretability and guarantees of correctness. They also often support only a subset of First-Order logic, limiting the types of formulas they can represent. In particular, programs with recursion, default negation, hard and soft constraints and choice, are not supported, preventing many Answer Set Programs from being integrated into neural-symbolic models.

A second approach that has gained popularity recently maintains symbolic inference through a symbolic solver (Manhaeve et al., 2021a; Yang et al., 2020a; Tsamoura et al., 2021). Support for gradient-based training is accomplished by letting the symbolic solver compute a complex loss function over network predictions and labels (Xu et al., 2018). This function may involve full-scale reasoning, unlocking the full representation power of the symbolic language and exact inference. However, this comes at the price of reduced scalability, as the training signal for the network may be very expensive to compute. In practice, these approaches are limited to small-scale tasks, preventing their use outside research.

At present, a scalable approach to neuro-symbolic reasoning and learning with full symbolic representation capabilities has yet to be achieved. Such an approach should ideally allow any program that a symbolic solver (such as an ASP solver) supports. It should be capable of both neuro-symbolic reasoning and learning. It should be as close to exact logical computations as possible, to provide strong guarantees of correctness. And, crucially, it should scale as much as possible, in order to be of practical use.

1.4 Contributions

In this work, we introduce novel methods for realising more scalable neuro-symbolic reasoning and learning. In the following, we provide a summarised description of our main contributions.

1. **A novel matrix-based representation of normal logic programs to realise logic-based computations in vector spaces.**
   As a first step towards neural representations of logic programs, we introduce a novel characterisation of arbitrary normal logic programs in linear algebraic spaces. In this characterisation, program interpretations are mapped to 0-1 vectors and normal programs are mapped to matrices using the notion of a program reduct, in a one-to-one manner. The matrix allows us to perform deduction with default negation in vector spaces by realising the Immediate Consequence operator by means of matrix-vector multiplication and application of a non-linear operation. We provide proofs of the correctness of both the mapping and the operation.

2. **A differentiable approach to logic-based deduction that supports gradient-based optimisation**
   To move closer to neural computations, we relax our vector-based representation of vectors to allow continuous values, and introduce the notion of "semantic equivalence" to give semantic meaning to vectors with real values. A differentiable approximation of the Immediate Consequence operator is then introduced, allowing us to perform logical deduction in a differentiable manner. We perform an analysis for the conditions under which this approximation maintains correct two-valued semantics with guarantees, with proofs.

3. **A novel gradient-based algorithm for computing supported and stable models**
   To demonstrate the potential of differentiable deduction, we develop a new algorithm for computing stable and supported models of arbitrary normal logic programs. Our approach involves characterising supported models (which is a superset of stable models) as roots of a differentiable function. We use the Newton method to find these roots. Our notion of semantic equivalence allows us to map these roots back to symbolic models. We demonstrate the feasibility of the algorithm on programs containing positive loops, negative loops and/or choice.

4. **A novel neural network layer that encapsulates a normal logic program to support end-to-end training from downstream labels**
   We move towards neuro-symbolic reasoning by showing how our matrix-based approach can be incorporated into a logic program. We first modify our representation in order to relax some of the assumptions made previously. We show how a hierarchical normal logic program can be encapsulated into a sequence of neural layers to perform logic-based deduction differentiably. We refer to these layers as "Logic Program Layers". The notion of semantic equivalence remains here to guarantee the maintenance of two-valued semantics under certain conditions. We evaluate the approach on end-to-end neuro-symbolic reasoning tasks. On the MNIST addition task, we show this approach scales better than systems that were state-of-the-art at the time. On another task, Syllogism, we show our approach's success depends on its symbolic representation, with implications on scalability, suggesting a different approach may be required to be able to scale further.

5. **A gradient-based method for learning normal logic programs from program transitions**
   As a step towards neuro-symbolic learning, we show how our program matrix can be learned when symbolic knowledge is missing. We introduce a new approach for learning from interpretation vector transitions. We develop a novel loss function so that the matrix can be learned using gradient-based optimisation to realise the Immediate Consequence operation. We demonstrate the feasibility of the approach to learn normal logic programs using several toy tasks and a classification task. We analyse the advantages and disadvantages of our approach.

6. **A scalable approach to neuro-symbolic reasoning based on neural pre-training and clustered embeddings**
   Drawing on insights from our matrix-based approach, we develop a completely novel approach to scalable neuro-symbolic reasoning that broke away from all existing approaches at the time. We introduce the idea of training a two-stage neural network architecture, perception and reasoning, as a first step towards neuro-symbolic modelling. We show that post-training relevant symbolic concepts can be identified in latent space as clusters of vector embeddings. This insight allows us to lift the purely neural model into a symbolic one. We introduce an optimisation task that can correctly label these clusters with symbolic latent concepts, allowing a neuro-symbolic model to be constructed while maintaining the perception component. In evaluation, we show that our approach, Embed2Sym, not only achieves comparable accuracy to state-of-the-art neuro-symbolic reasoning systems, but it also scales orders of magnitude more. Embed2Sym is the first neuro-symbolic system to solve the MNIST Addition task on a large number of digits (namely, 15). We also show it supports learning in the presence of more complex perception (CIFAR-10) and can even induce missing knowledge (using the Forth Sort task).

7. **A scalable approach to neuro-symbolic learning based on weak-labelling and inductive learning**
   Finally, we show how Embed2Sym can be generalised to support neuro-symbolic learning of First-Order Answer Set Programs. We first consider how an Inductive Logic Programming system can be used to support both labelling of latent concept clusters and the induction of downstream task rules. We explain why this approach, although theoretically possible, will not scale in practice. We then introduce a novel approach of using a publicly available Vision-Language Model as a weak labeller of latent concepts. We then show how these weak labels can be used to find the optimal cluster labels. A state-of-the-art symbolic learner, ILASP 4, is used to learn the correct downstream rules in the presence of noisy predictions. We evaluate the resulting system, Embed2Rule, on several neuro-symbolic benchmarks involving complex reasoning, a large number of raw inputs or latent concepts. We demonstrate that Embed2Rule not only achieves good latent accuracy while inducing the correct rules, but it does so while being far more scalable than state-of-the-art neuro-symbolic reasoning and learning systems.

The results achieved here serve as a promising new direction for neuro-symbolic reasoning and learning, and have already inspired work by others (van Krieken et al., 2023; Daniele et al., 2024). This foundation can serve as a starting point for further research, as we seek to build scalable neuro-symbolic reasoning and learning systems with real-world applications.

1.4.1 List of Publications

Most of the work presented here has previously appeared in several publications:

- **Aspis Y, Broda K, Russo A, Lobo J.** Stable and supported semantics in continuous vector spaces. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning* 2020 Jul 9 (Vol. 17, No. 1, pp. 59-68).

  Covered by work presented in Chapter 3.

- **Aspis Y, Broda K, Lobo J, Russo A.** Embed2Sym - scalable neuro-symbolic reasoning via clustered embeddings. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning* 2022 Jul 30 (Vol. 19, No. 1, pp. 421-431).

  Covered by work presented in Chapter 5.

- **Aspis Y, Albinhassan M, Lobo J, Russo A.** Embed2Rule - scalable neuro-symbolic learning via latent space weak-labelling. In *International Conference on Neural-Symbolic Learning and Reasoning* 2024 Sep 9 (pp. 195-218).

  Covered by work presented in Chapter 6.

1.4.2 Structure of This Work

This work is structured as follows:

- Chapter 2 provides background information on topics relevant to this work. We cover topics in Logic Programming, including syntax, semantics and Answer Set Programming. We also cover topics in machine learning, including Inductive Logic Programming, Clustering and Artificial Neural Networks.

- Chapter 3 covers contributions 1-3, including our novel matrix representation, differentiable approximation of logical deduction, and a gradient-based algorithm for computing supported and stable models and its evaluation.

- Chapter 4 covers contributions 4-5. We introduce the theoretical framework for neuro-symbolic tasks. We go over the theory of Logic Program Layers before evaluating them in the reasoning case. Next, we show that the program matrix can be learned differentiably from data, and provide an initial evaluation on several tasks.

- Chapter 5 covers contribution 6. Our system, Embed2Sym, is introduced. The various stages of the algorithm are covered, including fully neural training, cluster identification, cluster labelling, and neuro-symbolic inference. Results on several neuro-symbolic benchmarks are presented to demonstrate the scalability and capabilities of the approach.

- Chapter 6 covers contribution 7. Our scalable system for neuro-symbolic learning, Embed2Rule, is introduced. We explain which pieces are missing in Embed2Sym to realise learning and how these can be achieved using weak labelling, cluster labelling and inductive learning. A systematic evaluation and comparison to other systems is then performed, confirming high accuracy and better scalability, and demonstrating the potential of this approach.

- Chapter 8 summarises the work and provides an overview of possible future research directions in neuro-symbolic reasoning and learning.

Chapter 2

Background

This chapter provides the reader with background information on topics required to understand the rest of the thesis. We first cover important topics in Logic Programming, starting with syntax and proceeding to semantics. In this work, we focus on the stable semantics of programs with default negation, which is the foundation of Answer Set Programming, a popular symbolic method for representing agent knowledge. In the second part of this chapter, we cover topics in machine learning. We explain Inductive Logic Programming, the paradigm for learning Logic Programs, and specifically focus on the learning of Answer Set Programs. We then proceed to cover topics in statistical machine learning, with particular focus on neural networks. We cover basic topics that apply to all neural networks, before delving deeper into some of the more popular architectures, including some of the more recent developments.

2.1 Logic Programming

Logic Programming is a programming paradigm aimed at expressing computation via the language of formal logic. A logic program comprises a series of statements (or rules), typically first-order logic formulas, in a declarative manner. The statements express knowledge about a domain of interest and can be used to store factual information or to assist in performing logic-based inference. Examples of programming languages include Prolog, Datalog and Answer Set Programming (ASP). In this work, we mostly employ Answer Set Programming as our language of choice. For the interested reader, a more thorough introduction to the theory of Logic Programming can be found in Hogger (1990). Answer Set Programming is covered in more detail by Gebser et al. (2012).

The typical structure for a rule in a programming language is of the abstract form:

$$\text{consequence} \leftarrow \text{condition} \tag{2.1}$$

which can be understood as an if-then statement: if the condition is true then the consequence follows.[1]

2.1.1 Language & Syntax

Symbols

A first-order theory (or formula), may include several types of symbols:

[1] Programming languages often use the ASCII-friendly ':-' characters in place of the ← symbol.

- Operators: $\leftarrow$ (implication), ',' (comma - conjunction), $\neg$ (classical negation), not (negation-as-failure, or default negation).
- Constants: typically denoted using lower-case letters. When given abstract names, we typically start with 'a'. For example $a, b, c, ...$
- Variables: typically denoted using upper-case letters. We typically start with the letter 'V' when given abstract names. For example $V, W, X, Y, ...$
- Functions: functions have a name and take arguments. The number of arguments is called the *arity* of the function. Function names are typically denoted using lower-case letters, starting with 'f'. For example $f(a), g(V, b)$. f is a function of arity 1 and g of arity 2. Constants can be thought of as functions of arity 0.
- Predicates: A predicate is denoted with a lower-case letter, and also takes arguments (and hence has an arity). When given an abstract name, we tend to start with 'p'. For example: $p, q, r, ...$

A function of arity n can also be referred to as an n-ary function, for example 0-ary or 1-ary.

Example 2.1.1. *Suppose we wish to model a family tree. The names of the people in the family will be our constants: alice, bob, charlie. We can introduce a function for the mother of a given person. For example mother(charlie) is a 1-ary function.*

Definition 2.1.1. *Two special predicates of arity 0 are $\bot$ and $\top$, representing false and true, respectively.*

Definition 2.1.2. *A term is one of the following:*

1. *A constant.*
2. *A variable.*
3. *An n-ary function $f(t_1, t_2, \ldots, t_n)$ where $t_1, t_2, \ldots, t_n$ are terms.*

A term is ground if it contains no variables.

For example, the following is a term: $f(g(a, V), X, h(b))$, which is not ground. The following are also non-ground terms: $g(a, V)$, V, X while a, $h(b)$ and b are ground terms.

Definition 2.1.3. *An atom is comprised of an n-ary predicate and n terms. An atom is ground if all its terms are ground.*

For example, $p(a, X)$ is a (non-ground) atom of arity 2 and $q(f(b, c))$ is a ground atom of arity 1 (whose term is a function of arity 2).

Example 2.1.2. *Returning to our family tree example, suppose we define a predicate parent of arity two that signifies that its first term is the parent of the second term. We can express the fact that Alice is the parent of Charlie by the atom parent(alice, charlie). We can express the fact that Bob is the parent of the mother of Charlie by the fact parent(bob, mother(charlie)). An even simpler example is a 1-ary predicate person which we can define for every person constant in our domain: person(alice), person(bob), person(charlie).*

Note that while functions and atoms are written similarly, their meaning is different. Functions are terms that represent values in our domain, for example, people, colours or numbers. Atoms represent statements that can be either true or false.

Negation

In logic programming, there are two types of negation: classical negation $\neg$ and negation-as-failure *not*. For an atom A, its classical negation $\neg A$ is the statement that "A is false". However, from a computational point of view, it is often intractable to demonstrate $\neg A$. This is because it often requires us to have complete knowledge about all possible ways of deriving A, and to assess that none of them successfully show A. If our knowledge of the world is incomplete, this may not be possible. For this reason, many programming languages, such as Prolog and ASP, introduce a *closed-world assumption*. Under this assumption, we have been given all the knowledge required about the problem domain to infer whether a given ground atom is true. Hence the statement "A is false" can be replaced with the statement "A has not been demonstrated to be true" - under the closed-world assumption this is enough to assume A is false. To distinguish between this type of negation and classical negation, a new symbol is introduced, *not*, known as negation-as-failure (as in, failure to demonstrate A means A is false). In this work we deal exclusively with negation-as-failure.

Rules

Definition 2.1.4. *A clause r is a rule of the form:*

$$h^r \leftarrow b_1^r, b_2^r, \ldots, b_n^r, not\ c_1^r, not\ c_2^r, \ldots, not\ c_m^r \tag{2.2}$$

where h^r is an atom (known as the head atom) and b_1^r, ..., b_n^r, and c_1^r, ..., c_m^r are atoms known collectively as the body of the rule. Specifically, we refer to the set $B^r = \{b_1^r, ..., b_n^r\}$ as the positive body and $\{c_1^r, \ldots, c_m^r\}$ as the negative body. A clause is ground if all its atoms are ground.

Definition 2.1.5. *A clause is definite if its negative body is the empty set.*

Definite clauses are useful because logic programs comprising definite clauses only (definite programs) have simpler semantics and time-efficient inference algorithms. However, there are some relations that are more practical to define using negation-as-failure.

Example 2.1.3. *The grandparent relation can be defined using the following definite clause:*

$$grandparent(X, Y) \leftarrow parent(X, Z), parent(Z, Y) \tag{2.3}$$

The sibling relationship is easier to define using negation-as-failure:

$$sibling(X, Y) \leftarrow parent(Z, X), parent(Z, Y), not\ same_person(X, Y) \tag{2.4}$$

where we can define:

$$same_person(X, X) \leftarrow person(X). \tag{2.5}$$

The alternative would have been to explicitly define a not_same_person relationship for every pair of people in the domain.

Definition 2.1.6. *A fact is a clause with an empty set for the body:*

$$h^r \leftarrow \tag{2.6}$$

Facts[2] represent the notion that the head is unconditionally true. For example, we can encode

[2]The ASCII-friendly way of writing facts is using '.'. For example $H.$ instead of $H \leftarrow$

the fact that Alice is a person:

$$\text{person}\,(alice) \leftarrow \tag{2.7}$$

Note that facts are definite clauses.

Definition 2.1.7. *A normal logic program is a set of clauses. A program is definite if it is comprised of definite clauses only. A program is ground if all of its rules are ground.*

The class of normal programs includes all definite programs. When speaking about a general normal logic program, we will often refer to it simply as a "program". When discussing definite logic programs we will mention this explicitly ("definite program").

Having introduced the syntax of First-Order logic, we turn our attention to a special case known as propositional logic.

Definition 2.1.8. *A 0-ary predicate is known as a propositional atom. A logic program containing only propositional atoms is known as a propositional program.*

Propositional logic admits no constants, variables or functions. An example of a propositional program is:

$$\begin{aligned} p &\leftarrow q, \text{not } r \\ q &\leftarrow s \\ s &\leftarrow \end{aligned} \tag{2.8}$$

2.1.2 Semantics of Propositional Logic

We will define the semantics of logic programs through the propositional case. We will later see that the semantics of first-order programs emerges naturally from the propositional case. In the rest of this section, all programs are assumed to be propositional. The theorems in this section were formulated and proven in works such as van Emden and Kowalski (1976), Apt et al. (1988), Gelfond and Lifschitz (1988) and Fages (1990). In this section, we provide our own formulation and proofs, and cite the original work where the theorem first appeared.

Definition 2.1.9. *The Herbrand base of a propositional logic program P, denoted B_P, is the set of all atoms in P. More formally:*

$$B_P = \bigcup_{r \in P} \{h^r\} \cup B^r \cup C^r \tag{2.9}$$

Example 2.1.4. *For the program in equation 2.8, the Herbrand base is:*

$$B_P = \{p, q, s, r\} \tag{2.10}$$

Definition 2.1.10. *An interpretation of a program P is a subset of its Herbrand base B_P.*

Interpretations are usually denoted by I. Besides being sets of atoms, interpretations can also be thought of as assignments of truth values to the atoms of the Herbrand base. That is, if an atom $p \in I$, then p has been assigned the value "True". Otherwise, p is "False".

Example 2.1.5. *Some interpretations of the program in equation 2.8 are:*

$$\emptyset, \{r\}, \{p, s\}, \{p, q, r, s\} \tag{2.11}$$

Definition 2.1.11. *The Immediate Consequence operator T_P of a logic program P is a mapping from interpretation to interpretation defined as follows:*

$$T_P : 2^{B_P} \to 2^{B_P} \tag{2.12}$$

$$T_P(I) = \{h^r \mid r \in P, B^r \subseteq I, C^r \cap I = \emptyset\} \tag{2.13}$$

Example 2.1.6. *For the program in equation 2.8, we have:*

$$\begin{aligned} T_P(\emptyset) &= \{s\} \\ T_P(\{q\}) &= \{p, s\} \\ T_P(\{p, q, s\}) &= \{p, q, s\} \\ T_P(\{q, r\}) &= \{s\} \end{aligned} \tag{2.14}$$

Definition 2.1.12. *An interpretation I satisfies a rule r if $B^r \subseteq I$ and $C^r \cap I = \emptyset$ implies $h^r \in I$.*

Note that an interpretation I satisfies the rule r if $B^r \not\subseteq I$ or $C^r \cap I \neq \emptyset$. In such a case, we say that I trivially satisfies r.

Definition 2.1.13. *An interpretation M is called a model of a logic program P if M satisfies every rule in P.*

This notion of a program's model is sometimes referred to as a "classical model".

Theorem 2.1.1. *Let P be a normal logic program and M an interpretation. Then M is a model of P if and only if $T_P(M) \subseteq M$. (van Emden and Kowalski, 1976)*

Proof. Suppose M is a model of P. Let $a \in T_P(M)$. Then, there exists $r \in P$ such that $h^r = a$, $B^r \subseteq M$ and $C^r \cap M = \emptyset$. But, as M is a model, M satisfies r hence $h^r = a \in M$. Conversely, suppose $T_P(M) \subseteq M$. Let $r \in P$. We need to show that M satisfies r. If $B^r \not\subseteq M$ or $C^r \cap M \neq \emptyset$ then M trivially satisfies r. Otherwise, $h^r \in T_P(M) \subseteq M$, and M still satisfies r. □

Definition 2.1.14. *A model M of a logic program P is said to be minimal if there does not exist a model M' of P such that $M' \subsetneq M$.*

Example 2.1.7. *The following program:*

$$\begin{aligned} p &\leftarrow q \\ q &\leftarrow p \end{aligned} \tag{2.15}$$

has two models: $\emptyset$ and $\{p, q\}$. $\emptyset$ is minimal but not $\{p, q\}$.

Example 2.1.8. *The following program:*

$$\begin{aligned} p &\leftarrow not\ q \\ q &\leftarrow not\ p \end{aligned} \tag{2.16}$$

has three models: $\{p\}$, $\{q\}$ and $\{p, q\}$. $\{p\}$ and $\{q\}$ are minimal but not $\{p, q\}$.

Definition 2.1.15. *The powers of the Immediate Consequence operator are defined inductively for all $k \geq 0$ as follows:*

$$T_P^0(I) = I \tag{2.17}$$

$$T_P^{k+1}(I) = T_P\left(T_P^k(I)\right) \tag{2.18}$$

Theorem 2.1.2. *For a definite program P and for all $k \geq 0$:*

$$T_P^k(\emptyset) \subseteq T_P^{k+1}(\emptyset) \tag{2.19}$$

(van Emden and Kowalski, 1976)

Proof. By induction over k. For $k = 0$, we trivially have:

$$T_P^0(\emptyset) = \emptyset \subseteq T_P^1(\emptyset) \tag{2.20}$$

Suppose, as induction hypothesis, that for $k \geq 0$ we have $T_P^k(\emptyset) \subseteq T_P^{k+1}(\emptyset)$. Let $a \in T_P^{k+1}(\emptyset)$. Then there exists a (definite) rule $r \in P$ such that $a = h^r$ and $B^r \subseteq T_P^k(\emptyset)$. By assumption $B^r \subseteq T_P^k(\emptyset) \subseteq T_P^{k+1}(\emptyset)$. Then by definition of T_P we have $a = h^r \in T_P\left(T_P^{k+1}(\emptyset)\right) = T_P^{k+2}(\emptyset)$. □

Note that the theorem does not hold for the general case of normal programs. For example, successive applications of T_P for Program 2.16 results in the infinite chain: $\{\emptyset\}$, $\{p, q\}$, $\{\emptyset\}$, $\{p, q\}$, ...

Theorem 2.1.2 shows that if the Herbrand base of a definite program P is finite, there exists some $k \geq 0$ such that $T_P^k(\emptyset) = T_P^{k+1}(\emptyset)$. To see this, consider the sequence:

$$T_P^0(\emptyset) \subseteq T_P^1(\emptyset) \subseteq T_P^2(\emptyset) \subseteq T_P^3(\emptyset) \subseteq \ldots \tag{2.21}$$

and that for all $k \geq 0$, $T_P^k(\emptyset) \subseteq B_P$ (by definition). As B_P is finite and an upper bound of $T_P^k(\emptyset)$, the chain of subset operations must eventually become an equality. For such a k, the interpretation $T_P^k(\emptyset)$ is called a fixed point of T_P. We denote this fixed point as $T_P^\infty(\emptyset)$.

Since $T_P\left(T_P^\infty(\emptyset)\right) = T_P^\infty(\emptyset)$, by theorem 2.1.1, $T_P^\infty(\emptyset)$ is always a model of P. In fact, it is contained in every model of P, as shown by the next theorem.

Theorem 2.1.3. *Let P be a definite logic program and M a model of P. Then $T_P^\infty(\emptyset) \subseteq M$. (van Emden and Kowalski, 1976)*

Proof. Let $a \in T_P^\infty(\emptyset)$. There exists a minimal integer k such that $a \in T_P^k(\emptyset)$. We will show that $a \in M$ by induction over k. For $k = 1$, $a \in T_P(\emptyset) = \{h^r \mid r \in P, B^r \subseteq \emptyset\}$. In other words, there exists a fact $r \in P$ such that $h^r = a$. For r we have $B^r = \emptyset \subseteq M$, and as M is a model we have $a \in M$. Suppose for all $l < k$, we have $T_P^l(\emptyset) \subseteq M$. Since $a \in T_P^k(\emptyset) = T_P\left(T_P^{k-1}(\emptyset)\right) = \left\{h^r \mid r \in P, B^r \subseteq T_P^{k-1}(\emptyset)\right\}$. Therefore, there exists $r \in P$ such that $h^r = a$ and $B^r \subseteq T_P^{k-1}(\emptyset) \subseteq M$. Since M is a model, M satisfies r and therefore $a \in M$. □

A consequence of the above theorem is that for every definite program P, $T_P^\infty(\emptyset)$ is a unique minimal model. This model is often referred to as the "least model" or "least Herbrand model" and denoted $LHM(P)$. It is relatively straightforward to compute $LHM(P)$, by means of the following algorithm: Set $I = \emptyset$, then compute $I = T_P(I)$ until $I = T_P(I)$.

The semantics of definite programs is therefore rather straightforward. The least model of the program represents all the conclusions that can be derived. However, the general case of

normal programs is more complex. Consider again the program in equation 2.16. Recall its models are $\{p\}$, $\{q\}$ and $\{p, q\}$. It therefore does not have a unique minimal model. Intuitively, the models $\{p\}$ and $\{q\}$ are preferable to the model $\{p, q\}$, since $\{p, q\}$ cannot be derived while assuming both p and q are true. In fact, $\{p, q\}$ is only a model because it does not satisfy the body of any rule. Under such circumstances, p and q represent assumptions unsupported by any observations. Given this, it may seem like minimal models should be preferred, but even this is not always the case. Consider:

$$p \leftarrow \text{not } p \tag{2.22}$$

This program has only a single model, $\{p\}$, which is minimal. However, just as in the previous case, p represents an unsupported assumption. We should therefore demand that every atom in a model has a justification. This is captured by the notion of supported models.

Definition 2.1.16. *Let P be a normal logic program. An interpretation M is a supported model of P if:*

1. *M is a model of P.*
2. *For every $a \in M$ there exists a rule $r \in P$ such that $a = h^r$, $B^r \subseteq M$, $C^r \cap M = \emptyset$.*

Supported models are also given by the classical models of the Clark's Completion of a program (Clark, 1978; Apt et al., 1988).

Example 2.1.9. *The program in equation 2.16 has two supported models, $\{p\}$ and $\{q\}$, both of which are minimal.*

Example 2.1.10. *The program in equation 2.15 has two supported models, $\emptyset$ and $\{p, q\}$. Only $\emptyset$ is minimal.*

Example 2.1.11. *The program in equation 2.22 has no supported models.*

Theorem 2.1.4. *Let P be a normal program and M an interpretation. Then M is a supported model of P if and only if $T_P(M) = M$. (Apt et al., 1988)*

Proof. Suppose M is a supported model. Since M is a model, by theorem 2.1.1 we have $T_P(M) \subseteq M$ and we need to show that $M \subseteq T_P(M)$. Suppose $a \in M$. Since M is supported, there exists $r \in P$ such that $a = h^r$, $B^r \subseteq M$ and $C^r \cap M = \emptyset$. Therefore $a \in T_P(M)$. Conversely, suppose $T_P(M) = M$. From theorem 2.1.1, M is a model. Let $a \in M$. Then $a \in T_P(M) = \{h^r \mid r \in P, B^r \subseteq M, C^r \cap M = \emptyset\}$. Then, M is supported. □

The supported models of a program are therefore exactly the fixed points of T_P. It might therefore seem that minimal supported models of a normal program form a reasonable semantics. Unfortunately, this is not the case either. Consider:

$$\begin{aligned} p &\leftarrow p \\ q &\leftarrow \text{not } p \end{aligned} \tag{2.23}$$

This program has two supported models, $\{p\}$ and $\{q\}$, both of which are minimal. However, the supported model $\{p\}$ again seems unjustifiable, as p is only supported due to circular reasoning: p is concluded to be true only because it was assumed to be true. In contrast, $\{q\}$ does

not suffer from this problem, it has justification external to itself. An even stronger notion than minimal supported models is therefore required. This is successfully captured by the notion of a *stable model*, first introduced by Gelfond and Lifschitz (1988).

Definition 2.1.17. *Let P be a normal program and I an interpretation. The program reduct of P with respect to I, denoted P^I, is the following definite program:*

$$P^I = \{h^r \leftarrow b_1^r, b_2^r, \ldots, b_{n^r}^r \mid r \in P, C^r \cap I = \emptyset\} \tag{2.24}$$

Less formally, P^I is constructed from P using the following process: for every rule $r \in P$, if some atom $c \in C^r$ is also in I, r is removed from P. Otherwise, r remains but its negative body is removed. The result is therefore a definite program, which has a least Herbrand model.

Definition 2.1.18. *M is a stable model of P if $M = LHM\left(P^M\right)$*

Example 2.1.12. *Let P be the following program:*

$$\begin{aligned} p &\leftarrow r, not\ q \\ q &\leftarrow not\ r \\ r &\leftarrow not\ q \\ s &\leftarrow t \\ t &\leftarrow s \end{aligned} \tag{2.25}$$

If $I = \{r\}$, then P^I is:

$$\begin{aligned} p &\leftarrow r \\ r &\leftarrow \\ s &\leftarrow t \\ t &\leftarrow s \end{aligned} \tag{2.26}$$

which has a least Herbrand model of $\{p, r\}$. Therefore, $\{r\}$ is not a stable model. In fact, for $I = \{p, r\}$ we have the same program reduct. Therefore $\{p, r\}$ is a stable model of P.

If $I = \{q\}$ then P^I is:

$$\begin{aligned} q &\leftarrow \\ s &\leftarrow t \\ t &\leftarrow s \end{aligned} \tag{2.27}$$

for which the least Herbrand model is I. Therefore, $\{q\}$ is another stable model of P.

If $I = \{p, q, r, s\}$ then P^I is:

$$\begin{aligned} s &\leftarrow t \\ t &\leftarrow s \end{aligned} \tag{2.28}$$

for which the empty set is the least Herbrand model. In fact, it can be seen that s and t can never be part of any stable model since they have no external support.

The intuition behind stable models can be explained as follows: suppose an agent forms the belief that a set of atoms M is true. If this agent then applies this belief to its knowledge base

P and follows all steps of deduction reaches the conclusion that M is exactly the set of all true atoms, then M is considered to be a "stable" view of the world.

Note that every definite program has a unique stable model: its least Herbrand model. This is because for a definite program P and every interpretation I, $P^I = P$. Therefore $LHM(P) = LHM\left(P^{LHM(P)}\right)$. In general, a normal program may have zero, one or multiple stable models. The program 2.22 has no stable models. The program 2.8 has one: $\{p, q, s\}$. The program 2.16 has two: $\{p\}$ and $\{q\}$. Stable models allow us to model problem domains where multiple possibilities may occur at any given moment. For example, the result of tossing a coin may have two possible outcomes. This can be modelled by a program such as the one in equation 2.16, where for example p may represent the statement "the result is heads" while q represents the statement "the result is tails".

We next show that stable models are a subclass of supported models. First, we show that applying the Immediate Consequence operator for the reduct is equivalent to applying it for the full program.

Theorem 2.1.5. *Let P be a normal logic program and I an interpretation. Then $T_{P^I}(I) = T_P(I)$.*

Proof.

$$\begin{aligned} a \in T_{P^I}(I) \quad &\Leftrightarrow \quad \exists r \in P^I, h^r = a, B^r \subseteq I \\ &\Leftrightarrow \quad \exists r \in P, h^r = a, B^r \subseteq I, C^r \cap I = \emptyset \quad \Leftrightarrow \quad a \in T_P(I) \end{aligned} \tag{2.29}$$

□

Theorem 2.1.6. *Let P be a normal logic program and M a stable model of P. Then M is a supported model of P.*

Proof. We have:

$$T_P(M) = T_{P^M}(M) = T_{P^M}\left(LHM\left(P^M\right)\right) = LHM\left(P^M\right) = M \tag{2.30}$$

and from theorem 2.1.4, M is a supported model. □

Theorem 2.1.7. *Let P be a normal program and M a stable model. Then M is a minimal model of P. (Gelfond and Lifschitz, 1988)*

Proof. Let $M' \subseteq M$ be a model of P. To show that M is minimal, we need to prove that $M' = M$. To do this, we will show that M' is a model of P^M and therefore, since M is a minimal model of P^M (being its least Herbrand model), it follows that $M = M'$ as required.

Let $r \in P^M$. We need to show that M' satisfies r. If $B^r \not\subseteq M'$ then M' satisfies r trivially. Otherwise, $B^r \subseteq M'$ and we need to show that $h^r \in M'$. Since $r \in P^M$, from the definition of the Program Reduct, there exists a rule r' in P from which r was derived. It holds that $h^r = h^{r'}$ and $B^r = B^{r'}$. If $C^{r'} \cap M' \neq \emptyset$ there exists some $a \in C^{r'} \cap M' \subseteq C^{r'} \cap M$. But then, from the definition of Program Reduct, $r \notin P^M$ which is a contradiction. Therefore, $C^{r'} \cap M' = \emptyset$. Since M' is a model of P, $B^r \subseteq M'$, $C^r \cap M' = \emptyset$, it follows that $h^{r'} = h^r \in M'$ and therefore M' satisfies r. Therefore, M' is a model of P^M, as required. □

Therefore stable models are minimal supported models while the converse is not true. Recall that what prevents a supported model from being stable is circular logic: an atom can only be

shown to be true if it is assumed to be true. In other words, the atom has no "external support". This can be formalised by the notion of the positive dependency graph of a program.

Definition 2.1.19. *Let P be a normal logic program. The positive dependency graph of P is the directed graph $G = \left(B^P, E\right)$ where:*

$$E = \{(b, h^r) \mid r \in P, b \in B^r\} \tag{2.31}$$

The positive dependency graph is a graph where all the atoms of the program are nodes, and there is a directed edge from every atom that appears in a positive body of a rule to the head of that rule. We can equivalently define the negative dependency graph where B^r is replaced with C^r. In other words, where there is an edge from an atom in the negative body to the head of the rule.

Definition 2.1.20. *A positive (resp. negative) loop in a program P is a (non-trivial[3]) cycle in its positive (resp. negative) dependency graph.*

Positive loops can create supported but non-stable models, while negative loops can cause programs to have zero or multiple stable models.

Example 2.1.13. *The program in equation 2.15 has a positive loop $\{(p,q),(q,p)\}$. $\{p,q\}$ is a supported but non-stable model.*

Example 2.1.14. *The program in equation 2.16 has a negative loop $\{(p,q),(q,p)\}$ and two stable models $\{p\}$ and $\{q\}$.*

Example 2.1.15. *The program in equation 2.22 has a negative loop $\{(p,p)\}$ and no stable models.*

Positive loops do not necessarily create non-stable supported models.

Example 2.1.16. *The program:*

$$\begin{aligned} p &\leftarrow p, r \\ q &\leftarrow not\, r \end{aligned} \tag{2.32}$$

has a positive loop $\{(p,p)\}$. However, it does not have a supported but non-stable model. It has a single stable model $\{q\}$.

The concept of "stable models as supported models without positive loops" is captured formally in the notion of well-supported models (Fages, 1990).

Definition 2.1.21. *Let P be a normal logic program and M a supported model. M is well-supported if there exists a well-founded partial ordering $\prec$ of the atoms in I such that for all $r \in P$ and for all $b \in B^r$, $b \prec h^r$.*

Example 2.1.17. *Consider again the program:*

$$\begin{aligned} p &\leftarrow p \\ q &\leftarrow not\, p \end{aligned} \tag{2.33}$$

$\{p\}$ is not a well-supported model since it would require a partial ordering where $p \prec p$. However, $\{q\}$ is a well-supported model since there is no need to include p in the ordering.

[3] A trivial cycle in a graph is a path of length 0. That is, a single vertex and no edges.

Example 2.1.18. *Consider the program in equation 2.25. The supported model $\{p, r, s, t\}$ is not well-supported since a partial ordering $\prec$ would require both $s \prec t$ and $t \prec s$. Therefore, s and t cannot be part of a well-supported model. $\{p, r\}$ is well-supported, as is $\{q\}$.*

Theorem 2.1.8. *Let P be a normal program and M an interpretation. M is a stable model if and only if M is a well-supported model.*

The proof is omitted but given in Fages (1990).

2.1.3 Semantics of First-Order Programs

Definition 2.1.22. *Let P be a first-order program. The Herbrand universe of P, denoted H_P is the set of all ground terms that can be obtained from the terms of P. Formally, suppose P is defined over the set of constants C and the set of functions F, then H_P is constructed as follows:*

- $C \subseteq H_P$.
- *If $f \in F$ and $t_1, t_2, \ldots, t_n \in H_P$, then $f(t_1, t2, \ldots, t_n) \in H_P$*

H_P is therefore the set of all ground terms that can be constructed from constants and function symbols in P.

Example 2.1.19. *Suppose a program P contains the constants $alice, bob, charlie$ and the 1-ary function $mother(\cdot)$. Then the Herbrand universe is:*

$$\begin{aligned} &alice, bob, charlie, \\ &mother(alice), mother(mother(alice)), mother(mother(mother(alice))), \ldots \\ &mother(bob), mother(mother(bob)), mother(mother(mother(bob))), \ldots \\ &mother(charlie), mother(mother(charlie)), mother(mother(mother(charlie))), \ldots \end{aligned} \tag{2.34}$$

Definition 2.1.23. *Let P be a first-order program, and $r \in P$. Let T be the set of terms that appear in r and V the set of variables in r. A ground substitution θ_r is a function that maps terms in r to ground terms, while acting as an identity function on ground terms. Formally:*

$$\begin{aligned} \theta_r &: T \to H_P \\ \forall t \in T \setminus V \quad &\theta_r(t) = t \end{aligned} \tag{2.35}$$

We define the application of a ground substitution θ_r to atoms in r:

$$\theta_r(p(t_1, t_2, \ldots, t_n)) = p(\theta_r(t_1), \theta_r(t_2), \ldots, \theta_r(t_n)) \tag{2.36}$$

And to the entire rule r:

$$\begin{aligned} &\theta_r(h^r \leftarrow b_1^r, b_2^r, \ldots, b_n^r, not\ c_1^r, not\ c_2^r, \ldots, not\ c_m^r) = \\ &\quad \theta_r(h^r) \leftarrow \theta_r(b_1^r), \theta_r(b_2^r), \ldots, \theta_r(b_n^r), not\ \theta_r(c_1^r), not\ \theta_r(c_2^r), \ldots, not\ \theta_r(c_m^r) \end{aligned} \tag{2.37}$$

We denote Θ_r as the set of all ground substitutions for r.

Example 2.1.20. *Let P be the following program:*

$$\begin{aligned} person\left(alice\right) &\leftarrow \\ person\left(bob\right) &\leftarrow \\ person\left(charlie\right) &\leftarrow \\ parent\left(bob, alice\right) &\leftarrow \\ parent\left(alice, charlie\right) &\leftarrow \\ same_person\left(X, X\right) &\leftarrow person\left(X\right) \\ grandparent\left(X, Y\right) &\leftarrow parent\left(X, Z\right), parent\left(Z, Y\right) \end{aligned} \tag{2.38}$$

We have $H_P = \{alice, bob, charlie\}$.
For the rule $r = grandparent\left(X, Y\right) \leftarrow parent\left(X, Z\right), parent\left(Z, Y\right)$, we have $T = \{X, Y, Z\}$. Define:

$$\theta_r\left(X\right) = bob \quad \theta_r\left(Y\right) = charlie \quad \theta_r\left(Z\right) = alice \tag{2.39}$$

then:

$$\theta_r\left(r\right) = grandparent\left(bob, charlie\right) \leftarrow parent\left(bob, alice\right), parent\left(alice, charlie\right) \tag{2.40}$$

Definition 2.1.24. *Let P be a logic program. The ground version (or grounding) of a rule $r \in P$, denoted $ground\left(r\right)$, is defined as the following set of rules:*

$$ground\left(r\right) = \{\theta_r\left(r\right) | \theta_r \in \Theta_r\} \tag{2.41}$$

Definition 2.1.25. *Let P be a logic program. The ground version (or grounding) of P, denoted $ground\left(P\right)$, is defined as the following program:*

$$ground\left(P\right) = \bigcup_{r \in P} ground\left(r\right) \tag{2.42}$$

The ground version of a first-order program can be intuitively thought of as the program received by replacing variables in rules by all their possible instantiations.

Example 2.1.21. *If P is Program 2.38, then $ground(P)$ is:*

$$\begin{aligned}
person(alice) &\leftarrow \\
person(bob) &\leftarrow \\
person(charlie) &\leftarrow \\
parent(bob, alice) &\leftarrow \\
parent(alice, charlie) &\leftarrow \\
same_person(alice, alice) &\leftarrow person(alice) \\
same_person(bob, bob) &\leftarrow person(bob) \\
same_person(charlie, charlie) &\leftarrow person(charlie) \\
grandparent(alice, alice) &\leftarrow parent(alice, alice), parent(alice, alice) \\
grandparent(alice, alice) &\leftarrow parent(alice, bob), parent(bob, alice) \\
grandparent(alice, alice) &\leftarrow parent(alice, charlie), parent(charlie, alice) \\
grandparent(alice, bob) &\leftarrow parent(alice, alice), parent(alice, bob) \\
grandparent(alice, bob) &\leftarrow parent(alice, bob), parent(bob, bob) \\
grandparent(alice, bob) &\leftarrow parent(alice, charlie), parent(charlie, bob) \\
&\vdots
\end{aligned} \tag{2.43}$$

As can be seen, the ground version of a program can be substantially larger than its non-ground counterpart, motivating the need for first-order programs. Note that as a standard, variables in rules are "universally quantified". Intuitively, this means the rule must hold for any grounding of its variables. For example $p(X, Y) \leftarrow q(X, Y)$ holds for any values replacing X and Y.

Note that once a program is grounded, every atom acts as a 0-ary predicate. Because of this, ground programs are in fact propositional programs (more accurately, they are isomorphic to propositional programs). The semantics of a first-order program P are the semantics of its ground version $ground(P)$. As a result, all definitions and theorems from the propositional case apply here as well. For example, the Herbrand base of a first-order program is the Herbrand base of its ground program.

Example 2.1.22. *For the program in equation 2.38 (and its ground version), the Herbrand base is:*

$$\begin{aligned}
B_P = \{&person(alice), person(bob), person(charlie), \\
&same_person(alice, alice), same_person(alice, bob), same_person(alice, charlie), \ldots \\
&parent(alice, alice), parent(alice, bob), parent(alice, charlie), parent(bob, alice), \ldots \\
&grandparent(alice, alice), grandparent(alice, bob), grandparent(alice, charlie), \ldots\}
\end{aligned} \tag{2.44}$$

There is one notable caveat, however: in general, there is no guarantee that $ground(P)$ is finite. Note that as long as the program contains one constant and one function symbol (of arity 1 or more), the Herbrand universe is infinite, and hence the program's grounding is infinite. For this reason, programming languages such as Datalog use a function-free variant of first-order

logic. Other languages, like ASP, allow function symbols but use other methods to prevent infinite grounding, which we will discuss later. In this work we will assume $ground(P)$ is always finite, and therefore all results from the previous section apply.

2.1.4 Answer Set Programming

Answer Set Programming (ASP) is a logic programming paradigm, aimed at solving computationally difficult problems. ASP extends the syntax of first-order normal logic programs to support the representation of rich declarative knowledge, and to facilitate formal non-monotonic reasoning. This makes ASP particularly suitable not only for optimisation problems, but also for common-sense reasoning, which requires default negation to deal with incomplete and changing information. ASP has been applied in practice to a large variety of problem domains, including optimisation, planning, bioinformatics, robotics, diagnosis, classification, question answering (QA) and many more (Falkner et al., 2018).

At the heart of ASP is the stable semantics of normal logic programs. A problem domain is represented through a series of normal rules as well as additional constructs such as *constraints* and *choice rules*. The stable models of the resulting program (referred to as *Answer Sets* within the context of ASP) correspond to solutions of the original problem. To compute the Answer Sets, an *Answer Set solver* is employed. The existence of efficient answer set solvers allowed ASP to grow in popularity and become widely used.

In the following, we give a short introduction to some of the extensions ASP provides that we employ in this work. For a complete overview of ASP and how it can be used in practice, see Gebser et al. (2012).

Definition 2.1.26. *A choice rule r is of the form:*

$$l\,\{h_1^r, h_2^r, \ldots, h_{k^r}^r\}\,u \leftarrow b_1^r, b_2^r, \ldots, b_{n^r}^r, not\ c_1^r, not\ c_2^r, \ldots, not\ c_{m^r}^r \tag{2.45}$$

where l, u are non-negative integers such that $l \leq u$ and $h_1^r, h_2^r, \ldots, h_{k^r}^r$ are atoms known collectively as the choice head of the rule. When l is omitted, it is implied that $l = 0$ and when u is omitted, it is implied that $u = k^r$.

Definition 2.1.27. *An interpretation I satisfies a (ground[4]) choice rule r if $B^r \subseteq I$ and $I \cap C^r = \emptyset$ implies $l \leq I \cap \{h_1^r, h_2^r, \ldots, h_{l^r}^r\} \leq u$.*

Example 2.1.23. *The statement "a flipped coin is either heads or tails" can be encoded using a choice rule as follows:*

$$1\,\{heads(X), tails(X)\}\,1 \leftarrow coin(X) \tag{2.46}$$

where the choice rule ensures that for a coin, either heads or tails is true, but not both.

Intuitively, a choice rule states that when its body is true, between l and u atoms from the choice head must be chosen to be true. For example, if $l = u = 1$, exactly one atom must be chosen. Choice rules are syntactic sugar and can be compiled into a set of normal rules that is quadratic in k^r. For the special case where $l = 0$ and $u = k^r$ there exists a transformation linear in size, which we will show here. For such a choice rule r:

$$0\,\{h_1^r, h_2^r, \ldots, h_{k^r}^r\}\,k^r \leftarrow b_1^r, b_2^r, \ldots, b_{n^r}^r, \text{not } c_1^r, \text{not } c_2^r, \ldots, \text{not } c_{m^r}^r \tag{2.47}$$

[4]The semantics of an Answer Set Program is again given by the semantics of its ground version.

introduce $k^r + 1$ new symbols that are not in B_P: $d^r, g_1^r, g_2^r, \ldots, g_{k^r}^r$. Then r is replaced with the following set of rules:

$$
\begin{aligned}
d^r &\leftarrow b_1^r, b_2^r, \ldots, b_{n^r}^r, \text{not } c_1^r, \text{not } c_2^r, \ldots, \text{not } c_{m^r}^r \\
h_1^r &\leftarrow d^r, \text{not } g_1^r \\
g_1^r &\leftarrow \text{not } h_1^r \\
h_2^r &\leftarrow d^r, \text{not } g_2^r \qquad (2.48) \\
g_2^r &\leftarrow \text{not } h_2^r \\
&\vdots \\
h_{k^r}^r &\leftarrow d^r, \text{not } g_{k^r}^r \qquad (2.49) \\
g_{k^r}^r &\leftarrow \text{not } h_{k^r}^r \\
& \qquad (2.50)
\end{aligned}
$$

Intuitively, d^r represents the statement that the body of r is true, and g_i^r represents the negation of h_i^r. The rules enforce that if d^r is true, either h_i^r or g_i^r is true, but not both. The transformation for the general case can be found in Gebser et al. (2012).

Definition 2.1.28. *A hard constraint (or simply constraint) r is of the form:*

$$\leftarrow b_1^r, b_2^r, \ldots, b_n^r, \textit{not } c_1^r, \textit{not } c_2^r, \ldots, \textit{not } c_m^r \qquad (2.51)$$

Hard constraints have no head atom and intuitively represent conditions that should never be true. Formally, if I satisfies the body of a constraint, it is not a model. Constraints can be used to rule out undesirable solutions. A hard constraint is also syntactic sugar: the constraint above can be replaced by a single normal rule by introducing a new symbol $d^r \notin B_P$ and writing:

$$d^r \leftarrow b_1^r, b_2^r, \ldots, b_n^r, \text{not } c_1^r, \text{not } c_2^r, \ldots, \text{not } c_m^r, \text{not } d^r \qquad (2.52)$$

the resulting rule ensures that when an interpretation I satisfies the positive body $b_1^r, b_2^r, \ldots, b_n^r$ and also satisfies not $c_1^r, \text{not } c_2^r, \ldots, \text{not } c_m^r$, then it is not an answer set: if $d^r \notin I$, then I does not satisfy r, and if $d^r \in I$, I is not a supported model as there is no rule in the program that proves d^r.

Example 2.1.24. *To demonstrate how Answer Set programs can be used to solve computationally hard problems, we demonstrate how to encode a 3-colouring task. Specify the graph using facts:*

$$
\begin{aligned}
\textit{vertex}(1 \ldots N) \leftarrow \quad & \textit{// Shorthand notation for } \textit{vertex}(1) \leftarrow, \ldots, \textit{vertex}(N) \leftarrow \\
\textit{edge}(1,2) \leftarrow \quad & \qquad (2.53) \\
\textit{edge}(2,3) \leftarrow \quad & \\
\vdots \quad & \qquad (2.54)
\end{aligned}
$$

We will specify the colour of a vertex using an atom. For example, if vertex 1 is to be coloured red, this will be demonstrated by $\textit{red}(1)$. *Introduce a choice rule to specify that each vertex must*

have exactly one colour:

$$1\left\{red\left(X\right), green\left(X\right), blue\left(X\right)\right\}1 \leftarrow vertex\left(X\right) \tag{2.55}$$

Finally, introduce hard constraints to prevent two neighbouring vertices from being assigned the same colour:

$$\begin{aligned} &\leftarrow edge\left(X,Y\right), red\left(X\right), red\left(Y\right) \\ &\leftarrow edge\left(X,Y\right), green\left(X\right), green\left(Y\right) \\ &\leftarrow edge\left(X,Y\right), blue\left(X\right), blue\left(Y\right) \end{aligned} \tag{2.56}$$

The Answer Sets (stable models) of this program represent valid 3-colourings of the graph.

Up until now the new constructs introduced were merely syntactic sugars. We will now introduce a construct which is a true extension.

Definition 2.1.29. *A soft constraint r is of the form:*

$$\sim b_1^r, b_2^r, \ldots, b_n^r, not\ c_1^r, not\ c_2^r, \ldots, not\ c_m^r\ \ [w^r] \tag{2.57}$$

Soft constraints, unlike hard constraints, do not rule out Answer Sets. That is, a model of a program can still satisfy the body of a soft constraint. However, when it does so, it incurs the "penalty" w^r. This allows us to specify that certain answer sets are preferable over others. We formalise this using the notion of an interpretation score:

Definition 2.1.30. *Let P be an Answer Set program with R soft constraints: $r_1, r_2, \ldots, r^R$. The score of an interpretation I of P is given by:*

$$score\left(I\right) = \sum_{\substack{i=1 \\ B^{r_i} \subseteq I \\ I \cap C^{r_i} = \emptyset}}^{R} w^r \tag{2.58}$$

Definition 2.1.31. *An Answer Set A is optimal if $score\left(A\right) \leq score\left(A'\right)$ for any other answer set A'.*

Example 2.1.25. *We can generalise the graph colouring problem by turning it into an optimisation task. Instead of forbidding neighbouring vertices from having the same colour, we can penalise such a colouring. This can allow us to specify that, for example, two adjacent vertices being coloured red is better than two adjacent vertices being coloured green or blue:*

$$\begin{aligned} &\sim edge\left(X,Y\right), red\left(X\right), red\left(Y\right)\ \ [1] \\ &\sim edge\left(X,Y\right), green\left(X\right), green\left(Y\right)\ \ [2] \\ &\sim edge\left(X,Y\right), blue\left(X\right), blue\left(Y\right)\ \ [2] \end{aligned} \tag{2.59}$$

red edges will now receive a penalty of 1 while blue and green edges a penalty of 2. In this way, if there exists a graph colouring with all neighbours having a different colour, it will still be optimal. However, if such a colour does not exist, an optimal Answer Set will contain more red-coloured edges than other colours.

2.2 Machine Learning

This section provides an introduction to topics in machine learning relevant to the rest of the work. The reader is assumed to be familiar with the very basics of machine learning (for an introduction, see Bishop and Nasrabadi (2006)). In the following, we first discuss Inductive Logic Programming, a paradigm for learning logic programs from data, with particular focus on learning Answer Set programs. We then move to statistical methods: we give an introduction to clustering algorithms, followed by an overview of neural networks and various architectures which appear in this work, up to state-of-the-art methods, including Large Language Models and Vision Language Models.

2.2.1 Inductive Logic Programming

The purpose of Inductive Logic Programming (ILP, Muggleton (1991)) is to learn logic programs from data. Given a set of positive and negative examples (E^+, E^-), ILP attempts to induce a logic program H, known as a hypothesis, that together with a background knowledge B (also a program) logically entails the positive examples and none of the negative examples. To make the search feasible, ILP systems usually limit their search to a *hypothesis space* S_M. ILP systems may introduce further optimality criteria to guide their search towards an optimal hypothesis, such as requiring H to be short, or balancing example coverage with program length (to deal with noisy examples).

Some early examples of ILP include FOIL (Quinlan, 1990) and Progol (Muggleton, 1995). More modern approaches, such as Metagol Muggleton et al. (2014) and Popper (Cropper and Morel, 2021), employ meta-programs: logic programs that represent a search problem, whose solution corresponds to models of the program and can be found using a symbolic solver. However, they do not support learning clauses with default negation, and therefore cannot learn Answer Set programs.

Learning from Answer Sets

Learning from Answer Sets (LAS) (Law et al., 2014, 2020) is an ILP framework for inducing Answer Set Programs. LAS supports learning many constructs from ASP, including default negation, choice rule and constraints. In LAS, examples are made of partial interpretations that can be used to describe desirable properties of the Answer Sets of the program to be learned.

Definition 2.2.1. *A partial interpretation e_{pi} is a tuple $e_{pi} = \left(e^{inc}, e^{exc}\right)$ where e^{inc}, e^{exc} are sets of ground atoms.*

Definition 2.2.2. *An interpretation I extends e_{pi} if $e^{inc} \subseteq I$ and $I \cap e^{exc} = \emptyset$.*

Definition 2.2.3. *A weighted context-dependent example e is a tuple $e = (e_{id}, e_{pen}, e_{pi}, e_{ctx})$ where e_{id} is an identifier of the example (must be unique for each example), e_{pen} is known as penalty and is either a positive integer or infinity, e_{pi} is a partial interpretation and e_{ctx} is an Answer Set program known as context.*

The context of an example can be used to encode special facts or conditions that apply to that example only. For example, if the example represents a sample from a dataset, e_{ctx} can encode the sample's features as facts.

Definition 2.2.4. *An Answer Set program P accepts an example e if there exists an Answer Set of P that extends e.*

A common way to define the hypothesis space is using *mode declarations*.

Definition 2.2.5. *A mode declaration m is a literal (an atom or its negation) with abstracted terms that can be either:*

- *A constant*
- *$var(t)$ where t is a type.*
- *$const(t)$ where t is a type.*

A literal is compatible with m if it can be constructed from m by replacing every $var(t)$ with a variable of type t and every $const(t)$ with a const of type t.

Types are usually defined as ground facts in the background knowledge.

Definition 2.2.6. *A language bias is a tuple (M_h, M_b) where M_h is a set of mode declarations known as the head mode declarations and M_b is a set of mode declarations known as the body mode declarations.*

A rule r is in the hypothesis space S_M if and only if:

- h^r is compatible with a mode declaration in M_h.
- For every $b^r \in B^r$, b^r is compatible with a mode declaration in M_b.
- For every $c^r \in C^r$, not c^r is compatible with a mode declaration in M_b.

Definition 2.2.7. *A LAS task T is a tuple $T = (B, S_M, E^+, E^-)$, where B is an Answer Set program known as background knowledge, S_M is a hypothesis space, E^+ is a set of positive examples and E^- is a set of negative examples.*

Definition 2.2.8. *For a hypothesis $H \in S_M$ and a weighted context-dependent example e, we denote $H \models e$ if $B \cup H \cup e_{ctx}$ accepts e. Similarly, $H \not\models e$ if $B \cup H \cup e_{ctx}$ does not accept e.*

Definition 2.2.9. *The length of a hypothesis $H \in S_M$ is given by:*

$$|H| = \sum_{r \in H} (1 + |B^r| + |C^r|) \tag{2.60}$$

Definition 2.2.10. *The score of a hypothesis H is given by:*

$$score(H) = |H| + \sum_{\substack{e \in E^+ \\ H \not\models e}} e_{pen} + \sum_{\substack{e \in E^- \\ H \models e}} e_{pen} \tag{2.61}$$

Definition 2.2.11. *The optimal solution of a LAS task T is the hypothesis $H^* \in S_M$ with minimal score. Formally:*

$$H^* = \arg\min_{H \in S_M} score(H) \tag{2.62}$$

LAS is solvable by modern systems such as ILASP (Law, 2023) and FastLAS (Law et al., 2022). ILASP searches for an optimal hypothesis using ASP meta-programs. It iteratively constructs a set of constraints on the hypothesis space by analysing examples uncovered by the current hypothesis. FastLAS can learn a more limited range of programs, but far faster than ILASP. It analyses all examples in order to construct a smaller hypothesis space that is nonetheless guaranteed to contain an optimal hypothesis. It then uses a metaprogram to find this optimal hypothesis quickly. In Chapter 6, we use ILASP as an ILP solver to enable neuro-symbolic learning, leveraging its ability to learn more general ASP programs even from noisy examples. In Chapter 4, we compare one of our neuro-symbolic learning approaches to FastLAS.

Algorithm 1 K-Means

Construct an initial cluster assignment function f_0 by assigning each datapoint $\boldsymbol{x}_j \in \mathcal{D}$ to one of the clusters $\{1, 2, \ldots, K\}$.
repeat
Compute the cluster centres according to Equation 2.63.

Compute the cluster assignment function f_{i+1} according to Equation 2.64.

until $f_{i+1} = f_i$

2.2.2 Clustering

Clustering refers to a family of unsupervised machine learning algorithms, that is, learning algorithms where datapoints are unlabelled. Even though we do not possess labels, we may still want to classify our datapoints into a given set of labels. The idea behind clustering algorithms is to group points together if they share similar features (i.e. to cluster them together). Clustering algorithms assume that, if two datapoints are close to one another, they likely represent the same class and therefore should be classified the same.

Consider a dataset $\mathcal{D}$ of N samples with no labels, each sample having M features. We can assume these samples can be represented by real-valued vectors (feature vectors). That is, $\mathcal{D} = \left\{\boldsymbol{x}_i \middle| \boldsymbol{x}_i \in \mathbb{R}^M\right\}_{i=1}^{N}$. For some given number of clusters K, a clustering is a mapping from $f : \mathbb{R}^M \to \{1, 2, \ldots, K\}$, where the indices $\{1, 2, \ldots, K\}$ represent the K different clusters.

One approach to find this mapping is to introduce a metric function $d : \mathbb{R}^M \times \mathbb{R}^M \to \mathbb{R}^+$ which acts as a similarity function between datapoints. Ideally, this metric function should give a low value (distance) for two datapoints if they share similar features of interest, and therefore should be classified together. In the *K-Means* algorithm (MacQueen, 1967), the metric function is given by the squared Euclidean distance $d(\boldsymbol{x}, \boldsymbol{y}) = |\boldsymbol{x} - \boldsymbol{y}|^2$. The K-Means algorithm iteratively assigns datapoints to clusters by greedily decreasing the average value of the metric function d between the datapoints and the cluster centres as much as possible. Suppose that at iteration i we have a cluster assignment f_i. The "cluster centre" of cluster k, $\boldsymbol{c}_{i,k}$ is computed as the mean of all datapoints assigned to it. This is given by:

$$\boldsymbol{c}_{i,k} = \sum_{\boldsymbol{x}_j \in \mathcal{D}, f_i(x_j) = k} \frac{\boldsymbol{x}_j}{|\boldsymbol{x}_j \in \mathcal{D} | f_i(\boldsymbol{x}_j) = k|} \tag{2.63}$$

In the next iteration, $i+1$, a datapoint $\boldsymbol{x}_j$ is assigned to the cluster for which its distance to the cluster centre is minimal. Formally, the new clustering function is given by:

$$f_{i+1}(\boldsymbol{x}) = \underset{k \in \{1, \ldots, K\}}{\arg\min}\ d(\boldsymbol{x}, \boldsymbol{c}_{i,k}) \tag{2.64}$$

The operation is repeated iteratively until the datapoints no longer change cluster assignments. That is, $f_i = f_{i+1}$. The algorithm is given formally in Algorithm 1. When used in practice, the initial assignment of datapoints is either random, or some K samples in $\mathcal{D}$ are chosen as initial centres. K-Means is guaranteed to converge, but not necessarily to a global optimum. To account for this, it is often run multiple times.

While the Euclidean distance is used by the K-Means algorithm, other metric functions are possible and may be useful depending on the distribution of the data. In addition, clustering algorithms exist that do not depend on a distance function. These include Hierarchy-based Clustering (where datapoints are grouped together in a tree-like hierarchy) and density-based

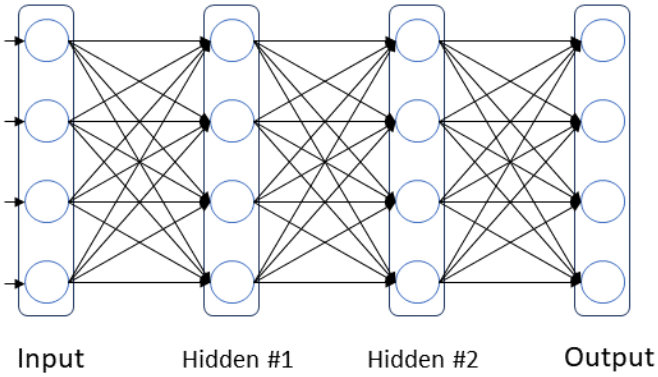


Figure 2.1: Architecture of a feedforward network.

algorithms such as DBSCAN (where clusters are discovered by identifying areas in $\mathbb{R}^M$ with a high density of datapoints). For further information on clustering algorithms, see Everitt et al. (2011).

2.2.3 Neural Networks

Artificial neural networks (ANNs) are computational models inspired by their biological counterparts (Suk, 2017). An artificial neural network is essentially a directed graph, where each node is called a *neuron* and the edges are weighted. The value a neuron holds is computed by using its inbound connections. Typically, a weighted average is performed over the values of all inbound neurons, with the weights given by the connecting edges. Possibly, a bias is then added to the value and then a non-linear activation function is applied. In this way, computation can flow through the network.

Neural network architectures group neurons into *layers*. One layer is designated as the input of the model, while another can be designated as the output. All other layers are considered *hidden*. The model's computation is then performed by first assigning values to the input, allowing the flow of information to the output layer. Training of the model is done by updating the weights of the network, typically in an attempt to minimise a goal function.

The most basic neural network architecture is the *Multi-Layered Perceptron* (MLP), also known as a *Feedforward Neural Network*. Feedforward networks are composed of a sequence of layers, where the nodes of each layer are fully connected to the nodes of the next (known as *fully-connected layers*). Hence, the graph of a feedforward network contains no cycles. Each fully-connected layer performs the following computation on its input:

$$\boldsymbol{y} = \sigma(W\boldsymbol{x} + \boldsymbol{b}) \tag{2.65}$$

where $\boldsymbol{x}$ is the input to the layer and $\boldsymbol{y}$ its output. $\boldsymbol{x}$ and $\boldsymbol{y}$ are an ordered collection of nodes and therefore can be thought of as vectors. W and $\boldsymbol{b}$ are the parameters of the layer. W is a matrix known as *weights* and $\boldsymbol{b}$ is a vector known as *bias*. σ is a non-linear *activation* function. Common choices for σ include the sigmoid function:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.66}$$

the Tanh function:

$$\sigma(x) = \tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2.67}$$

or the Rectified Linear Unit (ReLU):

$$\sigma(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases} \tag{2.68}$$

The entire network architecture is defined by composing layers in this manner. The first layer would then be used as input and the last as output. The simplicity and flexibility of feedforward networks allow for their use as general function approximators. In fact, it has been shown that even a feedforward network with a single hidden layer can theoretically approximate any continuous function over a compact set (Hornik, 1991). Feedforward networks can be used by themselves (for instance, as classifiers), or as part of a larger network architecture (such as the last few layers of a CNN architecture (Krizhevsky et al., 2017), see Section 2.2.5).

2.2.4 Training, Loss & Backpropagation

Training a neural network involves updating its parameters in order to improve the predictive value of the model. Training always involves a dataset of input-output pairs $(\boldsymbol{x}, \boldsymbol{y})$[5]. At the beginning of training, the weights and bias of the network are initialised, often randomly. One can then apply the model on a specific input $\boldsymbol{x}$ to receive some prediction of the network $\boldsymbol{o}$.

In order to test how well the network is currently performing on the task, a *loss function* $L(\boldsymbol{o}, \boldsymbol{y})$ is defined to compare the network output $\boldsymbol{o}$ and the ground truth objective $\boldsymbol{y}$. The loss function is non-negative and equal to 0 only when $\boldsymbol{o}$ is exactly equal to $\boldsymbol{y}$. To keep training tractable, the loss function should also be differentiable with respect to the model parameters, and ideally should be convex. The loss over a set of samples (such as an entire dataset) is usually defined as the average loss per sample, or the sum. Some commonly used loss functions include the mean-squared error:

$$L(\boldsymbol{o}, \boldsymbol{y}) = ||\boldsymbol{o} - \boldsymbol{y}||^2 \tag{2.69}$$

Mean absolute error:

$$L(\boldsymbol{o}, \boldsymbol{y}) = ||\boldsymbol{o} - \boldsymbol{y}|| \tag{2.70}$$

Cross-entropy (when $\boldsymbol{o}$ is a probability vector):

$$L(\boldsymbol{o}, \boldsymbol{y}) = -\sum_i y_i \log(o_i) \tag{2.71}$$

The last loss function, cross-entropy, is often used in conjunction with the *Softmax* function:

$$\text{softmax}(\boldsymbol{x})_i = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{2.72}$$

The Softmax function converts a vector $\boldsymbol{x}$ into a probability vector (i.e. all its entries become positive and sum up to one). Since the Softmax function is monotonic, it does not change the relative order of the values of the entries. It does change their relative magnitude: larger input values result in far larger output values relative to small input values. Since the Softmax function guarantees a probability vector as an output, it is commonly used as the final layer of a neural network, resulting in the so-called "Softmax Cross-Entropy" loss, which has the nice property of having an incredibly simple gradient:

[5] Even in unsupervised learning tasks, where $\boldsymbol{y}$ would be computed from $\boldsymbol{x}$ in some fashion.

$$\nabla L\left(\boldsymbol{o}, \boldsymbol{y}\right) = \boldsymbol{o} - \boldsymbol{y} \tag{2.73}$$

Gradient Descent

In order to improve the network's prediction, the parameters of all the layers of the network, denoted $\boldsymbol{\theta}$, need to be updated in order to minimise the value of the loss function. The optimisation algorithm most often used is *Gradient Descent*. Gradient Descent works iteratively by computing the gradient of the loss function with respect to the parameters, $\nabla_\theta L$, and then updating the parameters in the direction that decreases the loss the most. The update rule is therefore:

$$\theta_{i+1} = \theta_i - \eta \nabla_{\theta_i} L \tag{2.74}$$

where η is a hyperparameter of the learning algorithm, known as the *Learning Rate*. The learning rate controls the step size at every iteration, and thus has a significant effect on both the speed of convergence and the quality of the final parameter values. If the step size is too small, training could proceed very slowly. But if the step size is too large, the minimum could be missed altogether.

To compute the gradient itself the backpropagation algorithm is used (Rumelhart et al., 1986). Backpropagation is an efficient algorithm that iteratively applies the chain rule to compute the gradient of the loss with respect to all parameters. The error is computed backwards along the network, starting with the loss function and then through the output of each layer. Computing the gradient of the layer output with respect to its parameters is then a simple task.

The algorithm described above is the most basic form of gradient-based optimisation. It is sometimes referred to as "Vanilla Gradient Descent". While it is straightforward to implement, it suffers from slow convergence due to its 'zig-zag' behaviour: the update direction often flips. To address this, *momentum* is usually employed: instead of updating the parameters in the direction of the negative gradient, an average is taken with the previous update direction:

$$\theta_{i+1} = \theta_i - \eta \nabla_{\theta_i} L + \alpha \left(\theta_i - \theta_{i-1}\right) \tag{2.75}$$

where α is a hyperparameter. Even more advanced methods compute a per-parameter learning rate, such as the popular optimisation algorithm Adam (Kingma and Ba, 2014).

The gradient of the loss function needs to be computed over the samples in the dataset. In principle this may be done over the entire dataset:

$$\nabla_{\theta_i} L = \sum_{(\boldsymbol{x}, \boldsymbol{y}) \in \mathcal{D}} \nabla_{\theta_i} L_{(\boldsymbol{x}, \boldsymbol{y})} \tag{2.76}$$

In practice, the dataset size is often too large to compute this in every iteration. For this reason, the dataset is often randomly split into small "mini-batches" (or simply "batches") of samples. In a given iteration, the gradient is computed over the mini-batch only. The size of the mini-batch is another hyperparameter of the training algorithm, with popular sizes ranging from 64 to 2048. One loop over the entire dataset is known as an *epoch*.

2.2.5 Convolutional Neural Networks

While feedforward neural networks are very flexible, fully connected layers can be computationally prohibitive. This is particularly noticeable in computer vision tasks. For instance, if the model input is supposed to represent a 320×240 image of RGB values, and suppose the first

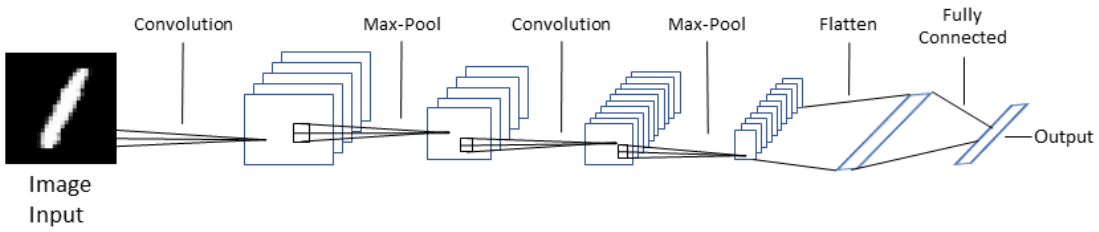


Figure 2.2: Architecture of a CNN applied to images.

hidden layer has 1000 neurons, then this layer alone has $230,400,000$ weights to be trained. Not only is this computationally expensive even by modern standards, but it also runs a serious risk of overfitting.

Convolutional neural networks (CNN) have been developed to support computer vision tasks using certain observations from studies of real-life animal vision (Fukushima and Miyake, 1982). Certain neurons in the visual cortex process information from a limited area of the visual field, with some overlapping between neighbouring neurons, in a similar fashion to a convolution operator.

A convolutional layer is different from a feedforward layer in several aspects (Stanford CS231n, n.d.). First, the input is typically a 3-dimensional tensor[6], with the 3 dimensions representing the width, height and channel of an image. The channel dimension is determined by the number of colour values each pixel has. For instance, greyscale images have a single channel while RGB images have three.

Instead of a fully-connected set of weights, convolutional layers employ *filters*. A filter is a relatively small 3-dimensional tensor that can be applied to any part of the image. For instance, a filter could be of size $5\times5\times3$, if it is applied to an image section of size 5×5 that has 3 colours. The filter is slid across the image, where for each patch a weighted average is computed by multiplying the input values by the filter weights. When this operation is applied across the entire image, it is essentially a convolution. One can stack many of these filters together to allow the output to also have multiple channels. Different filters can be thought of as attempting to capture different features in the image, such as different shapes or textures. Formally, a convolutional layer performs the following operation on an image patch, whose top-left coordinate is I, J and size is $w \times h$, with K input channels:

$$y_{IJl} = \sigma\left(\sum_{i=1}^{w}\sum_{j=1}^{h}\sum_{k=1}^{K} W_{ijkl} x_{(I+i-1),(J+j-1),k} + b_l\right) \tag{2.77}$$

l is an index indicating the filter applied, W the weights and b the bias. σ is again an activation function. Note that by employing the same weights across the entire image, we are introducing the assumption of shift-invariance. That is, if the image was translated spatially, it should not affect the output. This assumption is often valid in computer vision tasks, such as object detection, since the location of an object does not affect its identity.

The number of parameters of each convolutional layer is therefore kept relatively small, but the size of the output can still be quite large. Many convolutional layers are often employed in succession until the output size is small enough to apply fully-connected layers. In this context, *max pool* layers are also often employed to further reduce output size. Just like a convolutional

[6]There also exist convolutional layers with 2-dimensional inputs, for instance for speech data, or 4-dimensional inputs, such as video.

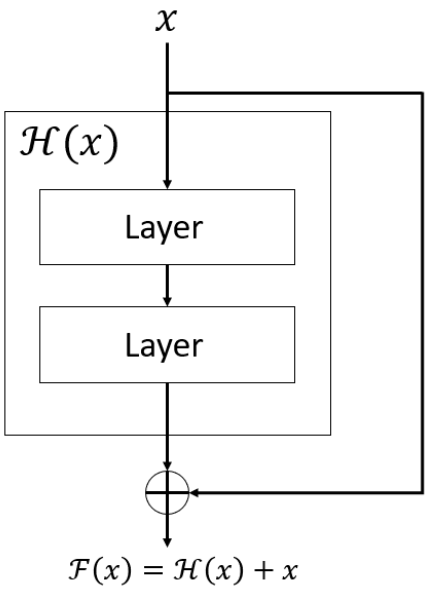


Figure 2.3: An example of a ResNet block.

layer, a max pool layer applies a small sliding window along the input. A max operation is performed on all values in the window, and this becomes the layer's output.

2.2.6 ResNets

Networks with more layers (i.e. deeper networks) can perform better than networks with fewer layers (i.e. shallower networks), since each layer adds an additional opportunity to perform computation, allowing more complex functions to be learned more easily. However, deep networks encounter the issue of *vanishing gradients*. As the backpropagation algorithm is applied from layer to layer, the gradients from each layer are multiplied. Since many of the values in the gradients are between -1 and 1, the result of the multiplication tends to 0. By the time the parameter updates in the earlier layers are computed, their updates diminish to 0. As a result, learning becomes inefficient. In addition, even when the vanishing gradient issue is addressed, deeper networks have been shown to be more difficult to optimise (Srivastava et al., 2015).

To make the training of deep networks more efficient, Residual Networks (ResNets) were introduced (He et al., 2016). The idea is to group layers into blocks so that in each block, instead of learning a mapping, we learn its residual. Formally, if a block is to compute a mapping $\mathcal{F}(\boldsymbol{x})$, we have it compute instead the residual $\mathcal{H}(\boldsymbol{x}) = \mathcal{F}(\boldsymbol{x}) - \boldsymbol{x}$. This is achievable by introducing a "skip connection" to the network: the input to the block is added to its output. A graphical depiction of a skip connection can be seen in Figure 2.3.

This provides two advantages: first, it has been empirically shown to be easier to learn a residual, as loss functions become smoother (Li et al., 2018); second, gradient information can now flow through the skip connection, mitigating the vanishing gradient issue. At the time of their introduction, various ResNet architectures achieved state-of-the-art performance on several computer vision benchmarks, and residual blocks have since become a standard in many architectures, including transformers.

2.2.7 Attention & Transformers

Many tasks involve computation over sequences. For example, natural language processing tasks can be thought of as acting upon sequences of words. Over the years, many neural network architectures have been proposed to deal with sequential data, such as the Recurrent Neural Network (Williams and Zipser, 1990) and its extensions, LSTM (Hochreiter and Schmidhuber, 1997) and GRU (Cho et al., 2014). Over long sequences, RNNs run into a "forgetfulness" problem - values computed over the beginning of the sequence are generally lost by the time the end of the sequence is processed. In many cases, however, only part of the sequence is of importance at any given moment. For example, suppose we are given a page of text and then asked a question, where the answer can be found somewhere in the text. Only a small portion of the text is actually relevant to answer the question (or "query") correctly. The *Attention Mechanism* is exactly suitable for situations such as this.

Given an input sequence of tokens $\{t_i\}_{i=1}^{n}$ of key-value pairs $t_i = (k_i, v_i)$, and a query $\boldsymbol{q}$, attention computes a weighted average over the sequence of values, where the weights are computed using the keys and query:

$$\begin{aligned} \boldsymbol{w}_i &= \text{softmax}\left(\text{f}\left(\boldsymbol{k_i}, \boldsymbol{q}\right)\right) \\ \text{attn}\left(\boldsymbol{k_i}, \boldsymbol{v_i}, \boldsymbol{q}\right) &= \sum_{i=1}^{n} w_i \boldsymbol{v_i} = \boldsymbol{w}^T V \end{aligned} \tag{2.78}$$

where f is an alignment function meant to measure how "relevant" the i'th entry is for the query $\boldsymbol{q}$. Nowadays, f is usually taken to be the "scaled dot-product":

$$f\left(k_i, q\right) = \frac{\boldsymbol{k_i}^T \boldsymbol{q}}{\sqrt{D}} \tag{2.79}$$

where D is the dimension of $\boldsymbol{k_i}$. An intuitive way of thinking about attention is that the query is checked against all tokens in the sequence. The key is used to tell how relevant the token is for a given query. If it is highly relevant, the value of the token is highly represented in the output. Otherwise, it does not receive much representation. This allows the output to be affected only by the most relevant tokens in the sequence.

The Transformer architecture Vaswani et al. (2017) was introduced to remove the need for recurrence when dealing with sequential data, as recurrence is time costly. Instead, it uses the attention mechanism almost exclusively. Transformers introduce multi-headed attention, a layer that transforms a sequence into another by performing the attention mechanism multiple times in parallel (each called an "attention head") and concatenating the results before projecting them down to a given dimension. A transformer block performs multi-headed attention followed by a fully-connected layer, with a skip connection. These blocks can be stacked to allow for an arbitrarily deep network. Allowing sequences to be processed in parallel rather than sequentially enabled the training of deep networks on datasets much larger than were previously feasible.

Transformer architectures come in three flavours: encoder-only, decoder-only and encoder-decoder. The encoder can process a given input sequence in parallel but the output sequence must be of the same length. The decoder receives an input sequence and can generate an output token one at a time (this is sometimes referred to as autoregressive modelling). The encoder-decoder architecture combines the two to allow the mapping of input to output sequences of arbitrary lengths.

While the encoder-decoder architecture is very useful for some tasks (for example, machine translation), the decoder-only architecture is suitable for tasks of text generation. For this reason, the model has recently become standard for the task of language modelling (the so-called "Large Language Models", LLM). This is accomplished by using a decoder-only model to learn the probability function $p(t_i|t_1, \ldots, t_{i-1})$ over some predefined vocabulary of tokens (Radford et al., 2018). The task can be learned in a self-supervised manner by taking a large corpus of text and sampling sequences, learning to regenerate them.

2.2.8 Vision-Language Models

Large Vision-Language Models (VLMs) extend the capabilities of Large Language Models by allowing them to receive an image input in addition to text. The task then becomes the modelling of the following probability distribution:

$$p(t_i|t_1, \ldots, t_{i-1}, I) \tag{2.80}$$

where I is an image. This allows models to perform tasks that involve both vision and language, such as visual QA. For example, the user may present the model an image of a car, and provide a prompt such as "What colour is the car?". The prompt is translated into a sequence of tokens that, together with the image, allow the model to generate the rest of the sequence, ideally the correct colour of the car that appears in the image.

Different VLMs take different approaches to incorporate image information in the input of the model. Many employ a pretrained visual encoder, which may be a ResNet or a Vision Transformer (Dosovitskiy et al., 2021). This translates the image into visual tokens that can then be incorporated in the rest of the architecture, for example as a query for an attention layer (sometimes called "cross-attention", Alayrac et al. (2022)).

In Chapter 6, we use the off-the-shelf open-source Vision Language Model BLIP-2 Li et al. (2023a). BLIP-2 leverages two pre-trained models: an image encoder and a text encoder. The image encoder is an open-source Vision Transformer that was previously pretrained using contrastive learning Radford et al. (2021). The text encoder is an open-source LLM. Training proceeds in two stages. In the first stage, a Querying Transformer (Q-Former) is trained to bridge the gap between the vision and language modalities by means of both (masked) self-attention and cross-attention. The training objective at this stage is three-fold: Image-Text Contrastive Learning learns to maximise the mutual information between matching images and text; Image-grounded Text Generation learns to generate text conditioned on an input image; Image-Text Matching learns to predict if an image and text match through binary classification. The second stage of training involves connecting the Q-Former to a frozen LLM and training either in a decoder-only or an encoder-decoder setting. BLIP-2 was trained on a dataset of 129M images from various sources, such as COCO (Lin et al., 2014) and Visual Genome (Krishna et al., 2017).

Chapter 3

Stable and Supported Semantics in Continuous Vector Spaces

Any method that incorporates a logic program in neural network computations must achieve two goals:

1. Be able to represent an arbitrary logic program, including its rules and atoms, in a joint neural-symbolic architecture.

2. Be capable of bridging the gap between continuous and symbolic values to perform logical operations on non-discrete data and facilitate a joint neural-symbolic computation.

In this chapter, we present a novel representation of normal logic programs that satisfy these two criteria and supports negation as failure. The representation is based on a characterisation of logic programs as real-valued matrices in a one-to-one mapping. Previous authors who have explored such representations (Sakama et al., 2017; Sato, 2017a,b; Nguyen et al., 2021b,a) have either limited their investigation to positive programs only or otherwise utilised an exponential-sized tensor to represent programs and interpretations, resulting in intractable algorithms. Our approach is novel in that it represents normal programs through the concept of program reduct. We show how the program reduct with respect to interpretations can be represented as a polynomial-sized matrix, allowing for arbitrary normal programs to be represented while keeping the computation tractable. Our matrix representation also realises the T_P operation via linear-algebraic operations, providing a computational method for performing deduction.

The representation proposed here is a first step towards developing a novel neuro-symbolic architecture that can embed arbitrary logic programs in a neural network and support back-propagation. To this end, we show how our matrix representation of program reduct can also be applied when given real-valued program interpretations. The representation naturally extends from the discrete case to the real-valued one, allowing for seamless integration of real-valued representations of data (such as the output of a neural network) with exact symbolic knowledge. We show that under certain conditions, exact inference is maintained even in the real-valued case.

In order to demonstrate the potential uses of this real-valued representation, we present a novel gradient-based algorithm for computing supported and stable models of a normal program. As both our program representation and interpretations are real-valued algebraic entities, they can be integrated naturally into differentiable functions, allowing us to restate the problem of

finding supported and stable models as a computational optimisation task. Namely, we write an equation whose roots are guaranteed to correspond to supported models, and which can be solved using Newton's method. In a systematic evaluation, we show the algorithm can find supported and stable models of semantically complex programs that contain positive loops, negative loops and/or choice rules.

This chapter is structured as follows: we begin by laying out our assumption on the structure of the normal program, through a condition we refer to as the "Multiple Definitions Condition". We show that every normal program can be transformed in linear time to one that satisfies the condition while maintaining its semantics, hence this assumption is without loss of generality (section 3.1). Next, we introduce our novel representation for normal programs using matrices. To ease the presentation, we first show how definite programs can be represented using matrices and prove that correct deduction can be achieved using matrix multiplication and the non-linear Heaviside operation. We then generalise this matrix representation to normal programs by showing how a program reduct is translated to a matrix, and prove an analogous deduction theorem for the normal program case (section 3.2). In order to move towards real-valued representation of interpretations and gradient-based computations, we replace the Heaviside operation with a Sigmoid operation in the next section. We develop and prove the necessary conditions to ensure exact inference is maintained in continuous space (section 3.3). To demonstrate the usefulness of this differentiable operation, we introduce and formalise a novel algorithm that employs Newton's root-finding algorithm to compute stable and supported models of an arbitrary normal program (section 3.4). Next, we perform an initial evaluation of this algorithm to show its feasibility, exploring its performance under challenging scenarios involving both positive and negative loops, showing both its success and failure cases as well as strategies to improve the success rate (section 3.5). Finally, we summarise the main results of the chapter (section 3.6).

The work presented in this chapter has previously been published in the paper "Stable and Supported Semantics in Continuous Vector Spaces" (Aspis et al., 2020).

3.1 Multiple Definitions Condition

Our matrix representation of normal logic program reducts is inspired by the work of Sakama et al (Sakama et al., 2017), who first introduced a matrix representation for Horn programs. Our work goes beyond theirs by introducing a novel representation that supports normal programs and is extended to real-valued program interpretations. Similar to their work, we assume programs obey a "Multiple Definitions Condition" (MD condition). This condition, which we will define in a moment, restricts the number and length of rules that define the same head atom. As will be shown later, this condition is needed to ensure the proposed one-to-one mapping between programs and matrices is correct and that the deduction process is sound. Furthermore, a simple one-to-one mapping exists from any normal program to a program satisfying the MD condition, so we may impose this condition without loss of generality.

3.1.1 Other Assumptions

Before discussing the MD condition, we introduce here two further assumptions that we make about our programs without loss of generality. Consider a normal propositional program P. To simplify some of the formulation in this chapter, we add an additional restriction on P: the positive and negative body of every clause are non-empty. To impose this restriction without loss of generality, we use the two constant atoms: $\bot$ and $\top$ (recall these represent the values "False"

and "True", respectively). We can transform P to a program that satisfies this condition by replacing every empty positive body with $\{\top\}$ and every empty negative body with $\{\bot\}$. Every interpretation of P is also assumed to contain $\top$ and not to contain $\bot$ (for clarity of presentation, when writing an interpretation explicitly we will omit $\top$). For technical reasons, we also assume that every program contains the clause $\top \leftarrow \top, \text{not } \bot$, which is a tautology and will be omitted when writing the program explicitly.

Example 3.1.1. *The following normal program:*

$$\begin{aligned} a &\leftarrow \\ b &\leftarrow c \\ d &\leftarrow not\ e \end{aligned} \tag{3.1}$$

is replaced with the program:

$$\begin{aligned} \top &\leftarrow \top, not\ \bot \\ a &\leftarrow \top, not\ \bot \\ b &\leftarrow c, not\ \bot \\ d &\leftarrow \top, not\ e \end{aligned} \tag{3.2}$$

which is semantically equivalent. For ease of presentation, we will write programs as in Equation 3.1.

In addition to these, we also assume in this chapter that our programs do not explicitly contain constraints. However, constraints can be transformed into clauses as follows. Suppose you have the following constraint:

$$\leftarrow b_1^r, b_2^r, \ldots, b_{n^r}^r, \text{not } c_1^r, \text{not } c_2^r, \ldots, \text{not } c_{m^r}^r \tag{3.3}$$

Remove the constraint from P, introduce a new atom $\bot^r$ and add the following clause to P:

$$\bot^r \leftarrow b_1^r, b_2^r, \ldots, b_{n^r}^r, \text{not } c_1^r, \text{not } c_2^r, \ldots, \text{not } c_{m^r}^r, \text{not } \bot^r \tag{3.4}$$

This ensures no supported model of P can satisfy the body of the constraint.

3.1.2 Definition of the MD condition

Definition 3.1.1. *We call a rule long if it has more than two body atoms. Otherwise, we refer to it as short.*

For a given propositional program P and an atom $h \in B_P$, define long_rules (h) to be the set of long rules in P with head h. Formally:

Definition 3.1.2. *Let P be a propositional program and h an atom in B_P. The set of long rules of h is*

$$long_rules(h) = \{r \in P \mid h^r = h, |B^r| + |C^r| > 2\} \tag{3.5}$$

Example 3.1.2. *For the following program:*

$$\begin{aligned} h &\leftarrow b_1, b_2 \\ h &\leftarrow not\, c_1, not\, c_2 \\ h &\leftarrow d \end{aligned} \tag{3.6}$$

we have:

$$long_rules\,(h) = \{h \leftarrow b_1, b_2, \quad h \leftarrow not\, c_1, not\, c_2\} \tag{3.7}$$

Definition 3.1.3. *A normal program satisfies the Multiple Definitions Condition if for every* $h \in B_P$ *we have* $|long_rules\,(h)| \leq 1$.

In other words, the MD condition states that an atom h can be the head of at most one long rule. The program 3.6 does not satisfy the MD condition as h is the head of two long rules (the first two). Note that there is no restriction on the number of short rules h can be the head of, so the third rule has no implication on satisfying the MD condition.

3.1.3 Transformation

The observation that there is no restriction on the number of short rules allows us to introduce an efficient transformation from an arbitrary normal program to a program that satisfies the MD condition. Suppose h is the head of $k > 1$ long rules:

$$\begin{aligned} h &\leftarrow b_1^1, b_2^1, \ldots, b_{n^1}^1, \text{not } c_1^1, \text{not } c_2^1, \ldots, \text{not } c_{m^1}^1 \\ h &\leftarrow b_1^2, b_2^2, \ldots, b_{n^2}^2, \text{not } c_1^2, \text{not } c_2^2, \ldots, \text{not } c_{m^2}^2 \\ &\vdots \\ h &\leftarrow b_1^k, b_2^k, \ldots, b_{n^k}^k, \text{not } c_1^k, \text{not } c_2^k, \ldots, \text{not } c_{m^k}^k \end{aligned} \tag{3.8}$$

We can introduce k new atoms to the Herbrand Base, which we denote $h_1, h_2, \ldots, h_k$. We replace the head h with these new atoms for each long rule. This ensures long_rules (h) becomes empty[1]. Then we add k short rules to the program, with h as the head and one of the new atoms in the body. The resulting set of rules is as follows:

$$\begin{aligned} h_1 &\leftarrow b_1^1, b_2^1, \ldots, b_{n^1}^1, \text{not } c_1^1, \text{not } c_2^1, \ldots, \text{not } c_{m^1}^1 \\ h_2 &\leftarrow b_1^2, b_2^2, \ldots, b_{n^2}^2, \text{not } c_1^2, \text{not } c_2^2 \ldots, \text{not } c_{m^2}^2 \\ &\vdots \\ h_k &\leftarrow b_1^k, b_2^k, \ldots, b_{n^k}^k, \text{not } c_1^k, \text{not } c_2^k, \ldots, \text{not } c_{m^k}^k \end{aligned} \tag{3.9}$$

$$\begin{aligned} h &\leftarrow h_1 \\ h &\leftarrow h_2 \\ &\vdots \\ h &\leftarrow h_k \end{aligned}$$

[1]We could have also defined the transformation as leaving one long rule for h and transforming the rest, which will still satisfy the MD condition. We chose to replace all long rules for simplicity of presentation.

After this transformation, h no longer causes a violation of the MD condition. We can apply this transformation to every atom h that is the head of two or more long rules. The result is a new program $P^{'}$ that satisfies the MD condition.

Example 3.1.3. *For the program in 3.6 the transformation would look as follows:*

$$\begin{aligned} h_1 &\leftarrow b_1, b_2 \\ h_2 &\leftarrow not\, c_1, not\, c_2 \\ h &\leftarrow h_1 \\ h &\leftarrow h_2 \\ h &\leftarrow d \end{aligned} \tag{3.10}$$

Next, we wish to establish that the semantics of the program P is preserved under this transformation. Intuitively, it is obvious that a model of $P^{'}$, once the new auxiliary atoms are removed, becomes a model of P. But we must also establish that this is true for supported and stable models, i.e. that they remain supported and stable models after this removal. This is in fact the case as is formalised in the next set of theorems:

Theorem 3.1.1. *Suppose P is a normal logic program and let $P^{'}$ be the program after the transformation described above. Then:*

1. *If M' is a model of $P^{'}$ then $M = M' \cap B_P$ is a model of P.*
2. *If M is a model of P then there exists a model M' of $P^{'}$ such that $M = M' \cap B_P$.*
3. *If M' is a supported model of $P^{'}$ then $M = M' \cap B_P$ is a supported model of P.*
4. *If M is a supported model of P then there exists a supported model M' of $P^{'}$ such that $M = M' \cap B_P$.*
5. *If M' is a stable model of $P^{'}$ then $M = M' \cap B_P$ is a stable model of P.*
6. *If M is a stable model of P then there exists a stable model M' of $P^{'}$ such that $M = M' \cap B_P$.*

Proof. 1. To show M is a model, we need to show that for every $r \in P$ such that $B^r \subseteq M$ and $C^r \cap M = \emptyset$, we have $h^r \in M$. Let $r \in P$ be such a rule. Either $r \in P^{'}$ or $r \notin P^{'}$.

- If $r \in P^{'}$, then r is a short rule or the only long rule. Since $r \in P$ we have $h^r \in B_P$ and $B^r, C^r \subseteq B_P$. Note that $B^r \subseteq M \subseteq M'$. Since $C^r \subseteq B_P$ we have $\emptyset = C^r \cap M = C^r \cap B_P \cap M' = C^r \cap M'$. As M' is a model of $P^{'}$, we have $h^r \in M'$, and since $h^r \in B_P$ we have $h^r \in M$.
- If $r \notin P^{'}$. Then from the construction of $P^{'}$, we have $r \in \text{long_rules}\,(h^r)$. Therefore, there exists an atom $h^r_i \in B_{P'}$ and two rules $r_1, r_2 \in P^{'}$ such that $h^{r_1} = h^r, B^{r^1} = \{h^r_i\}, h^{r_2} = h^r_i, B^{r_2} = B^r$ and $C^{r_2} = C^r$ where $h^r \in B_P$ and $B^r, C^r \subseteq B_P$. Since $r_2 \in P^{'}$ and since M' is a model of $P^{'}$, as before we have $h^r_i \in M'$, and since $r_1 \in P^{'}$ we have $h^r \in M'$, implying $h^r \in M$.

2. We construct a model M' of P' as follows. Let:

$$I = \left\{ h^r \mid r \in P^{'}, h^r \notin B_P, B^r \subseteq M, C^r \cap M = \emptyset \right\} \tag{3.11}$$

$$M' = M \cup I \tag{3.12}$$

From the construction, note that $I \cap B_P = \emptyset$ and we have $M = M' \cap B_P$. We will show that M' is a model of $P^{'}$. Let $r \in P^{'}$ such that $B^r \subseteq M'$ and $C^r \cap M' = \emptyset$. We need to show $h^r \in M'$. From the construction of $P^{'}$, there are three options: $r \in P$, $r \notin P$ and there exists an atom $h_i^r \notin B_P$ such that $h^r = h_i^r$ and $B^r, C^r \subseteq B_P$, or $r \notin P$ and there exists an atom $h_i^r \notin B_P$ such that $B^r = \{h_i^r\}$ and $C^r = \{\bot\}$.

- If $r \in P$ then $B^r, C^r \subseteq B_P$. Hence $B^r \subseteq B_P \cap M' = M$ and $\emptyset = C^r \cap M' = (C^r \cap M) \cup (C^r \cap I) = (C^r \cap M) \cup \emptyset = C^r \cap M$. Since M is a model of P, we have $h^r \in M \subseteq M'$.
- If $r \notin P$ and $h^r = h_i^r \notin B_P$, then $B^r, C^r \subseteq B_P$. As in the previous case, this implies $B^r \subseteq M$ and $C^r \cap M = \emptyset$, therefore $h^r \in I \subseteq M'$.
- If $r \notin P$ and $B^r = \{h_i^r\} \subseteq M'$, then $h_i^r \in M'$. Since $h_i^r \notin B_P$ we have $h_i^r \notin M$ and therefore $h_i^r \in I$. Therefore there exists a rule $r' \in P^{'}$ such that $h^{r^{'}} = h_i^r$, $B^{r^{'}} \subseteq M$ and $C^{r^{'}} \cap M = \emptyset$. So $h^{r^{'}} \in M$. But from the construction of $P^{'}$ it must hold that $h^{r^{'}} = h^r$. Therefore $h^r \in M \subseteq M'$.

3. From 1 we know that M is a model of P, and it remains to show that it is supported. Let $a \in M$. We need to show there exists a rule $r \in P$ such that $h^r = a$, $B^r \subseteq M$ and $C^r \cap M = \emptyset$. Since $a \in M \subseteq M'$, and since M' is supported, there exists a rule $r \in P'$ such that $h^r = a$, $B^r \subseteq M'$ and $C^r \cap M' = \emptyset$. There are two cases:

   - If $r \in P$, then $B^r, C^r \subseteq B_P$ and we obtain the result immediately.
   - If $r \notin P$ then because $a = h^r \in B_P$ from the construction of P' there exists an atom $h_i^r \notin B_P$ such that $B^r = \{h_i^r\}$, $C^r = \{\bot\}$ and a rule $r' \in P^{'}$ such that $h^{r^{'}} = h_i^r$, $B^{r^{'}} = B^r$, $C^{r^{'}} = C^r$, which is the only rule with h_i^r as its head. Since $B^r = \{h_i^r\} \subseteq M'$ we have $h_i^r \in M'$ and because M' is supported this implies $B^r \subseteq M'$ and $C^r \cap M' = \emptyset$, in addition to $B^r, C^r \subseteq B_P$. Therefore $B^r \subseteq M$ and $C^r \cap M = \emptyset$.

4. Construct M' as in Equation 3.12. We know M' is a model and it remains to show that it is supported. Let $a \in M'$. We need to show that there exists $r \in P'$ such that $a = h^r$, $B^r \subseteq M'$ and $C^r \cap M' = \emptyset$. Either $a \in M$ or $a \notin M$.

   - If $a \in M$, since M is a supported model of P there exists $r \in P$ such that $B^r \subseteq M \subseteq M'$ and $C^r \cap M = \emptyset$. But since $C^r \subseteq B^p$ we also have $C^r \cap M' = (C^r \cap M) \cup (C^r \cap I) = \emptyset \cup \emptyset = \emptyset$, as required.
   - If $a \notin M$, then $a \in I$ and therefore there exists $r \in P'$ such that $h^r = a$, $B^r \subseteq M \subseteq M'$ and $C^r \cap M = \emptyset$. Since $a \notin B_P$, from the construction of P' it must be the case that $C^r \subseteq B_P$. Therefore as in the previous case, $C^r \cap M' = \emptyset$, as required.

5. We know M is a supported model of P, and it remains to show that it is well-supported[2]. Since M' is a well-supported model of P', it establishes a well-founded partial ordering $\prec$ of the atoms in M', such that for all $a \in M'$, there exist $r \in P'$ such that $h^r = a$, $B^r \subseteq M'$, $C^r \cap M' = \emptyset$ and for all $b \in B^r$, $b \prec a$. Since $M \subseteq M'$ we can apply the same ordering over the atoms of M. Let $a \in M$. To show that M is well-supported, we need to demonstrate that there exists a rule $r \in P$ such that $h^r = a$, $B^r \subseteq M$, $C^r \cap M = \emptyset$ and for every $b \in B^r$, $b \prec a$. Since M is supported, there exists a rule $r \in P$ such that $h^r = a$,

[2]Recall the well-supported and stable models of a normal program coincide.

$B^r \in M$ and $C^r \cap M = \emptyset$. Let $b \in B^r$. If $r \in P^{'}$, then $b \prec a$. Otherwise, there exist two rules $r_1, r_2 \in P^{'}$ such that $B^{r_1} = B^r$, $C^{r_1} = C^r$, $h^{r_2} = h^r = a$ and $B^{r_2} = \{h^{r_1}\}$. Therefore $b \prec h^{r_1} \prec a$.

6. Construct M' as in Equation 3.12. We have shown M' is a supported model so it remains to show that it is well-supported. Since M is a well-supported model there exists a well-founded ordering $\prec$ over M such that for all $a \in M$, there exist $r \in P$ such that $h^r = a$, $B^r \subseteq M$ and $C^r \cap M = \emptyset$ and for all $b \in B^r$, $b \prec a$. We extend $\prec$ to the rest of M'. Note this means we must extend the ordering to the atoms in I. Let $a \in I$ From the construction of $P^{'}$, there exist two rules r_1, r_2 such that:

$$\begin{aligned} r_1 =& a \leftarrow b_1, b_2, ..., b_n, \text{not } c_1, \text{not } c_2, ..., \text{not } c_m \\ r_2 =& h \leftarrow a \end{aligned} \tag{3.13}$$

where $h, b_1, b_2, \ldots, b_n, c_1, c_2, \ldots, c_m \in B_P$. In addition, since P contained the rule:

$$h \leftarrow b_1, b_2, ..., b_n, \text{not } c_1, \text{not } c_2, ..., \text{not } c_m \tag{3.14}$$

we have $b_i \prec h$ for all $1 \leq i \leq n$. We can therefore extend $\prec$ so that $b_i \prec a \prec h$ for all i. Since r_1 and r_2 are the only rules in $P^{'}$ where a appears, the condition for a well-supported model is maintained with respect to a. As a is an arbitrary atom in I, we have shown that M' is a well-supported model.

□

A consequence of the theorem above is that finding supported or stable models of $P^{'}$ is equivalent to finding supported or stable models of P: one must simply remove the auxiliary atoms in the given model M' to construct M. We can therefore assume without loss of generality that a program satisfies the MD condition. If this is not the case, we can apply the transformation as a pre-processing step, and remove the auxiliary atoms during post-processing.

3.2 Matrix Representation of Normal Programs

We are now ready to present the representation of a normal program and its interpretations in vector space. Consider a normal program P satisfying all conditions mentioned in the previous section. We begin by placing an ordering over the elements of the Herbrand base of P:

$$B_P = \{p_1 = \bot, p_2 = \top, p_3, ..., p_N\} \tag{3.15}$$

Definition 3.2.1. *(Atom embedding) Each element of B_P is mapped to one of the elements of the standard basis of $\mathbb{R}^N$. Specifically, if the standard basis of $\mathbb{R}^N$ is $\{\boldsymbol{e_i}\}_{i=1}^N$, then p_i is mapped to $\boldsymbol{e_i}$. This is also known as the "one-hot encoding". We denote the embedding of atom p_i as $\boldsymbol{v}^{p_i} = \boldsymbol{e_i}$.*

Example 3.2.1. *Consider the following program:*

$$\begin{aligned} p &\leftarrow not\, q \\ q &\leftarrow not\, p \\ r &\leftarrow p, s, not\, q, not\, t \\ t &\leftarrow p, not\, s, not\, r \end{aligned} \tag{3.16}$$

we set the following order over its Herbrand Base:

$$B_P = \{p_1 = \bot, p_2 = \top, p_3 = p, p_4 = q, p_5 = r, p_6 = s, p_7 = t\} \tag{3.17}$$

and each element is embedded as its corresponding standard basis vector:

$$\boldsymbol{v}^{\bot} = \begin{bmatrix}1\\0\\0\\0\\0\\0\\0\end{bmatrix} \quad \boldsymbol{v}^{\top} = \begin{bmatrix}0\\1\\0\\0\\0\\0\\0\end{bmatrix} \quad \boldsymbol{v}^{p} = \begin{bmatrix}0\\0\\1\\0\\0\\0\\0\end{bmatrix} \quad \boldsymbol{v}^{q} = \begin{bmatrix}0\\0\\0\\1\\0\\0\\0\end{bmatrix} \tag{3.18}$$

$$\boldsymbol{v}^{r} = \begin{bmatrix}0\\0\\0\\0\\1\\0\\0\end{bmatrix} \quad \boldsymbol{v}^{s} = \begin{bmatrix}0\\0\\0\\0\\0\\1\\0\end{bmatrix} \quad \boldsymbol{v}^{t} = \begin{bmatrix}0\\0\\0\\0\\0\\0\\1\end{bmatrix}$$

Definition 3.2.2. *(Set embedding) A set of atoms $A \subset B_P$ is embedded by summing over the embedding of its elements. We denote the embedding of A as $\boldsymbol{v}^A$. Formally:*

$$\boldsymbol{v}^A = \sum_{p \in A} \boldsymbol{v}^p \tag{3.19}$$

$\boldsymbol{v}^A$ can also be thought of as a binary vector where an element is set to 1 if the corresponding atom is in A, otherwise, it is 0.

Example 3.2.2. *The embedding of the set $\{\bot, p, s\}$ is:*

$$\boldsymbol{v}^{\{\bot,p,s\}} = \boldsymbol{v}^{\bot} + \boldsymbol{v}^{p} + \boldsymbol{v}^{s} = \begin{bmatrix}1\\0\\0\\0\\0\\0\\0\end{bmatrix} + \begin{bmatrix}0\\0\\1\\0\\0\\0\\0\end{bmatrix} + \begin{bmatrix}0\\0\\0\\0\\0\\1\\0\end{bmatrix} = \begin{bmatrix}1\\0\\1\\0\\0\\1\\0\end{bmatrix} \tag{3.20}$$

For an interpretation I, we always have $v_1^I = 0$ and $v_2^I = 1$.

3.2.1 Embedding Definite Programs

Definition 3.2.3. *(Definite clause embedding) Let $r \in P$ be a definite clause, where $|B_P| = N$ and $|B^r| = n^r \geq 1$. r is embedded into a matrix $D^r \in \mathbb{R}^{N \times N}$, which is computed as follows:*

$$D^r = \frac{1}{n^r} \boldsymbol{v}^{h^r} \left(\boldsymbol{v}^{B^r} \right)^\intercal \tag{3.21}$$

Note that had we allowed clauses to have empty positive bodies, the definition in Equation 3.21 would require special handling of the case $n^r = 0$. The constraint $n^r \geq 1$ simplifies this and what follows without sacrificing generality.

Example 3.2.3. *The clause $r \leftarrow p$ is embedded as:*

$$D^{r \leftarrow p} = \frac{1}{|\{p\}|} \boldsymbol{v}^r \left(\boldsymbol{v}^{\{p\}} \right)^\intercal = \frac{1}{1} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} = \begin{array}{c} \\ \bot \\ \top \\ p \\ q \\ r \\ s \\ t \end{array} \begin{array}{c} \begin{array}{ccccccc} \bot & \top & p & q & r & s & t \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array} \tag{3.22}$$

Example 3.2.4. *The clause $r \leftarrow p, q$ is embedded as:*

$$D^{r \leftarrow p,q} = \frac{1}{|\{p,q\}|} \boldsymbol{v}^r \left(\boldsymbol{v}^{\{p,q\}} \right)^\intercal = \frac{1}{2} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.23}$$

Example 3.2.5. *The clause $r \leftarrow$ is embedded as:*

$$D^{r \leftarrow} = \frac{1}{|\{\top\}|} \boldsymbol{v}^r \left(\boldsymbol{v}^{\{\top\}} \right)^\intercal = \frac{1}{1} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.24}$$

Definition 3.2.4. *(Definite program embedding) The embedding of a definite program P is given by the sum of the embedding of its rules. Formally:*

$$D^P = \sum_{r \in P} D^r \tag{3.25}$$

Example 3.2.6. *For the following definite program P:*

$$\begin{aligned} p &\leftarrow \\ r &\leftarrow p, q \\ t &\leftarrow s \end{aligned} \tag{3.26}$$

the matrix representation is as follows:

$$D^P = D^{\top\leftarrow\top} + D^{p\leftarrow\top} + D^{r\leftarrow p,q} + D^{t\leftarrow s} = \begin{array}{c} \\ \bot \\ \top \\ p \\ q \\ r \\ s \\ t \end{array} \begin{array}{c} \begin{array}{ccccccc} \bot & \top & p & q & r & s & t \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array} \tag{3.27}$$

This definition coincides with the matrix embedding for definite programs given by Sakama et al. Sakama et al. (2017).

Immediate Consequences for Definite Programs

Using the matrix representation introduced above, we can realise the Immediate Consequences operator for definite programs. Recall that for definite programs $T_P(I) = \{h^r \mid r \in P, B^r \subseteq I\}$. The idea is to multiply the matrix representation of P with the vector representation of I. If $B^r \subseteq I$, the product of the row corresponding to h^r and $\boldsymbol{v}^I$ will be equal to exactly one. Otherwise, the product will be less than one. We can then threshold the values of the resulting vector so that values less than 1 are set to 0.

Definition 3.2.5. *(Threshold function)*

$$H_1(x) = \begin{cases} 1 & x \geq 1 \\ 0 & x < 1 \end{cases} \tag{3.28}$$

H_1 is a Heaviside function but with the threshold value being 1 instead of 0.

Theorem 3.2.1. *Let P be a definite program satisfying the MD condition and I an interpretation. Then:*

$$\boldsymbol{v}^{T_P(I)} = H_1\left(D^P \boldsymbol{v}^I\right) \tag{3.29}$$

where H_1 is applied element-wise. Note that for any atom $a \in B_P$ we have $H_1(\boldsymbol{v}^a) = \boldsymbol{v}^a$.

Proof. Let $a \in B_P$. Either $a \in T_P(I)$ or not.

- If $a \in T_P(I)$, there exists $r \in P$ such that $h^r = a$ and $B^r \subseteq I$. In particular, for every

$b \in B^r$ we have $\left(\boldsymbol{v}^b\right)^\intercal \boldsymbol{v}^I = 1$. Therefore:

$$\begin{aligned} D^P \boldsymbol{v}^I &= \sum_{r' \in P, r' \neq r} D^{r'} \boldsymbol{v}^I + D^r \boldsymbol{v}^I \geq D^r \boldsymbol{v}^I = \frac{1}{n^r} \boldsymbol{v}^a \left(\boldsymbol{v}^{B^r}\right)^\intercal \boldsymbol{v}^I \\ &= \frac{1}{n^r} \boldsymbol{v}^a \sum_{b \in B^r} \left(\boldsymbol{v}^b\right)^\intercal \boldsymbol{v}^I = \frac{1}{n^r} \boldsymbol{v}^a \sum_{b \in B^r} 1 = \frac{1}{n^r} \boldsymbol{v}^a \cdot n^r = \boldsymbol{v}^a \\ &\Rightarrow \left(\boldsymbol{v}^a\right)^\intercal H_1 \left(D^P \boldsymbol{v}^I\right) = \left(\boldsymbol{v}^a\right)^\intercal H_1 \left(\boldsymbol{v}^a\right) = \left(\boldsymbol{v}^a\right)^\intercal \boldsymbol{v}^a = 1 \end{aligned} \tag{3.30}$$

as required.

- If $a \notin T_P(I)$, then for every rule $r \in P$ with $h^r = a$ we have $B^r \not\subseteq I$. For every such rule, there exists $b \in B^r$ such that $\left(\boldsymbol{v}^b\right)^\intercal \boldsymbol{v}^I = 0$. Let $\{r_1, r_2, \ldots, r_k\}$ be the set of rules with $h^r = a$. We have:

$$\begin{aligned} \left(\boldsymbol{v}^a\right)^\intercal H_1 \left(D^P \boldsymbol{v}^I\right) &= \left(\boldsymbol{v}^a\right)^\intercal H_1 \left(\sum_{i=1}^k D^{r_i} \boldsymbol{v}^I\right) = \left(\boldsymbol{v}^a\right)^\intercal H_1 \left(\sum_{i=1}^k \frac{1}{n^{r_i}} \boldsymbol{v}^a \left(\boldsymbol{v}^{B^{r_i}}\right)^\intercal \boldsymbol{v}^I\right) = \\ &= \left(\boldsymbol{v}^a\right)^\intercal H_1 \left(\sum_{i=1}^k \frac{\left(\boldsymbol{v}^{B^{r_i}}\right)^\intercal \boldsymbol{v}^I}{n^{r_i}}\right) \boldsymbol{v}^a = H_1 \left(\sum_{i=1}^k \frac{\left(\boldsymbol{v}^{B^{r_i}}\right)^\intercal \boldsymbol{v}^I}{n^{r_i}}\right) \end{aligned} \tag{3.31}$$

Since P satisfies the MD condition, there exists at most one long rule with a as its head. For such a rule, we have $\left(\boldsymbol{v}^{B^{r_i}}\right)^\intercal \boldsymbol{v}^I < n^{r_i}$. All other rules are short, so we have $\left(\boldsymbol{v}^{B^{r_i}}\right)^\intercal \boldsymbol{v}^I = 0$. Putting it all together, we have:

$$\sum_{i=1}^k \frac{\left(\boldsymbol{v}^{B^{r_i}}\right)^\intercal \boldsymbol{v}^I}{n^{r_i}} < 1 \tag{3.32}$$

and therefore:

$$\left(\boldsymbol{v}^a\right)^\intercal H_1 \left(D^P \boldsymbol{v}^I\right) = H_1 \left(\sum_{i=1}^k \frac{\left(\boldsymbol{v}^{B^{r_i}}\right)^\intercal \boldsymbol{v}^I}{n^{r_i}}\right) = 0 \tag{3.33}$$

as required.

□

Example 3.2.7. *Let P be the program from Equation 3.26 and $I = \{q, s, r\}$. We have:*

$$H_1 \left(D^P \boldsymbol{v}^I\right) = H_1 \left(\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 0 \end{bmatrix} \right) = H_1 \left(\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ \frac{1}{2} \\ 0 \\ 1 \end{bmatrix} \right) = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{3.34}$$

which corresponds to the interpretation $\{p, t\} = T_P(I)$. Note that without adding the tautology $\top \leftarrow \top$ to P, the entry for $\top$ would have become 0 after this operation, resulting in a non-interpretation.

An implication of the above theorem is that the Least Herbrand Model of every definite program can be computed in a vector space. Recall that when applying the Tarski operator of a

definite program in succession starting with the empty interpretation, a fixed point is guaranteed to be reached, which is equal to the least Herbrand model of the program. Due to Theorem 3.2.1, the same can be said when computing the immediate consequence in a vector space, starting with the vector representation of the empty interpretation. This is formalised in the following algorithm:

Algorithm 2 Computing $LHM\left(P\right)$ in Vector Space

Pre-process P to ensure it satisfies the MD condition, has no empty rule bodies and contains the tautology $\top \leftarrow \top$ to receive the program $P^{'}$.

Set an initial vector to the representation of the empty interpretation:

$$\boldsymbol{v}^0 = \boldsymbol{v}^{\emptyset} = \begin{bmatrix} 0 & 1 & 0 & \dots & 0 \end{bmatrix}^\intercal$$

repeat

Compute $\boldsymbol{v}^{k+1} = H_1\left(D^{P^{'}}\boldsymbol{v}^k\right)$

until $\boldsymbol{v}^{k+1} = \boldsymbol{v}^k$

Let I be the interpretation corresponding to the vector $\boldsymbol{v}^k$, the fixed point computed above. Return $I \cap B_P$.

Necessity of the MD Condition

Consider the following logic program, P_1 that does not follow the MD condition:

$$\begin{aligned} h &\leftarrow p, q \\ h &\leftarrow r, s \end{aligned} \tag{3.35}$$

For this program we would have the following matrix:

$$D^{P^1} = \begin{array}{c} \\ \bot \\ \top \\ h \\ p \\ q \\ r \\ s \end{array} \begin{array}{c} \begin{array}{ccccccc} \bot & \top & h & p & q & r & s \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array} \tag{3.36}$$

Now, note that the matrix above also represents the following program, P_2:

$$\begin{aligned} h &\leftarrow p, r \\ h &\leftarrow q, s \end{aligned} \tag{3.37}$$

and therefore could not possibly correctly compute the immediate consequence for both. Indeed, for an interpretation $I = \{p, r\}$ we would have:

$$H_1\left(D^{P^1}\boldsymbol{v}^I\right) = H_1\left(\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}\right) = H_1\left(\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}\right) = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{3.38}$$

suggesting $T_{P_1}(I) = \{h\}$, which is wrong. The MD condition is therefore necessary for the correctness of Theorem 3.2.1.

3.2.2 Embedding Program Reduct

To generalise the matrix representation to normal programs, we utilise the concept of a program reduct. For a given interpretation I of program P, P^I is a definite program. Theoretically, one could first compute P^I from P and I and then embed it in a matrix as in the previous subsections. However, there are benefits for embedding the program reduct directly from the original program, as we shall see later.

We begin by modifying the embedding of rules to account for negation-as-failure. In a program reduct, a rule is either deleted, if any of the atoms in the negative body are true in the interpretation, or otherwise only the negative body is deleted. Our rule embedding is aimed at capturing this logic.

Definition 3.2.6. *(Program reduct clause embedding) Let $r \in P$ be a normal clause, where $|B_P| = N$, $|B^r| = n^r \geq 1$ and $|C^r| = m^r \geq 1$. Let I be an interpretation of P. The embedding of r with respect to I is a matrix $D^{r,I} \in \mathbb{R}^{N\times N}$, which is computed as follows:*

$$D^{r,I} = \frac{1}{n^r}\frac{\left(\mathbf{1} - \boldsymbol{v}^I\right)^\intercal \boldsymbol{v}^{C^r}}{m^r}\boldsymbol{v}^{h^r}\left(\boldsymbol{v}^{B^r}\right)^\intercal \tag{3.39}$$

When compared to the embedding of a definite clause, this definition adds the extra term $\frac{\left(\mathbf{1}-\boldsymbol{v}^I\right)^\intercal \boldsymbol{v}^{C^r}}{m^r}$ which is a scalar. The value $\left(\mathbf{1} - \boldsymbol{v}^I\right)^\intercal \boldsymbol{v}^{C^r}$ is equal to the number of atoms in C^r that are not in I, and ranges from 0 to m^r. Therefore, when $C^r \cap I = \emptyset$, the extra term will equal exactly one, and we recover the embedding representation of the positive part of the rule r, essentially "removing" C^r. If $C^r \cap I \neq \emptyset$, the extra term will be less than 1, where the H_1 thresholding operation will ensure it never "fires". This is semantically equivalent to removing the rule from the program. In this way, the operation of computing P^I is simulated in linear algebra.

Definition 3.2.7. *(Program reduct embedding) Let P be a normal program I an interpretation. The embedding of the program reduct P^I is given by the sum of the embedding of its rules. Formally:*

$$D^{P,I} = \sum_{r\in P} D^{r,I} \tag{3.40}$$

Example 3.2.8. *Consider the program from Equation 3.16*

$$\begin{aligned} p &\leftarrow not\ q \\ q &\leftarrow not\ p \\ r &\leftarrow p, s, not\ q, not\ t \\ t &\leftarrow p, not\ s, not\ r \end{aligned} \tag{3.41}$$

and the interpretation $I = \{p, r, t\}$*. The matrix representation of the clause* $p \leftarrow not\ q$ *is computed as follows:*

$$\begin{aligned} D^{p \leftarrow not\ q, \{p,r,t\}} &= \frac{1}{|\{\top\}|} \frac{\left(\mathbf{1} - \boldsymbol{v}^{\{p,r,t\}}\right)^\intercal \boldsymbol{v}^{\{q\}}}{|\{q\}|} \boldsymbol{v}^p \left(\boldsymbol{v}^{\{\top\}}\right)^\intercal = \\ &= \frac{1}{1} \frac{\left(\begin{bmatrix} 1\\1\\1\\1\\1\\1\\1 \end{bmatrix} - \begin{bmatrix} 0\\1\\1\\0\\1\\0\\1 \end{bmatrix} \right)^\intercal \begin{bmatrix} 0\\0\\0\\1\\0\\0\\0 \end{bmatrix}}{1} \begin{bmatrix} 0\\0\\1\\0\\0\\0\\0 \end{bmatrix} \left(\begin{bmatrix} 0\\1\\0\\0\\0\\0\\0 \end{bmatrix} \right)^\intercal = \\ &= \left(\begin{bmatrix} 1\\0\\0\\1\\0\\1\\0 \end{bmatrix} \right)^\intercal \begin{bmatrix} 0\\0\\0\\1\\0\\0\\0 \end{bmatrix} \begin{bmatrix} 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0 \end{bmatrix} = \\ &= 1 \cdot \begin{bmatrix} 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0 \end{bmatrix} = \begin{bmatrix} 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0 \end{bmatrix} \end{aligned} \tag{3.42}$$

the resulting matrix represents the definite clause $p \leftarrow$ *as required.*

For the clause $r \leftarrow p, s, not\ q, not\ t$ *we have:*

$$D^{p\leftarrow p,s,not\ q,not\ t,\{p,r,t\}} = \frac{1}{|\{p,s\}|}\frac{\left(\mathbf{1}-\boldsymbol{v}^{\{p,r,t\}}\right)^{\intercal}\boldsymbol{v}^{\{q,t\}}}{|\{q,t\}|}\boldsymbol{v}^{r}\left(\boldsymbol{v}^{\{p,s\}}\right)^{\intercal} =$$

$$= \frac{1}{2}\frac{\left(\begin{bmatrix}1\\1\\1\\1\\1\\1\\1\\1\end{bmatrix}-\begin{bmatrix}0\\1\\1\\0\\1\\0\\0\\1\end{bmatrix}\right)^{\intercal}\begin{bmatrix}0\\0\\0\\1\\0\\0\\0\\1\end{bmatrix}}{2}\begin{bmatrix}0\\0\\0\\0\\1\\0\\0\\0\end{bmatrix}\left(\begin{bmatrix}0\\0\\1\\0\\0\\1\\0\\0\end{bmatrix}\right)^{\intercal} = \tag{3.43}$$

$$= \frac{1}{2}\left(\begin{bmatrix}1\\0\\0\\1\\0\\1\\0\\0\end{bmatrix}\right)^{\intercal}\begin{bmatrix}0\\0\\0\\1\\0\\0\\0\\1\end{bmatrix}\begin{bmatrix}0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\0&0&\frac{1}{2}&0&0&\frac{1}{2}&0\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\end{bmatrix} =$$

$$= \frac{1}{2}\cdot\begin{bmatrix}0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\0&0&\frac{1}{2}&0&0&\frac{1}{2}&0\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\end{bmatrix} = \begin{bmatrix}0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\0&0&\frac{1}{4}&0&0&\frac{1}{4}&0\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\end{bmatrix}$$

any interpretation vector multiplied by this matrix will not result in any entry equal to 1. Therefore this rule will never fire. Essentially, the rule $r \leftarrow p, s, not\ q, not\ t$ *is deleted, as required. After computing the matrix for the other two rules and the tautology* $\top \leftarrow \top, not\ \bot$*, and adding all resulting matrices, we get:*

$$D^{P,I} = \begin{array}{c} \\ \bot \\ \top \\ p \\ q \\ r \\ s \\ t \end{array}\begin{array}{c}\begin{array}{ccccccc}\bot & \top & p & q & r & s & t\end{array}\\\begin{bmatrix}0&0&0&0&0&0&0\\0&1&0&0&0&0&0\\0&1&0&0&0&0&0\\0&0&0&0&0&0&0\\0&0&\frac{1}{4}&0&0&\frac{1}{4}&0\\0&0&0&0&0&0&0\\0&0&\frac{1}{2}&0&0&0&0\end{bmatrix}\end{array} \tag{3.44}$$

Immediate Consequences for Program Reduct

Theorem 3.2.2. *Let* P *be a normal program satisfying the MD condition and let* I, M *be interpretations.* $\boldsymbol{v}^{T_{P^M}(I)} = H_1\left(D^{P,M}\right)\boldsymbol{v}^{I}$.

Proof. For any $p \in B_P$ we have:

$$\begin{aligned}(\boldsymbol{v}^p)^\intercal H_1\left(D^{P,M}\boldsymbol{v}^I\right) &= (\boldsymbol{v}^p)^\intercal H_1\left(\sum_{r\in P, h^r=p}\frac{1}{n^r}\frac{\left(\mathbf{1}-\boldsymbol{v}^I\right)^\intercal \boldsymbol{v}^{C^r}}{m^r}\boldsymbol{v}^p\left(\boldsymbol{v}^{B^r}\right)^\intercal \boldsymbol{v}^I\right) = \\ &= (\boldsymbol{v}^p)^\intercal H_1\left(\sum_{r\in P, h^r=p}\frac{\left(\mathbf{1}-\boldsymbol{v}^I\right)^\intercal \boldsymbol{v}^{C^r}\left(\boldsymbol{v}^{B^r}\right)^\intercal \boldsymbol{v}^I}{n^r m^r}\right)\boldsymbol{v}^p = \\ &= H_1\left(\sum_{r\in P, h^r=p}\frac{\left(\mathbf{1}-\boldsymbol{v}^I\right)^\intercal \boldsymbol{v}^{C^r}\left(\boldsymbol{v}^{B^r}\right)^\intercal \boldsymbol{v}^I}{n^r m^r}\right)\end{aligned} \tag{3.45}$$

If $p \in T_{P^M}(I)$, then there exists a rule $r \in P^M$ such that $B^r \subseteq I$ and $M \cap C^r = \emptyset$. Therefore $\left(\boldsymbol{v}^{B^r}\right)^\intercal \boldsymbol{v}^I = n^r$ and $\left(\mathbf{1}-\boldsymbol{v}^M\right)^\intercal \boldsymbol{v}^{C^r} = m^{C^r}$ which implies $\frac{\left(\mathbf{1}-\boldsymbol{v}^I\right)^\intercal \boldsymbol{v}^{C^r}\left(\boldsymbol{v}^{B^r}\right)^\intercal \boldsymbol{v}^I}{n^r m^r} = 1$ and therefore:

$$H_1\left(\sum_{r\in P, h^r=p}\frac{\left(\mathbf{1}-\boldsymbol{v}^I\right)^\intercal \boldsymbol{v}^{C^r}\left(\boldsymbol{v}^{B^r}\right)^\intercal \boldsymbol{v}^I}{n^r m^r}\right) = 1 \tag{3.46}$$

If $p \notin T_{P^M}(I)$, for a rule r such that $h^r = p$, there are two cases:

1. r is the single long rule with p as its head, then either $B^r \not\subseteq I$ and hence $\left(\boldsymbol{v}^{B^r}\right)^\intercal \boldsymbol{v}^I < n^r$ or $C^r \cap M \neq \emptyset$ and hence $\left(\mathbf{1}-\boldsymbol{v}^M\right)^\intercal \boldsymbol{v}^{C^r} < m^r$. Either way we have $\frac{\left(\mathbf{1}-\boldsymbol{v}^I\right)^\intercal \boldsymbol{v}^{C^r}\left(\boldsymbol{v}^{B^r}\right)^\intercal \boldsymbol{v}^I}{n^r m^r} < 1$.
2. r is a short rule with p as its head. Then either $B^r \not\subseteq I$ and hence $\left(\boldsymbol{v}^{B^r}\right)^\intercal \boldsymbol{v}^I = 0$ or $C^r \cap M \neq \emptyset$ and hence $\left(\mathbf{1}-\boldsymbol{v}^M\right)^\intercal \boldsymbol{v}^{C^r} = 0$. Either way we have $\frac{\left(\mathbf{1}-\boldsymbol{v}^I\right)^\intercal \boldsymbol{v}^{C^r}\left(\boldsymbol{v}^{B^r}\right)^\intercal \boldsymbol{v}^I}{n^r m^r} = 0$.

Putting these cases together we get:

$$H_1\left(\sum_{r\in P, h^r=p}\frac{\left(\mathbf{1}-\boldsymbol{v}^I\right)^\intercal \boldsymbol{v}^{C^r}\left(\boldsymbol{v}^{B^r}\right)^\intercal \boldsymbol{v}^I}{n^r m^r}\right) = 0. \tag{3.47}$$

Hence $p \in T_{P^M}(I)$ implies the entry for p in $H_1\left(D^{P,M}\right)\boldsymbol{v}^I$ is 1 and $p \notin T_{P^M}(I)$ implies the entry for p in $H_1\left(D^{P,M}\right)\boldsymbol{v}^I$ is 0. □

Example 3.2.9. *Consider the program in Equation 3.16 and interpretations $I = \{p, r, t\}$ and $J = \{p, s\}$. Then:*

$$H_1\left(D^{P,I}\boldsymbol{v}^J\right) = H_1\left(\begin{bmatrix}0&0&0&0&0&0&0\\0&1&0&0&0&0&0\\0&1&0&0&0&0&0\\0&0&0&0&0&0&0\\0&0&\frac{1}{4}&0&0&\frac{1}{4}&0\\0&0&0&0&0&0&0\\0&0&\frac{1}{2}&0&0&0&0\end{bmatrix}\cdot\begin{bmatrix}0\\1\\1\\0\\0\\1\\0\end{bmatrix}\right) = H_1\left(\begin{bmatrix}0\\1\\1\\0\\\frac{1}{2}\\0\\\frac{1}{2}\end{bmatrix}\right) = \begin{bmatrix}0\\1\\1\\0\\0\\0\\0\end{bmatrix} \tag{3.48}$$

which corresponds to the interpretation $\{p\}$ which is equal to $T_{P^I}(J)$, as required.

A corollary of Theorem 3.2.2 is that for supported model M of a normal P the following equality holds:

$$\boldsymbol{v}^M = H_1\left(D^{P,M}\boldsymbol{v}^M\right) \tag{3.49}$$

which is the vector space equivalent of the equality $T_{P^M}(M) = M$ which holds for all supported models. Solving this equation therefore amounts to computing supported models. However, it

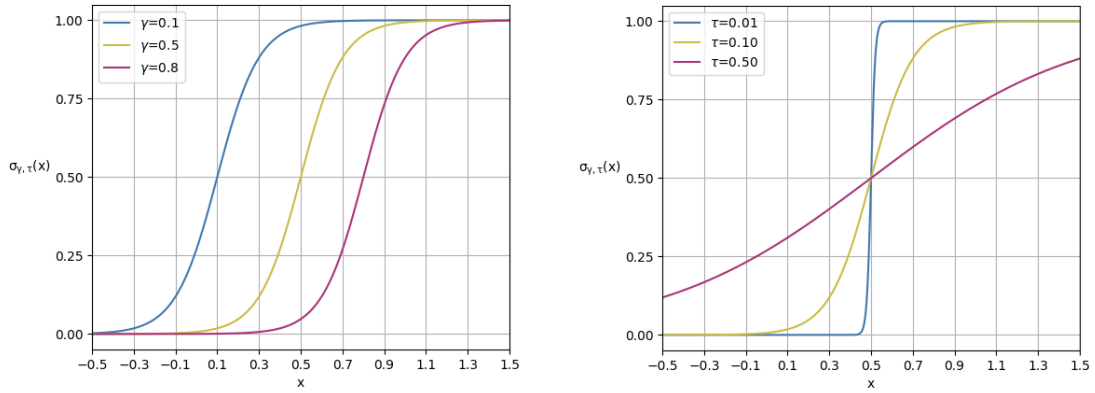


Figure 3.1: Graphs of $\sigma_{\gamma,\tau}(x)$ for various values of γ and τ. Left: varying γ. Right: varying τ.

is difficult to solve for $\boldsymbol{v}^M$ since H_1 is a non-differentiable operation. For this reason, in the next section, we replace H_1 with a differentiable approximation.

3.3 Differentiable Immediate Consequences

The function H_1, being a Heaviside Step function, has several differentiable approximations (Bracewell, 2000). In this work, we choose to focus on one, specifically Sigmoid. Other common choices are *tanh* and ReLU. *tanh* is in fact simply a sigmoid whose image is extended to $(-1, 1)$, as can be seen by the identity $tanh(x) = 2\sigma(2x) - 1$. But we require our truth values to be between 0 and 1. For x values between 0 and 1, $ReLU(x) = x$. In other words, it is a linear function which is insufficient to realise the logical operations of conjunction and disjunction. It therefore would not allow us to maintain two-valued semantics as desired. For this reason, we choose the sigmoid function, although other suitable activation functions may exist.

Definition 3.3.1. *(Sigmoid approximation of H_1) For $0 < \gamma < 1$ and $\tau > 0$, define:*

$$\sigma_{\gamma,\tau}(x) = \frac{1}{1 + \exp\left(\frac{\gamma - x}{\tau}\right)} \tag{3.50}$$

The effect of γ and τ on the approximation can be seen best in Figure 3.1. γ is the x value for which $\sigma_{\gamma,\tau}(x) = \frac{1}{2}$ and can thus be seen as a boundary between real values representing 'True' $(x > \gamma)$ and 'False' $(x < \gamma)$. τ is called 'temperature' and affects the slope of the function. When $\tau \to 0$, the slope of $\sigma_{\gamma,\tau}(x)$ becomes infinite and the approximation of the heaviside function is better, but the gradient everywhere else approaches 0, becoming less useful for optimisation. Crucially, we would like to ensure that exact inference is maintained when replacing H_1 with $\sigma_{\gamma,\tau}$. We must therefore select the values of γ and τ carefully.

3.3.1 Constraints for Exact Inference

Consider a clause:

$$h^r \leftarrow b_1, b_2, ..., b_{n^r}, \text{not } c_1, \text{not } c_2, ..., \text{not } c_{m^r} \tag{3.51}$$

For exact inference to work, we need the following conditions to hold:

- If all $b_i's$ have a truth value greater than γ, and all $c_i's$ have a truth value less than γ, then h^r is "true" and it should receive a value greater than γ.

- if any of the b_i's have a value less than γ or any of the c_i's greater than γ, then h^r is not proven, so it should not receive a value greater than γ (at least not by this rule).

To ensure these hold, we are going to derive constraints on the values of γ and τ. These constraints are going to be based solely on the structure of the program, namely the lengths of the rules and the number of rules with the same head. The following is a derivation of the conditions on γ and τ that guarantee exact inference.

Let $p \in B_P$. We denote:

$$\text{rules}\,(p) = \{r \in P \mid h^r = p\} \tag{3.52}$$

.

$$R_p = |\text{rules}\,(p)| \tag{3.53}$$

Let M and I be interpretations of P. When replacing H_1 with $\sigma_{\gamma,\tau}$, the truth value of p after a single step of deduction is equal to:

$$\sigma_{\gamma,\tau}\left(\sum_{r\in\text{rules}(p)} \frac{\left(\boldsymbol{v}^{B^r}\right)^\intercal \boldsymbol{v}^I \left(\mathbf{1}-\boldsymbol{v}^M\right)^\intercal \boldsymbol{v}^{C^r}}{n^r m^r}\right) \tag{3.54}$$

(see similar derivation in Equation 3.45). Further, we denote:

$$x^p = \sum_{r\in\text{rules}(p)} \frac{\left(\boldsymbol{v}^{B^r}\right)^\intercal \boldsymbol{v}^I \left(\mathbf{1}-\boldsymbol{v}^M\right)^\intercal \boldsymbol{v}^{C^r}}{n^r m^r} \tag{3.55}$$

The entries of $\boldsymbol{v}^M$, $\boldsymbol{v}^I$ are defined to be 0 or 1. However, when performing gradient-based optimisation these will be replaced by real-valued vectors, resulting in a value for p that can range from 0 to 1. Later we will formally define the semantic meaning of real-valued vectors with regard to interpretations of the program, for now, we will treat $\boldsymbol{v}^M$ and $\boldsymbol{v}^I$ as real-valued vectors while we derive the constraints for γ and τ.

We begin with a straightforward case: suppose p is the head of a single rule:

$$p \leftarrow a \tag{3.56}$$

If a is true in I, the value of p must be true after a single deduction step. We have $(\boldsymbol{v}^a)^\intercal \boldsymbol{v}^I > \gamma$. In particular:

$$x^p = \frac{\left(\boldsymbol{v}^{\{a\}}\right)^\intercal \boldsymbol{v}^I \left(\mathbf{1}-\boldsymbol{v}^M\right)^\intercal \boldsymbol{v}^{\{\bot\}}}{1} > \gamma \tag{3.57}$$

And since p must be true after applying $\sigma_{\gamma,\tau}$, we demand:

$$\sigma_{\gamma,\tau}\left(x^p\right) > \gamma \tag{3.58}$$

for all $x^p > \gamma$. And as $\sigma_{\gamma,\tau}$ is monotonically increasing this condition becomes:

$$\begin{aligned} &\sigma_{\gamma,\tau}\left(x^p\right) > \sigma_{\gamma,\tau}\left(\gamma\right) > \gamma \\ &\Rightarrow \quad \frac{1}{1+\exp\left(\frac{\gamma-\gamma}{\tau}\right)} > \gamma \quad \Rightarrow \quad \gamma < \frac{1}{2} \end{aligned} \tag{3.59}$$

On the other hand, if a is false, since this is the only rule with p as its head, p must be false.

This time we have $(\boldsymbol{v}^a)^\intercal \boldsymbol{v}^I < \gamma$ and therefore $x^p < \gamma$, leading to:

$$\begin{aligned} &\sigma_{\gamma,\tau}\left(x^p\right) < \sigma_{\gamma,\tau}\left(\gamma\right) < \gamma \\ \Rightarrow \quad &\frac{1}{1+\exp\left(\frac{\gamma-\gamma}{\tau}\right)} < \gamma \quad \Rightarrow \quad \gamma > \frac{1}{2} \end{aligned} \tag{3.60}$$

So γ must be both greater and smaller than $\frac{1}{2}$ to ensure our deduction is sound. Clearly, this will not work. To remedy the situation, we introduce two further hyperparameters $\gamma^\perp, \gamma^\top$ such that $0 < \gamma^\perp < \gamma < \gamma^\top < 1$. $\gamma^\perp$ is an upper bound on false values that variables can take. $\gamma^\top$ is a lower bound on true values. Values between $\gamma^\perp$ and $\gamma^\top$ are transitional and are not considered true or false.

Having introduced the necessary hyperparameters, let us consider the general case. Suppose there exists a rule $r \in \text{rules}\left(p\right)$ whose body is satisfied (so that p must be deduced). We have:

$$x^p = \sum_{r' \in \text{rules}(p)} \frac{\left(\boldsymbol{v}^{B^{r'}}\right)^\intercal \boldsymbol{v}^I \left(\mathbf{1}-\boldsymbol{v}^M\right)^\intercal \boldsymbol{v}^{C^{r'}}}{n^{r'} m^{r'}} \geq \frac{\left(\boldsymbol{v}^{B^r}\right)^\intercal \boldsymbol{v}^I \left(\mathbf{1}-\boldsymbol{v}^M\right)^\intercal \boldsymbol{v}^{C^r}}{n^r m^r} \tag{3.61}$$

The second inequality being true as the sum is over non-negative numbers. For each $b \in B^r$ we have $\left(\boldsymbol{v}^b\right)^\intercal \boldsymbol{v}^I \geq \gamma^\top$ and for each $c \in C^r$ we have $(\boldsymbol{v}^c)^\intercal \boldsymbol{v}^M \leq \gamma^\perp$. Plugging these in we get:

$$x^p \geq \frac{n^r \gamma^\top m^r \left(1-\gamma^\perp\right)}{n^r m^r} = \gamma^\top (1-\gamma^\perp) \tag{3.62}$$

And therefore our condition to ensure p is deduced must be:

$$\begin{aligned} &\sigma_{\gamma,\tau}\left(x^p\right) \geq \sigma_{\gamma,\tau}\left(\gamma^\top(1-\gamma^\perp)\right) \geq \gamma^\top \\ \Rightarrow \quad &\frac{1}{1+\exp\left(\frac{\gamma-\gamma^\top(1-\gamma^\perp)}{\tau}\right)} \geq \gamma^\top \\ \Rightarrow \quad &\frac{1}{\gamma^\top} - 1 \geq \exp\left(\frac{\gamma-\gamma^\top(1-\gamma^\perp)}{\tau}\right) \\ \Rightarrow \quad &\ln\left(\frac{1}{\gamma^\top}-1\right) \geq \frac{\gamma-\gamma^\top(1-\gamma^\perp)}{\tau} \\ \Rightarrow \quad &\tau \leq \frac{\gamma-\gamma^\top(1-\gamma^\perp)}{\ln\left(\frac{1}{\gamma^\top}-1\right)} \end{aligned} \tag{3.63}$$

To ensure τ is positive, we also add the following constraints:

$$\gamma < \gamma^\top \left(1-\gamma^\perp\right) \tag{3.64}$$

$$\ln\left(\frac{1}{\gamma^\top}-1\right) < 0 \quad \Rightarrow \quad \gamma^\top > \frac{1}{2} \tag{3.65}$$

Now, suppose for all rules in $\text{rules}\left(p\right)$ the body of the rule is not satisfied, so that p must not be deduced. We wish to find an upper bound for x^p, which we denote $x^{p,\perp}$, so that we can demand $\sigma_{\gamma,\tau}\left(x^{p,\perp}\right) \leq \gamma^\perp$. The contribution of a rule $r \in \text{rules}\left(p\right)$ to $x^{p,\perp}$ is given by:

$$\frac{\left(\boldsymbol{v}^{B^r}\right)^\intercal \boldsymbol{v}^I \left(\mathbf{1}-\boldsymbol{v}^M\right)^\intercal \boldsymbol{v}^{C^r}}{n^r m^r} \tag{3.66}$$

which receives its maximum value under one of two conditions:

- All atoms in B^r are true in I except one, while all atoms in C^r are false in M. At most, all true atoms of B^r would have a value of 1 and the one false atom would have a value of $\gamma^\perp$, while all false atoms of C^r have a value of 0. The contribution of r to $x^{p,\perp}$ is then:

$$\frac{n^r - 1 + \gamma^\perp}{n^r} \tag{3.67}$$

- All atoms in B^r are true in I while all atoms in C^r are false in M except one. At most, all true atoms of B^r would have a value of 1, while all false atoms of C^r have a value of 0 and the true atom has a value of $\gamma^\top$. The contribution of r to $x^{p,\perp}$ is then:

$$\frac{m^r - \gamma^\top}{m^r} \tag{3.68}$$

Therefore, for each rule $r \in \text{rules}(p)$, its contribution to $x^{p,\perp}$ is given by the maximum of the two values. So we have:

$$x^{p,\perp} = \sum_{r \in \text{rules}(p)} \max\left\{\frac{n^r - 1 + \gamma^\perp}{n^r}, \frac{m^r - \gamma^\top}{m^r}\right\} \tag{3.69}$$

Our condition to ensure p is not deduced is therefore:

$$\begin{aligned}
&\sigma_{\gamma,\tau}\left(x^p\right) \leq \sigma_{\gamma,\tau}\left(x^{p,\perp}\right) \leq \gamma^\perp \\
\Rightarrow \quad &\frac{1}{1 + \exp\left(\frac{\gamma - x^{p,\perp}}{\tau}\right)} \leq \gamma^\perp \\
\Rightarrow \quad &\frac{1}{\gamma^\perp} - 1 \leq \exp\left(\frac{\gamma - x^{p,\perp}}{\tau}\right) \\
\Rightarrow \quad &\ln\left(\frac{1}{\gamma^\perp} - 1\right) \leq \frac{\gamma - x^{p,\perp}}{\tau} \\
\Rightarrow \quad &\tau \leq \frac{\gamma - x^{p,\perp}}{\ln\left(\frac{1}{\gamma^\perp} - 1\right)}
\end{aligned} \tag{3.70}$$

As τ is a global parameter we need to generalise this condition to all $p \in B_P$. This can be done by plugging the maximum value of $x^{p,\perp}$ over all p:

$$x^\perp = \max_{p \in B_P} x^{p,\perp} \tag{3.71}$$

into Equation 3.70:

$$\tau \leq \frac{\gamma - x^\perp}{\ln\left(\frac{1}{\gamma^\perp} - 1\right)} \tag{3.72}$$

To ensure τ is positive, we also add the following constraints:

$$\gamma > x^\perp \tag{3.73}$$

$$\ln\left(\frac{1}{\gamma^\perp} - 1\right) > 0 \quad \Rightarrow \quad \gamma^\perp < \frac{1}{2} \tag{3.74}$$

To summarise, our analysis shows the conditions on $\gamma, \tau, \gamma^\bot, \gamma^\top$ to ensure sound deduction using $\sigma_{\gamma,\tau}$ are as follows:

1. $\gamma^\top > \frac{1}{2}$
2. $\gamma^\bot < \frac{1}{2}$
3. $\gamma < \gamma^\top \left(1 - \gamma^\bot\right)$
4. $\gamma > x^\bot$
5. $\tau \leq \frac{\gamma - \gamma^\top\left(1-\gamma^\bot\right)}{\ln\left(\frac{1}{\gamma^\top}-1\right)}$
6. $\tau \leq \frac{\gamma - x^\bot}{\ln\left(\frac{1}{\gamma^\bot}-1\right)}$

what remains to show is that for every normal program that satisfies the MD condition, there exist $0 < \gamma^\bot < \gamma < \gamma^\top < 1$ and $\tau > 0$ that satisfy the above conditions. Before showing this, we will first formalise the analysis above.

3.3.2 Deduction over Continuous Vectors

The following definition generalises the concept of interpretation embedding to real-valued vectors.

Definition 3.3.2. *(Semantic equivalence between vectors) Suppose $\boldsymbol{v}, \boldsymbol{u} \in [0,1]^N$ and let $0 < \gamma^\bot < \gamma^\top < 1$. We say $\boldsymbol{v}$ and $\boldsymbol{u}$ are semantically equivalent if for all $1 \leq i \leq N$:*

1. *$v_i \geq \gamma^\top$ if and only if $u_i \geq \gamma^\top$.*
2. *$v_i \leq \gamma^\bot$ if and only if $u_i \leq \gamma^\bot$.*

In such a case we write $\boldsymbol{v} \sim_{\gamma^\bot}^{\gamma^\top} \boldsymbol{u}$.

A vector $\boldsymbol{v}$ in $[0,1]^N$ can therefore be semantically equivalent to $\boldsymbol{v}^I$ for some interpretation I, essentially representing the interpretation.

Example 3.3.1. *Suppose the Herbrand base of a program is $\{\bot, \top, p, q\}$ and $\gamma^\bot = 0.25$, $\gamma^\top = 0.75$. The following vectors are semantically equivalent:*

$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \sim_{\gamma^\bot}^{\gamma^\top} \begin{bmatrix} 0.1 \\ 0.9 \\ 0.85 \\ 0.15 \end{bmatrix} \tag{3.75}$$

the vector on the left representing the interpretation $\{p\}$, and the vector on the right being semantically equivalent to it, essentially representing the same interpretation using real values.

Note that if for some entry of $\boldsymbol{v}$ we have $\gamma^\bot < v_i < \gamma^\top$ then it does not represent any interpretation. Next, we generalise the embedding of program reduct to continuous vectors.

Definition 3.3.3. *(Continuous program reduct embedding) Let P be a normal program and $\boldsymbol{v} \in [0,1]^N$. The embedding of the continuous program reduct $P^{\boldsymbol{v}}$ is given by:*

$$D^{P,\boldsymbol{v}} = \sum_{r \in P} \frac{(\mathbf{1} - \boldsymbol{v})^\intercal \, \boldsymbol{v}^{C^r} \boldsymbol{v}^{h^r} \left(\boldsymbol{v}^{B^r}\right)^\intercal}{n^r m^r} \tag{3.76}$$

The summand term in Equation 3.76 is almost identical to Equation 3.39, except the interpretation vector $\boldsymbol{v}^I$ has been replaced with the continuous vector $\boldsymbol{v}$. The following theorem is a generalisation of Theorem 3.2.2 to the continuous case that formalises the analysis above.

Theorem 3.3.1. *Let P be a normal program and I, M be interpretations. Let $0 < \gamma^\perp < \gamma < \gamma^\top < 1$ and $\tau > 0$ such that:*

1. $\gamma^\top > \frac{1}{2}$
2. $\gamma^\perp < \frac{1}{2}$
3. $\gamma < (1-\gamma^\perp)\gamma^\top$
4. $\gamma > x^\perp$
5. $\tau \leq \min\left\{ \frac{\gamma-(1-\gamma^\perp)\gamma^\top}{\ln(\frac{1}{\gamma^\top}-1)}, \frac{\gamma-x^\perp}{\ln(\frac{1}{\gamma^\perp}-1)} \right\}$

Suppose that for vectors $\boldsymbol{v}, \boldsymbol{u} \in [0,1]^N$ we have $\boldsymbol{v} \sim_{\gamma^\perp}^{\gamma^\top} \boldsymbol{v}^M$ and $\boldsymbol{u} \sim_{\gamma^\perp}^{\gamma^\top} \boldsymbol{v}^I$. Then:

$$H_1\left(D^{P,M}\boldsymbol{v}^I\right) \sim_{\gamma^\perp}^{\gamma^\top} \sigma_{\gamma,\tau}\left(D^{P,\boldsymbol{v}}\boldsymbol{u}\right) \tag{3.77}$$

Proof. Define:

$$\boldsymbol{w} = H_1\left(D^{P,M}\boldsymbol{v}^I\right) \qquad \boldsymbol{z} = \sigma_{\gamma,\tau}\left(D^{P,\boldsymbol{v}}\boldsymbol{u}\right) \tag{3.78}$$

and let $p \in B_{P^M}$. Recall:

$$x^p = \sum_{r \in \text{rules}(p)} \frac{\left(\boldsymbol{v}^{B^r}\right)^\intercal \boldsymbol{v}^I \left(\boldsymbol{1}-\boldsymbol{v}^M\right)^\intercal \boldsymbol{v}^{C^r}}{n^r m^r} \tag{3.79}$$

Suppose $p \in T_{P^M}(I)$. From Theorem 3.2.2 we have $(\boldsymbol{v}^p)^\intercal \boldsymbol{w} = 1$. In addition, there exists a rule $r \in P^M$ such that $p = h^r$, $B^r \subseteq I$ and $C^r \cap M = \emptyset$. Therefore:

- For every $b \in B^r$ we have $\left(\boldsymbol{v}^b\right)^\intercal \boldsymbol{v}^I = 1$, therefore $\left(\boldsymbol{v}^b\right)^\intercal \boldsymbol{u} \geq \gamma^\top$.
- For every $c \in C^r$ we have $(\boldsymbol{v}^c)^\intercal \boldsymbol{v}^I = 0$, therefore $(\boldsymbol{v}^c)^\intercal \boldsymbol{u} \leq \gamma^\perp$.

Therefore $x^p \geq \left(1-\gamma^\top\right)\gamma^\top$. In addition, since $\tau \geq \frac{\gamma-\left(1-\gamma^\perp\right)\gamma^\top}{\ln\left(\frac{1}{\gamma^\top}-1\right)}$ we have:

$$\sigma_{\gamma,\tau}(x^p) \geq \frac{1}{1+\exp\left(\frac{\gamma-(1-\gamma^\perp)\gamma^\top}{\tau}\right)} \geq \frac{1}{1+\exp\left(\ln\left(\frac{1}{\gamma^\top}-1\right)\right)} = \gamma^\top \tag{3.80}$$

Therefore $(\boldsymbol{v}^p)^\intercal \boldsymbol{w} = 1$ entails $(\boldsymbol{v}^p)^\intercal \boldsymbol{z} = 1$.

Now suppose $p \notin T_{P^M}(I)$, hence $(\boldsymbol{v}^p)^\intercal \boldsymbol{w} = 0$. Then, for every rule $r \in P^M$ one of the following occurs:

- $p \neq r$ and the contribution of this rule to x^p is 0.
- Some atom $b \in B^r$ is false in I, and therefore $\left(\boldsymbol{v}^b\right)^\intercal \boldsymbol{u} \leq \gamma^\perp$. The contribution of r to x^p is less than or equal to $\frac{n^r-1+\gamma^\perp}{n^r}$.
- Some atom $c \in C^r$ is true in I, and therefore $(\boldsymbol{v}^c)^\intercal \boldsymbol{u} \geq \gamma^\top$. The contribution of r to x^p is less than or equal to $\frac{m^r-\gamma^\top}{m^r}$.

we therefore have:

$$x^p \leq \sum_{r \in P^M, p=h^r} \max\left\{\frac{n^r - 1 + \gamma^\perp}{n^r}, \frac{m^r - \gamma^\top}{m^r}\right\} \leq x^\perp \tag{3.81}$$

and since $\tau \leq \frac{\gamma - x^\perp}{\ln\left(\frac{1}{\gamma^\perp} - 1\right)}$ we have:

$$\sigma_{\gamma,\tau}\left(x^p\right) \leq \frac{1}{1 + \exp\left(\frac{\gamma - x^\perp}{\tau}\right)} \leq \frac{1}{1 + \exp\left(\ln\left(\frac{1}{\gamma^\perp} - 1\right)\right)} = \gamma^\perp \tag{3.82}$$

Therefore $(\boldsymbol{v}^p)^\intercal \boldsymbol{w} = 0$ entails $(\boldsymbol{v}^p)^\intercal \boldsymbol{z} = 0$.

Now, suppose $(\boldsymbol{v}^p)^\intercal \boldsymbol{z} \geq \gamma^\top$. Since $p \notin T_{P^M}(I)$ entails $(\boldsymbol{v}^p)^\intercal \boldsymbol{z} \leq \gamma^\perp$, we must have $p \in T_{P^M}(I)$. Therefore, $(\boldsymbol{v}^p)^\intercal \boldsymbol{w} \geq \gamma^\top$. A similar argument shows that $(\boldsymbol{v}^p)^\intercal \boldsymbol{z} \leq \gamma^\perp$ entails $(\boldsymbol{v}^p)^\intercal \boldsymbol{w} \leq \gamma^\perp$. We have therefore shown that $\boldsymbol{w} \sim_{\gamma^\perp}^{\gamma^\top} \boldsymbol{z}$. □

3.3.3 A Solution for the Hyperparameters

Given a normal program P satisfying the MD condition, how can we select the values of $\gamma, \gamma^\perp, \gamma^\top, \tau$ while ensuring all the constraints are met? Consider again the conditions:

1. $\gamma^\top > \frac{1}{2}$
2. $\gamma^\perp < \frac{1}{2}$
3. $\gamma < (1 - \gamma^\perp)\gamma^\top$
4. $\gamma > x^\perp$
5. $\tau \leq \min\left\{\frac{\gamma - (1-\gamma^\perp)\gamma^\top}{\ln(\frac{1}{\gamma^\top} - 1)}, \frac{\gamma - x^\perp}{\ln(\frac{1}{\gamma^\perp} - 1)}\right\}$

As τ is only constrained by condition 5, we can select its value after selecting $\gamma, \gamma^\perp, \gamma^\top$. Conditions 1-3 are relatively straightforward, but condition 4 hides quite a lot of complexity. Recall:

$$x^\perp = \max_{p \in B_P} x^{p,\perp} = \max_{p \in B_P} \sum_{r \in \text{rules}(p)} \max\left\{\frac{n^r - 1 + \gamma^\perp}{n^r}, \frac{m^r - \gamma^\top}{m^r}\right\} \tag{3.83}$$

Let:

$$p^* = \arg\max_{p \in B_P} x^{p,\perp} \tag{3.84}$$

So that we have:

$$x^\perp = \sum_{r \in \text{rules}(p^*)} \max\left\{\frac{n^r - 1 + \gamma^\perp}{n^r}, \frac{m^r - \gamma^\top}{m^r}\right\} \tag{3.85}$$

Denote:

$$n = \max_{r \in \text{rules}(p^*)} n^r \qquad m = \max_{r \in \text{rules}(p^*)} m^r \qquad R = |\text{rules}\left(p^*\right)| \tag{3.86}$$

Since P satisfies the MD condition, p^* is the head of at most one long rule. We therefore have $n^r = m^r = 1$ for all $r \in \text{rules}\left(p^*\right)$ except perhaps one. We can therefore write:

$$x^\perp = \max\left\{\frac{n - 1 + \gamma^\perp}{n}, \frac{m - \gamma^\top}{m}\right\} + (R - 1)\max\left\{\gamma^\perp, 1 - \gamma^\top\right\} \tag{3.87}$$

The rest of our analysis will be divided into two cases: $n > m$ and $n \leq m$.

Suppose $n > m$. In this case, we will choose $\gamma^\perp, \gamma^\top$ such that $\gamma^\perp > 1 - \gamma^\top$. We therefore have:

$$\frac{n-1+\gamma^\perp}{n} > \frac{n-1+1-\gamma^\top}{n} = \frac{n-\gamma^\top}{n} > \frac{m-\gamma^\top}{m} \tag{3.88}$$

this last inequality being true since $f(x) = \frac{x-a}{x}$ is monotonically increasing when $a > 0$. We therefore have:

$$\begin{aligned} x^\perp = \frac{n-1+\gamma^\perp}{n} + (R-1)\gamma^\perp &< \gamma \\ \Rightarrow \qquad \frac{n-1}{n} + \left(R - 1 + \frac{1}{n}\right)\gamma^\perp &< \gamma \\ \Rightarrow \qquad \gamma^\perp &< \frac{\gamma - \frac{n-1}{n}}{R-1+\frac{1}{n}} \end{aligned} \tag{3.89}$$

in particular we see that we require $\gamma > \frac{n-1}{n}$ to ensure $\gamma^\perp > 0$.

Now suppose $m \geq n$. In this case, we will choose $\gamma^\perp < 1 - \gamma^\top$ so that we have:

$$\frac{m-\gamma^\top}{m} > \frac{m-1+\gamma^\perp}{m} > \frac{n-1+\gamma^\perp}{n} \tag{3.90}$$

and:

$$\begin{aligned} x^\perp = \frac{m-\gamma^\top}{m} + (R-1)\left(1-\gamma^\top\right) &< \gamma \\ \Rightarrow \qquad R + \left(1 - R - \frac{1}{m}\right)\gamma^\top &< \gamma \\ \Rightarrow \qquad \gamma^\top &> \frac{R-\gamma}{R-1+\frac{1}{m}} \end{aligned} \tag{3.91}$$

and to ensure $\gamma^\top < 1$ we also demand:

$$\frac{R-\gamma}{R-1+\frac{1}{m}} < 1 \qquad \Rightarrow \qquad \gamma > \frac{m-1}{m} \tag{3.92}$$

Putting all these observations together, we can choose $\gamma, \gamma^\perp, \gamma^\perp, \tau$ using the following process:

1. Find atom $p \in B_P$ for which either n or m is greatest.
2. If $n > m$, select:
   (a) $\frac{n-1}{n} < \gamma < 1$
   (b) $\gamma^\perp < \frac{\gamma - \frac{n-1}{n}}{R-1+\frac{1}{n}}$
   (c) $\gamma^\top > \max\left\{1-\gamma^\perp, \frac{\gamma}{1-\gamma^\perp}\right\}$
3. If $m \geq n$, select:
   (a) $\frac{m-1}{m} < \gamma < 1$
   (b) $\gamma^\top > \frac{R-\gamma}{R-1+\frac{1}{m}}$
   (c) $\gamma^\perp < \min\left\{1-\gamma^\top, \frac{\gamma^\top - \gamma}{\gamma^\top}\right\}$
4. Select $\tau \leq \min\left\{\frac{\gamma-(1-\gamma^\perp)\gamma^\top}{\ln(\frac{1}{\gamma^\top}-1)}, \frac{\gamma-x^\perp}{\ln(\frac{1}{\gamma^\perp}-1)}\right\}$

3.4 Gradient-Based Computation of Supported Models

We are now ready to present the main theorems on the characterisation of supported models in continuous space. These theorems characterise the supported models of a normal program as solutions of an equation involving only differentiable functions.

Theorem 3.4.1. *Let P be a normal program satisfying the MD and M an interpretation of P. Let $N = |B_P|$ and let $0 < \gamma^\perp < \gamma < \gamma^\top < 1$ and $\tau > 0$ be selected as above. Suppose $\boldsymbol{v} \sim_{\gamma^\perp}^{\gamma^\top} \boldsymbol{v}^M$ and the following equality holds:*

$$\boldsymbol{v} = \sigma_{\gamma,\tau}\left(D^{P,\boldsymbol{v}}\boldsymbol{v}\right) \tag{3.93}$$

Then M is a supported model of P.

Proof. From Theorem 3.2.2 we know that $\boldsymbol{v}^{T_{P^M}(M)} = H_1\left(D^{P,M}\boldsymbol{v}^M\right)$. Therefore, from Theorem 3.3.1 we also know that $\boldsymbol{v} \sim_{\gamma^\perp}^{\gamma^\top} H_1\left(D^{P,\boldsymbol{v}^M}\boldsymbol{v}^M\right) = \boldsymbol{v}^{T_{P^M}(M)}$ meaning $\boldsymbol{v}^M \sim_{\gamma^\perp}^{\gamma^\top} \boldsymbol{v} \sim_{\gamma^\perp}^{\gamma^\top} \boldsymbol{v}^{T_{P^M}(M)}$. Hence $M = T_{P^M}(M)$ and M is a supported model. □

Theorem 3.4.2. *Let P be a normal program satisfying the MD condition and M a supported model of P. Let $N = |B_P|$ and let $0 < \gamma^\perp < \gamma < \gamma^\top < 1$ and $\tau > 0$ be selected as above. Then there exists $\boldsymbol{v} \in [0,1]^N$ such that $\boldsymbol{v} \sim_{\gamma^\perp}^{\gamma^\top} \boldsymbol{v}^M$ and:*

$$\boldsymbol{v} = \sigma_{\gamma,\tau}\left(D^{P,\boldsymbol{v}}\boldsymbol{v}\right) \tag{3.94}$$

Proof. Define:

$$S = \left\{\boldsymbol{v} \in [0,1]^N \mid \boldsymbol{v} \sim_{\gamma^\perp}^{\gamma^\top} \boldsymbol{v}^M\right\} \tag{3.95}$$

S is a closed hyperrectangle and therefore a compact convex set. Let:

$$\begin{aligned} &f : S \to [0,1]^N \\ &f(\boldsymbol{v}) = \sigma_{\gamma,\tau}\left(D^{P,\boldsymbol{v}}\boldsymbol{v}\right) \end{aligned} \tag{3.96}$$

f is a continuous function with domain S. Since M is a supported model, from Theorem 3.3.1 every $\boldsymbol{v} \in S$ is mapped to a point in S. f is therefore a continuous function mapping a compact convex set unto itself. From Brouwer's fixed-point theorem, there exists some $\boldsymbol{v} \in S$ such that $f(\boldsymbol{v}) = \boldsymbol{v}$, as required. □

Theorems 3.4.1 and 3.4.2 lend themselves naturally to a gradient-based algorithm for computing supported models of a normal program. The theorems essentially say that computing supported models is equivalent to solving the following equation:

$$\boldsymbol{v} = \sigma_{\gamma,\tau}\left(D^{P,\boldsymbol{v}}\boldsymbol{v}\right) \tag{3.97}$$

for vectors $\boldsymbol{v}$ that are semantically equivalent to an interpretation. Define the following mapping:

$$F(\boldsymbol{v}) = \sigma_{\gamma,\tau}\left(D^{P,\boldsymbol{v}}\boldsymbol{v}\right) - \boldsymbol{v} \tag{3.98}$$

The supported models of P are the roots of F that are semantically equivalent to an interpretation. Therefore, we can compute supported models of P by searching for roots of F. To do so, we employ Newton's method. This is formalised in Algorithm 3.

Newton's method for finding roots of a mapping such as $F(\boldsymbol{v})$ proceeds by first picking an initial vector $\boldsymbol{v}^0$. At each iteration k the Jacobian of F at point $\boldsymbol{v}^k$, denoted $J_{\boldsymbol{v}^k}$, is computed

Algorithm 3 Newton Deduction

Input: $\epsilon > 0$, $MaxK \geq 1$, $0 < \gamma < 1$, $\tau > 0$

Pick an initial vector:

$$\boldsymbol{v}^0 = \begin{bmatrix} 0 \\ 1 \\ v_3 \\ \vdots \\ v_N \end{bmatrix}$$

repeat

Solve $J_{\boldsymbol{v}^k}(\boldsymbol{v}^{k+1} - \boldsymbol{v}^k) = \boldsymbol{v}^k - \sigma_{\gamma,\tau}(D^{P^{\boldsymbol{v}^k}} \cdot \boldsymbol{v}^k)$

until $||\boldsymbol{v}^{k+1} - \boldsymbol{v}^k||_2 < \epsilon$ **or** $k \geq MaxK$

if $k < MaxK$ **then**

return $\boldsymbol{v}^k$

else

return 'FAIL'

and used to find the next vector $\boldsymbol{v}^{k+1}$. The update rule is given by:

$$J_{\boldsymbol{v}^k}(\boldsymbol{v}^{k+1} - \boldsymbol{v}^k) = -F\left(\boldsymbol{v}^k\right) = \boldsymbol{v}^k - \sigma_{\gamma,\tau}\left(D^{P,\boldsymbol{v}^k}\boldsymbol{v}^k\right) \tag{3.99}$$

The Jacobian is an $N \times N$ matrix with elements:

$$\begin{aligned} J_{i,j} = \frac{\partial F\left(\boldsymbol{v}\right)_i}{\partial v_j} = \frac{\partial \sigma_{\gamma,\tau}\left(D^{P,\boldsymbol{v}}\boldsymbol{v}\right)_i}{\partial v_j} - \frac{\partial \boldsymbol{v}_i}{\partial v_j} = \\ = \frac{1}{\tau}\sigma_{\gamma,\tau}\left(D^{P,\boldsymbol{v}}\boldsymbol{v}\right)_i\left(1 - \sigma_{\gamma,\tau}\left(D^{P,\boldsymbol{v}}\boldsymbol{v}\right)_i\right)\frac{\partial\left(D^{P,\boldsymbol{v}}\boldsymbol{v}\right)_i}{\partial v_j} - \delta_{ij} \end{aligned} \tag{3.100}$$

where:

$$\delta_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases} \tag{3.101}$$

We have:

$$\begin{aligned} \frac{\partial\left(D^{P,\boldsymbol{v}}\boldsymbol{v}\right)_i}{\partial v_j} = \frac{\partial \sum_{k=1}^{N} D_{ik}^{P,\boldsymbol{v}} v_k}{\partial v_j} = \sum_{k=1}^{N}\frac{\partial\left(D_{ik}^{P,\boldsymbol{v}} v_k\right)}{\partial v_j} = \sum_{k=1}^{N}\left[\frac{\partial D_{ik}^{P,\boldsymbol{v}}}{\partial v_j}v_k + D_{ik}^{P,\boldsymbol{v}}\frac{\partial v_k}{\partial v_j}\right] = \\ = \sum_{k=1}^{N}\left[\frac{\partial D_{ik}^{P,\boldsymbol{v}}}{\partial v_j}v_k + D_{ik}^{P,\boldsymbol{v}}\delta_{jk}\right] = \sum_{k=1}^{N}\left[\frac{\partial D_{ik}^{P,\boldsymbol{v}}}{\partial v_j}v_k\right] + D_{ij}^{P,\boldsymbol{v}} \end{aligned} \tag{3.102}$$

$$\begin{aligned} \frac{\partial D_{ik}^{P,\boldsymbol{v}}}{\partial v_j} = \frac{\partial \sum_{r\in P}\left[\frac{(\mathbf{1}-\boldsymbol{v})^\intercal \boldsymbol{v}^{C^r}\boldsymbol{v}^{h^r}\left(\boldsymbol{v}^{B^r}\right)^\intercal}{n^r m^r}\right]_{ik}}{\partial v_j} = \sum_{r\in P}\frac{\boldsymbol{v}_i^r \boldsymbol{v}_k^{B^r}}{n^r m^r}\cdot\frac{\partial\left[(\mathbf{1}-\boldsymbol{v})^\intercal \boldsymbol{v}^{C^r}\right]}{\partial v_j} = \\ = \sum_{r\in P}\frac{\boldsymbol{v}_i^r \boldsymbol{v}_k^{B^r}}{n^r m^r}\cdot\frac{\partial \sum_{l=1}^{N}(1-v_l)\boldsymbol{v}_l^{C^r}}{\partial v_j} = -\sum_{r\in P}\frac{\boldsymbol{v}_i^r \boldsymbol{v}_k^{B^r}}{n^r m^r}\cdot\sum_{l=1}^{N}\boldsymbol{v}_l^{C^r}\delta_{lj} = -\sum_{r\in P}\frac{\boldsymbol{v}_i^r \boldsymbol{v}_k^{B^r}\boldsymbol{v}_j^{C^r}}{n^r m^r} \end{aligned} \tag{3.103}$$

Plugging this result back in Equation 3.102 we get:

$$\frac{\partial \left(D^{P,\boldsymbol{v}}\boldsymbol{v}\right)_i}{\partial v_j} = -\sum_{k=1}^{N}\sum_{r\in P}\left[\frac{\boldsymbol{v}_i^r \boldsymbol{v}_j^{C^r} \boldsymbol{v}_k^{B^r} v_k}{n^r m^r}\right] + D_{ij}^{P,\boldsymbol{v}} =$$

$$= -\sum_{r\in P}\left[\frac{\boldsymbol{v}_i^r \boldsymbol{v}_j^{C^r} \sum_{k=1}^{N} \boldsymbol{v}_k^{B^r} v_k}{n^r m^r}\right] + D_{ij}^{P,\boldsymbol{v}} = \tag{3.104}$$

$$-\sum_{r\in P}\left[\frac{\boldsymbol{v}_i^r \boldsymbol{v}_j^{C^r} \left(\boldsymbol{v}^{B^r}\right)^\intercal \boldsymbol{v}}{n^r m^r}\right] + D_{ij}^{P,\boldsymbol{v}} \tag{3.105}$$

Hence an element of the Jacobian is:

$$J_{i,j} = \frac{1}{\tau}\sigma_{\gamma,\tau}\left(D^{P,\boldsymbol{v}}\boldsymbol{v}\right)_i \left(1 - \sigma_{\gamma,\tau}\left(D^{P,\boldsymbol{v}}\boldsymbol{v}\right)_i\right)\left(D_{ij}^{P,\boldsymbol{v}} - \sum_{r\in P}\frac{\boldsymbol{v}_i^r \boldsymbol{v}_j^{C^r} \left(\boldsymbol{v}^{B^r}\right)^\intercal \boldsymbol{v}}{n^r m^r}\right) - \delta_{ij}$$

which in matrix form is given by:

$$J_{\boldsymbol{v}} = \frac{1}{\tau}\mathrm{diag}\big(\sigma_{\gamma,\tau}(D^{P,\boldsymbol{v}}\boldsymbol{v}) \times (\mathbf{1} - \sigma_{\gamma,\tau}(D^{P,\boldsymbol{v}}\boldsymbol{v}))\big) \cdot \left(D^{P,\boldsymbol{v}} - \sum_{r\in P}\frac{\boldsymbol{v}^{h^r}(\boldsymbol{v}^{B^r})^\intercal \boldsymbol{v}(\boldsymbol{v}^{C^r})^\intercal}{n^r m^r}\right) - \mathbb{I} \tag{3.106}$$

where $\times$ indicates element-wise multiplication and $\mathrm{diag}(\boldsymbol{v})$ is a matrix with the entries of $\boldsymbol{v}$ along its main diagonal. For different initial vectors $\boldsymbol{v}^0$, the algorithm will return different results. We always set $v_1 = 0$ and $v_2 = 1$ representing that $\bot$ is false and $\top$ is true, respectively. For the initial values of $v_3, ..., v_N$, we consider two methods:

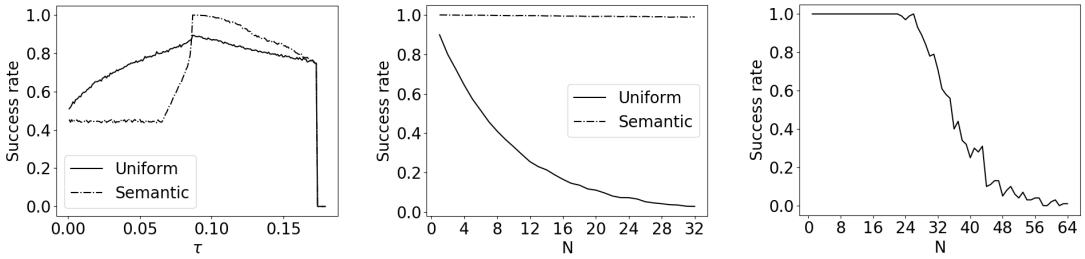
- **Uniform sampling** - The values are picked uniformly from $[0, 1]$.
- **Semantic sampling** - The values are picked uniformly from $[0, \gamma^\bot] \cup [\gamma^\top, 1]$. In other words, the initial vector is always semantically equivalent to an interpretation.

Newton's method does not guarantee convergence, which is why we limit the number of iterations to $MaxK$ before declaring non-convergence. When convergence occurs, the discovered root $\boldsymbol{v}^k$ is returned. However, this root is not necessarily semantically equivalent to an interpretation. If it is semantically equivalent to $\boldsymbol{v}^M$ for some interpretation M, it is guaranteed by Theorem 3.4.1 that M is a supported model. When a root semantically equivalent to an interpretation is returned, we consider it a successful case for the algorithm. In addition, we can easily check if the discovered supported model is stable by computing the Least Herbrand Model of the program reduct.

3.5 Evaluation

To test the proposed method, a simple implementation was made. Efficiency was not a goal of the implementation, and therefore we do not test its time performance. Instead, we focus on the rate of success for computing supported and stable models under various conditions. We define the rate of success for a given program to be the number of applications of Algorithm 3 for the program which converged to a root semantically equivalent to a supported model, divided by the total number of attempts.

The test programs were generated so as to study the algorithm's success rate for relatively difficult problems. Randomly generated programs tend to be simple to solve. The reason for this is that their structure tends not to contain loops. As mentioned in the Background chapter,



(a) One negative loop. (b) N negative loops. (c) N negative loops within 1 sec.

Figure 3.2: Success rate of converging to a supported model with negative loops. (a) A single negative loop for varying temperatures. (b) N negative loops for $\tau = 0.087$. (c) N negative loops for $\tau = 0.087$ and when given as many attempts as possible within 1 second.

positive loops tend to create supported but non-stable models. Our algorithm is still capable of finding these models, although it is generally preferable to find stable models. It is therefore worth considering how often we would converge to a non-stable supported model when positive loops are presented. Negative loops can be used to model choice. Hence, checking the success rate in the presence of choice is of great interest. Since randomly generated programs tend to have no negative loops (sometimes referred to as "stratified programs"), they are not of much interest. In fact, evaluation of our algorithm on stratified programs revealed a 100% success rate. Instead, we intentionally generated programs containing loops. Recall from Section 2.1.4 that any hard constraint or choice rule can be mapped to a sequence of normal rules containing positive and negative loops. Therefore, while the programs we evaluate on are constructed and seemingly engineered, they in fact represent the structure of real-world programs, while ensuring the solving difficulty that stratified programs do not have.

We begin by considering the simple negative loop:

$$\begin{aligned} p &\leftarrow \text{not } q \\ q &\leftarrow \text{not } p \end{aligned} \tag{3.107}$$

The program has four Herbrand interpretations: $\emptyset$, $\{p\}$, $\{q\}$, $\{p, q\}$. Two of these are both supported and stable models, namely $\{p\}$ and $\{q\}$. Simply guessing an interpretation and checking if it is stable has a 50% success rate. To test the rate of success of finding a supported or stable model in this case, we repeatedly applied Algorithm 3. We tested for 200 different values of τ from just above 0 to 0.179 (which is slightly above the maximum allowed temperature value) in equal intervals. For each τ, we applied the algorithm 10,000 times, with randomly selected values of $\boldsymbol{v}^0$, and calculated the portion for which it converged to either $\{p\}$ or $\{q\}$. We repeated this experiment separately for both uniform and semantic sampling.

From Figure 3.2a, one can see that the rate of success approaches 50% at the limit of τ reaching zero, for both methods of sampling. This behaviour can be understood in the following way: When τ approaches zero, the Sigmoid begins behaving like a Heaviside function such as H_1. The computation is therefore changing from a continuous nature to a discrete one. In other words, in the limit where τ goes to zero, the Sigmoid implements T_P, and the algorithm behaves like a purely symbolic guess and check. Hence, the success rate approaches 50%.

As τ increases, the gradient information helps to direct the search and the rate of success increases, peaking at 89% for uniform sampling and near 100% for semantic sampling around $\tau = 0.087$. It is interesting that the increase is delayed for the semantic sampling case, not

showing any improvement for $\tau < 0.07$. This is due to the approximation of T_P being more pronounced for vectors semantically equivalent to interpretations, and providing weaker gradient information at low temperatures.

For $\tau = 0.087$, it is interesting to ask what went wrong in the 11% of the cases where a stable model was not found, and why semantic sampling does not seem to have this problem. As it turns out, for cases where uniform sampling failed, the algorithm always converged to the same root, $[0, 1, 0.5, 0.5]$. This root is not semantically equivalent to any interpretation, hence it is still considered a failure case for the algorithm. We do not consider it in this work, but it is possible that this root still holds some semantic meaning of the program. For instance, it may have a link to the Well-Founded Model of the program (Van Gelder et al., 1991). For now, however, we simply consider it as an undesired output of the algorithm. The set of initial vectors that result in converging to $[0, 1, 0.5, 0.5]$ is mostly clustered around the root. Semantic sampling therefore avoids most of these points, improving the chance of success.

The maximum value of τ for which Theorem 3.3.1 holds is around 0.173. At that point we see a sudden drop in the success rate to 0. In other words, the conditions imposed on τ not only guarantee the correctness of Theorem 3.3.1 but are also important to converge to a meaningful fixed point. The main takeaway from this experiment is that both the temperature and the method of sampling have a significant impact on the performance of Algorithm 3.

Next, we consider how the algorithm fares as negative loops are added to the program. Consider a program with N pairs of clauses of the form:

$$\begin{aligned} p_i &\leftarrow \text{not } q_i \\ q_i &\leftarrow \text{not } p_i \end{aligned} \tag{3.108}$$

For each pair, we have an 89% chance of success with uniform sampling and an optimal choice of τ. Since each pair is independent of the rest, one can think of their embedded semantics as existing in independent two-dimensional subspaces. Consequently, the chance of finding a stable model is equivalent to tossing N coins, each with a 89% chance of landing on heads, and observing N heads. We confirm this intuition experimentally, as shown in Figure 3.2b. In contrast, for semantic sampling, the high probability of success implies a very slow drop as N increases. This is also confirmed in Figure 3.2b.

So far, we have only considered the chance of success when applying Algorithm 3 once. But we can also repeatedly apply the algorithm on the same program, for different initial vectors, until a supported/stable model is found. Consider, for instance, a scenario in which we are given 1 second to find a supported/stable model, and within that second we are allowed to apply Algorithm 3 as often as we can. We performed this experiment for uniform sampling on a computer with the following specifications:

- **CPU:** Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz.
- **Memory:** 15.4 GB RAM.
- **Implementation Language:** Python 3.6.9.

Note that the implementation is single-threaded. As can be seen from Figure 3.2c, even with an inefficient implementation we can achieve a perfect success rate for up to 32 pairs. Past 32, we see a fast drop in performance. This can be understood when considering the chance of success as N increases. For N pairs and K applications of the algorithm, the chance of success is given by $1-(1-0.89^N)^K$. For a large enough K, one can retrieve the graph in Figure 3.2c. Therefore,

N	M	Success rate (supported)	Success rate (stable)
10	5	70%	69%
	10	84%	66%
	20	99%	7%
40	20	46%	44%
	40	68%	39%
	60	88%	13%
Overall		76%	40%

Table 3.1: Success rate with uniform sampling for programs with positive and negative loops.

as N increases one must also increase K by allowing further time. Although not shown, we expect that using semantic sampling would improve performance in this experiment too.

We now turn our attention to the case where programs contain both positive and negative loops. As before, we first add N negative loop pairs of the form in equation (3.108) to the program. We then add M pairs of definite clauses using the following process: We select two indices $1 \leq i, j \leq N$ such that $i \neq j$ and add to the program a pair of clauses chosen randomly from one of the following forms:

$$\begin{array}{cccc} p_i \leftarrow p_j & p_i \leftarrow q_j & q_i \leftarrow p_j & q_i \leftarrow q_j \\ q_j \leftarrow q_i & p_j \leftarrow q_i & q_j \leftarrow p_i & p_j \leftarrow p_i \end{array} \tag{3.109}$$

We repeat this selection process M times. Note that each time we may select a different form for the pairs. As M becomes larger, positive loops will begin to appear more often in the program. This process will therefore generate programs containing many positive loops without enforcing any kind of structure over these loops.

We generated 100 different programs for several values of N and M. For each program we applied Algorithm 3 100 times. We report the average success rate for each case in Table 3.1. Results for semantic sampling were very similar and are not reported here.

From the table we see the clear effect of positive loops on the success rate. Positive loops help the system discover supported models. However, they have the opposite effect on stable models. As the number of positive loops increases, the number of supported models that are not stable also increases, hence stable models become more difficult to find.

Finally, we consider the case where a choice between multiple options is required. Suppose we have N options, represented by the N variables $p_1, p_2, ..., p_N$, and we are required to select only one. This can be modelled by the following set of N clauses:

$$\begin{array}{c} p_1 \leftarrow \text{not}\, p_2, \text{not}\, p_3, ..., \text{not}\, p_N \\ p_2 \leftarrow \text{not}\, p_1, \text{not}\, p_3, ..., \text{not}\, p_N \\ \vdots \\ p_N \leftarrow \text{not}\, p_1, \text{not}\, p_2, ..., \text{not}\, p_{N-1} \end{array} \tag{3.110}$$

Then $\{p_1\}, \{p_2\}, ..., \{p_N\}$ are the supported (and stable) models of the program. When $N = 2$ we retrieve the case of a single pair. Results for the case $N = 2$ to 9 appear in Figure 3.3 without the label "short". As can be seen, the success rate falls quickly as the number of choices increases. This is a result from the conditions for τ imposed by equations (3.63) and (3.70). As

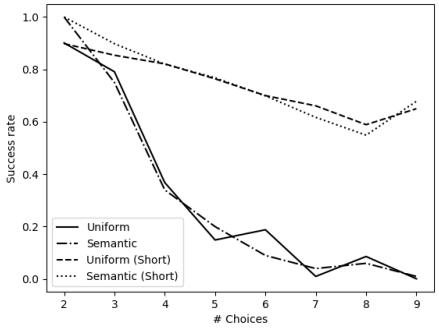


Figure 3.3: Success rate for N choices under various settings. Short stands for the representation of choice as a set of short clauses.

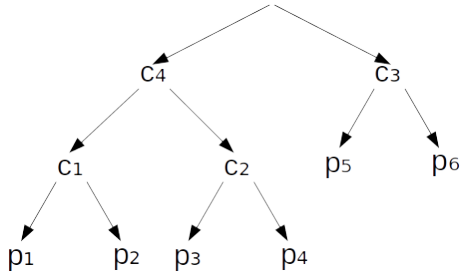


Figure 3.4: The binary tree corresponding to the encoding of $N = 6$ choices using short clauses only.

the length of rules increases, τ must be made smaller to guarantee exact inference holds. For a very small τ, the gradient information is insufficient, hurting the chance of success. Clearly, we must prefer representing our programs using shorter rules.

To accommodate this, we transform the problem of N choices into a sequence of choices between two atoms. We divide the variables $p_1, p_2, ..., p_N$ into pairs $(p_1, p_2), (p_3, p_4), ...$ and for each i pair we create a new atom c_i representing a choice between the two. We can encode these choices using each pair and its corresponding new atom as follows:

$$\begin{aligned} p_1 &\leftarrow c_1, \text{not}\, p_2 \\ p_2 &\leftarrow c_1, \text{not}\, p_1 \\ p_3 &\leftarrow c_2, \text{not}\, p_4 \\ p_4 &\leftarrow c_2, \text{not}\, p_3 \\ &\vdots \end{aligned} \tag{3.111}$$

Note that the number of new atoms is $\lfloor \frac{N}{2} \rfloor$. We then take the new atoms $c_1, c_2, ...$ and possibly p_N if N is odd, and encode a choice among them using the same process. By repeating this process, we are eventually left with only two atoms, for which we can encode a choice using a regular negative loop as in equation (3.107). This encoding is best visualised using a binary tree structure. For example, for $N = 6$, the encoding above results in the binary tree depicted in

figure 3.4, which corresponds to the program:

$$\begin{array}{cc} p_1 \leftarrow c_1, \text{not } p_2 & p_2 \leftarrow c_1, \text{not } p_1 \\ p_3 \leftarrow c_2, \text{not } p_4 & p_4 \leftarrow c_2, \text{not } p_3 \\ p_5 \leftarrow c_3, \text{not } p_6 & p_6 \leftarrow c_3, \text{not } p_5 \\ c_1 \leftarrow c_4, \text{not } c_2 & c_2 \leftarrow c_4, \text{not } c_1 \\ c_3 \leftarrow \text{not } c_4 & c_4 \leftarrow \text{not } c_3 \end{array} \tag{3.112}$$

In this way, we successfully encoded a choice among N variables using only short rules, and by adding only $O(N)$ new variables and clauses. It is easy to verify that the stable models of the new representation correspond to stable models of the original, once the auxiliary atoms are removed. The drawback to this representation is that it does not guarantee an equal probability of finding each of the stable models. Figure 3.3 shows the results with this representation, under the label "Short". As can be seen, with a representation that uses short rules, the success rate scales far better. With both representations, there does not seem to be a significant difference between uniform and semantic sampling.

3.6 Summary

In this chapter, we laid the foundation for a novel approach to neuro-symbolic reasoning by exploring how logical operations can be realised in continuous vector space. We introduced a method for embedding normal propositional programs in matrices using the notion of a program reduct. We showed how this program matrix can be used to realise the immediate consequence operation in a completely differentiable manner. As a first contribution, we developed a completely novel method for computing stable and supported models of a program using a gradient-based optimisation process and demonstrated the feasibility of the approach on programs containing both positive and negative loops. Experimental results demonstrated the importance of selecting optimal hyperparameters (namely, γ and τ) and that restructuring the program for shorter rules greatly improves the rate of success in computing stable models. From a scalability point of view, we saw that random restarts help the scalability of the approach, suggesting massive parallelisation can be helpful to find stable models more quickly. Given that the computations are done in a vector space, they can be implemented on GPU hardware, making it easy to parallelise. An interesting research direction would be the combination of traditional CDNL approaches alongside gradient-based search for greater parallelisation. In the next chapter, we build on the matrix-based representation developed here to allow the integration of logic programs with neural networks. We do so by incorporating the matrix in a new type of layer: "Logic Program Layer".

Chapter 4

Logic Program Layer

Deductive reasoning refers to the ability of an agent to arrive at correct conclusions from a given set of premises and rules. In a neural-symbolic setting, these rules provide additional information to help link multiple observable concepts together. In symbolic inference, observations are usually encoded as facts or hard constraints. In neuro-symbolic inference, observations can also be given by outputs of a neural network, and have fuzzy truth values. Since these observations often come from unstructured data, such as images or audio, we can think of the neural network as performing a perception task. For instance, suppose a model is trained to categorise locations that appear in an image. If it is shown an image of a bedroom, the following rule may help in its task:

$$\text{location(bedroom)} \leftarrow \text{location(indoors)}, \text{object(bed)} \tag{4.1}$$

since the task of detecting a bedroom is now reduced to the task of distinguishing between indoor/outdoor settings and identifying certain objects such as a bed. Suppose we have a perception machine learning model M (or set of models) that can correctly infer that location(indoors) and object(bed) are true. Then we can apply the T_P operator (perhaps multiple times) to infer location(bedroom). Thus, we have broken down the task of detecting bedrooms into a chain of two simpler tasks: a perception task (detecting objects in an image) and a reasoning task (which objects likely indicate a bedroom). As breaking down a reasoning task into simpler steps is known to improve reasoning performance (Wei et al., 2022), this can be highly beneficial.

However, during training the problem is reversed. We receive a label indicating that the atom location(bedroom) is true, without being told anything about the contents of the image, such as whether location(indoors) and object(bed) are true. Given that, can we use the above rule to help train M to begin with? That is the core question of neural-symbolic reasoning we aim to address in this work.

In this chapter, our approach for neural-symbolic reasoning is to use our differentiable form of T_P to train such a model. We can encapsulate this operation in a new layer we refer to as "Logic Program Layer". This layer does not have trainable parameters. Instead, it uses a matrix representation to encode the rules of a logic program. Since this representation allows the application of T_P in a differentiable manner, backpropagation through the layer is possible. Therefore, we can train models with labels such as location(bedroom) if we wish.

We refer to the intermediate observations produced by the perception model as "latent concepts" as these are not typically labelled, and our task is not to predict those. In our example, "bed" and "indoors" are latent concepts. Our Logic Program Layer performs the reasoning

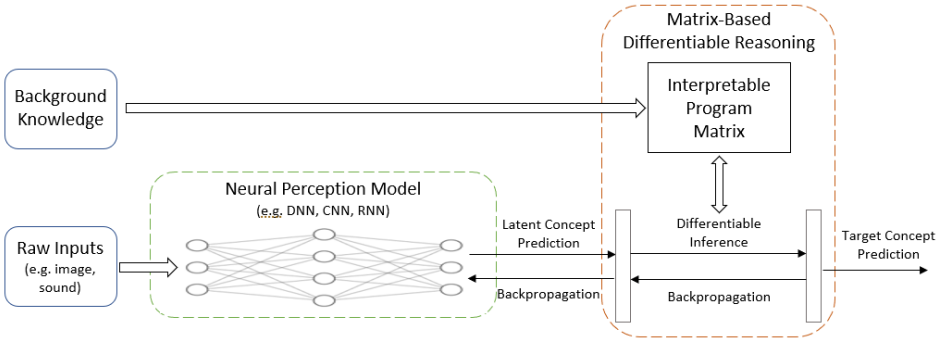


Figure 4.1: General neural-symbolic architecture.

task, taking the latent concepts to our "target concepts" or "downstream concepts" (in this case "bedroom"). As we train end-to-end from target concept labels, our perception model should become better at predicting latent concepts, improving overall downstream accuracy.

While neural-symbolic reasoning allows a model to incorporate certain background information and improve the learning process, it does depend on a given *background knowledge* relevant to the task. The background knowledge is often given as a set of logical rules (a logic program) specified by a human. However, in many tasks, it is either difficult or impossible to provide these rules beforehand. They may not be known or are too complicated to be reasonably enumerated. In such settings, it would be beneficial to learn the rules from the data together with other model parameters. That is in essence the task of *neural-symbolic learning*.

As discussed in the background, learning logic programs is the subject of Inductive Logic Programming (ILP). Doing so is rather challenging, as the size of the hypothesis space is exponential in the size of the language, requiring the introduction of *language bias*. Using our matrix-based approach and working in continuous vector spaces could potentially alleviate this issue. A search over an exponentially sized hypothesis space can be done by a gradient-based optimisation process. This would involve treating our matrix-vector representation as model parameters to be optimised with an appropriate loss function. In the latter part of this chapter, we detail our initial approach towards this goal, by tackling the problem of gradient-based matrix learning of *purely symbolic* tasks involving a single T_P transition. An existing symbolic learning framework, known as "Learning from Interpretation Transitions" Inoue et al. (2014) serves as our starting point. We show how this framework can be adapted to learning in continuous vector space, where we gain the ability to handle noisy input, and perform an initial study of our new framework.

Note that while in the previous chapter, we were using a differentiable T_P operator to perform stable and supported reasoning over normal programs, here our goal is different. We simply wish to perform several steps of T_P to arrive at new conclusions. For this purpose, we do not need to compute supported models, and so we adapt our approach to this new setting. Namely, we assume our programs are *hierarchical*. Hierarchical programs can be thought of as being composed of *levels*, similar to how neural networks are composed of layers, where the sets of atoms for different levels are disjoint. Specifically, they contain no loops. We believe a large variety of interesting neural-symbolic reasoning tasks feature hierarchical programs. Any task that can be modelled as a Directed Acyclic Graph is representable by a hierarchical program. For example, path planning in an environment (Sensors $\rightarrow$ Objects $\rightarrow$ Path) or verifying legal compliance (documents $\rightarrow$ facts $\rightarrow$ compliance), so this restriction is not severe.

This chapter is structured as follows: we begin by describing the theoretical framework for

neuro-symbolic reasoning and learning we use in the rest of this work. We then specifically describe our Logic Program Layer approach, beginning with neuro-symbolic reasoning. We first simplify our matrix representation a bit further, lifting some of the assumptions from the previous chapter, such as the MD condition, and removing the dependency on Program Reducts by introducing a matrix-vector form of normal logic programs. We then evaluate this approach on end-to-end neuro-symbolic reasoning tasks, showing some of its benefits and limitations. We then turn our attention to the task of learning the program matrix, laying the theoretical foundation for our gradient-based optimisation process and then evaluating our approach on several tasks to demonstrate its feasibility. Finally, we summarise the chapter.

4.1 Neuro-Symbolic Theoretical Framework

In neuro-symbolic settings, we are generally interested in learning functions of the form $h = g \circ f : \mathcal{X} \to \mathcal{Z} \to \mathcal{Y}$, where h is decomposed into a perception function $f : \mathcal{X} \to \mathcal{Z}$ (or a set of functions) and a reasoning function $g : \mathcal{Z} \to \mathcal{Y}$. $\mathcal{X}$ is the space of inputs, $\mathcal{Y}$ is the space of target labels, and $\mathcal{Z}$ is a space of latent concept values relevant to the task.

While the description above is a useful way of thinking about neuro-symbolic functions, it is imprecise in several ways. In the following, we provide a more accurate definition of a neuro-symbolic reasoning and learning task and the function h which we wish to learn to solve them. To help explain some of the definitions, we will use two neuro-symbolic tasks as running examples:

1. The MNIST Addition task (Manhaeve et al., 2019). We are given two images of MNIST digits (LeCun et al., 2010a) and must output their sum. Each sample in the dataset consists of two 28×28 grayscale images and the sum as a label. It is assumed the images themselves are unlabelled. To solve the task a model would have to correctly predict which digit appears in each image, and compute the sum of the digits correctly.

2. The Member task (Tsamoura et al., 2021). We are given three MNIST images and a digit in symbolic form. Our task is to determine whether the symbolic digit appears in one of the images. The images are again unlabelled. Instead, labels have one of the values "true" or "false" depending on whether the digit is a member of the list of digits in the images. To solve the task, the model should predict the digit in each image, and correctly compute if it is equal to the symbolic digit.

A task input can be either raw (unstructured) or symbolic.

Definition 4.1.1. *(Raw input) A raw input x_i is a tuple $(n_{x_i}, \mathcal{X}_i, \boldsymbol{v}_{x_i})$ where n_{x_i} is the name of the raw input, $\mathcal{X}_i$ is the raw input space and $\boldsymbol{v}_{x_i}$ is a vector in $\mathcal{X}_i$. For $i \neq j$, we may have $\mathcal{X}_i = \mathcal{X}_j$.*

Example 4.1.1. *In the MNIST addition task, we have two images as raw inputs, these being $x_1 = (\mathit{digit}(1), \mathcal{X}_1, \boldsymbol{v}_1)$ and $x_2 = (\mathit{digit}(2), \mathcal{X}_2, \boldsymbol{v}_2)$. They both belong to the same raw input space $\mathcal{X}_1 = \mathcal{X}_2 = [0,1]^{28\times 28}$, which is the set of all possible 28×28 grayscale images.*

Definition 4.1.2. *(Symbolic input) A symbolic input s_i is a tuple (n_{s_i}, S_i, a_{s_i}) where n_{s_i} is the name of the symbolic input, S_i a finite set of symbols and $a_{s_i} \in S_i$. For $i \neq j$, we may have $\mathcal{S}_i = \mathcal{S}_j$.*

Example 4.1.2. *In the Member task, besides the three raw inputs, we are also given a symbolic input $s_1 = (\mathit{target_digit}, S_1, a_1)$ where $a_1 \in S_1 = \{0,1,2,3,4,5,6,7,8,9\}$.*

If the task has n raw inputs and m symbolic inputs, the set of task inputs can be written as $x_1, x_2, \ldots x_n, s_1, s_2, \ldots, s_m$. We can write this in short-hand form using the cartesian product $\bigotimes_{i=1}^{n} x_i \otimes \bigotimes_{j=1}^{m} s_j$. To keep the notation less cluttered, we will drop the indices under and above $\otimes$.

Definition 4.1.3. *(Latent concept) A latent concept c_k is a tuple $(n_{c_k}, V_{c_k}, \mathcal{Z}_{c_k})$, where n_{c_k} is the name of the latent concept, V_{c_k} is a finite set of symbols, known as the latent concept values, $\mathcal{Z}_{c_k}$ is known as the latent concept space. Each raw input x_i is associated with a single latent concept c_k. For $i \neq j$, x_i and x_j may be both associated with the same latent concept c_k.*

Example 4.1.3. *Each MNIST image is associated with the same latent concept c_1, which is the tuple $(\mathit{digit}, \{0,1,2,3,4,5,6,7,8,9\}, \mathcal{Z}_{c_1})$. A possible latent concept space $\mathcal{Z}_k$ could be the probability simplex over the set of possible digits:*

$$\mathcal{Z}_{c_1} = \left\{ \boldsymbol{z} \in [0,1]^{10} \middle| \sum_{i=1}^{10} z_i = 1 \right\} \tag{4.2}$$

While there could be many latent concepts, we assume without loss of generality that each x_i is associated with a single latent concept. To deal with multiple latent concepts per input, we can construct a composite latent concept. For example, if x_i should be associated with both $c_1 = (n_{c_1}, V_{c_1}, \mathcal{Z}_{c_1})$ and $c_2 = (n_{c_2}, V_{c_2}, \mathcal{Z}_{c_2})$ then we construct $c_{1,2} = \left(n_{c_{1,2}}, V_{c_1} \times V_{c_2}, \mathcal{Z}_{c_{1,2}}\right)$. Exactly how the latent space $\mathcal{Z}_{c_{1,2}}$ is constructed depends on the nature of $\mathcal{Z}_{c_1}$ and $\mathcal{Z}_{c_2}$. For example, if these are probability simplexes, we can construct a new probability simplex for $V_{c_1} \times V_{c_2}$.

Definition 4.1.4. *(Perception function) A perception function maps a raw input to an element of its latent concept space. Formally, for a raw input x_i associated with latent concept $c_j = (n_j, V_j, \mathcal{Z}_j)$ the perception function is $f_j : \mathcal{X}_i \to \mathcal{Z}_j$.*

When we have a collection of raw inputs $\bigotimes x_i$, where raw input x_i is associated with latent concept c_{j_i}, we write the application of all perception functions to all raw inputs as $f\left(\bigotimes x_i\right) = \bigotimes f_{j_i}\left(x_i\right)$. f is a short-hand notation for all the perception functions according to the order of the raw inputs. We refer to f as the "perception functions".

Example 4.1.4. *A perception function for the MNIST Addition task could be a convolutional neural network that maps an MNIST image to a vector of probabilities over the digits 0 to 9.*

Definition 4.1.5. *(Symbol grounding function) For a latent concept c_k, a symbol grounding function $G_k : \mathcal{Z}_{c_k} \to V_{c_k}$ maps elements of the latent concept space to latent concept values.*

Example 4.1.5. *In the MNIST Addition task, the symbol grounding function needs to map a probability vector (the output of the convolutional neural network) to one of the digits 0 to 9. This can be done with an argmax function.*

Definition 4.1.6. *(Target concept) A target concept or downstream concept y_i is defined to be a tuple $(n_{y_i}, T_{y_i}, t_{y_i})$, where n_{y_i} is the name of the target concept, T_{y_i} is a finite set of symbols known as the target concept values and $t_{y_i} \in T_{y_i}$.*

Example 4.1.6. *For MNIST addition, the target concept is $y_1 = (\mathit{sum}, \{0, 1, \ldots, 18\}, t_{y_1})$. In the Member task the target concept is $y_1 = (\mathit{member}, \{\mathit{true}, \mathit{false}\}, t_{y_1})$*

Definition 4.1.7. *(Reasoning function) The reasoning function g maps the latent concept space for all raw inputs and the symbolic inputs to target concepts. Formally, if for a raw input x_i we have a latent concept space $\mathcal{Z}_i$, then $g : \bigotimes \mathcal{Z}_i \otimes \bigotimes S_j \to \bigotimes \mathcal{Y}_k$.*

Even though a latent concept and a symbolic input both can take a finite set of symbolic values, their function is quite distinct. A symbolic input is a *given* input to the task. It is part of the dataset. A latent concept is *not given* as part of the dataset. Instead, each raw input is associated with a latent concept whose value must be inferred from said raw input. Hence, learning to map raw inputs to latent concept values is an important aspect of solving a neuro-symbolic task, while symbolic inputs only play a role at the reasoning stage.

Example 4.1.7. *In the MNIST addition task, the reasoning task involves computing the sum of the digits that appear in the images. The images are first mapped by the perception function* f_1 *to probabilities over the set of digits. Namely,* $z_1 = f_1(x_1)$ *and* $z_2 = f_2(x_2)$ *are two probability vectors of size 10 each. These are fed into a reasoning function* g *to compute the sum* $y_1 = g(z_1, z_2) \in \{0, 1, \ldots, 18\}$.

In the Member task, the three images are fed into the perception function f_1 *to receive the probabilities* z_1, z_2, z_3. *These are fed into* g *in addition to the symbolic input* s_1 *(the target digit) to compute whether* s_1 *is a member of the list* $y_1 = g(z_1, z_2, z_3, s_1) \in \{true, false\}$.

Definition 4.1.8. *The neuro-symbolic function* h *maps raw inputs and symbolic inputs to target concepts by mapping raw inputs to latent concept values, and applying the reasoning function on the latent concept values and symbolic inputs. Formally, let* $\bigotimes x_i$ *be the raw inputs,* $\bigotimes s_k$ *the symbolic inputs and* $\bigotimes y_l$ *the targets. Then:*

$$\begin{aligned} &h : \bigotimes \mathcal{X}_i \otimes \bigotimes S_k \to \bigotimes \mathcal{Y}_l \\ &h\left(\bigotimes x_i, \bigotimes s_k\right) = g\left(f\left(\bigotimes x_i\right), \bigotimes s_k\right) \end{aligned} \tag{4.3}$$

The reasoning function itself can be represented as a logic program together with a symbol grounding function. We can encode the output of the symbol grounding functions as facts and add these to the program. By applying T_P repeatedly, we can deduce further atoms until the correct target concept values are computed.

Example 4.1.8. *In the MNIST Addition task, we can take the argmax as the symbol grounding function, translating the probabilities given by the perception function into latent concept values. This latent concept value, being the digit with the highest probability for a given image, can be encoded as a symbolic atom. For example, for an image* I *(*$I = 1, 2$*) and digit* D *between 0 and 9, we can write the atom* $digit(I, D)$. *We can then encode the rules for computing the sum as follows:*

$$\begin{aligned} &sum(0) \leftarrow digit(1, 0), digit(2, 0) \\ &sum(1) \leftarrow digit(1, 1), digit(2, 0) \\ &sum(1) \leftarrow digit(1, 0), digit(2, 1) \\ &sum(2) \leftarrow digit(1, 2), digit(2, 0) \\ &sum(2) \leftarrow digit(1, 1), digit(2, 1) \\ &\quad\vdots \end{aligned} \tag{4.4}$$

For two images, suppose we have the following latent concept predictions:

$$\begin{aligned} \boldsymbol{z}_1 &= [0.05, 0.05, 0.05, 0.20, 0.30, 0.05, 0.05, 0.05, 0.15, 0.05] \\ \boldsymbol{z}_2 &= [0.26, 0.30, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.19] \end{aligned} \tag{4.5}$$

these are mapped to the two following facts, which are added to the program:

$$\begin{aligned} &\mathit{digit}(1,4) \leftarrow \\ &\mathit{digit}(2,1) \leftarrow \end{aligned} \tag{4.6}$$

by applying T_P twice we deduce $\mathit{sum}(5)$ as expected.

For a general logic program successive applications of T_P may not be appropriate (for example, programs that contain negative loops). For this reason, we assume our logic program is hierarchical (see definition 4.2.1). In this chapter, we take logic programs such as 4.4 and embed them into matrices to be incorporated as additional layers of a neural network ("Logic Program Layer"). Since we wish to keep the process differentiable end-to-end, we do not apply the symbol grounding function at this stage (but we will do so in later chapters). Instead, the sequence of Logic Program Layers acts as our reasoning function g.

To train our model, we are given a dataset of samples $\mathcal{D} = \{X, y\} \subset \mathcal{X} \times \mathcal{Y}$. Crucially, $\mathcal{D}$ does not contain any information about the correct latent labels in $\mathcal{Z}$. To perform well at the task, the model would have to learn how to correctly map a raw input x_i to its latent concept value, which is the responsibility of the f_i functions.

4.2 Reasoning with Logic Program Layers

4.2.1 Hierarchical Logic Programs

Consider the program 4.4. In total, the program comprises 100 rules. Note that the set of head atoms and body atoms are disjoint, resulting in a loop-free program. This is typical of neuro-symbolic tasks. More generally, the head atoms of rules, such as the sum atoms, may be fed into further rules to derive additional downstream concepts. However, the program will still be loop-free.

Example 4.2.1. *The MNIST addition task can be generalised to a multi-digit variant. In this variant, we are again given two numbers in the form of MNIST digits, but this time each number consists of N digits (so there are $2N$ images). The latent concept values are now represented as atoms of the form $\mathit{digit}(I, L, D)$ where I and D play the same role as before, and L is the index of the position of the digit within the number (1 means the ones digit, 2 means the tens, and so forth). The task is still to output the sum of the two numbers in symbolic form. A naive encoding of the program would include every possible combination of low-level observations, and their associated sum. For example, in the case of $N = 2$, the first few rules would be:*

$$\begin{aligned} &\mathit{sum}(0) \leftarrow \mathit{digit}(1,2,0), \mathit{digit}(1,1,0), \mathit{digit}(2,2,0), \mathit{digit}(2,1,0) \\ &\mathit{sum}(1) \leftarrow \mathit{digit}(1,2,0), \mathit{digit}(1,1,1), \mathit{digit}(2,2,0), \mathit{digit}(2,1,0) \\ &\mathit{sum}(1) \leftarrow \mathit{digit}(1,2,0), \mathit{digit}(1,1,0), \mathit{digit}(2,2,0), \mathit{digit}(2,1,1) \\ &\mathit{sum}(2) \leftarrow \mathit{digit}(1,2,0), \mathit{digit}(1,1,2), \mathit{digit}(2,2,0), \mathit{digit}(2,1,0) \\ &\mathit{sum}(2) \leftarrow \mathit{digit}(1,2,0), \mathit{digit}(1,1,1), \mathit{digit}(2,2,0), \mathit{digit}(2,1,1) \\ &\qquad\vdots \end{aligned}$$

which in total would include 10,000 rules. For a general N, there are $10^{2N} \sim \mathcal{O}\left(100^N\right)$ rules, preventing this approach from scaling up. A smarter encoding breaks down the sum operation

into steps, each involving the addition of two digits in the same position. Introduce the atom $sum(L, S)$ *which encodes the proposition that the sum up until the digit in position* L *(inclusive) is* S*. For* $N = 2$*, the first step can be encoded with the following set of rules:*

$$
\begin{aligned}
sum(1,0) &\leftarrow digit(1,1,0), digit(2,1,0)\\
sum(1,1) &\leftarrow digit(1,1,1), digit(2,1,0)\\
sum(1,1) &\leftarrow digit(1,1,0), digit(2,1,1)\\
sum(1,2) &\leftarrow digit(1,1,2), digit(2,1,0)\\
sum(1,2) &\leftarrow digit(1,1,1), digit(2,1,1)\\
&\vdots
\end{aligned}
$$

which includes 100 rules. The rules for the second step are:

$$
\begin{aligned}
sum(2,0) &\leftarrow digit(1,2,0), digit(2,2,0), sum(1,0)\\
sum(2,1) &\leftarrow digit(1,2,1), digit(2,2,0), sum(1,0)\\
sum(2,1) &\leftarrow digit(1,2,0), digit(2,2,1), sum(1,0)\\
sum(2,2) &\leftarrow digit(1,2,2), digit(2,2,0), sum(1,0)\\
sum(2,2) &\leftarrow digit(1,2,1), digit(2,2,1), sum(1,0)\\
&\vdots\\
sum(2,1) &\leftarrow digit(1,2,0), digit(2,2,0), sum(1,1)\\
sum(2,2) &\leftarrow digit(1,2,1), digit(2,2,0), sum(1,1)\\
sum(2,2) &\leftarrow digit(1,2,0), digit(2,2,1), sum(1,1)\\
sum(2,3) &\leftarrow digit(1,2,2), digit(2,2,0), sum(1,1)\\
sum(2,3) &\leftarrow digit(1,2,1), digit(2,2,1), sum(1,1)\\
&\vdots
\end{aligned}
$$

Since there are 19 distinct $sum(1, S)$ *atoms, and 100 rules in the second step per* S*, the second step has a total of 1,900 rules. We therefore encoded the rules of 2-digit summation using 2000 rules (across the two steps) instead of 10,000. It can be shown that using this method, for a general* N *there are* $\frac{200}{9}\left(10^N - 1\right) - 100N$ *rules, or* $\mathcal{O}\left(10^N\right)$*. While the number of rules still grows exponentially with* N*, it does so much more slowly than the previous approach.*

When looking at the rules of every step individually, the head atoms of the rules are disjoint from the body atoms. Therefore, the program is still loop-free.

The steps in the previous example create a natural division of the program into *layers*, with each layer deriving further high-level concepts using the output of the previous layer, not dissimilar to the multilayered architecture of neural networks. This motivates the next definition of hierarchical programs.

Definition 4.2.1. *(Hierarchical logic program) A hierarchical logic program* P *is a union of disjoint logic programs* $P = \bigcup_{i=1}^{L} P_i$ *where for all* $a \in B_P$ *and for all* $r \in P_i$*,* $r' \in P_j$ *(*$1 \leq i, j \leq L$*) the following holds:*

- *If* $h^r = a$ *and* $h^{r'} = a$ *then* $i = j$*.*
- *If* $h^r = a$ *and* $a \in B^{r'} \cup C^{r'}$ *then* $j = i + 1$*.*

The program P_i is referred to as the i'th layer of P.

The two bullet points define exactly the concept of a program that can be divided into a hierarchy of layers. The first bullet point states that an atom can only be defined in exactly one layer. The second bullet point says that if an atom appears in the body of a layer, it could only have been defined in the previous layer. These two conditions rule out any loops in the program. Note that the first bullet point can be directly derived from the second one if one adds the condition that every atom that appears in the head of a rule also appears in the body of a rule (except for the last layer).

Example 4.2.2. *Consider the program described in example 4.2.1 to solve the multi-digit MNIST task. The steps described in the example correspond to the different layers of the program. P_1 defines the rules of summing the ones digit. P_2 encodes the rules of summing the tens digit, with the accumulated sum from P_1 encoded using the sum$(1, S)$ atoms. The rules for the hundreds digit become P_3, and so forth.*

In a hierarchical program, the heads of the previous layer become body atoms of the next layer. Since the atoms of the heads and the body are disjoint, we can express the Herbrand base of each layer as the union of two sets.

Definition 4.2.2. *(Inbound and outbound Herbrand base)*

1. *The inbound Herbrand base of layer i is:*

$$B_{P_i}^{in} = \{a \in B_P | \exists r \in P_i, a \in B^r \cup C^r\} \tag{4.7}$$

2. *The outbound Herbrand base of layer i is:*

$$B_{P_i}^{out} = \{a \in B_P | \exists r \in P_i, a = h^r\} \tag{4.8}$$

The Herbrand base of layer P_i can therefore be expressed as:

$$B_{P_i} = B_{P_i}^{\text{in}} \cup B_{P_i}^{\text{out}} \tag{4.9}$$

and for the entire program we have:

$$B_P = \bigcup_{i=1}^{L} B_{P_i} \tag{4.10}$$

Definition 4.2.3. *(Inbound interpretation, outbound interpretation) An inbound interpretation I^{in} of layer P_i is a subset of $B_{P_i}^{in}$. An outbound interpretation I^{out} of layer P_i is a subset of $B_{P_i}^{out}$. The immediate consequence operator of layer P_i is defined as:*

$$\begin{aligned} &T_{P_i} : 2^{B_{P_i}^{in}} \to 2^{B_{P_i}^{out}} \\ &T_{P_i}\left(I^{in}\right) = \left\{h^r \middle| r \in P_i, B^r \subseteq I^{in}, C^r \cap I^{in} = \emptyset\right\} \end{aligned} \tag{4.11}$$

Recall that for $h \in B_P^{out}$ we define rules$(h) = \{r \in P \mid h = h^r\}$ - the set of rules with h as their head.

4.2.2 Matrix-Vector Representation

In the previous chapter, our matrix-based representation of programs relied on programs following the MD condition. While this can be achieved for a general program using an efficient

transformation, the consequence of this transformation is that the application of T_P of the original program corresponds to two applications of T'_P of the transformed program. This is undesirable in deep learning, since it would mean the introduction of an additional auxiliary layer.

To counter this, we will now develop a matrix-based representation that does not require programs to follow the MD condition. The issue is that T_P is not a linear operation, as it involves conjunction followed by disjunction. Our matrix-based approach from the previous chapter is trying to implement this operation using a linear operation followed by an element-wise non-linearity, which is not possible, unless the program follows the MD condition. To remedy this, we will implement only the conjunction step using the linear operation, and then perform disjunction using max, as is commonly used in fuzzy logic (van Krieken et al., 2020).

Definition 4.2.4. *(Rule vector embedding) Let P be a normal program with R rules. Introduce an order over the rules of P: r_1, r_2, ..., r_R. Each rule is mapped to a vector in the standard basis of $\mathbb{R}^R$. Namely, the rule r_i is mapped to the standard basis vector $\boldsymbol{e}_i$. We denote $\boldsymbol{v}^{r_i} = \boldsymbol{e}_i$.*

Example 4.2.3. *Suppose that P_i is the following program that does not adhere to the MD condition:*

$$\begin{aligned} r_1 &: p \leftarrow a, b \\ r_2 &: p \leftarrow c, d \\ r_3 &: q \leftarrow a, c \\ r_4 &: q \leftarrow b, not\, d \end{aligned} \tag{4.12}$$

$R = 4$ and each rule has a vector of the standard basis of $\mathbb{R}^4$ associated with it:

$$\boldsymbol{v}^{r_1} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad \boldsymbol{v}^{r_2} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad \boldsymbol{v}^{r_3} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad \boldsymbol{v}^{r_4} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{4.13}$$

We maintain the same atom embedding and atom set embeddings as in chapter 3, but we can remove the $\top$ and $\bot$ constants from our Herbrand bases as they are no longer needed. Consequently, B^r and C^r can now be empty and programs do not contain the tautology $\top \leftarrow \top$. While programs can have facts, we do not encode these in the layers of the program. Instead, these are treated as symbolic inputs to the reasoning function. Hence, we can assume that $|B^r| + |C^r| \geq 1$.

Definition 4.2.5. *(Normal clause embedding) Let P_i be a layer of a normal hierarchical logic program and r a clause. We embed r as a matrix-vector tuple (D^r, b^r) defined as follows:*

$$D^r = \boldsymbol{v}^r \left(\frac{\boldsymbol{v}^{B^r} - \boldsymbol{v}^{C^r}}{n^r + m^r} \right)^\intercal \tag{4.14}$$

$$\boldsymbol{b}^r = \frac{m^r}{n^r + m^r} \boldsymbol{v}^r \tag{4.15}$$

where $n^r = |B^r|$ and $m^r = |C^r|$.

Definition 4.2.6. *(Normal program embedding) Let P_i be a layer of a normal hierarchical logic.*

We embed P as a matrix-vector tuple $\left(D^{P_i}, b^{P_i}\right)$ defined as follows:

$$D^{P_i} = \sum_{r \in P_i} D^r \tag{4.16}$$

$$\boldsymbol{b}^{P_i} = \sum_{r \in P_i} \boldsymbol{b}^r \tag{4.17}$$

Recall that in the previous chapter, we embedded normal rules into matrices using the notion of a program reduct. Here we have removed the dependency on program reducts and embedded a rule directly. P is encoded into a matrix-vector pair $\left(D^P, \boldsymbol{b}^P\right)$, where $D^P \in \mathbb{R}^{R \times N}$ and $\boldsymbol{b}^P \in \mathbb{R}^R$. $\boldsymbol{b}^P$ can be thought of as a "bias vector" that always adds $\frac{m^r}{n^r+m^r}$ to the value for r. When computing the immediate consequence for interpretation I^{in}, we can multiply $D^r \boldsymbol{v}^{I^{\text{in}}}$ and add $\boldsymbol{b}^r$. If the body of r is satisfied, we would get $\left(\boldsymbol{v}^{B^r}\right)^\intercal \boldsymbol{v}^{I^{\text{in}}} = n^r$ and $\left(\boldsymbol{v}^{C^r}\right)^\intercal \boldsymbol{v}^{I^{\text{in}}} = 0$ so $D^r \boldsymbol{v}^{I^{\text{in}}} + \boldsymbol{b}^r = 1$ as required. If it is not satisfied, the contribution of $\boldsymbol{v}^{B^r}$ would be less than n^r, so the result is guaranteed to be less than 1 when adding $\boldsymbol{b}^r$. One can think of the bias vector as implementing negation-as-failure - the atoms in C^r are assumed to always contribute to the truth value of the head of r via the bias, unless they are shown not to, by virtue of being in I^{in}.

Example 4.2.4. *The matrix encoding of the program 4.12 is:*

$$D^{P_i} = \begin{array}{c} \\ p^1 \\ p^2 \\ q^1 \\ q^2 \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & \frac{1}{2} & 0 & -\frac{1}{2} \end{bmatrix} \end{array} \tag{4.18}$$

and the vector embedding is:

$$\boldsymbol{b}^{P_i} = \begin{array}{c} \\ p^1 \\ p^2 \\ q^1 \\ q^2 \end{array} \begin{array}{c} \\ \begin{bmatrix} 0 \\ 0 \\ 0 \\ \frac{1}{2} \end{bmatrix} \end{array} \tag{4.19}$$

where we have:

$$rules(p) = \{r_1, r_2\} \qquad rules(q) = \{r_3, r_4\} \tag{4.20}$$

Theorem 4.2.1. *Let P_i be a layer of a normal hierarchical logic program. Let I^{in} be an inbound interpretation and $I^{out} = T_{P_i}\left(I^{in}\right)$. Then:*

$$\boldsymbol{v}^{I^{out}} = \sum_{h \in B_P^{out}} \boldsymbol{v}^h \left[\max_{r \in rules(h)} \left(\boldsymbol{v}^r\right)^\intercal H_1 \left(D^{P_i} \boldsymbol{v}^{I^{in}} + \boldsymbol{b}^{P_i} \right) \right] \tag{4.21}$$

To prove the above theorem, we will first prove two lemmas.

Lemma 4.2.1. *Let $\boldsymbol{e}_j$ be a vector in the standard basis of $\mathbb{R}^N$ (for some N) and let $\boldsymbol{v} \in \mathbb{R}^N$. Then:*

$$\boldsymbol{e}_j^\intercal H_1\left(\boldsymbol{v}\right) = H_1\left(\boldsymbol{e}_j^\intercal \boldsymbol{v}\right) \tag{4.22}$$

Proof. Recall that:

$$\delta_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases} \tag{4.23}$$

We have:

$$\begin{aligned} \boldsymbol{e}_j^{\intercal} H_1(\boldsymbol{v}) &= \sum_{i=1}^{N} (e_j)_i H_1(v)_i = \sum_{i=1}^{N} \delta_{ij} H_1(v_i) = H_1(v_j) = \\ &= H_1\left(\sum_{i=1}^{N} \delta_{ij} v_i\right) = H_1\left(\sum_{i=1}^{N} (e_j)_i v_i\right) = H_1\left(\boldsymbol{e}_j^{\intercal} \boldsymbol{v}\right) \end{aligned} \tag{4.24}$$

□

Lemma 4.2.2. *Let S be a non-empty finite set of real numbers. Then:*

$$\max_{s \in S} H_1(s) = H_1\left(\max_{s \in S} s\right) \tag{4.25}$$

Proof. Suppose there exists $s \in S$ such that $s \geq 1$. Then:

$$\max_{s' \in S} H_1(s') \geq H_1(s) = 1 \tag{4.26}$$

and since $H_1(s')$ is upper bounded by 1, the inequality is strict. In addition, since H_1 is a monotonically non-decreasing function:

$$H_1\left(\max_{s' \in S} s'\right) \geq H_1(s) = 1 \tag{4.27}$$

as required. Conversely, suppose for all $s \in S$ we have $s < 1$. Then $\max_{s \in S} s < 1$ and $H_1(\max_{s \in S} s) = 0$. Let $s' \in S$ be the value for which $H_1(s)$ is maximal. Then:

$$\max_{s \in S} H_1(s) = H_1(s') = 0 \tag{4.28}$$

as required. □

We can now move to prove Theorem 4.2.1.

Proof. Let $a \in B_{P_i}$ and let $\boldsymbol{u}$ equal the right side in equation 4.21. We need to show $\boldsymbol{v}^{I^{\text{out}}} = \boldsymbol{u}$. We have:

$$\begin{aligned} (\boldsymbol{v}^a)^{\intercal} \boldsymbol{u} &= (\boldsymbol{v}^a)^{\intercal} \sum_{h \in B_P^{out}} \boldsymbol{v}^h \left[\max_{r \in rules(h)} (\boldsymbol{v}^r)^{\intercal} H_1\left(D^{P_i} \boldsymbol{v}^{I^{\text{in}}} + \boldsymbol{b}^{P_i}\right)\right] = \\ &= \sum_{h \in B_P^{out}} (\boldsymbol{v}^a)^{\intercal} \boldsymbol{v}^h \left[\max_{r \in rules(h)} (\boldsymbol{v}^r)^{\intercal} H_1\left(D^{P_i} \boldsymbol{v}^{I^{\text{in}}} + \boldsymbol{b}^{P_i}\right)\right] = \\ &= \sum_{h \in B_P^{out}} \delta_{ah} \left[\max_{r \in rules(h)} (\boldsymbol{v}^r)^{\intercal} H_1\left(D^{P_i} \boldsymbol{v}^{I^{\text{in}}} + \boldsymbol{b}^{P_i}\right)\right] = \\ &= \max_{r \in rules(a)} (\boldsymbol{v}^r)^{\intercal} H_1\left(D^{P_i} \boldsymbol{v}^{I^{\text{in}}} + \boldsymbol{b}^{P_i}\right) \end{aligned} \tag{4.29}$$

we can now apply Lemmas 4.2.1 and 4.2.2 to get:

$$
\begin{aligned}
&= \max_{r \in rules(a)} H_1 \left((\boldsymbol{v}^r)^\intercal \, D^{P_i} \boldsymbol{v}^{I^{\text{in}}} + (\boldsymbol{v}^r)^\intercal \, \boldsymbol{b}^{P_i} \right) = \\
&= H_1 \left[\max_{r \in rules(a)} \left((\boldsymbol{v}^r)^\intercal \, D^{P_i} \boldsymbol{v}^{I^{\text{in}}} + (\boldsymbol{v}^r)^\intercal \, \boldsymbol{b}^{P_i} \right) \right] = \\
&= H_1 \left[\max_{r \in rules(a)} \left((\boldsymbol{v}^r)^\intercal \sum_{r' \in P_i} D^{r'} \boldsymbol{v}^{I^{\text{in}}} + (\boldsymbol{v}^r)^\intercal \sum_{r' \in P_i} \boldsymbol{b}^{r'} \right) \right] = \\
&= H_1 \left[\max_{r \in rules(a)} \left((\boldsymbol{v}^r)^\intercal \sum_{r' \in P_i} \boldsymbol{v}^{r'} \left(\frac{\boldsymbol{v}^{B^{r'}} - \boldsymbol{v}^{C^{r'}}}{n^{r'} + m^{r'}} \right)^\intercal \boldsymbol{v}^{I^{\text{in}}} + (\boldsymbol{v}^r)^\intercal \sum_{r' \in P_i} \frac{m^{r'}}{n^{r'} + m^{r'}} \boldsymbol{v}^{r'} \right) \right] = \quad (4.30) \\
&= H_1 \left[\max_{r \in rules(a)} \left(\sum_{r' \in P_i} \delta_{rr'} \left(\frac{\boldsymbol{v}^{B^{r'}} - \boldsymbol{v}^{C^{r'}}}{n^{r'} + m^{r'}} \right)^\intercal \boldsymbol{v}^{I^{\text{in}}} + \sum_{r' \in P_i} \delta_{rr'} \frac{m^{r'}}{n^{r'} + m^{r'}} \right) \right] = \\
&= H_1 \left[\max_{r \in rules(a)} \left(\left(\frac{\boldsymbol{v}^{B^r} - \boldsymbol{v}^{C^r}}{n^r + m^r} \right)^\intercal \boldsymbol{v}^{I^{\text{in}}} + \frac{m^r}{n^r + m^r} \right) \right] = \\
&= H_1 \left[\max_{r \in rules(a)} \frac{\left(\boldsymbol{v}^{B^r} \right)^\intercal \boldsymbol{v}^{I^{\text{in}}} - \left(\boldsymbol{v}^{C^r} \right)^\intercal \boldsymbol{v}^{I^{\text{in}}} + m^r}{n^r + m^r} \right]
\end{aligned}
$$

Now, suppose $a \in I^{\text{out}}$ so that $(\boldsymbol{v}^a)^\intercal \, \boldsymbol{v}^{I^{\text{out}}} = 1$. Then there exists $r' \in P_i$ such that $h^{r'} = a$, $B^{r'} \subseteq I^{\text{in}}$, $C^{r'} \cap I^{\text{in}} = \emptyset$. We have $\left(\boldsymbol{v}^{B^{r'}} \right)^\intercal \boldsymbol{v}^{I^{\text{in}}} = n^r$ and $\left(\boldsymbol{v}^{C^{r'}} \right)^\intercal \boldsymbol{v}^{I^{\text{in}}} = 0$, so:

$$
\begin{aligned}
\max_{r \in rules(a)} \frac{\left(\boldsymbol{v}^{B^r} \right)^\intercal \boldsymbol{v}^{I^{\text{in}}} - \left(\boldsymbol{v}^{C^r} \right)^\intercal \boldsymbol{v}^{I^{\text{in}}} + m^r}{n^r + m^r} &\geq \frac{\left(\boldsymbol{v}^{B^{r'}} \right)^\intercal \boldsymbol{v}^{I^{\text{in}}} - \left(\boldsymbol{v}^{C^{r'}} \right)^\intercal \boldsymbol{v}^{I^{\text{in}}} + m^{r'}}{n^{r'} + m^{r'}} = \quad (4.31) \\
&= \frac{n^{r'} - 0 + m^{r'}}{n^{r'} + m^{r'}} = 1 \quad \Rightarrow \quad (\boldsymbol{v}^a)^\intercal \, \boldsymbol{u} = 1
\end{aligned}
$$

Conversely, suppose $a \notin I^{\text{out}}$ so that $(\boldsymbol{v}^a)^\intercal \, \boldsymbol{v}^{I^{\text{out}}} = 0$. Then for every $r \in rules(a)$ we have either $B^r \not\subseteq I^{\text{in}}$ or $C^r \cap I^{\text{in}} \neq \emptyset$. Therefore either $\left(\boldsymbol{v}^{B^{r'}} \right)^\intercal \boldsymbol{v}^{I^{\text{in}}} < n^r$ or $\left(\boldsymbol{v}^{C^{r'}} \right)^\intercal \boldsymbol{v}^{I^{\text{in}}} > 0$. Therefore for all $r' \in rules(a)$:

$$
\max_{r \in rules(a)} \frac{\left(\boldsymbol{v}^{B^r} \right)^\intercal \boldsymbol{v}^{I^{\text{in}}} - \left(\boldsymbol{v}^{C^r} \right)^\intercal \boldsymbol{v}^{I^{\text{in}}} + m^r}{n^r + m^r} < \frac{n^{r'} - 0 + m^{r'}}{n^{r'} + m^{r'}} = 1 \quad \Rightarrow \quad (\boldsymbol{v}^a)^\intercal \, \boldsymbol{u} = 0 \qquad (4.32)
$$

Putting it all together, we showed that if $a \in I^{\text{out}}$ then $(\boldsymbol{v}^a)^\intercal \, \boldsymbol{u} = 1$ and if $a \notin I^{\text{out}}$ then $(\boldsymbol{v}^a)^\intercal \, \boldsymbol{u} = 0$, which means $\boldsymbol{v}^{I^{\text{out}}} = \boldsymbol{u}$ as required. □

Example 4.2.5. *For the interpretation $I^{in} = \{a, b, d\}$ we first perform:*

$$D^{P_i}\boldsymbol{v}^{I^{in}} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & \frac{1}{2} & 0 & -\frac{1}{2} \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ \frac{1}{2} \\ \frac{1}{2} \\ 0 \end{bmatrix} \tag{4.33}$$

$$D^{P_i}\boldsymbol{v}^{I^{in}} + \boldsymbol{b}^{P_i} = \begin{bmatrix} 1 \\ \frac{1}{2} \\ \frac{1}{2} \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ \frac{1}{2} \end{bmatrix} = \begin{bmatrix} 1 \\ \frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{2} \end{bmatrix} \tag{4.34}$$

$$H_1\left(D^{P_i}\boldsymbol{v}^{I^{in}} + \boldsymbol{b}^{P_i}\right) = H_1\left(\begin{bmatrix} 1 \\ \frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{2} \end{bmatrix}\right) = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{4.35}$$

We then perform the max operation for each head separately:

$$\max_{r \in rules(p)} (\boldsymbol{v}^r)^\intercal H_1\left(D^{P_i}\boldsymbol{v}^{I^{in}} + \boldsymbol{b}^{P_i}\right) = \\ = \max\left\{ \left(\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}\right)^\intercal \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \left(\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}\right)^\intercal \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \right\} = \max\{1, 0\} = 1 \tag{4.36}$$

$$\max_{r \in rules(q)} (\boldsymbol{v}^r)^\intercal H_1\left(D^{P_i}\boldsymbol{v}^{I^{in}} + \boldsymbol{b}^{P_i}\right) = \\ = \max\left\{ \left(\begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}\right)^\intercal \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \left(\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}\right)^\intercal \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \right\} = \max\{0, 0\} = 0 \tag{4.37}$$

Then sum over all the heads in the Outbound Herbrand base:

$$\boldsymbol{v}^{I^{out}} = \boldsymbol{v}^p \cdot 1 + \boldsymbol{v}^q \cdot 0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \tag{4.38}$$

which represents $\{p\}$ as expected.

As in the previous chapter, we make our T_{P_i} operation differentiable by replacing H_1 with $\sigma_{\gamma,\tau}$. The resulting operation is then considered a Logic Program Layer encapsulating P_i. We can stack these layers one after another to embed the entire hierarchical program into a neural network. As before, the optimal values of γ and τ depend solely on the lengths of the rules and therefore can be computed before training begins.

4.2.3 Evaluation

In the following, we discuss experiments evaluating our Logic Program Layer approach on two tasks involving both low-level perception and high-level reasoning. The first task is the generalised MNIST Addition which involves arithmetic operations on images of digits, and the second involves syllogistic arguments over geometric shapes and their relations. In our evaluation, we

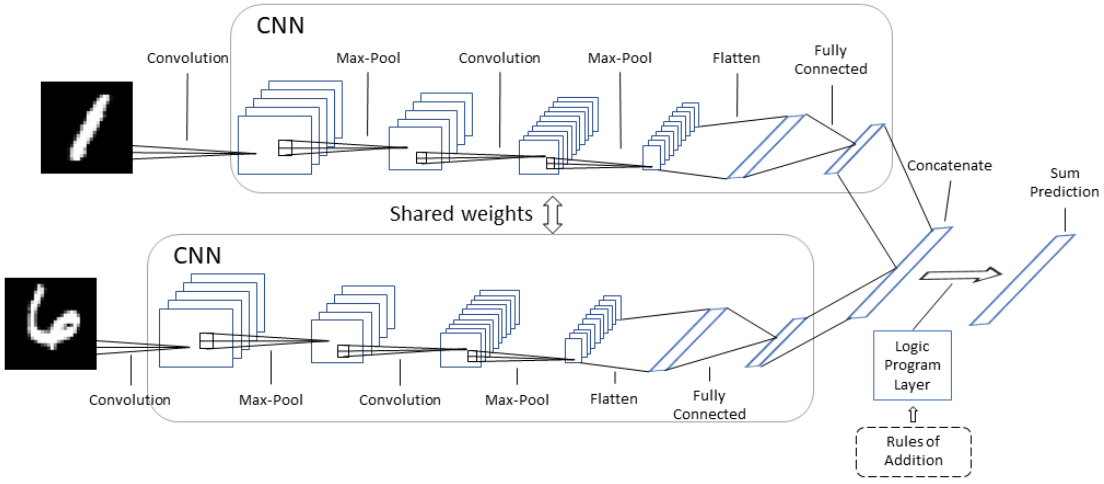


Figure 4.2: Architecture for neural-symbolic MNIST Addition task.

focus on the success of our approach in training the perception module and compare it to existing neural-symbolic architectures as well as purely neural models.

N-digit MNIST Addition

The generalised MNIST Addition task was previously introduced to illustrate some of the concepts of our neural-symbolic end-to-end training approach. Here we define it formally for a general N. The task involves summing two numbers, indexed $I = 1, 2$. Each summand has N digits, which are given in MNIST image form. The task has raw inputs $x_{I,D} = \left(\text{digit}\left(I, D\right), [0,1]^{28\times 28}, \boldsymbol{v}_{I,D}\right)$ for $I = 1, 2$ and $D = 1 \ldots N$. All the raw inputs map to the same latent concept $c = (\text{digit}, \{0, \ldots, 9\}, \mathcal{Z})$. The target concept is the sum of the numbers $y = \left(\text{sum}, \left\{0, \ldots, 2 \cdot 10^N - 2\right\}, t\right)$.

The perception function f we use to map images to digit latent concept space is a LeNet (Lecun et al., 1998). The model architecture in the case of single-digit numbers ($N = 1$) appears in Figure 4.2. As all the images are associated with the same latent concept, we apply the LeNet, a common CNN architecture, to each image and the network's weights are shared across all inputs. The network output consists of 10 softmax probabilities, one for each possible digit (which is our latent concept space $\mathcal{Z}$). Hence this network is trained to be an MNIST digit recogniser in a weakly-supervised manner. We therefore have a vector of size 10 for each image, which are then concatenated into a single vector. This vector becomes a (fuzzy) inbound interpretation.

The reasoning function consists of the rules of addition with accumulated sum, given in the form of a logic program and captured by a sequence of Logic Program Layers. For N-digit addition, there is a single layer per digit from 1 to N. The i^{th} layer is captured by a program P_i. The i^{th} layer corresponds to summing the i^{th} digit (1 being least significant, N most significant) with the accumulated sum from the previous layer. A first-order version of P_i is given by:

$$\begin{gathered}\text{sum}(i, Sum) \leftarrow \text{digit}\left(1, i, N_1\right), \text{digit}\left(2, i, N_2\right), \text{sum}\left(i-1, S\right), \\ Sum = N_1 \cdot 10^{i-1} + N_2 \cdot 10^{i-1} + S.\end{gathered} \tag{4.39}$$

where N_1 and N_2 range from 0 to 9. The values of $\text{digit}\left(1, i, N_1\right)$ and $\text{digit}\left(2, i, N_2\right)$ are

Dataset	1 Digit	2 Digits	3 Digits
Train	29500	14750	9833
Validation	500	250	166
Test	5000	2500	1666

(a) Sizes of datasets used.

Train Num Digits	1 Digit	2 Digits	3 Digits
1	97.81%	95.73%	93.57%
2	97.91%	95.88%	93.87%
3	97.19%	94.5%	91.89%

(b) Accuracy on the test set.

Table 4.1: Dataset information and accuracy for the MNIST Addition experiment.

given by the inbound interpretation, directly fed from the perception model. The value for $\text{sum}(i-1, S)$ is given by the previous layer (for the first layer it is dropped), where S ranges over the possible sums from the previous layer (0 to $2 * 10^{i-1} - 2$). The rules are grounded and the matrix-vector form is then constructed from them. Recall that each sample comes with a label for the sum. We can construct from this label the outbound interpretation of the final layer. This allows us to backpropagate through the Logic Program layers and into the CNN, training the perception component.

As a loss function, we use binary cross-entropy so that the truth value of each atom in the outbound interpretation is evaluated independently. We test the accuracy of the model in its ability to predict the correct sum. Note that our ability to change the rules of a Logic Program Layer means a model can be trained and tested on numbers of different lengths. So during training, we can show the model single-digit numbers, and then test it on numbers with two digits by changing the Logic Program Layer to encode the appropriate rules. This is also applicable the other way - we can train on two-digit numbers and test on single-digits. Table 4.1a details the sizes of the datasets used, while Table 4.1b reports the average test accuracy over 10 runs for various training and test number lengths. For instance the first row of Table 4.1b refers to test accuracy achieved when training on single-digit numbers. The second column talks about test accuracy when testing over two-digit numbers. So the test accuracy for two-digit numbers, when we trained on single-digit numbers, is 95.73%.

We also compare our performance to two competing neural-symbolic systems, NeurASP (Yang et al., 2020b) and DeepProbLog (Manhaeve et al., 2019) in terms of accuracy and training time. As can be seen in Tables 4.2a and 4.2b, the accuracy is similar to competing approaches, but training time is much shorter (Logic Program Layers appears as LPL) owing to a much shorter per-epoch time. While DeepProbLog and NeurASP only run for 2-3 epochs, their epochs can last between minutes to hours (depending on number of digits) while our Logic Program Layers only require between a few seconds (for a single digit) to a few minutes (3 digits) per epoch. We also compare to a baseline CNN that attempts to solve the problem end-to-end (following the architecture proposed in Manhaeve et al. (2019)). We outperform the baseline CNN significantly. This demonstrates the data efficiency of our neural-symbolic approach compared to pure deep learning. The baseline CNN would perform much better if it were trained on a much larger dataset covering all possible cases. But breaking down the task into a perception task requiring training and a reasoning task requiring no training greatly reduces the number of samples required for successful training.

It is worth noting that while our approach scales up better than the baseline approaches, it still suffers from some scalability issues. As N increases, the number of rules still grows exponentially, and so does the size of our matrix encoding. In our experimental setup, we weren't able to run training for $N = 4$, although this is likely possible with more GPU memory. Even then, $N = 5$ would not be possible to train. In the next chapter, we introduce a far more scalable approach to neuro-symbolic training.

Method	1 Digit	2 Digits
LPL (Ours)	97.81%	95.88%
DeepProbLog	97.20%	95.16%
NeurASP	97.57%	N/A
CNN Baseline	94.84%	79.38%

(a) Test accuracy comparison. NeurASP do not report results other than single-digit addition.

Method	1 Digit	2 Digits	3 Digits
LPL	1m 11s	8m 36s	1h 5m 35s
DPLog	28m 51s	4h 22m 11s	Timeout
NASP	1m 50s	N/A	N/A

(b) Training time comparison. NeurASP (NASP) do not report results other than single-digit addition. DeepProbLog (DPLog) training was stopped after 24 hours without significant progress.

Table 4.2: Accuracy comparison and training time for the MNIST Addition experiment.

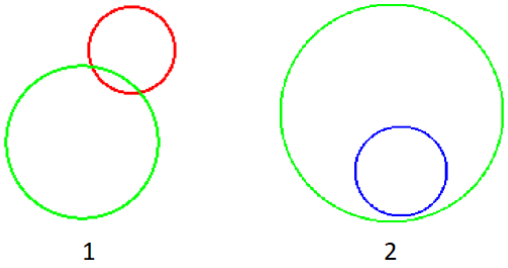


Figure 4.3: Input to the Syllogism task. There are two input images, each an Euler diagram describing a relation between two sets.

Syllogism

Syllogism is a logical deduction where a conclusion is drawn based on two premises. This experiment tests the performance of our Logic Program Layer approach in such a setting. The task, originally appearing in Wang et al. (2018a), involves two Euler Diagrams, such as the ones appearing in Figure 4.3. These diagrams describe relations between sets. One diagram illustrates the relation between the red and green set, while the other between green and blue. If we look at a pair of sets, such as red and green, there are four possibilities:

1. Red contains green.
2. Red is contained in green.
3. Red and green intersect (but do not contain each other).
4. Red and green are disjoint.

Given the input images such as in Figure 4.3, the red-green relation and the green-blue relation are known, and these become our premises. Our conclusion is the *possible* relations between the red and blue sets. In Figure 4.3 for instance, red and green intersect and blue is contained in green. We therefore conclude that one of the following holds:

1. Red contains blue.
2. Red intersects with blue.
3. Red and blue are disjoint.

The task then, is given the two input images, output all possible red-blue relations. It's worth noting that in Euler diagrams the size and exact position of each circle are irrelevant. Only the abstract relation between the sets is conveyed. Therefore a machine learning model cannot learn to use simple geometric information such as these. Instead, it must analyse the relative

geometric information between the two circles. For example, is one fully contained in another? Do they intersect anywhere?

In Wang et al. (2018a), a neural architecture is proposed to solve this task. The authors divide the architecture into two parts - a perception network and a reasoning network. The perception network is a CNN that processes each image individually and outputs an embedding vector - a representation of the diagram in high-dimensional vector space that is not easily interpretable. The two vectors are concatenated and fed into the reasoning network that is supposed to simulate the logical deduction process and output all possible conclusions about the red-blue pair. The authors generated 96000 samples for training and reported a high accuracy of 99.5%.

But an entirely neural network-based solution to the task hurts the generality of the solution. The reasoning network is only trained to work in a specific embedding space. It would be utterly useless if the circles were replaced with squares, for instance, even though the underlying logic did not change. In addition, training a reasoning network requires a larger number of samples, while specifying the logical rules in this case is very simple. We therefore would like to replace the reasoning network with a Logic Program Layer in the hope of achieving both better generality and a smaller sample size requirement.

In this task, we have two raw inputs:

$$x_1 = \left(\text{red_green}, [0,1]^{64\times64\times3}, \boldsymbol{v_1}\right) \qquad x_2 = \left(\text{green_blue}, [0,1]^{64\times64\times3}, \boldsymbol{v_2}\right) \tag{4.40}$$

where the raw input space is 64×64 RGB images (normalised to $[0,1]$). They are both associated with the latent concept $c = (\text{relation}, \{\text{contained}, \text{contains}, \text{intersect}, \text{disjoint}\}, \mathcal{Z})$. Just as in the MNIST Addition task, we use a CNN as a perception function to process each image and output softmax probabilities. In this case, the output size would be 4, for the four possible relations. During training this CNN will essentially learn to recognise set relations from Euler diagrams. These softmax probabilities for both images will then be concatenated and become a fuzzy inbound interpretation. The mapping from red-green and green-blue to red-blue relations can be specified in a simple logic program that is encoded in a Logic Program Layer. This layer will be applied to the inbound interpretation to output all the possible red-blue relations. The ground truth label specifies all these relations, so a binary cross-entropy loss is applied. Information can then backpropagate to the CNN.

How we choose to represent the rules affects our target concept. The mapping from the red-green and green-blue relations to the red-blue relation can be summarised in Table 4.3. We can treat the red-green and green-blue premises as body atoms. But for each premise pair, we need to output multiple conclusions. The most straightforward way to encode these mappings in a logic program is by using a different rule for each possible conclusion, given a premise pair.

With this approach, we have four target concepts:

$$y_1 = (\text{possibly_contained}, \{false, true\}, t_1) \qquad y_2 = (\text{possibly_contains}, \{false, true\}, t_2) \tag{4.41}$$

$$y_3 = (\text{possibly_intersect}, \{false, true\}, t_3) \qquad y_4 = (\text{possibly_disjoint}, \{false, true\}, t_4)$$

Red-Green \ Green-Blue	**Contained**	**Contains**	**Intersect**	**Disjoint**
Contained	contained	contained, contains, intersect, disjoint	contained, intersect, disjoint	disjoint
Contains	contained, contains, intersect	contains	contains, intersect	contains, intersect, disjoint
Intersect	contained, intersect	contains, intersect, disjoint	contained, contains, intersect, disjoint	contains, intersect, disjoint
Disjoint	contained, intersect, disjoint	disjoint	contained, intersect, disjoint	contained, contains, intersect, disjoint

Table 4.3: Consistent conclusions about red-blue pairs given red-green and green-blue premises.

and the rules:

$$
\begin{aligned}
&\text{possibly_contained}(\text{red}, \text{blue}) \leftarrow \text{contained}(\text{red}, \text{green}), \text{contained}(\text{green}, \text{blue}) \\
&\text{possibly_contained}(\text{red}, \text{blue}) \leftarrow \text{contained}(\text{red}, \text{green}), \text{contains}(\text{green}, \text{blue}) \\
&\text{possibly_contains}(\text{red}, \text{blue}) \leftarrow \text{contained}(\text{red}, \text{green}), \text{contains}(\text{green}, \text{blue}) \\
&\text{possibly_intersect}(\text{red}, \text{blue}) \leftarrow \text{contained}(\text{red}, \text{green}), \text{contains}(\text{green}, \text{blue}) \\
&\text{possibly_disjoint}(\text{red}, \text{blue}) \leftarrow \text{contained}(\text{red}, \text{green}), \text{contains}(\text{green}, \text{blue}) \\
&\text{possibly_contained}(\text{red}, \text{blue}) \leftarrow \text{contained}(\text{red}, \text{green}), \text{intersect}(\text{green}, \text{blue}) \\
&\quad \vdots \\
&\text{possibly_disjoint}(\text{red}, \text{blue}) \leftarrow \text{disjoint}(\text{red}, \text{green}), \text{disjoint}(\text{green}, \text{blue})
\end{aligned}
\tag{4.42}
$$

We performed experiments to compare three models.

- **Logic Program Layer**: The model using a CNN to generate latent concept predictions (we used sigmoid as the final activation layer), followed by the Logic Program Layer specified above.
- **Neural Reasoning model**: We replace the Logic Program Layer with a reasoning network as in Wang et al. (2018a). The CNN still outputs 4 values (with sigmoid activation at the final layer). The reasoning network can choose to use these values in any way to simulate the reasoning. In particular, they no longer have to carry the same meaning as the Logic Program Layer case. The reasoning network consists of several fully connected layers.

Method	**Contained**	**Contains**	**Intersect**	**Disjoint**	**Overall**
Logic Program Layer	72.01%	76.79%	74.85%	79.00%	75.66%
Neural Reasoning	85.48%	83.44%	75.93%	88.13%	83.24%
Embedding	98.12%	98.81%	98.50%	99.31%	98.69%

Table 4.4: Validation accuracy results for red-blue relation (final label).

	Run #1	**Run #2**	**Run #3**	**Run #4**	**Run #5**
Contained	75.00%	75.00%	45.12%	75.00%	25.00%
Contains	96.38%	74.75%	75.00%	75.50%	75.00%
Intersect	25.00%	75.00%	70.19%	75.00%	25.00%
Disjoint	96.13%	52.06%	77.25%	52.63%	75.00%

(a) Red-Green latent concept accuracy for Logic Program Layer.

	Run #1	**Run #2**	**Run #3**	**Run #4**	**Run #5**
Contained	73.94%	75.00%	75.00%	97.56%	67.87%
Contains	56.06%	58.13%	75.00%	25.00%	75.00%
Intersect	75.00%	25.00%	73.06%	25.00%	49.00%
Disjoint	75.00%	75.00%	25.00%	75.00%	75.00%

(b) Green-Blue latent concept accuracy for Logic Program Layer.

Table 4.5: Latent accuracy results for the Syllogism experiment.

- **Embedding model**: Similar to the neural reasoning model, but the CNN outputs a vector of arbitrary size (with ReLU activation as the final layer). This model is essentially the one used in Wang et al. (2018a). The representation of latent concepts (the premises) is vectors in an embedding space. We use an embedding size of 128. The reasoning network is unchanged from the Neural Reasoning model.

Table 4.4 compares the average accuracy over 5 training runs of the different models. The comparison here is based on their predictive ability of the target label - i.e. the red-blue relations. The accuracy for each distinct label is shown as well as the overall accuracy. We can see that the Logic Program Layer model does not perform as well as the pure neural network architectures. The embedding model reaches a very high accuracy of 98.69%, in line with the results reported in Wang et al. (2018a). The Neural Reasoning model, which uses the same type of output for the CNN as the Logic Program Layer model but has to learn the reasoning part, still performs better. This is not intuitive at first glance, since the Logic Program Layer already implements the ground truth function. In other words, despite the learning task being harder, the Neural Reasoning model performs better. To understand why this is happening, we examined the performance of the Logic Program Layer model in predicting the *latent concepts*. The results for the 5 different training runs appear in Table 4.5.

Inspection of the table shows accuracy in predicting latent concepts tends to be between 25% and 75%, and these specific values are the most common. This can be explained by the fact that the dataset is generated with a uniform distribution over latent concepts. For example, 25% of datapoints have "contained" as the red-green ground truth relation. If the CNN converges to a constant function that always states that contained $(\text{red}, \text{green})$ is false, it will be correct on 75% of datapoints. In fact, any value between 25% and 75% indicates the CNN is randomly guessing the truth value of contained $(\text{red}, \text{green})$. In contrast, values greater than 75% indicate successful learning of the latent concept, while values less than 25% indicate learning the opposite function

(i.e. not contained (red, green)). This is of course true for all latent concepts as much as it is true for contained (red, green). What these tables show us, then, is that the Logic Program Layer model fails to learn the latent concepts and therefore does not perform well on the task.

Why does the CNN fail to train? The reason lies in a certain ambiguity in the rules that affects only backpropagation. When doing forward propagation, the truth value of the latent concepts together with the rules allows the Logic Program Layer to correctly deduce the truth values of target concepts. But backpropagation involves reasoning in the other direction - from the truth value of the target concepts we must reason which latent concepts should have been true. This is known as *abductive reasoning*. The key difficulty in abductive reasoning is that multiple premises could indicate the same conclusion. For instance, suppose we are told (via the label) that possibly_contained (red, blue) is true for this particular sample. There are many rules with this atom as their head, and each such rule provides evidence for the atoms that appear in its body. Since we have:

$$\text{possibly_contained}\,(\text{red}, \text{blue}) \leftarrow \text{contained}\,(\text{red}, \text{green})\,, \text{contained}\,(\text{green}, \text{blue}) \tag{4.43}$$

then it's possible we should conclude contained (red, green) and contained (green, blue) are true. But we could have also considered:

$$\text{possibly_contained}\,(\text{red}, \text{blue}) \leftarrow \text{contained}\,(\text{red}, \text{green})\,, \text{contains}\,(\text{green}, \text{blue}) \tag{4.44}$$

which tells us that contained (red, green) and contained (green, blue). Since these options are mutually exclusive, we encounter an ambiguity in the signal being backpropagated through the Logic Program Layer to the CNN. In fact, possibly_contained (red, blue) is the head of many rules, exacerbating the problem.

One could argue that this ambiguity is easily removed when one considers all the rules together, rather than each individually. For instance, suppose that possibly_contained (red, blue) is true and possibly_contains (red, blue) is false, then the combination of latent concept values {contained (red, green) , contained (green, blue)} is still possible, but not the combination of concepts {contained (red, green) , contained (green, blue)} since the latter would also imply that possibly_contains (red, blue) is true. In other words, our label needs to be the *set* of all target concepts, rather than a vector which treats each one as an independent concept. With this in mind, we can define a single target concept for the task:

$$\begin{aligned} y = (&\text{consistent_conclusions}, \{ \\ &\emptyset, \{\text{contained}\}, \{\text{contains}\}, \{\text{intersect}\}, \{\text{disjoint}\}, \\ &\{\text{contained, contains}\}, \{\text{contained, intersect}\}, \\ &\{\text{contained, disjoint}\}, \{\text{contains, intersect}\}, \\ &\{\text{contains, disjoint}\}, \{\text{intersect, disjoint}\}, \\ &\{\text{contained, contains, intersect}\}, \\ &\{\text{contained, contains, disjoint}\}, \\ &\{\text{contained, intersect, disjoint}\}, \\ &\{\text{contains, intersect, disjoint}\}, \\ &\{\text{contained, contains, intersect, disjoint}\}, \\ &\}, t) \end{aligned} \tag{4.45}$$

Method	**Contained**	**Contains**	**Intersect**	**Disjoint**	**Overall**
Logic Program Layer	72.01%	76.79%	74.85%	79.00%	75.66%
Logic Program Layer #2	N/A	N/A	N/A	N/A	98.19%
Neural Reasoning	85.48%	83.44%	75.93%	88.13%	83.24%
Embedding	98.12%	98.81%	98.50%	99.31%	98.69%

Table 4.6: Validation accuracy results for red-blue relation (final label).

Concept	**Run #1**	**Run #2**	**Run #3**	**Run #4**	**Run #5**
Contained	99.50%	99.37%	99.37%	99.25%	99.50%
Contains	99.81%	99.81%	99.56%	99.81%	99.62%
Intersect	99.12%	98.37%	99.06%	98.50%	98.87%
Disjoint	98.37%	97.43%	98.50%	98.37%	99.00%

(a) Red-Green latent concept accuracy for Logic Program Layer #2.

Concept	**Run #1**	**Run #2**	**Run #3**	**Run #4**	**Run #5**
Contained	99.25%	99.37%	99.56%	99.56%	99.62%
Contains	99.12%	99.06%	99.19%	99.50%	99.37%
Intersect	98.75%	99.44%	98.69%	98.94%	98.50%
Disjoint	99.00%	98.19%	98.56%	98.94%	98.69%

(b) Green-Blue latent concept accuracy for Logic Program Layer #2.

Table 4.7: Results for the Syllogism Experiments with more informative target labels.

and rewrite the rules as follows:

$$
\begin{aligned}
&\text{consistent_conclusions}(\text{contained}) \leftarrow \text{contained}(\text{red}, \text{green}), \text{contained}(\text{green}, \text{blue})\\
&\text{consistent_conclusions}(\text{contained}, \text{contains}, \text{intersect}, \text{disjoint})\\
&\qquad\qquad \leftarrow \text{contained}(\text{red}, \text{green}), \text{contains}(\text{green}, \text{blue})\\
&\text{consistent_conclusions}(\text{contained}, \text{intersect}, \text{disjoint})\\
&\qquad\qquad \leftarrow \text{contained}(\text{red}, \text{green}), \text{intersect}(\text{green}, \text{blue})\\
&\text{consistent_conclusions}(\text{disjoint}) \leftarrow \text{contained}(\text{red}, \text{green}), \text{disjoint}(\text{green}, \text{blue})\\
&\text{consistent_conclusions}(\text{contained}, \text{contains}, \text{intersect})\\
&\qquad\qquad \leftarrow \text{contains}(\text{red}, \text{green}), \text{contained}(\text{green}, \text{blue})\\
&\qquad\qquad\quad \vdots\\
&\text{consistent_conclusions}(\text{contained}, \text{contains}, \text{intersect}, \text{disjoint})\\
&\qquad\qquad \leftarrow \text{disjoint}(\text{red}, \text{green}), \text{disjoint}(\text{green}, \text{blue})
\end{aligned}
\tag{4.46}
$$

In this manner, the full set of labels indicates exactly which latent concepts are true. Note that there are fewer rules with the same head as well. We denote the model using this Logic Program Layer as "Logic Program Layer #2" in Table 4.6 where we compare its performance to the other models. The results indicate the rewritten rules significantly outperform the previous Logic Program Layer model, bringing it comparable accuracy to the embedding model. Table 4.7 confirms that latent concepts are indeed learned by the CNN.

The syllogism example demonstrates that the format of the logic program and the labels is significant for reasoning in neural-symbolic contexts. This adds to the challenge of designing a logic program that allows for efficient backpropagation. Note that by replacing the four target

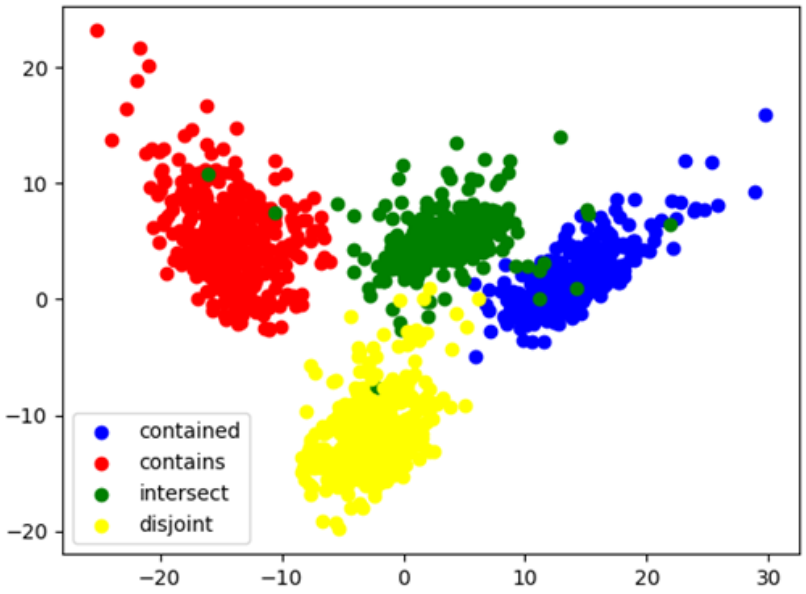


Figure 4.4: PCA visualisation of the latent space of the Embedding model. Four clusters appear to form, corresponding to the four latent concept values.

concepts with a single one, we also replaced the target concept value set from a fixed size (two values per target concept) to almost the power set of all possible red-blue relations. In general, we may not be able to avoid an exponential explosion of the size of the target concept value set. Hence, while this approach worked well in this simple case, it may not scale up well.

As a final note on this experiment, it is noteworthy that the Embedding model was able to perform so well on this task, despite no explicit knowledge given or built into its structure. The model is not interpretable, however, in an effort to understand it better we investigated its latent space post-training. This latent space is the space of the embedding vector produced by the perception function (the CNN). We applied the trained CNN on all images in the dataset to build a collection of latent space vectors. In our notation, this will be the set $E = \{\boldsymbol{z}_i | \boldsymbol{z}_i = f(\boldsymbol{v_i}), (x_i, y_i) \in \mathcal{D}\}$. To try to gain insight into what the CNN learned, we visualised E using PCA (Figure 4.4).

We can see that images corresponding to the same latent concept value (contained, contains, intersect, disjoint) are mapped to points closer to one another. In fact, four clusters appear to form. This indicates the CNN has learned to recognise these concepts in the images, despite no direct supervision and without the ground truth rules. This partially explains its ability to perform well on the downstream task. Of course, even with the CNN recognising the latent concept, we are still left with an uninterpretable reasoning network and no ability to generalise to other downstream tasks. In the next chapter, we build on the insight gained from this investigation and suggest a method to overcome these limitations of the Embedding model. For now, we'll turn to the question of how one could use the matrix-vector encoding to learn logic programs.

4.3 Learning Logic Program Layers

Up until now, we assumed the reasoning function can be represented as a logic program known in advance and engineered by a human. In many scenarios, the background knowledge is in-

complete, in which case we would like to learn the logic program from data. In this section, we detail our initial approach towards this goal. We tackle the problem of gradient-based matrix learning of *purely symbolic* tasks involving a single T_P transition. Our starting point is an existing symbolic learning framework, known as "Learning from Interpretation Transition" (Inoue et al., 2014). We adapt this into a new framework to support learning in continuous space, naming it "Learning from Interpretation Vector Transition". We show how the algorithm works in practice for several toy tasks as well as a simple classification task. Finally, we discuss the current limitations of the approach and how they may be lifted as well as other beneficial extensions.

4.3.1 Learning Framework

Learning from Interpretation Transition

A Learning from Interpretation Transition (LFIT) task as defined in Inoue et al. (2014), involves finding a normal logic program H over a fixed Herbrand base B_H. In this learning task, an example is a pair of interpretations (I, J). A program H satisfies an example (I, J) if $J = T_H(I)$. A solution is a program H that satisfies all examples in the dataset.

The task as defined above has a trivial solution: start with an empty program, and for every pair (I, J) and every $h \in J$ add to H the following clause:

$$h \leftarrow \bigwedge_{b \in I} b \wedge \bigwedge_{c \notin I} \text{not } c \tag{4.47}$$

This is equivalent to maintaining a lookup table for every example. But this solution obviously overfits. In Inoue et al. (2014) this issue is not formally addressed but the authors apply certain generalisation steps during training that result in a reasonable hypothesis.

In ILP literature, overfitting is typically avoided by defining an *optimal* solution to the task that involves a certain optimality criterion over the size of the program. The general consensus is that smaller programs that fit the data well should result in better generalisation. Let R be the number of rules in H and $|body(r)|$ be the length of a rule $r \in H$. Then the size of H is defined as:

$$|H| = R + \sum_{r \in H} |body(r)| \tag{4.48}$$

A straightforward but problematic way to define an optimal solution is a program H that satisfies all examples and for which $|H|$ is minimal. This definition only works well if the dataset contains no noise, such as mislabelled examples. To address this, most ILP systems define a loss function that balances data fit with program size.

Learning from Interpretation Vector Transition

We can adapt an LFIT task to the task of learning the matrix-vector representation of a single Logic Program Layer. In our case, examples are a pair of interpretation vectors $\left(\boldsymbol{v}^{I^{\text{in}}}, \boldsymbol{v}^{I^{\text{out}}}\right)$ instead of straight-up interpretations. $\boldsymbol{v}^{I^{\text{in}}}$ would represent the truth values of the atoms inbound Herbrand base and $\boldsymbol{v}^{I^{\text{out}}}$ would correspondingly represent the truth values for the outbound Herbrand base.

Definition 4.3.1. *(LFIVT task) A Learning from Interpretation Vector Transition (LFIVT) task is a triplet* (B_H^{in}, B_H^{out}, E) *where:*

- B_H^{in} *is the Inbound Herbrand Base.*
- B_H^{out} *is the outbound Herbrand Base.*
- E *is a set of vector pairs* $\left(\boldsymbol{v}^{I^{in}}, \boldsymbol{v}^{I^{out}}\right)$ *such that* $(\boldsymbol{v}^{I^{in}} \in \{0,1\}^{|B_H^{in}|}$ *and* $\boldsymbol{v}^{I^{out}} \in \{0,1\}^{|B_H^{out}|}$. *These pairs are known as examples.*

Definition 4.3.2. *(Satisfying an LFIVT example) A normal logic program H satisfies an example $\left(\boldsymbol{v}^I, \boldsymbol{v}^J\right)$ if $J = T_H(I)$.*

Note that it is possible that $B_H^{in} = B_H^{out}$ or $B_H^{in} \neq B_H^{out}$. In this sense, LFIVT is a generalisation of LFIT. As before we can define an optimal solution to the task by balancing program size and data fit.

As in the case of reasoning, we are going to assume H contains no facts. This is again mostly a matter of convenience, since the kind of programs we are interested in do not usually contain facts. An extension of the framework to allow facts is simple and will be addressed in the discussion section.

4.3.2 Model & Loss Function

Learning a Logic Program Layer involves learning a matrix $H \in \mathbb{R}^{M \times N}$ that represents a program with M rules and an Inbound Herbrand Base of size N. This matrix is equivalent to D^P from the reasoning case. Note that the bias vector $\boldsymbol{b}$ can be easily computed from D^P: simply take the sum of the negative entries in D^P and negating them:

$$b_i = \frac{m^{r^i}}{n^{r^i} + m^{r^i}} = \sum_{\substack{j=1 \\ D_{ij}^P < 0}}^{N} -D_{ij}^P \tag{4.49}$$

for $1 \leq i \leq N$.

Example 4.3.1. *For the program P:*

$$\begin{aligned} p &\leftarrow a, not\ b, not\ c \\ q &\leftarrow b, c, not\ a \end{aligned} \tag{4.50}$$

we have the following matrix:

$$D^P = \begin{array}{c} \\ p^1 \\ q^1 \end{array} \begin{array}{c} \begin{array}{ccc} a & b & c \end{array} \\ \begin{bmatrix} \frac{1}{3} & -\frac{1}{3} & -\frac{1}{3} \\ -\frac{1}{3} & \frac{1}{3} & \frac{1}{3} \end{bmatrix} \end{array} \tag{4.51}$$

and the bias vector is:

$$\boldsymbol{b}^{P_i} = \begin{array}{c} p^1 \\ q^1 \end{array} \begin{bmatrix} \frac{2}{3} \\ \frac{1}{3} \end{bmatrix} \tag{4.52}$$

Because of this, once H is known, we compute the bias vector $\boldsymbol{b}$ in the same manner:

$$b_i = \sum_{\substack{j=1 \\ H_{ij} < 0}}^{N} -H_{ij} \tag{4.53}$$

Therefore, H is the only parameter to be learned. Each row in H corresponds to a single rule. In order to translate the row back to a symbolic rule, we need to decide on a mapping of each row to an atom in the outbound Herbrand Base, ahead of learning. We assume each head atom $h \in B_H^{out}$ maps to a fixed number of rows R. So if we have $B_H^{out} = \{h_1, h_2, h_3, ...\}$, then the first R rows correspond to h_1, the next R rows correspond to h_2, and so on. Hence $H \in \mathbb{R}^{\left(R \cdot \left|B_H^{out}\right|\right) \times \left|B_H^{in}\right|}$ and $\boldsymbol{b} \in \mathbb{R}^{R \cdot \left|B_H^{out}\right|}$.

For simplicity of notation, we denote the application of a logic program layer on a vector $\boldsymbol{u}$ as:

$$LPL_H(\boldsymbol{u}) \stackrel{\text{def}}{=\!=} \sum_{h \in B_P^{out}} \boldsymbol{v}^h \left[\max_{r \in rules(h)} \left(\boldsymbol{v}^{h^r}\right)^\intercal \sigma_{\gamma,\tau} \left(H\boldsymbol{u} + \boldsymbol{b}\right) \right] \tag{4.54}$$

(this is a differentiable version of Equation 4.21).

An example $(\boldsymbol{v}^I, \boldsymbol{v}^J)$ is satisfied when:

$$\boldsymbol{v}^J \sim_{\gamma_\perp}^{\gamma^\top} LPL_H(\boldsymbol{v}^I) \tag{4.55}$$

To construct the loss function, we can follow the same line of derivation as binary cross-entropy loss. Note that the truth value of an atom $h \in B_H^{out}$ in an interpretation vector $\boldsymbol{v}^I$ is given by $\left(\boldsymbol{v}^h\right)^\intercal \boldsymbol{v}^I$. Therefore, for an example $\left(\boldsymbol{v}^I, \boldsymbol{v}^J\right)$, the value $\left(\boldsymbol{v}^h\right)^\intercal \boldsymbol{v}^J$ is the ground truth value of h in the outbound interpretation vector. We can therefore construct the following loss function:

$$L_{(\boldsymbol{v}^I, \boldsymbol{v}^J)}(H) = \prod_{h \in B_H^{out}} \left(\left(\boldsymbol{v}^h\right)^\intercal LPL_H\left(\boldsymbol{v}^I\right) \right)^{\left(\boldsymbol{v}^h\right)^\intercal \boldsymbol{v}^J} \cdot \left(1 - \left(\boldsymbol{v}^h\right)^\intercal LPL_H\left(\boldsymbol{v}^I\right) \right)^{1 - \left(\boldsymbol{v}^h\right)^\intercal \boldsymbol{v}^J} \tag{4.56}$$

Taking the negative log we get:

$$\begin{aligned} &L_{(\boldsymbol{v}^I, \boldsymbol{v}^J)}(H) = \\ &= \sum_{h \in B_H^{out}} -\left(\boldsymbol{v}^h\right)^\intercal \boldsymbol{v}^J \log\left[\left(\boldsymbol{v}^h\right)^\intercal LPL_H\left(\boldsymbol{v}^I\right) \right] - \left(1 - \left(\boldsymbol{v}^h\right)^\intercal \boldsymbol{v}^J\right) \log\left[1 - \left(\boldsymbol{v}^h\right)^\intercal LPL_H\left(\boldsymbol{v}^I\right) \right] \end{aligned} \tag{4.57}$$

This defines the loss for a single example. We can now define the data fit loss term by summing over the loss of all examples:

$$L_{\text{examples}}(H, E) = \sum_{(\boldsymbol{u}, \boldsymbol{v}) \in E} L_{(\boldsymbol{u}, \boldsymbol{v})}(H) \tag{4.58}$$

Next, we need to define a loss term that penalises large programs. This can be thought of as a regularisation term for H. It will both encourage H to be sparse and result in small values. One could apply a standard l^2-norm, such as the sum of squared values. This will not result in the shortest rules possible, however. For example, the rule:

$$p \leftarrow a, b \tag{4.59}$$

is longer than the rule:

$$p \leftarrow a \tag{4.60}$$

and one can see that the latter is "more general", since it logically implies the former. But in our standard matrix representation the first rule would result in two values of $\frac{1}{2}$ in the same row, whose squared sum is $\frac{1}{2}$. The second rule results in a row with a single 1 whose squared

sum is 1. Therefore, according to the sum of squares norm, the first rule is preferred, resulting in longer rules. To encourage short rules, we can instead take the sum of *square roots*. In this case, the first rule would give a penalty of $\frac{1}{\sqrt{2}} + \frac{1}{\sqrt{2}} = \sqrt{2}$, while the second rule has a penalty of 1. Therefore, the second rule is preferable, as expected. Formally:

$$L_{\text{size}}(H) = \sum_{i,j} \sqrt{|H_{ij}|} \tag{4.61}$$

with the loss function (so far) being:

$$L(H, E) = L_{\text{examples}}(H, E) + \mu_1 L_{\text{size}}(H) \tag{4.62}$$

where μ_1 is a hyperparameter to control the balance of data fit and generalisation.

Rule packing

Consider the following program:

$$\begin{aligned} h &\leftarrow a, b \\ h &\leftarrow b, c \end{aligned} \tag{4.63}$$

The program should be represented by the following matrix:

$$\begin{array}{c} \\ h \\ h \end{array} \begin{array}{c} \begin{array}{ccc} a & b & c \end{array} \\ \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} \end{bmatrix} \end{array} \tag{4.64}$$

but the way we defined our loss function so far, the following matrix works just as well:

$$\begin{array}{c} \\ h \\ h \end{array} \begin{array}{c} \begin{array}{ccc} a & b & c \end{array} \\ \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 \end{bmatrix} \end{array} \tag{4.65}$$

since all interpretation transitions for program 4.63 will be computed correctly by this matrix. Essentially, the two rules were "packed" into the same row, resulting in an ambiguous representation that nonetheless performs the T_P transition for program 4.63 correctly. This had not been an issue when we were doing reasoning, since we controlled the matrix representation. But when learning, this matrix could become a viable option for the optimisation algorithm.

We would like the learning algorithm not to produce such matrices. One approach is to add a loss term that requires each row to sum up to one particular value, such as 1, or the average of 1 and γ. When rules are packed together in the same row, the row values tend to have a larger sum. This loss term can therefore be beneficial:

$$L_{\text{packing}}(H) = \sum_{i=1}^{K} \left| \frac{1+\gamma}{2} - \sum_{j=1}^{N} H_{ij} \right| \tag{4.66}$$

The total loss function is then given by:

$$L(H, E) = L_{\text{examples}}(H, E) + \mu_1 L_{\text{size}}(H) + \mu_2 L_{\text{packing}}(H) \tag{4.67}$$

with μ_1 and μ_2 being hyperparameters.

Logic Program Layer compared to Fully Connected Layer A Logic Program Layer is quite similar to a Fully Connected Layer, which is one of its advantages. Due to its similarity to these layer it integrates naturally in any neural architecture. However, a Logic Program Layer is also distinct from a Fully Connected Layer in several ways:

1. Its weights are sparse rather than dense. Even when trained, it is encouraged to be as sparse as possible.
2. Its weights are derived from exact symbolic rules rather than statistical approximations. Note how the weights must sum up to 1.
3. Its activation also involves a max operation to approximate sound deduction.

Hence the Logic Program Layer acts as an approximation of the Immediate Consequence operator rather than learning to compute an arbitrary statistical function.

4.3.3 Evaluation

Here we present an initial evaluation of the framework to assess its feasibility. We begin with several toy tasks to give an idea of how the framework works in practice. We do this in order to demonstrate the program matrices our method learns and how they translate to a symbolic hypothesis. We also evaluated the method on the Zoo dataset, which is a classification task over structured data designed as a simple benchmark for machine learning models.

Toy Tasks

The following table contains a description of a simple toy task. It also lists a hypothesis H converged to when applying the method above for program learning.

B_P^{in}	B_P^{out}	Examples	Optimal Hypothesis	Discovered Hypothesis
$\{p, q\}$	$\{p, q\}$	$(\{p\}, \{q\})$	$q \leftarrow p$	$q \leftarrow p = 1.000$

The hypothesis in the rightmost column is just a more readable description of the following matrix:

$$H = \begin{array}{c} \\ p \\ q \end{array} \begin{array}{c} \begin{array}{cc} p & q \end{array} \\ \begin{bmatrix} 0 & 0 \\ 1.000 & 0 \end{bmatrix} \end{array} \tag{4.68}$$

where the rows are labelled with the head of the rule they represent. Convergence was achieved after 2500 iterations. While this task is very simple, it is useful as a sanity check - the matrix we get is indeed the matrix representing the program $q \leftarrow p$ and the rule is easily extracted from it. Sometimes tasks have multiple possible solutions. For example:

B_P^{in}	B_P^{out}	Examples	Optimal Hypothesis	Discovered Hypothesis
$\{p, q, r\}$	$\{p, q, r\}$	$(\{p, q\}, \{r\})$	1. $r \leftarrow p$ 2. $r \leftarrow q$	1. $r \leftarrow p = 1.000$ 2. $r \leftarrow q = 1.000$

where either $r \leftarrow p$ or $r \leftarrow q$ are acceptable hypotheses, and different initialisation of the parameter matrix H at the beginning of training will result in a different hypothesis at the end

of training. Note that in this task the example given does not suggest a minimal rule. For instance, the original LFIT algorithm given by Inoue et al. (2014) would translate $(\{p, q\}, \{r\})$ to the rule $r \leftarrow p, q$ and would not perform any further generalisation. The additional size term in our loss function, however, results in minimality. The next example shows that programs with multiple rules can be learned.

B_P^{in}	B_P^{out}	Examples	Optimal Hypothesis	Discovered Hypothesis
$\{p, q, r\}$	$\{p, q, r\}$	$(\{p\}, \{r\})$	$r \leftarrow p$	$r \leftarrow q = 1.000$
		$(\{q\}, \{r\})$	$r \leftarrow q$	$r \leftarrow p = 1.000$

In this case a single rule such as $r \leftarrow p$ does not cover both examples, so it is not a minimal solution. The solution found indeed captures both rules as expected. Rules with longer bodies can also be captured:

B_P^{in}	B_P^{out}	Examples	Optimal Hypothesis	Discovered Hypothesis
$\{p, q, r, s\}$	$\{p, q, r, s\}$	$(\{p, q\}, \{s\})$	$s \leftarrow p, q$	$s \leftarrow p = 0.3324, q = 0.3324$
		$(\{r\}, \{s\})$	$s \leftarrow r$	$s \leftarrow r = 1.0000$
		$(\{p\}, \{\})$		
		$(\{q\}, \{\})$		

Here we see a rule with two body atoms was learned. When we were given the rules, a conjunction with two body atoms was represented by an entry of $\frac{1}{2}$ in the corresponding columns. Here we get instead a value close to $\frac{1}{3}$. This is simply because the two values when summed up are greater than γ, which we set to $\frac{1}{2}$. So together they are sufficient to make the head true. The symbolic rule $s \leftarrow p, q$ can still be extracted correctly as long as one looks at a combination of entries that sum up to more than γ. Note that in this task, the last two examples are necessary to ensure $s \leftarrow p, q$ is minimal. Otherwise, a rule such as $s \leftarrow p$ would suffice. The next example shows rule packing, while mitigated by the L_{packing} term, can still occur:

B_P^{in}	B_P^{out}	Examples	Optimal Hypotheses	Discovered Hypotheses
$\{p, q, r, s, t\}$	$\{p, q, r, s, t\}$	$(\{p, q\}, \{r\})$	1. $r \leftarrow p$	1. $r \leftarrow p = 1.00, t = 1.00$
		$(\{s, t\}, \{r\})$	$r \leftarrow t$	2. $r \leftarrow q = 1.000$
			and 3 other possibilities	$r \leftarrow s = 1.000$

Here we see two different hypotheses discovered by the learning algorithm. The first one packs two rules into the same line in the matrix. Namely, the rules $r \leftarrow p$ and $r \leftarrow t$ are both captured in the same line. At other times the rules are correctly captured in two different rows, as can be seen in the second hypothesis. This highlights that more research into preventing rule packing is required. Finally, the learning algorithm can also learn normal rules. The last toy task we show is taken from Inoue et al. (2014).

B_P^{in}	B_P^{out}	Examples	Optimal Hypothesis	Discovered Hypothesis
$\{p, q, r\}$	$\{p, q, r\}$	$(\{q, r\}, \{p, r\})$	$p \leftarrow q$	$p \leftarrow q = 1.0000$
		$(\{p, r\}, \{q\})$	$q \leftarrow p, r$	$q \leftarrow p = 0.4951, r = 0.4951$
		$(\{q\}, \{p, r\})$	$r \leftarrow \text{not } p$	$r \leftarrow p = -0.8676$
		$(\{p, q, r\}, \{p, q\})$		
		$(\{p, q\}, \{p\})$		
		$(\{p\}, \{\})$		
		$(\{\}, \{r\})$		
		$(\{r\}, \{r\})$		

We can see the matrix has a negative entry for the third rule, representing that the body atom should be negated, i.e. $r \leftarrow \text{not } p$.

Zoo Task

The zoo dataset (Dua and Graff, 2017) is a simple benchmark for testing machine learning algorithms. In this task, each data point is an animal taken from a particular zoo. The animals are divided into 7 categories: mammal, bird, fish, reptile, bug, amphibian, and invertebrate. Each sample has 16 features relating to the animals such as the presence of feathers, the number of legs, or if it's aquatic among others. The purpose of the task is to identify the category of the animal from the features. All features are binary except "legs" which is multi-valued. This makes the task ideal for symbolic learning.

We evaluated our matrix learning approach on this dataset and compared the results to the exact symbolic solver FastLAS as well as a deep neural network. The training set is comprised of 144 samples and the test of 70 samples. In one of the runs, our approach converged to the following hypothesis after 10,000 iterations:

$$\begin{aligned}
\text{class}\,(\text{amphibian}) \leftarrow &\text{aquatic}\,(1) = 0.1930,\ \text{toothed}\,(1) = 0.0155,\ \text{legs}\,(4) = 0.1864,\\
&\text{catsize}\,(0) = 0.1253\\
\text{class}\,(\text{bird}) \leftarrow &\text{feathers}\,(1) = 1.0000\\
\text{class}\,(\text{bug}) \leftarrow &\text{aquatic}\,(0) = 0.1412,\ \text{legs}\,(6) = 0.5623\\
\text{class}\,(\text{fish}) \leftarrow &\text{breathes}\,(0) = 0.3151,\ \text{fins}\,(1) = 0.4240\\
\text{class}\,(\text{invertebrate}) \leftarrow &\text{backbone}\,(0) = 0.3652,\ \text{breathes}\,(0) = 0.1780,\ \text{airborne}\,(0) = 0.1019\\
\text{class}\,(\text{mammal}) \leftarrow &\text{milk}\,(1) = 1.0000\\
\text{class}\,(\text{reptile}) \leftarrow &\text{hair}\,(0) = 0.1282,\ \text{milk}\,(0) = 0.0583,\ \text{toothed}\,(1) = 0.0345,\\
&\text{aquatic}\,(0) = 0.0323, \text{venomous}\,(1) = 0.1384,\ \text{legs}\,(4) = 0.1455,\\
&\text{tail}\,(1) = 0.1090
\end{aligned} \tag{4.69}$$

where, for instance, $tail(1)$ indicates the presence of a tail. A quick inspection shows that the rules seem reasonable. For instance, the rule for fish is an animal that has fins but does not require air to breathe. Hence it even covers the more exceptional cases such as dolphins. For

comparison, FastLAS (Law et al., 2021), an exact solver, recovers the following solution:

$$\begin{aligned}
\text{class}\,(\text{mammal}) &\leftarrow \text{milk}\,(1)\\
\text{class}\,(\text{fish}) &\leftarrow \text{fins}\,(1)\,,\ \text{breathes}\,(0)\\
\text{class}\,(\text{bird}) &\leftarrow \text{feathers}\,(1)\\
\text{class}\,(\text{invertebrate}) &\leftarrow \text{legs}\,(0)\,,\ \text{backbone}\,(0)\\
\text{class}\,(\text{invertebrate}) &\leftarrow \text{aquatic}\,(1)\,,\ \text{backbone}\,(0)\\
\text{class}\,(\text{bug}) &\leftarrow \text{aquatic}\,(0)\,,\ \text{legs}\,(6)\\
\text{class}\,(\text{amphibian}) &\leftarrow \text{tail}\,(0)\,,\ \text{backbone}\,(1)\,,\ \text{eggs}\,(1)\\
\text{class}\,(\text{reptile}) &\leftarrow \text{fins}\,(0)\,,\ \text{feathers}\,(0)\,,\ \text{tail}\,(1)\,,\ \text{hair}\,(0)
\end{aligned} \tag{4.70}$$

We can see the rules are often quite similar, with a few differences. These are expected since there are multiple optimal hypotheses for the task, hence different runs could recover different rules. The rules recovered by FastLAS tend to be shorter, hence more optimal with respect to size. This is due to the exact nature of FastLAS as opposed to the approximation produced by a gradient-based optimisation process.

To evaluate how well the learned program matrix H performs on the test dataset, we must define some metrics. Given a test example (I^{in}, I^{out}), suppose $\boldsymbol{u}$ is a vector representing I^{in} and let $\boldsymbol{v}$ be the prediction produced by the program matrix, i.e.

$$\boldsymbol{v} = LPL_H\,(\boldsymbol{u}) \tag{4.71}$$

We consider three different metrics for evaluating the correctness of the prediction $\boldsymbol{v}$:

1. Atom-Based Accuracy: The average number of atoms that $\boldsymbol{v}$ and I^{out} agree on. Intuitively, if $v_i > \gamma$, this represents that the atom p_i (which corresponds to the i^{th} entry of $\boldsymbol{v}$) is predicted to be true. If it is true in I^{out}, then the prediction is correct. Similarly, if $v_i < \gamma$ then p_i is predicted to be false. We take the average number of correct predictions to be the atom-based accuracy for this example. Formally, $Acc_{atom} = \frac{1}{|I^{out}|}\sum_{i=1}^{|I^{out}|}[v_i > \gamma \wedge p_i \in I^{out}]$.

2. Interpretation-Based Accuracy: Equal to 1 if $\boldsymbol{v}$ represents the interpretation I^{out}, otherwise 0. In this context, representing means $\boldsymbol{v}$ agrees with I^{out} on every atom. Therefore interpretation-based accuracy is always lower than atom-based accuracy. However, atom-based accuracy could be skewed high by a large number of 0's in I^{out}.

3. Max-Based accuracy: 1 if the argmax of $\boldsymbol{v}$ is the i'th entry and $p_i \in I^{out}$. This is most suitable for classification tasks, and is essentially how neural networks are evaluated. It is applicable here as well since $\boldsymbol{v}$ is a vector of real truth values, and a higher value means an atom has more "support".

We use these three metrics to evaluate our approach. The hypothesis recovered by FastLAS can be evaluated on an example using a symbolic solver. We take the hypothesis as background knowledge and add the features as facts. The solver would then output the predicted class. We also trained a feedforward neural network with two hidden layers (sizes 7 and 21) and softmax outputs, which is evaluated in the standard manner. Results (averaged over 5 different runs) appear in table 4.8. Our algorithm's Max-based accuracy, which is the most suitable for a

Atom-Based Accuracy	Interpretation-Based Accuracy	Max-Based Accuracy	FastLAS	Neural Network
96.33%	80.00%	91.43%	88.57%	90.00%

Table 4.8: Accuracy in prediction for the Zoo task

classification task such as this, shows our model performs well compared to FastLAS and Deep Learning. These are promising results considering the preliminary nature of this method.

4.3.4 Discussion

The framework of Learning from Interpretation Vector Transition (LFIVT) is promising as a first step in learning logic programs in a differentiable manner. As seen, it naturally captures classification tasks, allowing for an interpretable hypothesis to be extracted at the end of training. In this chapter we only explored learning from structured data under this framework, but our ultimate goal is to learn from fuzzy data. One can imagine chaining a standard neural network to a logic program layer which learns the hypothesis matrix H. For example, chaining a CNN that classifies MNIST digits to a logic program layer that learns the rules of addition. However, in practice, it is difficult to learn in an end-to-end manner. At the beginning of training the hypothesis matrix would not represent useful rules and therefore the information backpropagated to the CNN would not be meaningful. One way to tackle this would be to partially pretrain the CNN, which is only possible when you have access to data labelled with the latent concepts. Another method to try is to learn iteratively: First, hold a random matrix H fixed and train the CNN. Then, hold the CNN fixed and train H. Then again fix H and fine-tune the CNN, and so on. Hopefully in this manner training the CNN and H would become easier since each iteration requires optimising fewer variables (a similar idea was explored in Cunnington et al. (2023b), although not using a matrix form). Still, it is not clear if such an approach can successfully converge to a desirable solution.

As noted in some of the examples, rule-packing is still a standing issue. This is mostly an issue of interpretability - when we extract the rules at the end of training, we have no way of knowing which atom belongs to which rule. One could perhaps try to "unpack" a packed rule by looking at all combinations of the body atoms that sum up to more than γ, and evaluating them on a given validation set. We hypothesise that the unpacked rules would be the minimal among these that also perform well.

While simple classification tasks are captured well by LFIVT, more complicated tasks would require a more sophisticated framework. For instance, suppose we'd like to learn the rules for what makes numbers even or odd. A number is even if it is the successor of an odd number, and vice versa. Hence, we need to learn recursive rules, or at least rules that require multiple steps of reasoning. But LFIVT only looks at a single step. One can imagine chaining multiple Logic Program Layers to represent multi-step reasoning and optimising their parameters all together. But this is still a finite reasoning process and would capture the rules for small examples only. For fully recursive rules, a recursive prediction process such as the one presented in Chapter 3 may be necessary. This would require changing the process of computing models to be entirely differentiable, while at the moment we depend on a loop in the Newton algorithm.

Many learning tasks in ILP also include a given background knowledge. This background knowledge is itself a logic program B, and the hypothesis H is to be added on top of it. For instance, in learning from entailment, a solution H is a logic program such that $B \cup H$ logically

entails every positive example, and does not entail every negative example. We can modify our framework to support background knowledge rules by representing those using additional fixed rows in the hypothesis matrix H.

4.4 Summary

In this chapter, we detailed an initial approach to neuro-symbolic reasoning and learning that we developed, based on the concept of embedding hierarchical logic programs in matrix-vector form. The approach allowed the seamless integration of logic programs that support negation-as-failure into neural network architectures while supporting backpropagation. This allows the end-to-end training of perception networks from downstream labels, as well as learning the program matrix itself, supporting a form of differentiable inductive logic programming.

Initial experiments in the neuro-symbolic reasoning setting showed our approach scales better than competing approaches. In the MNIST addition task, we were able to train the perception network orders of magnitude faster than the baseline approaches for 1 to 3 digits. However, scaling up further may prove challenging as the number of rules grows exponentially. Additionally, the Syllogism experiment shows that the structure of the program can prevent learning, due to the abductive nature of training, which in general is NP-hard.

The restriction of logic programs to a hierarchical structure is, as we mentioned, not severe, but it does prevent us from representing certain relations such as:

$$\text{ancestor}\left(X, Y\right) \leftarrow \text{parent}\left(X, Z\right), \text{ancestor}\left(Z, Y\right) \tag{4.72}$$

In principle, it's likely logic program layers can be generalised to deal with recursive relations using recurrent neural connections. However, RNNs often suffer from vanishing gradients, making them more difficult to train.

The approach presented here for Neuro-Symbolic Learning showed potential in initial experimentation, but suffers from several drawbacks. Coupled with the drawbacks in reasoning with Logic Program Layers, it encourages us to seek a new approach to neuro-symbolic reasoning and learning. In the next chapter, we introduce a method to facilitate neuro-symbolic reasoning, based on previous insights from section 4.2, which does not suffer from the same drawbacks, is not restricted to hierarchical programs only and can scale to far more complex tasks. We will return to neuro-symbolic learning in chapter 6.

Chapter 5

Embed2Sym - Scalable Neuro-Symbolic Reasoning via Clustered Embeddings

Up until now, our approach to neuro-symbolic reasoning and learning involved encoding logic programs in vector spaces, allowing differentiable reasoning and integration as neural network layers. While we have seen some scalability improvement compared to existing systems, it is still limited as the number of ground rules may be too large to feasibly represent as a matrix, or alternatively, the number of possible labels could grow exponentially, preventing efficient training.

Broadly speaking, our Logic Program Layer approach belongs to a family of neuro-symbolic methods that attempt to replace symbolic operations with continuous approximations (while our approach maintains guarantees of exact inference over some values of the continuous space. Namely, when the neural network is relatively confident in its predictions, giving probabilities higher than $\gamma^{\top}$ or lower than $\gamma^{\perp}$). Other examples of this approach include Logic Tensor Networks (Badreddine et al., 2022a), Neural Theorem Prover (Minervini et al., 2018) and Logical Neural Networks (Riegel et al., 2020). While these approaches are generally more scalable, they often come with limited representation power. For example, recursion is quite difficult to represent with this family of neuro-symbolic approaches.

A different approach that has recently gained attention involves extending the capabilities of symbolic solvers by (1) allowing the system to depend on the output of a neural component, and (2) using inconsistent results arising from the neural output, reasoning process and data to guide the training of the network. These systems see neuro-symbolic reasoning on raw data as a combination of two tasks: a perception task, handled by a neural component, and a reasoning downstream task, handled by the symbolic solver. Prominent examples include DeepProbLog (Manhaeve et al., 2021a) and NeurASP (Shakarian et al., 2023). These systems possess the significant advantage of maintaining the full representation capabilities of the symbolic solver at their core. However, experiments have shown that their training times can be significantly long even when the perception task is simple (Tsamoura et al., 2021). The difficulty in training is inherent in current approaches: in order to train, the symbolic component must "work backwards" from a given target label to a set of possible latent labels, which can be exponential in size. This has so far hindered most neuro-symbolic reasoning systems from being applied to tasks involving a larger number of inputs and more complex perception tasks are usually not

considered, with a few exceptions, e.g. Huang et al. (2021b).

In contrast, approaches such as Concept Bottleneck models (Koh et al., 2020) have demonstrated that performing the reasoning step with a second neural network can achieve competitive accuracy when latent concepts are effectively integrated into the neural architecture, without loss of scalability. But they require latent bottleneck concepts to be labelled to preserve interpretability and generalisation. They are also not typically applied in a neuro-symbolic setting where multiple raw inputs must be considered together.

Motivated by this and the clustering phenomenon we witnessed while training the Embedding Model in section 4.2.3, we introduce a novel approach to neuro-symbolic reasoning systems. This new approach, which we call Embed2Sym, is scalable, can produce neuro-symbolic models that are interpretable and generalise beyond the training task, and does not require latent concepts to be labelled. Embed2Sym works in three steps. First, a neural network composed of a perception component and a reasoning component is trained on the downstream task end-to-end. The architecture is similar to the Embedding Model used in Section 4.2.3 for diagrammatic reasoning (Wang et al., 2018b). Here we consider its wider application within the context of a general neuro-symbolic approach. By training the perception component in this manner, rather than through evaluation of a logical formula, we avoid the combinatorial difficulty that other neuro-symbolic reasoning systems confront. Second, we search for latent concepts that have been learned by the perception component by analysing the clustered structure of datapoints in embedding space. Third, we identify the discovered latent concepts and assign them their symbolic meaning by means of an optimisation task solved using a symbolic system (Clingo, Gebser et al. (2017)) and background knowledge of the task. In this way, we are able to lift a purely neural model into a neuro-symbolic one that is interpretable and can generalise to tasks the neural network alone could not perform.

We evaluate Embed2Sym on several tasks that have been previously introduced to benchmark neuro-symbolic systems. Results confirm that Embed2Sym achieves similar or better accuracy to competing systems while training faster by orders of magnitude. The scalability of Embed2Sym allows it to train on tasks involving at least dozens of raw inputs, such as adding two 15-digit numbers given as MNIST images, which other systems find infeasible. We also confirm that Embed2Sym can scale to tasks involving more challenging perception domains such as the CIFAR-10 dataset.

The chapter is structured as follows: we begin by covering the various aspects of the Embed2Sym algorithm for neuro-symbolic training and inference. First, we cover the pre-training stage of a Fully Neural model. Next, we discuss the process of identifying and labelling clusters in the latent space. We then show how these two stages allow us to perform neuro-symbolic reasoning. Following that, we present the results of an evaluation of the system confirming its performance and scalability. We then discuss the results before summarising.

The work described in this chapter has previously been published in Aspis et al. (2022).

5.1 Fully Neural Model

Recall that we wish to learn functions of the form $h = g \circ f$, where f is a set of perception functions $f_i : \mathcal{X}_j \to \mathcal{Z}_i$ and g is a reasoning function $g : \bigotimes \mathcal{Z}_i \otimes \bigotimes S_j \to \bigotimes \mathcal{Y}_k$. Up until now, the latent spaces $\mathcal{Z}_i$ were probability simplexes — each point z_i represented a probability distribution for the latent concept values. When the reasoning function g was implemented through vector and matrix operations, we could feed the z_i's directly into it. If instead we would

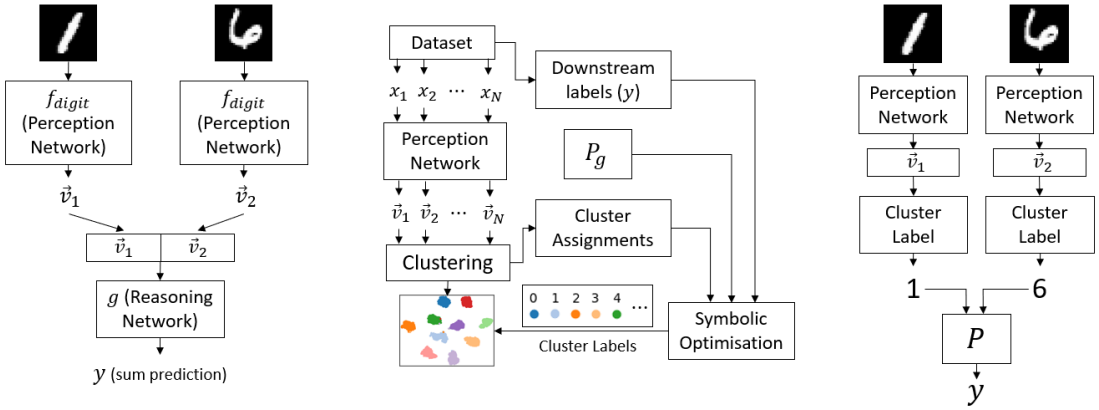


Figure 5.1: Embed2Sym architecture. Left: An example of a fully neural architecture for the MNIST Addition task. Middle: After training the fully neural network, clustering is applied to embedding vectors computed by the perception network. Symbolic optimisation is used to label the clusters together with the rules P_g. Right: An example inference on MNIST digits. The cluster labels are used as symbolic inputs to the program P (which may differ from P_g).

like g to be a symbolic program (with computation done by a symbolic solver), we can employ a symbol grounding function to transform the z_i's into ground atoms.

In Section 4.2.3, we considered an "Embedding model". This is a fully neural model that attempts to solve the end-to-end task, both perception and reasoning. The perception network, a CNN, is applied to each image and outputs an embedding vector, which is not interpretable. These vectors are concatenated before being fed to another network, which predicts the downstream probabilities. In our notation, this second network, which we call a "reasoning network", plays the role of the function g (but unlike our typical g functions, it is not interpretable). The latent spaces $\mathcal{Z}_i$ are simply vector spaces like $\mathbb{R}^K$ (for some dimension K), where the embedding vectors $\boldsymbol{z}_i$ live.

While the Embedding model is not neuro-symbolic, it is trained from downstream labels without labelling of the latent concepts, just as in neuro-symbolic tasks. One might expect the lack of latent labelling and the non-interpretability of the model to limit its usefulness for neuro-symbolic inference. However, as can be seen in Figure 4.4, the perception model learns to cluster images belonging to the same latent concept values together (in that case, the relation between two sets in an Euler diagram). We hypothesise this clustering behaviour generalises across many tasks: as the reasoning network trains, it guides the perception network to correctly recognise latent concepts in the raw data.

In our new approach to neuro-symbolic AI, we make use of this clustering behaviour. In this chapter, we deal exclusively in the neuro-symbolic reasoning paradigm, where the rules of the downstream task are assumed to be completely known in advance. Since these rules are known, training the perception component f becomes the primary goal. We begin by training an "Embedding Model" from the downstream label, which we rename "Fully Neural Model" to distinguish it from the "Neuro-Symbolic Model" which is our ultimate goal. Training the Fully Neural Model is useful in three ways:

1. The perception function and the neural reasoning function can represent data using dense continuous representations, which is exactly the regime where neural and gradient-based approaches excel.

2. As the learned latent space demonstrates clustering behaviour, it is useful for discovering

structure in the data that is relevant to the domain. We can use this to lift the differentiable model into a neuro-symbolic one with greater interpretability and generalisation capabilities.

3. As each training iteration involves only backpropagation, our approach does not suffer from the same scalability issues that other neuro-symbolic approaches encounter. Hence, we can scale very well with the number of raw inputs.

Figure 5.1 (left) illustrates this architecture for the single-digit MNIST Addition task. f_{digit} in this case could be a CNN such as a LeNet (Lecun et al., 1998). Note that typically the final layer of a LeNet is Softmax to produce a probability vector. In our case, since we are interested in dense embeddings, a Softmax will not be ideal: softmax use of exponents tends to cause one argument to dominate the others, sending a sparser signal to the reasoning component. We therefore modify the final layer to produce an embedding vector using the ReLU activation. Our modified CNN is then applied to both images, resulting in two embedding vectors $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$, which are then concatenated before being fed to the reasoning network. For this task, the reasoning network g is simply an MLP with an output layer of size 19, corresponding to the 19 possible sums, activated using softmax. We can then train from the sum labels, backpropagating through the reasoning networks into the CNN.

Consider a general task with n raw inputs $\{x_1, \ldots, x_n\}$, m symbolic inputs $\{s_1, \ldots, s_m\}$, k latent concepts $\{c_1, \ldots, c_k\}$ and t target concepts $\{y_1, \ldots, y_t\}$. Suppose raw input x_i is associated with latent concept c_{j_i} which has the perception function f_{j_i}. Then the fully neural model first applies the perception functions to the raw inputs:

$$f_{j_i}(x_i) = z_{j_i} \in R^{K_j} \tag{5.1}$$

where K_j is the dimensionality of the embedding space for latent concept c_j, chosen in advance as a hyperparameter. The latent embeddings are concatenated together and then to the symbolic inputs (which are embedded as vectors using the one-hot encoding) before being fed to the reasoning network g:

$$g\left(\bigotimes_{j=1}^{k} z_j \otimes \bigotimes_{l=1}^{m} s_l\right) = \bigotimes y_{p=1}^{t} \tag{5.2}$$

Once training of the f_j functions and g is complete, we can proceed to the next stage: identifying the learned clusters in embedding space.

5.2 Identifying Latent Concepts

The fully neural model described above can be trained to solve the task using $\mathcal{D}$. As the various f_i functions are trained, they tend to cluster datapoints corresponding to the same latent concept value together. Recall from Definition 4.1.3 that a latent concept is a tuple $c = (n_c, V_c, \mathcal{Z}_c)$ where V_c is the set of latent concept values. If, for example, we have $c_i = (n_{c_i}, \{v_1, v_2, v_3\}, \mathcal{Z}_i)$, f_i will converge to a function that maps datapoints to three distinct clusters in embedding space. An example of this is shown in Figure 5.2. The plot depicts a t-SNE visualisation (Van der Maaten and Hinton, 2008) of the embedding space of MNIST images after training the fully neural model depicted in Figure 5.1 (left) on the MNIST Addition task. Although this model did not receive any knowledge about the relevant latent concepts for the task, 10 clear clusters emerged, each associated with a specific digit.

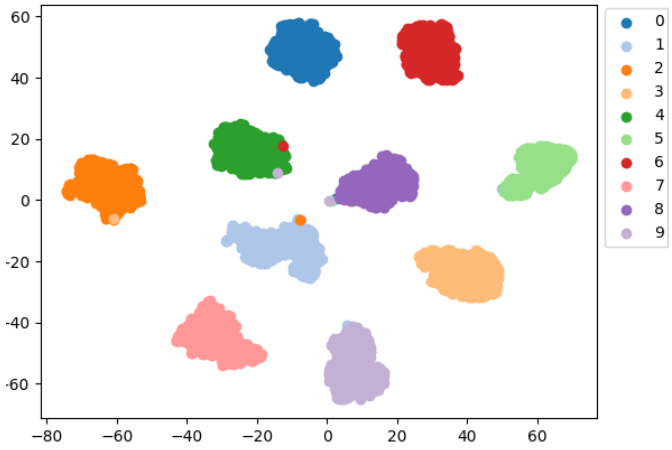


Figure 5.2: t-SNE visualisation of f_{digit} embedding space post-training, clearly showing clusters representing 10 different digits.

This is therefore our second example of a clustering behaviour emerging from fully neural training, the first being the Syllogism dataset, as in figure 4.4. Note that Figure 4.4 was created using PCA visualisation, while Figure 5.2 was created using t-SNE. The latent space can look quite different, depending on which dimensionality reduction method is used. The PCA algorithm attempts to maintain distance information between datapoints, while t-SNE is density-based (points that were close together remain close together). For this reason, clustering information is visualised better when t-SNE is employed. This explains why the clusters are more clearly visible in Figure 5.2 than in Figure 4.4.

This observation that clusters emerge in the latent space is key to lifting the fully neural model into a neuro-symbolic one. To achieve neuro-symbolic reasoning, we need to transform each f_{i_j} function into a classifier for the latent concept c_i. We can exploit the clustering behaviour to do so. After training f_i, we identify the various clusters in embedding space $\mathcal{Z}_i$. Then, we label each cluster with its latent concept value. The process for cluster discovery and labelling is depicted in Figure 5.1 (middle). When we wish to classify the latent concept value associated with a raw input x_j, we compute $f_{i_j}(x_i)$, check which cluster it belongs to and classify x_i with the label of that cluster.

5.2.1 Cluster Discovery

To perform the clustering in embedding space, we employ the K-Means Clustering algorithm. Recall that clustering is a mapping from $\mathbb{R}^M$ for some dimension M to a finite set $\{1, 2, \ldots, K\}$ (see Section 2.2.2). A clustering function can therefore almost act as a Symbol Grounding function (see Definition 4.1.5). This is because for a concept $c_j = \{n_{c_j}, V_{c_j}, \mathcal{Z}_{c_j}\}$ we have $\mathcal{Z}_{c_j} = \mathbb{R}^{M_{c_j}}$ and V_{c_j} is a finite set. Therefore, the domain of the clustering function is the same as the symbol grounding function. The co-domain is not the same, however. The clustering function maps to the set of integers $\{1, 2, \ldots, |V_{c_j}|\}$, not directly to the set V_{c_j}. In other words, the clustering function can help us map each raw input to a cluster index, but not to a latent concept value, which is what we need for our classifier. In the cluster discovery stage, which is the second stage of our Embed2Sym algorithm, we learn the cluster function for all the latent concepts. In the next stage, we will see how to use it to build the final symbol grounding function.

To build a dataset for the clustering algorithm, we can use the perception functions trained

in the previous stage. For a latent concept $c_j = \left\{n_{c_j}, V_{c_j}, \mathcal{Z}_{c_j}\right\}$, let $x_{j_1}, x_{j_2}, \ldots, x_{j_n}$ be the raw inputs associated with c_j. We construct a dataset consisting of embedding vectors for the raw inputs (where y_m are the downstream concept labels):

$$\mathcal{D}_j = \bigcup_{(\bigotimes x_i, \bigotimes y_m) \in \mathcal{D}} \left\{f_j\left(x_{i_k}\right) | 1 \leq k \leq n\right\} \tag{5.3}$$

For example, for the MNIST Addition task, we can construct the dataset $\mathcal{D}_{digit}$ by using the CNN f_{digit} on the MNIST images of the digits. In practice, we can take a subsample of $\mathcal{D}$ as learning a clustering function does not require as many samples as training a neural network on the downstream task.

As mentioned, the clustering algorithm we chose to employ is K-Means. K-Means has the advantages of being simple and fast, although it assumes clusters are isotropic in Euclidean distance. However, our experimental results (Section 5.4.6) demonstrate it performs well against a wide range of task, especially compared to other clustering algorithms such as DBSCAN (Ester et al., 1996). We therefore believe it is a particularly good fit for the Embed2Sym approach. Note that algorithms like K-means assume the number of clusters is known in advance. In our case, this number is assumed to be equal to the number of latent concept values, which is part of the task description. A future direction of research could explore how to discover the number of clusters automatically. For example, running HDBSCAN first to estimate the number of clusters.

5.2.2 Cluster Labelling

In the cluster discovery stage, we identified the clusters in the latent embedding spaces, by learning a function that maps the latent embedding space to cluster indices. But we have yet to discover which symbolic latent value is associated with each cluster. In other words, the clusters are unlabelled. To label the clusters, we take advantage of the fact that the reasoning component g can be given an alternative representation using the logic program P_g.

Using the perception and clustering functions we can map a raw input to a cluster index. What we are missing is a mapping from the cluster indices $\left\{1, 2, \ldots, \left|V_{c_j}\right|\right\}$ to V_{c_j}. This can be thought of as a permutation function over the concept latent values. Denote a permutation function for concept c_j as π_j, and the clustering function as $\mathcal{C}_j$. The symbol grounding function is then given by:

$$G_{c_j}\left(z_i\right) = \pi_j \circ \mathcal{C}_j\left(z_i\right) = \pi_j\left(\mathcal{C}_j\left(z_j\right)\right) \tag{5.4}$$

Our entire classifier for latent concept c_j, which we denote P_j, is given by:

$$P_j\left(x_i\right) = \pi_j \circ \mathcal{C}_j \circ f_j\left(x_i\right) = \pi_j\left(\mathcal{C}_j\left(f_j\left(x_i\right)\right)\right) \tag{5.5}$$

Our logic program P_g can be used to implement the downstream reasoning task. Suppose we have permutation functions for the latent concepts:

$$\pi = \bigotimes_j \pi_j \tag{5.6}$$

We construct a new logic program P_g^π by encoding the π_j as facts and adding these to P_g:

$$\text{cluster_mapping}\left(n_{c_j}, k, \pi\left(k\right)\right) \leftarrow \tag{5.7}$$

We can now implement g as follows: for a given sample $(\bigotimes x_i \otimes \bigotimes s_k, \bigotimes y_t) \in \mathcal{D}$, assign some identifier id. For each raw input x_i associated with latent concept c_j, add the following fact to P_g:

$$\text{raw_input_cluster}\left(id, n_{x_i}, \mathcal{C}_j\left(x_i\right)\right) \leftarrow \tag{5.8}$$

then the symbol grounding function can be computed using the following rule:

$$\text{holds}\left(ID, N, V\right) \leftarrow \text{raw_input_cluster}\left(ID, N, K\right), \text{cluster_mapping}\left(K, V\right) \tag{5.9}$$

where ID, N, V, K are variables representing the sample's identifier, a raw input's name, a latent concept value and cluster index, respectively. Symbolic inputs can be easily encoded as facts as well. For the symbolic input $s_k = \left(n_{s_k}, S_{s_k}, a_{s_k}\right)$ we write:

$$\text{holds}\left(id, n_{s_k}, a_{s_k}\right) \leftarrow \tag{5.10}$$

The rest of the rules of P_g are assumed to map the "holds" predicate to the target concept, which is also expressed using "holds":

$$\text{holds}\left(ID, n_{y_l}, t_l\right) \leftarrow \text{conditions to map to downstream concept} \tag{5.11}$$

Since the rules are assumed to be correct, we can also assume that only a single target concept value is predicted per output. Therefore, our program P_g^π can be passed to a symbolic solver to compute the predicted downstream concepts. Our entire end-to-end neuro-symbolic function is now:

$$h\left(\bigotimes x_i \otimes \bigotimes s_k\right) = P_g^\pi\left(\bigotimes \mathcal{C}_j\left(x_i\right) \otimes \bigotimes s_k\right) \tag{5.12}$$

Example 5.2.1. *For the single-digit MNIST addition task, we have two raw inputs for the two input images:* $x_1 = \left(digit\left(1\right), \mathcal{X}, \boldsymbol{v}_1\right)$ *and* $x_2 = \left(digit\left(2\right), \mathcal{X}, \boldsymbol{v}_2\right)$*. We have a single target concept* $y = \left(sum, \left\{0, \ldots, 18\right\}, t\right)$*. Suppose* x_1 *was mapped to cluster index 3 and* x_2 *to cluster index 8. Then we encode:*

$$\begin{aligned} &\leftarrow raw_input_cluster\left(id, digit\left(1\right), 3\right) \\ &\leftarrow raw_input_cluster\left(id, digit\left(2\right), 8\right) \end{aligned} \tag{5.13}$$

and the rules of summation can be encoded using the '+' operator:

$$holds\left(ID, sum, C\right) \leftarrow holds\left(ID, digit\left(1\right), A\right), holds\left(ID, digit\left(2\right), B\right), C = A + B \tag{5.14}$$

We can now define an optimisation task to find the optimal permutation functions π^*. We can define a score function for π as the number of samples in $\mathcal{D}$ incorrectly classified by our neuro-symbolic model:

$$\text{score}\left(\pi\right) = \sum_{\left(\bigotimes x_i \otimes \bigotimes s_k, \bigotimes y_t\right) \in \mathcal{D}} \left[P_g^\pi\left(\bigotimes \mathcal{C}_j\left(x_i\right) \otimes \bigotimes s_k\right) \neq \bigotimes y_t\right] \tag{5.15}$$

where $[\text{cond}] = 1$ when cond is true, otherwise 0. The optimal permutation is then given by the permutation that results in the smallest amount of misclassifications:

$$\pi^* = \arg\min_\pi \text{score}\left(\pi\right) \tag{5.16}$$

In this manner, we defined the task of labelling clusters as an optimisation problem. Since our logic programs P_g are typically going to be Answer Set Programs, it is most convenient to solve this optimisation task using Answer Set Programming. We take our modified program P_g^{π} that had the permutation encoded directly, and remove the permutation facts (as now these are to be found). Instead of these, we ask the Answer Set Solver to choose a permutation and then compute the number of misclassified samples in order to score it. To choose a permutation, we can use a choice rule per latent concept $c_j = \left(n_{c_j}, V_j, \mathcal{Z}_j\right)$ and cluster index k:

$$1\left\{\text{cluster_mapping}\left(n_{c_j}, k, v_{j_1}\right), , \ldots, \text{cluster_mapping}\left(n_{c_j}, k, v_{j_{|V_j|}}\right)\right\}1 \leftarrow \tag{5.17}$$

ASP actually supports a shorthand form for this set of rules:

$$\left\{\text{cluster_mapping}\left(n_{c_j}, I, V\right) : V = 1 \ldots \left|V_{c_j}\right|\right\} = 1 \leftarrow I = 1...K \tag{5.18}$$

where elsewhere we can define facts that map the integer values of V to latent concept values:

$$\text{latent_concept_value}\left(n_{c_j}, v_1, 1\right) \leftarrow \tag{5.19}$$

$$\vdots \tag{5.20}$$

to ensure the chosen "cluster_mapping" atoms conform to a one-to-one mapping, we also need to add the following constraints:

$$\leftarrow \text{cluster_mapping}\left(n_{c_j}, I_1, V\right), \text{cluster_mapping}\left(n_{c_j}, I_2, V\right), I_1 \neq I_2 \tag{5.21}$$

which ensures two different clusters are not mapped to the same latent concept value. Together with the choice rule above that ensures only a single cluster is chosen per latent concept value, we are guaranteed to choose a permutation function. All that remains is to score it.

Example 5.2.2. *For the MNIST addition task, we have a single latent concept:*

$$c_{digit} = \left(digit, \{0, \ldots, 9\}, \mathbb{R}^M\right) \tag{5.22}$$

We encode the choice of the permutation function as follows:

$$\begin{aligned}
&\textit{latent_concept_value}\left(digit, 0, 1\right) \leftarrow \\
&\textit{latent_concept_value}\left(digit, 1, 2\right) \leftarrow \\
&\textit{latent_concept_value}\left(digit, 2, 3\right) \leftarrow \\
&\textit{latent_concept_value}\left(digit, 3, 4\right) \leftarrow \\
&\textit{latent_concept_value}\left(digit, 4, 5\right) \leftarrow \\
&\textit{latent_concept_value}\left(digit, 5, 6\right) \leftarrow \\
&\textit{latent_concept_value}\left(digit, 6, 7\right) \leftarrow \\
&\textit{latent_concept_value}\left(digit, 7, 8\right) \leftarrow \\
&\textit{latent_concept_value}\left(digit, 8, 9\right) \leftarrow \\
&\textit{latent_concept_value}\left(digit, 9, 10\right) \leftarrow \\
&\left\{\textit{cluster_mapping}\left(digit, I, V\right) : V = 1 \ldots 10\right\} = 1 \leftarrow I = 1...10 \\
&\leftarrow \textit{cluster_mapping}\left(n_{c_j}, I_1, V\right), \textit{cluster_mapping}\left(n_{c_j}, I_2, V\right), I_1 \neq I_2
\end{aligned} \tag{5.23}$$

```
1  latent_concept_value(n_{c_j}, v_i, i).
2
3  raw_input_cluster(sample_id, n_{x_i}, C(j)).
4
5  {cluster_mapping(n_{c_j}, I, V) : V=1..|V_{c_j}|} = 1 :-
6      I=1..|V_{c_j}|.
7  :- cluster_mapping(n_{c_j}, I, V1),
8      cluster_mapping(n_{c_j}, I, V2), V1 != V2.
9
10 holds(Id, RawInputName, Value) :-
11     raw_input_cluster(Id, RawInputName, ClusterIdx),
12     cluster_mapping(C, ClusterIdx, ValueIdx).
13     latent_concept_value(C, Value, ValueIdx).
14
15 % Task-specific rules defining the label
16 % in terms of the ''holds'' predicate
17
18 :~ holds(sample_id, target, y). [-1]
```

Listing 5.1: ASP encoding for the cluster labelling task.

To compute the score of a chosen permutation, we can use soft constraints in ASP. Recall that when a soft constraint is broken, it adds the penalty of the constraint to the score of the Answer Set. An Answer Set with minimal penalty is considered optimal. We therefore add for every sample with identifier id and which has target concepts $y_1, \ldots, y_l$ with ground truth labels $t_1, \ldots, t_l$ the following soft constraint:

$$\sim \text{holds}\left(id, n_{y_1}, t_1\right), \ldots \text{holds}\left(id, n_{y_l}, t_l\right) \quad [-1] \tag{5.24}$$

so that correctly predicting all the target concepts results in a "penalty" of -1. The Answer Set with minimal penalty therefore also encodes the permutation with minimal score (as it gets the highest number of correct downstream predictions, which is equivalent to the fewest number of mistakes).

Example 5.2.3. *For the single-digit MNIST addition task, suppose a sample has a ground truth sum label of 16. We write the following soft constraint:*

$$\sim \mathit{holds}\left(id, sum, 16\right) \quad [-1] \tag{5.25}$$

We solve the optimisation task using Clingo 5.5.1 (Gebser et al., 2017). Clingo returns the optimal Answer Set, and from there it is simple to extract the optimal permutation function from the "cluster_mapping" atoms. In the unlikely event that multiple optimal answer sets exist, we can select one of them.

A summary of the ASP encoding of the optimisation task appears in Listing 5.1.

5.3 Training & Inference Algorithm

With the various stages of Embed2Sym laid out, we can now present the full Embed2Sym training and inference algorithms. These are listed in Algorithm 4 and 5, respectively. An example inference process is also depicted in Figure 5.1 (right). Below we explain the algorithms in further detail.

Algorithm 4 receives the neuro-symbolic task dataset $\mathcal{D}$, the set of latent concepts C, and

Algorithm 4 Embed2Sym Training

Input: $\mathcal{D}$, C, P_g
Output: f, $\mathcal{C}$, π

1: Train fully neural model $h = g \circ f$ on $\mathcal{D}$
2: Select $\mathcal{D}_2 \subseteq \mathcal{D}$
3: $\mathcal{C} \leftarrow \emptyset$
4: **for** $c_j \in C$ **do**
5: $\quad \mathcal{D}_j \leftarrow \emptyset$
6: $\quad$ **for** $(\bigotimes x_i \otimes \bigotimes s_k, \bigotimes y_t) \in \mathcal{D}_2$ **do**
7: $\quad\quad$ **for** x_i s.t. c_j is the latent concept of x_i **do**
8: $\quad\quad\quad$ Add $f_j(x_i)$ to $\mathcal{D}_j$
9: $\quad$ Train $\mathcal{C}_j$ on $\mathcal{D}_j$ using K-Means Clustering.
10: $\quad$ Add $\mathcal{C}_j$ to $\mathcal{C}$
11: Encode optimisation task O according to Listing 5.1 using $\mathcal{D}_2$, $\mathcal{C}$, P_g
12: Solve O and select optimal answer set A
13: $\pi \leftarrow \emptyset$
14: **for** $c_j \in C$ **do**
15: $\quad \pi_j \leftarrow \emptyset$
16: $\quad$ **for** cluster_mapping$\left(n_{c_j}, k, v\right) \in A$ **do**
17: $\quad\quad$ Assign $\pi_k(k) = v$
18: $\quad$ Add π_j to π
19: **return** f, $\mathcal{C}$, π

the rules of the downstream task P_g. Line 1 involves the first stage of Embed2Sym: end-to-end training of the fully neural network. In this case g is the reasoning network, as depicted in Figure 5.1 (left). In line 2, we make a selection of a subset of the dataset $\mathcal{D}$. This is motivated by the fact that both K-Means clustering and cluster labelling using Clingo are quite data-efficient. These steps can be sped up by selecting a much smaller subset of $\mathcal{D}$ than was required for training the perception functions. Selection would typically be done by uniformly sampling from $\mathcal{D}$ without replacement.

In lines 3 to 10, we perform the second stage of Embed2Sym, cluster discovery. $\mathcal{C}$ is a shorthand notation for all the clustering functions for all latent concepts. In lines 5-8 we build the dataset for the clustering algorithm for latent concept c_j (see Equation 5.3) and line 9 uses K-Means Clustering to find the clustering function.

Lines 11-18 are the third stage of Embed2Sym, cluster labelling. In line 11 we construct an Answer Set Program that encodes the optimisation task, as described in the previous section. We then use Clingo to solve the task in Line 12. Clingo gives us the optimal Answer Set, which encodes the optimal permutation function for mapping clusters to latent concept values. Lines 13-18 involve constructing the permutation function from the optimal Answer Set. These are encoded in the atoms "cluster_mapping", which we iterate over to construct the functions π_j. π is a shorthand notation for all the permutation functions for all the latent concepts. Line 19 returns the training algorithms output: the perception functions f, the clustering functions $\mathcal{C}$ and the permutation functions π.

Algorithm 5 is the final stage of Embed2Sym: neuro-symbolic inference. Here we receive raw and symbolic inputs, as well as the list of latent concepts and the outputs from the training stage. We also receive the rules for the downstream task P. An important aspect of Embed2Sym is that P_g of Algorithm 4 and P of Algorithm 5 need not be the same. This allows the rules of the reasoning component to be changed, as long as they are still concerned with the same latent concepts. For example, if the model was trained on the addition of MNIST images, it can

Algorithm 5 Embed2Sym Inference

Input: $\bigotimes x_i$, $\bigotimes s_k$, C, f, $\mathcal{C}$, π, P
Output: $\bigotimes y_t$

1: $\bigotimes \boldsymbol{z}_i \leftarrow f\left(\bigotimes x_i\right)$
2: $\bigotimes I_i \leftarrow \mathcal{C}(\bigotimes \boldsymbol{z}_i)$
3: $\bigotimes v_i \leftarrow \pi(\bigotimes I_i)$
4: $Facts \leftarrow \emptyset$
5: **for** $k \in \bigotimes I_i$ **do**
6: Add $holds(0, n_{x_i}, k)$ to $Facts$
7: **for** $s \in \bigotimes s_k$ **do**
8: Add $holds(0, n_{s_k}, s)$ to $Facts$
9: **for** Latent Concept $c_j \in C$ **do**
10: **for** index $k = 1 \ldots \left|V_{c_j}\right|$ **do**
11: Add cluster_mapping $\left(n_{c_j}, k, \pi_j\left(k\right)\right)$ to $Facts$
12: Let A be an answer set of $P \cup Facts$
13: **return** $\bigotimes y_l$ for all l s.t. $holds(0, n_{y_l}, t_l) \in A$

perform multiplication as well at inference time. Hence, despite starting with a non-generalising neural model, we achieved the same level of generalisation as other neuro-symbolic systems have, and the same level of interpretability.

In line 1 we use the perception functions to map the raw inputs to embeddings. We define the application of f to raw inputs as follows: if input x_i is associated with latent concept c_j, with perception function f_{i_j}, which outputs vector z_i, then:

$$f\left(\bigotimes x_i\right) = \bigotimes f_{i_j}\left(x_i\right) = \bigotimes z_i \tag{5.26}$$

In line 2 we apply the clustering functions to the embedding outputs to receive cluster indices:

$$\mathcal{C}\left(\bigotimes z_i\right) = \bigotimes \mathcal{C}_{i_j}\left(z_i\right) = \bigotimes I_i \tag{5.27}$$

In line 3 we apply the permutation functions to the cluster indices to receive the indices of the latent concept values:

$$\pi\left(\bigotimes I_i\right) = \bigotimes \pi_{i_j}\left(I_i\right) = \bigotimes v_i \tag{5.28}$$

In lines 5-12 we expand the program P with facts that encode information needed to perform the downstream classification, specifically information about the inputs and permutation functions. We use the dummy identifier 0 in place of a sample identifier. Lines 5-6 encode the cluster indices for the raw inputs $x_1, \ldots, x_n$. Lines 7-8 encode the symbolic inputs $s_1, \ldots, s_m$. Lines 9-11 encode the optimal permutation function for each latent concept using the "cluster_mapping" predicate. These facts are added to P in line 12 before being passed to Clingo to receive an Answer Set. This Answer Set encodes, among other things, the predicted values for the target concepts $y_1, \ldots, y_t$. Here we assumed the program $P \cup Facts$ has a single answer set, which is typically the case. However, there is nothing in our algorithm that prevents $P \cup Facts$ from having multiple Answer Sets, if such a scenario is necessary. In that case, there would be multiple prediction combinations for $\bigotimes y_t$, one per Answer Set.

5.4 Evaluation

In this section, we present results following an evaluation of our Embed2Sym implementation in benchmark tasks involving both perception and reasoning. In designing these experiments, we aimed to answer the following questions:

1. Is Embed2Sym capable of solving tasks involving both perception and reasoning?
2. Is Embed2Sym able to scale to more challenging tasks involving a larger number of raw inputs?
3. Can Embed2Sym cope with challenging perception tasks?
4. How does Embed2Sym compare to other neuro-symbolic reasoning systems in terms of accuracy and training time?
5. Can Embed2Sym learn to cluster the latent concepts even when the reasoning task does not have a unique answer set for a given dataset?

To answer questions 1 & 2, we tested Embed2Sym on three benchmark tasks introduced to evaluate neuro-symbolic systems, namely MNIST Addition, Member (Tsamoura et al., 2021) and Forth Sort (Bošnjak et al., 2017). We chose these tasks in particular as a means of testing the scalability of Embed2Sym when confronted with a large number of raw inputs, and increasing this number is straightforward for both. To answer question 3, we took the same approach previously introduced to benchmark the ABL-Sim algorithm (Huang et al., 2021b) and created a CIFAR-10 Addition task as a more challenging version of the MNIST Addition task. To answer question 4, we also compare our models to two state-of-the-art neuro-symbolic systems, DeepProbLog and NeurASP, as they are both well-known and have code available that allows for easy adaptation to the tasks we chose. To answer question 5, we performed an experiment on a variant of MNIST addition, namely *MNIST Addition mod 10*, where for every downstream label there are multiple sets of latent concept values that would derive it. In addition, to test if K-Means is a good choice as a clustering algorithm, we performed an ablation study comparing its performance against other contending clustering algorithms.

Our implementation of Embed2Sym was developed in Python 3.8 using the TensorFlow framework, and can be found at the following: https://github.com/YanivAspis/Embed2Sym. Experiments were performed on a system running Ubuntu 18.04.6 with the following specifications:

- **CPU:** Intel i7-6700 @ 3.40 GHz.
- **RAM:** 8 GB.
- **GPU:** A single GeForce GTX 970 GPU.

When training the various systems, we used the Adam optimiser (Kingma and Ba, 2014) with a learning rate of 0.001. To train Embed2Sym's fully neural model, we use a batch size of 128. The embedding size (output of the perception network) is 128. The number of epochs changes between experiments, with this value being determined by the performance on a validation set separate from the test set. The clustering and logical optimisation steps were performed using a subset of the dataset of 100 samples ($\mathcal{D}_2$ of algorithm 4). Other systems were trained over 3 epochs, except for DeepProbLog on the MNIST Addition task, where we found 1 epoch to be sufficient to achieve good accuracy.

When evaluating the different models on the various tasks, we present results for both the latent accuracy and downstream accuracy. Downstream accuracy is naturally interesting as this is the task we trained the model on. However, latent accuracy demonstrates how well the perception network performs, which allows its reuse for other downstream tasks involving the same latent concept. This is the major benefit of neuro-symbolic systems. Note that both the fully neural and neuro-symbolic models of Embed2Sym make predictions on the downstream task, and it is interesting to compare them, while only the neuro-symbolic model predicts the latent concepts. Baseline models, being neuro-symbolic, also predict both the latent concept and the downstream label. Finally, training time is compared between the systems to assess scalability.

We ran each experiment 5 times and report the average test accuracy and training time to account for the effects of random initialisation of the weights. If a system took longer than 4 hours to train for any run, we stopped the training and declared a timeout.

5.4.1 MNIST Addition

We used the single-digit MNIST Addition task as a running example throughout this chapter to illustrate our approach, and the multi-digit version was used in Chapter 4 as one of the benchmark tasks. Here we consider again the multi-digit task. Each summand has N digits and so the number of raw inputs the model receives is $2N$. For the fully neural model, we can follow the architecture depicted in Figure 5.1 (left) for small values of N. Namely, the reasoning network can have a softmax layer as its final output, with the number of units being the number of possible sums. However, as N grows, the number of target labels (possible sums) grows exponentially, resulting in an output layer of exponential size (for N-digit summation, there are $2 \cdot 10^N - 2$ possible sums). This sparsity of the final layer prevents the neural network from converging during training. To resolve this, we replace the single target concept "sum", with $2N$ target concepts, one for each digit of the sum:

$$y_l = (\text{sum_digit}, \{0, \ldots, 9\}, t_l) \tag{5.29}$$

for $l = 1, \ldots, N+1$. The number of output neurons now grows linearly. The architecture is otherwise unchanged. P_g is the set of rules for computing the sum with carry:

$$\begin{aligned}
\text{holds}(SampleId, \text{digit_carry}, 1, 0) \leftarrow & \text{sample}(SampleId) \\
\text{holds}(SampleId, \text{sum_digit}, I, S) \leftarrow & \text{holds}(SampleId, \text{digit}(1, I), D_1), \\
& \text{holds}(SampleId, \text{digit}(2, I), D_2), \\
& \text{holds}(SampleId, \text{digit_carry}, I, C), \\
& S = (D_1 + D_2 + C) \bmod 10 \\
\text{holds}(SampleId, \text{carry_digit}, I+1, C_1) \leftarrow & I > 0, \text{holds}(SampleId, \text{digit}(1, I), D_1), \\
& \text{holds}(SampleId, \text{digit}(2, I), D_2), \\
& \text{holds}(SampleId, \text{digit_carry}, I, C_2), \\
& C_1 = \left\lfloor \frac{D_1 + D_2 + C_2}{10} \right\rfloor \\
\text{holds}(SampleId, \text{sum_digit}, I, S) \leftarrow & \text{holds}(SampleId, \text{carry_digit}, N+1, S)
\end{aligned} \tag{5.30}$$

N	1	2	3	4	15
Embed2Sym - Fully Neural	97.40	82.10	22.50	00.73	00.00
Embed2Sym - Neuro-Symbolic	97.73	94.31	93.90	91.61	66.40
DeepProbLog	96.02	T/O	T/O	T/O	T/O
NeurASP	96.07	95.00	T/O	T/O	T/O

Table 5.1: Average test sum accuracy (%) on the MNIST Addition task for varying number of digits. T/O stands for "timeout".

N	1	2	3	4	15
Embed2Sym	98.86	98.54	98.96	98.92	98.64
DeepProbLog	98.04	T/O	T/O	T/O	T/O
NeurASP	98.02	98.72	T/O	T/O	T/O

Table 5.2: Average test digit accuracy (%) on the MNIST Addition task for varying number of digits. T/O stands for "timeout".

f_{digit} is again a LeNet-5 (Lecun et al., 1998) with an output layer of 128 and ReLU activation. The f_{digit} component is applied to each of the $2N$ MNIST images, and their embeddings are concatenated before the reasoning component is applied. We generated for every N a training set of 30,000 samples, 500 validation samples and 5000 test samples. Results are summarised in Tables 5.1, 5.2 and 5.3. The results reported in each column are for training and testing on the number of digits appearing at the top.

Table 5.1 shows that Embed2Sym is able to produce models that solve this task with close or better accuracy than that of competing systems. For $N = 1$ the fully neural model performs quite well, despite not having the advantage of being given any knowledge on the task. Note that in order to be considered a correct answer, the fully neural model must predict each of the sum digits correctly. Table 5.2 confirms that even though the fully neural model scores poorly on larger values of N, digit accuracy remains high, meaning the perception network still learns the correct latent concepts. As N grows, the reasoning task becomes very difficult, as statistically speaking the network is likely to get at least one digit in the sum wrong. Therefore the fully neural model by itself is incapable of solving the task for large N in a satisfying manner.

However, all of this is solved once the model is lifted into a neuro-symbolic form. Despite not succeeding in the downstream task, clusters still formed in the perception network's embedding space, and this can be exploited by Embed2Sym to create a neuro-symbolic model fully capable of adding MNIST numbers of large lengths. Note that for $N = 15$, a downstream average task accuracy of 66.40% is due to the statistical independence of predicting each individual input digit. An average accuracy of 98.64% in predicting each of 30 individual inputs correctly corresponds to a theoretical overall accuracy of 66.31% in predicting them all at once. Hence a result of

N	1	2	3	4	15
Embed2Sym - Fully Neural	38.2	65.0	93.7	115.7	412.3
Embed2Sym - Neuro-Symbolic	40.9	69.0	99.6	122.6	434.1
DeepProbLog	1033.4	T/O	T/O	T/O	T/O
NeurASP	919.7	3414.6	T/O	T/O	T/O

Table 5.3: Average training time (sec) on the MNIST Addition task for varying number of digits. Training time of "Embed2Sym - Neuro-Symbolic" includes the training time of "Embed2Sym - Fully-Neural". T/O stands for "timeout". Results reported here for DeepProbLog are after 1 epoch of training.

66.40% is in fact as high as one would expect for such a length.

The results suggest two important conclusions. First, even though the reasoning component did not achieve good accuracy, Embed2Sym is still able to extract the latent concepts from the perception component's embedding space. In this sense, the neuro-symbolic model can be used to improve upon the fully neural one in tasks where the reasoning steps may be more complex, by exploiting background knowledge. Second, this improvement can be achieved using only a limited portion of the dataset.

While other neuro-symbolic systems also provide a neural network with generalisation capabilities, we see the advantage of Embed2Sym in its scalability in terms of training time. For small N, training is still feasible for NeurASP and DeepProbLog. NeurASP trains faster than DeepProbLog as it relies on the fast Clingo engine, being able to train on 2-digit addition within the time limit. However, as we move to 3 digits or more, training becomes infeasible to both NeurASP and DeepProbLog, while Embed2Sym can train up to $N = 15$ within minutes, and theoretically could go higher. Analysing the results deeper, we see that Embed2Sym spends most of its training time at the fully neural stage, with the process of lifting to a neuro-symbolic one only taking a few more seconds. In fact, our analysis shows that those few seconds are mostly spent computing the embeddings using the perception component (line 8 of Algorithm 4), while K-Means clustering and Clingo each take less than a second on their part. This suggests training time can be further reduced by decreasing the number of epochs spent training the fully neural model, although at the possible cost of reduced accuracy.

5.4.2 CIFAR-10 Addition

CIFAR-10 (Krizhevsky and Hinton, 2009) is a dataset of images depicting 10 classes of objects, including animals and vehicles. In this task, we consider each class to be a digit instead. Following Huang et al. (2021c), we created a dataset for CIFAR-10 Addition where the MNIST images have been replaced with images from CIFAR-10, and the sum label is computed based on the digit each image label has been mapped to. Hence, the task involves the same level of reasoning as MNIST Addition but with a far more complex perception task. For the perception component, we use a ResNet56 architecture (He et al., 2016).

For this task only, we performed the experiment on a system with 4 Intel(R) Xeon(R) Silver 4110 CPU cores, 96 GB RAM and a single Quadro RTX 6000 GPU. We used SGD optimisation with a learning rate of 0.1 which decays to 0.01 after 100 epochs and momentum set to 0.9. We trained for 130 epochs in total. For DeepProbLog and NeurASP, each epoch takes a long time so we used a different set of hyperparameters: we trained DeepProbLog for 8 epochs with a batch size of 10. NeurASP only supports a batch size of 1 and could only train for one epoch before timing out. For these two systems, we used Adam optimisation with a learning rate of 0.001 since SGD did not converge for either. We trained these systems for 4 hours before stopping training and we report the accuracy at that point.

Results are summarised in Table 5.4. The experiment confirms Embed2Sym is capable of learning even in this more difficult setting, completing training in less than 2 hours. DeepProbLog did not achieve satisfying accuracy within the time limit, although results could be higher if more time is given. NeurASP was unable to learn as it could only train for 1 epoch. We suspect that the limitation to a batch size of 1 is also affecting the training process negatively.

	Task (%)	Image (%)	Training Time (sec)
Embed2Sym - Fully Neural	84.68	——	5883.6
Embed2Sym - Neuro-Symbolic	84.45	91.74	5908.6
DeepProbLog	55.29	73.75	(T/O) 14400.0
NeurASP	6.52	12.75	(T/O) 14400.0

Table 5.4: Results for the CIFAR-10 Single Digit Addition experiment. Task % = Test accuracy in predicting the sum correctly. Image % = Test accuracy in predicting the digit (object) correctly. The fully neural model produces embeddings for the latent concepts and therefore cannot be assessed on Image Accuracy. Training time is in seconds.

5.4.3 Member

In the Member task (Tsamoura et al., 2021), we are given a list of MNIST images as raw inputs, and another digit as a symbolic input. We must determine if the digit appears in one of the images in the list (In other words, if it is a member of the list). For lists of N elements, the task has the following specification:

- **Latent Concept:** $(\text{digit}, \{0, 1, ..., 9\}, \mathbb{R}^M)$
- **Raw inputs:** $x_i = \left(\text{element}\,(i)\,, [0,1]^{28\times 28}\,, \boldsymbol{v}_i\right)$, for $i = 1, \ldots, N$, each a 28x28 image.
- **Symbolic input:** $s = (\text{test_digit}, \{0, \ldots, 9\}\,, a)$, the digit to test for membership.
- **Target concept:** $y = (\text{is_member}, \{\text{true}, \text{false}\}\,, t)$.

From a time complexity point of view, this task is interesting as the number of ways one can assign digits to the list of raw inputs increases exponentially with N. However, the label gives minimal information: If it is true, we know one of the images is labelled a, but we do not know which. If it is false, all images must not be labelled a, but we have no further information about their label. Therefore, the label does not eliminate the exponential number of lists that can be interpreted. For every N, we generated 10,000 training samples, 1000 validation samples and 2000 test samples.

For the fully neural model, we keep the perception component unchanged. The symbolic input s is fed directly into the reasoning component, using one-hot encoding to represent the 10 possible values of t. This one-hot vector is transformed using fully connected layers to an embedding vector, which is joined to the other embedding vectors being fed into the g reasoning network component. For g, we use an attention-like mechanism where the embedding of s is compared to the embedding of each x_i, and the results are then passed through multiple layers before the true/false prediction is made. For all N, we train the fully neural model for 30 epochs. The rules in P_g are:

$$\begin{aligned} \text{holds}\,(SampleId, \text{is_member}, \text{true}) \leftarrow & \text{holds}\,(SampleId, \text{element}\,(I)\,, X)\,, \\ & \text{holds}\,(SampleId, \text{test_digit}, X)\,. \\ \text{holds}\,(SampleId, \text{is_member}, \text{false}) \leftarrow & \text{sample}\,(SampleId)\,, \\ & \text{not holds}\,(SampleId, \text{is_member}, \text{true})\,. \end{aligned} \tag{5.31}$$

Results of this experiment are listed in Tables 5.5, 5.6 and 5.7. In terms of accuracy, we see that Embed2Sym again achieves comparable or better accuracy to competing systems. In this case, the fully neural model is slightly better than the neuro-symbolic one, especially for

N	3	4	5	20
Embed2Sym - Fully Neural	98.67	98.22	98.57	99.49
Embed2Sym - Neuro-Symbolic	96.82	96.51	97.53	93.46
DeepProbLog	97.71	97.23	88.52	86.75
NeurASP	97.41	T/O	T/O	T/O

Table 5.5: Average test membership accuracy (%) on the Member task for a varying size of the set. T/O stands for "timeout".

N	3	4	5	20
Embed2Sym - Neuro-Symbolic	94.52	94.23	94.33	90.54
DeepProbLog	96.94	95.94	78.85	79.46
NeurASP	95.36	T/O	T/O	T/O

Table 5.6: Average test digit accuracy (%) on the Member task for varying number of digits. T/O stands for "timeout".

$N = 20$. This is most likely due to its more relaxed representation. The fully neural model can retrieve the correct answer to the question of membership, without having to commit to the exact identities of the images. For example, if it is asked if the digit 4 is in the set, it can answer "yes" as long as it believes there are several images that could be a 4. It need not decide between them. In contrast, the neuro-symbolic model must assign a digit to each image. While this does mean that the switch from the neural model to the neuro-symbolic one comes with some cost of accuracy, there is a gain in the form of sound reasoning and interpretability. The ability to correctly predict membership without correctly predicting all set members also explains why all systems demonstrate lower digit accuracy than one would expect based on their accuracy on the downstream task.

Once again, Embed2Sym is much faster to train than competing systems. NeurASP is unable to scale beyond small values, timing out already at $N = 4$. This is due to the time spent by NeurASP computing all answer sets that satisfy a label, as this number grows exponentially with N. DeepProbLog, on the other hand, relies only on the various proof paths to its query, which grows linearly with N. This allows DeepProbLog to scale well on the Member task, while other tasks such as MNIST Addition remain infeasible due to the exponential number of proofs. This is in contrast to the MNIST Addition task, where NeurASP performed better, showing that the scalability on tasks can vary significantly between systems. However, Embed2Sym is able to scale well on both tasks, and is about 5-6 times faster than DeepProbLog. We also noticed that DeepProbLog had more difficulty training for higher values of N, at times failing to converge, which explains the lower average accuracy.

N	3	4	5	20
Embed2Sym - Fully Neural	33.2	40.0	51.6	169.2
Embed2Sym - Neuro-Symbolic	35.6	42.6	54.9	177.2
DeepProbLog	190.4	239.8	304.2	998.9
NeurASP	2044.4	T/O	T/O	T/O

Table 5.7: Average training time (sec) on the Member task for varying number of digits. Training time of "E2S - NS" includes the training time of "E2S - FN". T/O stands for "timeout".

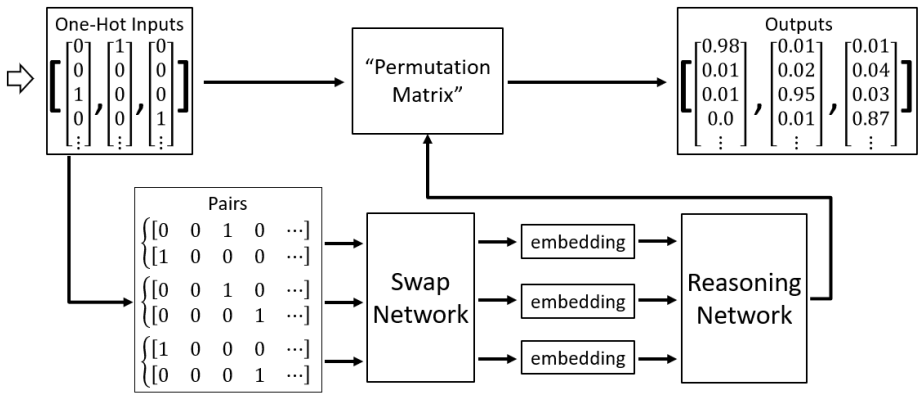


Figure 5.3: High-level diagram of the fully neural architecture for Forth Sort.

5.4.4 Forth Sort

The experiments so far have all been concerned with the classification of images into latent labels. However, Embed2Sym can be applied in a broader range of settings. To demonstrate this, we performed the Forth Sort experiment (Bošnjak et al., 2017). This experiment involves the induction of an operation, namely the decision to swap two elements in a list during sorting. The input to this task is a list of symbolic digits. Note the assumption that we do not know how to compare two digits, and therefore cannot use this as background knowledge. Instead, a "perception" network is employed to learn if two digits should be swapped or not. The latent concept "swap" therefore has two values: true and false. For a list of length N with digits $L_1, L_2, \ldots, L_N$ we have:

- **Latent Concept:** $(\text{swap}, \{\text{true}, \text{false}\}, \mathbb{R}^M)$
- **Raw inputs:** $x_{ij} = \left(\text{pair}\,(i,j), \{0,\ldots,9\}^2, \boldsymbol{v}_{ij}\right)$, for each pair of indices (i,j) of elements in the list. That is, for $1 \leq i \leq N$ and $1 \leq j \leq N$, the raw input is the pair (L_i, L_j). $\boldsymbol{v}_{ij}$ are pairs of the one-hot encoding of L_i and L_j.
- **Symbolic inputs:** None.
- **Target concepts:** $y_i = (\text{sorted_element}\,(i), \{0,\ldots,9\}, t)$, the elements of the sorted list.

Our approach for the fully neural network is depicted at a high level in Figure 5.3. We take the list of digits and convert them to vectors using the one-hot encoding. We then take each pair of input vectors, and pass them through the "Swap Network", which acts as our perception network f_{swap}. As always, this network produces an embedding vector $\boldsymbol{z}_{ij} \in \mathbb{R}^M$. We choose $M = 16$ for this experiment. These embeddings are concatenated and fed into the reasoning network, as usual. The reasoning network, instead of trying to produce the sorted list directly, produces instead a continuous approximation of a permutation matrix. When this matrix is multiplied by the original one-hot inputs, it produces a prediction of the sorted list. The loss function is the softmax cross-entropy applied to all the outputs. As this entire network is differentiable, the gradients can be passed to the swap network so it can learn when two pairs need to be swapped. In fact, the swap network can be thought of as learning the relation "greater than". We trained with a batch size of 256, using the SGD optimiser (as it performed better than Adam) and a learning rate of 0.1.

The background knowledge for the task involves sorting based on the decision of swap/not swap for every possible pair of digits. Since ASP is a declarative language, traditional sorting

N	3	4	5	6	7	8
Embed2Sym - Neuro-Symbolic	11.0	13.6	31.4	50.2	74.6	277.4
DeepProbLog (Bubble Sort)	16.1	70.2	271.4	2277.3	T/O	T/O
DeepProbLog (Quick Sort)	27.6	27.7	213.5	791.3	1602.8	T/O

Table 5.8: Average training time (sec) for the Forth Sort task on various list lengths. Time to reach 100% accuracy on the test set is reported. T/O stands for "timeout". B/Q = Bubble/Quick Sort.

algorithms are not well-suited for this implementation (as the ASP solver would essentially have to emulate them using a search for a stable model, which is inefficient). Instead, we encode the sorting task as a choice of a permutation over the list, with a constraint that the permutation must agree with the outputs of the swap network's prediction. Below is an example of P_g for a list of length 3:

$$
\begin{aligned}
\{\text{permute}\,(SampleId, I, J) : J = 1..3\} = 1 \leftarrow & \text{sample}\,(SampleId)\,, I = 1..3 \\
\leftarrow & \mathit{permute}(S, I1, J), \mathit{permute}(S, I2, J), I1 \neq I2 \\
\text{holds}\,(S, \text{sorted_element}\,(I)\,, D) \leftarrow & \text{holds}\,(SampleId, \text{input_element}\,(J)\,, D)\,, \\
& \text{permute}(SampleId, J, I) \\
\text{correct}\,(SampleId, I, I+1) \leftarrow & \text{holds}\,(SampleId, \text{sorted_element}\,(I)\,, D_1)\,, \\
& \text{holds}\,(SampleId, \text{sorted_element}\,(I+1)\,, D_2)\,, \\
& \text{permute}\,(SampleId, I_1, I)\,, \\
& \text{permute}\,(SampleId, J_1, I+1)\,, I_1 < J_1, \\
& \text{holds}\,(SampleId, \text{pair}\,(I_1, J_1)\,, \mathit{false}) \\
\text{correct}\,(SampleId, I, I+1) \leftarrow & \text{holds}\,(SampleId, \text{sorted_element}\,(I)\,, D_1)\,, \\
& \text{holds}\,(SampleId, \text{sorted_element}\,(I+1)\,, D_2)\,, \\
& \text{permute}\,(SampleId, I_1, I)\,, \\
& \text{permute}\,(SampleId, J_1, I+1)\,, I_1 > J_1, \\
& \text{holds}\,(SampleId, \text{pair}\,(J_1, I_1)\,, \mathit{true}) \\
\text{sorted}\,(SampleId) \leftarrow & \text{correct}\,(SampleId, 1, 2)\,, \text{correct}\,(SampleId, 2, 3) \\
\leftarrow & \text{sample}\,(SampleId)\,, \text{not sorted}\,(SampleId)
\end{aligned}
\tag{5.32}
$$

The "correct" predicate indicate whether the choice to place two adjacent elements in the output list (indices I and I+1) agrees with the prediction by the swap network. If it is true for all adjacent pairs, the list is considered sorted according to the swap network, which is exactly the expected neuro-symbolic output for the task.

Results are reported in Table 5.8. Following the DeepProbLog paper, we report the time it takes to reach 100% accuracy on a test set. In the DeepProbLog paper results for Bubble Sort alone are reported. As we are concerned with training time, we also report results for DeepProbLog using Quick Sort. The results not only confirm that Embed2Sym can be used to solve the Forth Sort task, but can do so much faster than DeepProbLog.

Figure 5.4 depicts the embedding space for the "swap" concept using PCA for dimensionality reduction. There are 100 points, one for each possible digit pair, and each coloured according to whether they belong to the "true" or "false" cluster. In this case, while the clustering algorithm

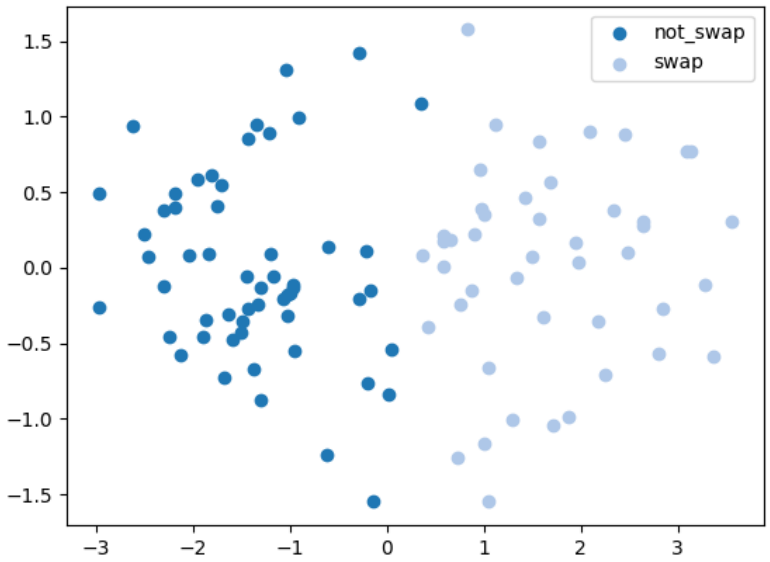


Figure 5.4: PCA visualisation of the embedding space of Swap.

performed well, the points in the latent space behave more similarly to a half-plane classifier, suggesting behaviours other than clustering may emerge. Analysing the space further, we noticed that pairs that are further apart, such as $(0, 9)$, belong to the points at the extreme left or right, while pairs that are closer, such as $(5, 4)$, are closer to the centre. In fact the points right at the centre belong to pairs of equal digits (e.g. $(8, 8)$, suggesting the swap network learned the relative magnitude of digits.

5.4.5 MNIST Addition Mod 10

While the MNIST Addition experiment has become a gold standard for evaluating Neuro-Symbolic models, it does suffer from one drawback. For certain target labels, MNIST Addition provides hidden direct supervision on the latent labels. For example, consider:

$$X + Y = 0 \tag{5.33}$$

where X and Y are MNIST images. Given the rules of addition, the only solution for X and Y is $X = Y = 0$. The CNN would therefore receive a direct signal from the reasoning component during training in these cases. In Embed2Sym, this is somewhat lessened as the reasoning network does not begin training knowing the rules in advance. However, very quickly the reasoning network will also learn to send direct supervision signals to the CNN. To see how Embed2Sym as well as other neuro-symbolic systems fare in situations where this hidden direct supervision signal is not available, we conducted an experiment using a variant of MNIST Addition, namely *MNIST Addition Mod 10*. Rather than provide the sum of the two digits as the downstream label, we provide the sum *mod 10*. Now the equation:

$$X + Y = 0 \tag{5.34}$$

has 10 solutions, so no direct supervision can be given. Even more so, note that:

$$X + Y \pmod{10} = X + 5 + Y + 5 \pmod{10} \tag{5.35}$$

	Run	Task (%)	Digit (%)	Training Time (sec)
Embed2Sym (Fully-Symbolic)	Run 1	97.43	——	66.59
	Run 2	97.65	——	61.70
	Run 3	97.38	——	59.59
	Run 4	97.59	——	58.71
	Run 5	97.21	——	56.93
	Average	97.45	——	60.70
Embed2Sym (Neuro-Symbolic)	Run 1	97.42	0.00	69.77
	Run 2	98.04	99.02	63.22
	Run 3	97.60	00.27	61.02
	Run 4	97.34	98.67	60.17
	Run 5	97.64	98.81	58.39
	Average	97.61	——	62.52
DeepProbLog	Run 1	96.84	98.38	916.16
	Run 2	11.82	10.09	1205.07
	Run 3	58.80	00.18	1143.91
	Run 4	96.74	98.34	906.52
	Run 5	96.22	98.06	899.37
	Average	72.08	——	1014.21
NeurASP	Run 1	9.82	8.92	644.04
	Run 2	10.12	10.28	626.36
	Run 3	9.82	9.8	619.6
	Run 4	95.02	0.56	508.03
	Run 5	9.32	9.58	395.46
	Average	26.82	——	558.698

Table 5.9: Results for the MNIST Addition Mod 10 experiment. Task % = Test accuracy in predicting the sum mod 10 correctly. Digit % = Test accuracy in predicting the digit correctly. The fully neural model produces embeddings for the latent concepts and therefore cannot be assessed on Image Accuracy.

in other words, if you shift all the digits by 5 ($0 \rightarrow 5$, $1 \rightarrow 6$,...), you would get the exact same solution for any addition mod 10 equation. Consequently, there are two sets of valid solutions to the latent labels of the MNIST images: one that corresponds to the normal meaning of these images ( $\rightarrow 0$, $\rightarrow 1$) and one that corresponds the normal meaning shifted by 5 ($\rightarrow 5$, $\rightarrow 6$)). Even if the CNN could perfectly distinguish between the 10 digits (e.g. in the case of Embed2Sym if it generated 10 perfect clusters), the neuro-symbolic model would still have to deal with this "double solution" issue (in logic terms, the abductive task of assigning labels to the clusters has two solutions).

In this task, we have:

- **Latent Concept:** $(\text{digit}, \{0, 1, ..., 9\}, \mathbb{R}^M)$
- **Raw inputs:** $x_1 = \left(\text{input_digit}(1), [0,1]^{28\times 28}, \boldsymbol{v}_1\right)$, $x_2 = \left(\text{input_digit}(2), [0,1]^{28\times 28}, \boldsymbol{v}_2\right)$.
- **Target concept:** $y = (\text{sum_mod_10}, \{0, 1, ..., 9\}, t)$.

We trained all models using the same approach as the regular MNIST Addition. For DeepProbLog and NeurASP we chose the number of epochs that achieved the best accuracy on the validation set. Results are summarised in table 5.9. Due to the "double solution" issue mentioned above, showing an average digit accuracy over 5 runs is meaningless, so we chose to show the results of each run in a separate row.

The results show Embed2Sym can learn the downstream task of MNIST Addition Mod 10 consistently and with similar downstream accuracy to the regular MNIST Addition task, with similar training time, showing that even without hidden direct supervision there is no hindrance in performance. The digit accuracy achieved (as can be seen in the 5 runs under the neuro-symbolic mode) reveals the "double solution" phenomenon. Embed2Sym either scores close to 99% digit accuracy, or close to 0%, depending on whether the cluster assignment corresponds to the normal human labelling of the digits or the one shifted by 5. Either way, the downstream accuracy is not affected due to the symmetry revealed by equation 5.35. For Embed2Sym, the "double solution" only becomes relevant at the symbolic optimisation stage, when clusters need to be assigned their digit label. In this case the answer set program constructed in step 11 of the training algorithm 4 has two optimal answer sets. In step 12, the ASP solver will choose one of them for the assignment of clusters to digit labels, resulting in the apparently random 99% or 0% accuracy. One can easily imagine, however, a variant of Algorithm 4 where we ask the ASP solver to return **all** optimal answer sets, showing the user that there are two equally valid assignments.

Unlike Embed2Sym, DeepProbLog and NeurASP struggle with this task compared to the regular MNIST Addition task. DeepProbLog is able to converge most of the time to one of the two solutions, but also fails to converge quite often, underperforming in the downstream task. NeurASP is almost never able to converge. For both systems we also see the phenomenon of high downstream accuracy together with a digit accuracy close to 0%, showing they are at times converging to the "shifted by 5" solution. Regardless, even when correctly converging, DeepProbLog and NeurASP still require at least an order of magnitude more training time to achieve this result, and this will become even worse with increased number of digits. We attribute DeepProbLog's and NeurASP's convergence struggles to the issue of multiple optimal solutions. Unlike Embed2Sym's reasoning network, DeepProbLog and NeurASP must assign images to digits during neural training, which means gradient information must flow from the downstream label to CNN pointing it towards specific digit labels. The "double solution" phenomenon means there are always at least two legitimate digit labels for any image, making convergence difficult. Depending on the random initialisation, backpropagation may be able to find a local minimum that corresponds to one of the optimal solutions but may also get stuck at a suboptimal local minimum.

5.4.6 Choice of Clustering Algorithm

Finally, we wanted to compare our choice of K-Means as the clustering algorithm to other candidates, to justify its choice. As an ablation study, we performed the MNIST Addition experiment comparing K-Means to the following three options:

1. **Gaussian Mixture Model (GMM):** a soft variant of K-Means that models the data as coming from one of K gaussians, each gaussian determining the probability of a datapoint being assigned to it (normalised over the entire distribution). The GMM distribution is discovered through the Expectation Maximisation algorithm, where at each iteration the probability of the datapoint assignments are computed, then the parameters of the gaussians are updated.

2. **HDBSCAN:** A popular variant of DBSCAN, a density-based algorithm. Unlike K-Means and GMM, DBSCAN does not require the number of clusters to be specified in advance. Instead it is dependent on an ϵ parameter describing how close two points need to be

	Fully Neural (%)	Neuro-Symbolic (%)	Digit (%)	Training Time (sec)
K-Means	96.87	97.53	98.75	68.2
GMM	97.52	97.19	98.59	71.0
HDBSCAN	97.36	94.22	97.07	74.3
MeanShift	97.21	72.32	64.46	166.1

Table 5.10: Results for the Clustering Ablation study on the MNIST Addition experiment. Fully Neural % = Test accuracy in predicting the sum correctly for the fully neural model. Neuro-Symbolic % = Test accuracy in predicting the sum correctly for the neuro-symbolic model. Digit % = Test accuracy in predicting the digit correctly.

in order to be considered part of the same cluster. This also allows dealing with outlier datapoints by labelling them as noise when they are too far apart from any other point. HDBSCAN expands on DBSCAN by not requiring a single value ϵ. Instead, it constructs many cluster candidates for a variaty of ϵ values. It then constructs a minimum spanning tree for all the potential clusters and prunes it. Note that both DBSCAN and HDBSCAN do not allow for the assignment of new datapoints to clusters after training. To perform inference in Embed2Sym, we choose to assign a new datapoint to the same cluster as its nearest neighbour.

3. **MeanShift:** Like DBSCAN, MeanShift does not require the number of clusters to be specified in advance. MeanShift works by assigning a "search window" around every data point, the size of which is controlled by a *bandwidth* hyperparameter. At every step, MeanShift calculates the mean of all points in the window and shifts the window to be centred around this mean. In this way the windows move towards high density areas. At the end of the algorithm, all datapoints falling within the same window are assigned to the same cluster. At inference time, a new datapoint is assigned to the cluster with the nearest centroid.

We kept all other architecture and hyperparameters the same, except we used 1000 samples for training the clustering algorithm in case 100 was not sufficient for some of the algorithms. Results are presented in table 5.10. The results show that K-Means and GMM perform best, with K-Means having perhaps a slight edge being slightly faster. This suggests the clusters generated by the CNN are sufficiently separate where a probabilistic assignment does not differ much from a hard assignment. HDBSCAN does slightly worse, likely because it labels quite a few datapoints as noise. MeanShift performs worst of all, not being able to consistently determine the correct number of clusters despite our attempt to fine-tune its bandwidth hyperparameter carefully, suggesting it is unsuitable for this task. Overall, the ablation study suggests that Embed2Sym learns to generate isotropic and distinct clusters which are quite suitable for K-Means.

5.5 Discussion

With regards to the five questions raised at the beginning of section 5.4, we can now provide the following answers:

1) Embed2Sym successfully solves tasks involving perception and reasoning, as confirmed by the Addition, Member and Forth Sort experiments.

2) Embed2Sym is capable of scaling to tasks involving dozens of raw inputs and potentially more. In particular, we have shown that Embed2Sym is capable of solving 15-digit MNIST

addition and 20-digit Member.

3) Embed2Sym can learn even when the perception task is more complex, as confirmed by the CIFAR-10 Addition experiment.

4) Embed2Sym achieves similar or better test accuracy to competing systems on all the tested tasks, while training faster by several orders of magnitude and scaling to tasks infeasible to other systems.

5) Embed2Sym is able to cluster latent concepts even if there is no unique answer set for a given dataset, as seen in the MNIST Addition Mod 10 experiment which had two, and does so much faster and more consistently than the competing systems.

Embed2Sym offers a more scalable approach to neuro-symbolic reasoning by embracing a different philosophy than other systems. While most approaches attempt to turn logical formulas into differentiable operations, Embed2Sym uses the neural and symbolic components as a means to complement each other. The difficulty in creating a scalable version of differentiable logic is highlighted by the combinatorial issues existing systems encounter. Backpropagation through a differentiable logic gate requires answering the question "Which input values would result in the correct output". Systems such as NeurASP and ABL link this question to logical abduction. In the case of ABL this is done only in the context of definite programs, whereas NeurASP uses the full expressivity of Answer Set Programming. The difficulty in using abduction is that these systems need to find the full set of solutions which in general can be exponential in size. DeepProbLog takes a proof-theoretical view to this question, constructing a (weighted) boolean formula that satisfies a given query. The time complexity of this construction can be $\#P$-complete (Fierens et al., 2015). We see these difficulties manifest in practice particularly in the MNIST Addition experiment, where even for small values of N, training becomes infeasible for both NeurASP and DeepProbLog.

Embed2Sym solves this issue by embracing the approximate nature of neural network inference. The training of the fully neural model creates a structure in embedding space that a symbolic system can later identify and exploit. This gives a symbolic reasoning system access to inference on raw data while circumventing the difficulties arising from differentiable logic operations. The logical optimisation step, which is the critical step in creating the neuro-symbolic model, scales well with an increased number of raw inputs because it builds upon the advances in solving an optimisation problem in Answer Set Programming. We see from experiments that the logical optimisation step is very fast (less than 1 second), and it is in fact the training of the neural network that has a significant effect on training time.

A question naturally arises as to Embed2Sym's assumption that clusters will appear in embedding space during training, as this is an empirical observation, not a theoretical one. In practice we see this happening across different datasets and tasks (see, for example, the MNIST Addition Mod 10 experiment) and more examples will be shown next chapter. This appears to be, if not a universal phenomenon, at least a common one. It is possible, however, that other structures might appear, for example in the Forth Sort experiment where the behaviour appeared more similar to a half plane classifier.

Another question one might wonder is whether clustering in embedding space was necessary. Could the images have been clustered directly, avoiding the need to train a perception and reasoning network from a downstream task? Note, however, that raw data contains many features that are not semantically meaningful for the downstream task. For example, MNIST

images contain many distracting visual features such as digit orientation, location or line thickness. Other datasets would also include colour or lighting information, or background objects. A direct clustering algorithm must take into account all of these features, resulting in suboptimal clustering for the task at hand. Embed2Sym, in contrast, would learn to ignore these as they are not useful for the downstream task.

It should be noted that while Embed2Sym works by identifying latent concepts discovered by a neural network, it should not be understood as an explainability method. Embed2Sym makes no attempt at explaining why the perception network associates a given raw input with a latent concept value. It merely states that it did. Specifically, it is not designed to discover if this relationship is due to a statistical correlation rather than a causal relation. Note that other neuro-symbolic systems do not address this issue either (an issue highlighted by Marconato et al. (2023)).

In this work, we used K-Means as a clustering algorithm, which we justified in an ablation study, though it is still possible that Embed2Sym may benefit from more sophisticated methods of discovering cluster information in embedding space (Guo et al., 2021). For the reasoning component during the neural training stage, we chose to employ a simple MLP, but in general more sophisticated models could be required, as reasoning gets more complex. For example, reasoning involving differentiating and integrating mathematical functions has been shown to be possible using tree-generating Seq2Seq models (Lample and Charton, 2020). The MNIST Addition Mod 10 experiment suggests that neuro-symbolic systems can benefit from jointly training the perception and reasoning components, rather than treating them as separate, as a less rigid reasoning component can overcome situations where the downstream label does not provide sufficient gradient information to the perception component.

Finally, we assumed in this chapter that the reasoning task can be perfectly specified using rules that are known in advance. While this can be the case in many tasks, in general, the rules may not be fully known. In the next chapter, we tackle this issue directly by combining Embed2Sym with an Inductive Logic Programming system. Namely, the ILASP system for learning Answer Set Programs from data.

5.6 Summary

In this chapter, we built on insights gained in the previous one to develop a novel approach to neuro-symbolic reasoning. We introduced Embed2Sym, a system that is capable of solving tasks involving both perception and reasoning, that is interpretable and can generalise to other downstream tasks. We explained in detail the four stages of Embed2Sym, namely: end-to-end pretraining of a fully neural network, cluster discovery in latent space, cluster labelling using logic-based optimisation and neuro-symbolic inference. In experiments, our system was shown to scale far better than competing systems, being capable of solving tasks involving a far larger amount of raw inputs and more complex perception tasks. We discussed the difference in approach between Embed2Sym and other neuro-symbolic systems, why it scales much better, and mentioned some aspects where Embed2Sym can be improved, particularly by removing the assumption that the rules of the reasoning task are fully given in advance. In the next chapter, we will turn our attention to this topic.

Chapter 6

Embed2Rule - Scalable Neuro-Symbolic Learning via Latent Space Weak-Labelling

In the previous chapter, we presented a novel method for neuro-symbolic reasoning, Embed2Sym, that overcomes the scalability issue encountered by existing systems. It does so by breaking the training task into multiple stages: end-to-end pre-training of a fully neural model, followed by identification of clusters in latent space, and labelling of the clusters using the symbolic knowledge of the downstream task. Like other neuro-symbolic reasoning systems, Embed2Sym assumes the rules of the downstream tasks are fully known in advance (specified as the ASP program P_g in Algorithm 4). Hence, a natural question arises: what happens if the rules of the downstream task are not fully known? Learning the symbolic rules jointly with the neural networks is exceedingly difficult, as the reasoning component depends on accurate neural predictions to learn meaningful rules while the neural component relies on accurate rules to receive a useful training signal. Hence, most approaches choose to hand-engineer the rules.

To avoid the need to engineer the symbolic knowledge, methods have recently been introduced for inducing rules from raw data, though these are usually restricted to the ground case (Wang et al., 2021; Daniele et al., 2023a; Defresne et al., 2023). Methods have also been proposed for performing Inductive Logic Programming (Muggleton, 1991) of first-order programs from raw data. MetaABD (Dai and Muggleton, 2021) can learn first-order datalog programs. NSIL (Cunnington et al., 2023b) and NeuralFastLAS (Charalambous et al., 2023) learn the more expressive answer set programs. However, these approaches suffer from the same scalability issues encountered by systems with engineered complete knowledge.

In this chapter, we build upon the Embed2Sym approach and introduce a scalable neuro-symbolic learning system that can learn answer set programs from raw data. Our approach, which we call Embed2Rule, does so by splitting the training of the neural network and the rule learning into two steps. After training the network, we discover latent symbolic concepts by clustering the learned latent space and optimally labelling the clusters using weak labels generated by an open-source Vision-Language Model (VLM) (Li et al., 2023a). The cluster labels are optimised with respect to these weak labels. Finally, the optimised labels are used by the (noise-robust) symbolic learner ILASP 4 (Law, 2023) to find optimal rules for solving the downstream task. Our key contributions are as follows:

- We propose Embed2Rule, a neuro-symbolic learning framework that can train neural networks for perception tasks while inducing answer set programs from raw data.
- We leverage a pre-trained VLM in a novel way to ground raw inputs into symbolic latent concepts and overcome scalability issues in learning rules from raw data.
- We evaluate Embed2Rule on existing benchmark tasks, demonstrating it achieves better accuracy than baselines while scaling up to tasks infeasible to other approaches.

In addition, Embed2Rule is modular with respect to both the VLM and the symbolic learner and will benefit from the continued improvement of either.

The chapter is structured as follows: section 6.1 provides a problem statement and an overview of our method. Section 6.2 describes our approach to labelling the clusters using a weak labeller. Section 6.3 details the symbolic learning process. Section 6.4 provides an evaluation of Embed2Rule on existing benchmark tasks and compares it to existing systems, and section 6.5 concludes the chapter.

The work described in this chapter has previously been published in Aspis et al. (2024).

6.1 Method Overview

In Embed2Sym, we assumed that we were given a fully specified program P_h that fully realises the reasoning function g (see Section 5.2.2). In Embed2Rule, this assumption is relaxed so that we now need to induce rules in P_h. Instead of being fully specified, we assume we are given some background knowledge B about the reasoning task and a hypothesis space S_M, specified through mode declarations or any other way (see Section 2.2.1 in the background). We need to find $H \in S_M$ such that g is realised by $P_g = B \cup H$.

Because P_g is now missing, the optimisation task for finding the optimal cluster labelling, as defined in equations 5.15 and 5.16 is now no longer possible to compute. This is because P_g^π cannot be constructed with some rules missing. In terms of the ASP optimisation as appearing in Listing 5.1, the task-specific rules cannot be specified[1].

At this point, we are able to obtain:

- The perception functions f, as we can learn these in the fully neural training stage without knowing the downstream rules.
- The cluster functions $\mathcal{C}$, as we can learn these using the K-Means clustering algorithm after training the fully neural model.

and we are yet unable to obtain:

- The permutation function π for mapping cluster indices to latent concept values.
- The missing rules H of P_g

As a consequence, the symbol grounding function $G = \pi \circ \mathcal{C}$ is also unavailable.

Recall that in the LAS framework, a symbolic learning task is a tuple (B, S_M, E^+, E^-). In Embed2Rule, B and S_M are already given, and E^+ and E^-, the positive and negative examples, remain to be constructed. In order to induce the rules of the downstream task, these examples should include information about the latent concept values for each sample.

[1] If we have some rules P_g for some other task involving the same set of images, we can apply Embed2Sym to learn the cluster labels at this point. However, this will not work if we are in a domain where such rules are not known, e.g. a medical imaging domain.

Without the permutation function π, we do not have access to the latent concept values, and therefore cannot construct the examples. Without the induced rules, we cannot find the correct permutation function π.

This inter-dependency between the optimal permutation π and the optimal program P_g is another example of why neuro-symbolic learning from raw data is so challenging. Most authors encounter difficulty as they require the training of a perception function to be done jointly with the induction of the optimal rules. The inter-dependency on the performance of the perception function and the quality of the rules makes convergence unlikely. In our case, the task is simplified to finding only an optimal permutation π, but it still remains challenging.

Formally, we define the new optimisation function using modified versions of Equations 5.15 and 5.16. For permutation functions π and $H \in S_M$, a score is given by:

$$\text{score}\left(\pi, H\right) = \sum_{\left(\bigotimes x_i \otimes \bigotimes s_k, \bigotimes y_t\right) \in \mathcal{D}} \left[P^{\pi}_{B \cup H}\left(\bigotimes \mathcal{C}_j\left(x_i\right) \otimes \bigotimes s_k\right) \neq \bigotimes y_t\right] \tag{6.1}$$

and the solution to our task is given by:

$$\left(\pi^*, H^*\right) = \arg\min_{\pi, H} \text{score}\left(\pi, H\right) \tag{6.2}$$

6.1.1 Joint Optimisation via Program Induction

Since π and H depend on one another, it would appear they must be optimised jointly. Inducing an Answer Set Program can be performed by ILASP, while in Embed2Sym we optimised π using an ASP solved by Clingo, which ILASP is based on. The most natural course then, is to have ILASP induce not only the optimal hypothesis H but also the permutation function π.

Note that this is not as simple as taking a modified encoding in Listing 5.1 and placing it in the background knowledge of the task. While this encoding asks the ASP solver to choose a permutation function, this is not suitable for ILASP. Recall that in ILASP, an example is a (weighted) context-dependent partial interpretation $e = (e_{id}, e_{pen}, e_{pi}, e_{ctx})$ where e_{pi} is the partial interpretation and e_{ctx} is the context. To cover the example, the answer sets of $B \cup H \cup e_{ctx}$ must extend e_{pi}. In other words, each example in ILASP can have different answer sets. Therefore, by encoding the choice of a permutation function in the background knowledge, we are asking ILASP to choose a different permutation function per example, rather than a global one.

For ILASP to choose a global permutation function, it must induce it as a set of facts. For a latent concept $c_j = \left(n_{c_j}, V_{c_j}, \mathcal{Z}_{c_j}\right)$, which has $\left|V_{c_j}\right|$ clusters marked with indices $1, \ldots, \left|V_{c_j}\right|$, we add the following set of facts to the hypothesis space:

$$\left\{\text{cluster_mapping}\left(n_{c_j}, i, v\right) \leftarrow \middle| i \in \left\{1, \ldots, \left|V_{c_j}\right|\right\}, v \in V_{c_j}\right\} \tag{6.3}$$

where if a fact $\text{cluster_mapping}\left(n_{c_j}, i, v\right)$ is induced, it means cluster i is mapped to latent concept value v. To ensure a one-to-one mapping, we must add the following constraints:

$$\begin{aligned}
&\leftarrow \text{cluster_mapping}\left(n_{c_j}, I_1, V\right), \text{cluster_mapping}\left(n_{c_j}, I_2, V\right), I_1 \neq I_2 \\
&\leftarrow \text{cluster_mapping}\left(n_{c_j}, I, V_1\right), \text{cluster_mapping}\left(n_{c_j}, I, V_2\right), V_1 \neq V_2 \\
&\text{chosen}\left(n_{c_j}, I\right) \leftarrow \text{cluster_mapping}\left(n_{c_j}, I, V\right) \\
&\leftarrow \text{not chosen}\left(n_{c_j}, I\right)
\end{aligned} \tag{6.4}$$

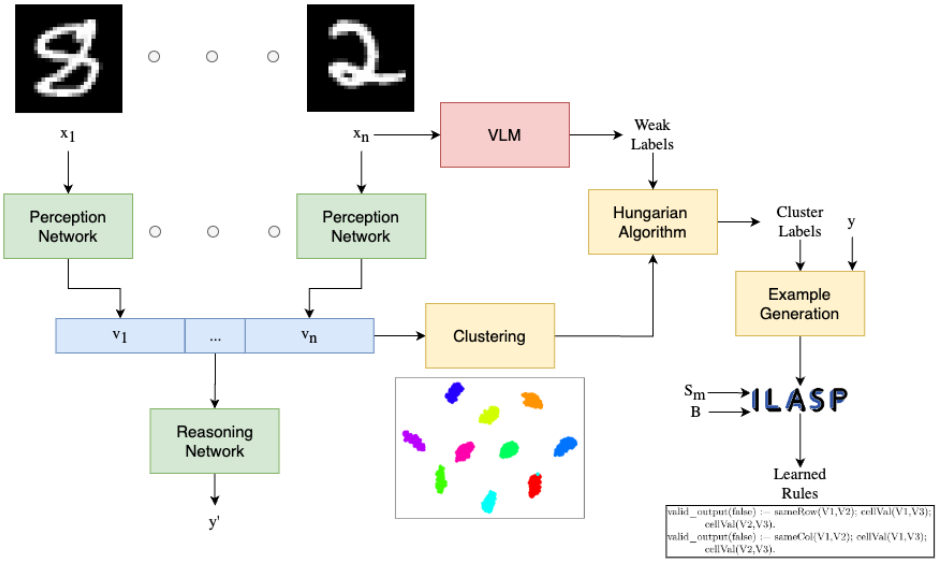


Figure 6.1: Architecture of the Embed2Rule system.

where the last two rules ensure every cluster index is mapped to some latent concept value.

Since ILASP is guaranteed to derive an optimal hypothesis, theoretically speaking the approach above solves the joint optimisation task. In practice, other than for the smallest tasks, this approach will fail. The reason is ILP systems prefer to induce shorter programs over longer ones, following the principle of Occam's Razor. This is formalised by their use of a penalty function over program length. Every fact induced incurs a penalty of 1 to the length. To induce the full permutation function, a total penalty of $\sum_j |V_{c_j}|$ must be incurred. This could be much larger than the penalty for the rules in H.

To find an optimal hypothesis efficiently, ILASP first searches for short programs over long ones. Therefore, a hypothesis that induces a large number of facts will not be considered for a long time. In addition, ILASP may choose not to cover $\sum_j |V_{c_j}|$ examples instead of inducing the "cluster_mapping" facts, as this can be equally optimal. To avoid this, we must add many examples that will not be covered if the "cluster_mapping" facts are not induced. In theory, this will be at least $\sum_j |V_{c_j}|+1$ examples. In practice, in order to learn the full permutation function, there will be many more. Even then, the search will likely take a very long time, preventing this approach from scaling up.

Joint learning has also been considered in other neuro-symbolic learning approaches (Dai and Muggleton, 2021; Cunnington et al., 2023b) with limited scalability results. As such, we propose to replace joint learning with a sequential learning approach: find the optimal permutation, followed by the optimal hypothesis. However, since the optimal hypothesis is unknown, we cannot find the optimal permutation with the current set of components in our system. Therefore, we introduce a new element: a weak labeller. Figure 6.1 shows an overview of our proposed architecture. On the left we see the fully-neural architecture which remains unchanged from Embed2Sym. We will explain the rest of the components in the next few sections.

6.2 Weak & Cluster Labelling

A weak labeller is a classifier of samples into a given set of labels, where the classification can be noisy. The accuracy of weak labellers can range from better-than-random to highly accurate. Weak labellers may be heuristic-based, rule-based or statistical models. They are often employed

Dataset	Accuracy (%)
MNIST	65.21
Playing Cards	78.46

Table 6.1: Accuracy of the classifier implemented using BLIP-2's Image-Text Matching head.

to label data where manual labelling can be costly, with the goal of creating a better classifier (Zhang et al., 2022).

For a latent concept c_j, a latent concept classifier is a function $P_j : \mathcal{X}_j \to V_{c_j}$ where $\mathcal{X}_j$ is a raw input space associated with latent concept c_j. In Equation 5.5, we defined a classifier using a perception network and a symbol grounding function. The backbone of this classifier is our perception function, which maps the raw input to the far more compact latent representation. The symbol grounding function then carries out the translation to a latent concept value, completing the classifier. We will now look at another classifier, which uses a Vision-Language Model (VLM) as a backbone. Namely, BLIP-2 (Li et al., 2023a), an open-source VLM with publicly available weights.

In Section 2.2.8 we described the training stages of BLIP-2. In the first stage, the Q-Former is trained to predict how well a given text describes an image through an Image-Text Matching head. This head produces a score between 0 and 1, with 1 being an excellent match. We can use the Image-Text Matching head to create a classifier for a latent concept. Denote the Image-Text Matching function as $\text{match}(I, T)$ where I is an image and T is some text. Given a raw input x_i associated with latent concept $c_j = \left(n_{c_j}, V_{c_j}, \mathcal{Z}_{c_j}\right)$, let T_{c_j} be a "description function" that maps the latent concepts values to text. Then the following is a classifier of images to latent concept values of c_j:

$$WP_j(x_i) = \underset{v \in V_{c_j}}{\arg\max}\, \text{match}\left(x_i, T_{c_j}(v)\right) \tag{6.5}$$

Example 6.2.1. *In the MNIST Addition task, the latent concept "digit" has ten values which are the set* $\{0, \ldots, 9\}$*. A possible description function* T_{digit} *takes each of these values and translates them to a sentence of the form "The digit X" where X is the digit. For example:*

$$T_{digit}(0) = \textit{"The digit 0"} \tag{6.6}$$

If X_i *is an MNIST image,* $\text{match}(X_i, \textit{"The digit 0"})$ *provides a value between* 0 *and* 1 *for how well the image matches the text "The digit 0". Ideally, the match function will give the highest score for the digit which appears in the image, giving us the classification.*

In Table 6.1 we see the classification accuracy of the proposed WP_j function on two datasets: MNIST and Playing Cards (Cunnington et al., 2023a). The latter dataset involves images of playing cards taken from a standard deck. There are 52 classes, one for each type of card. We see that BLIP-2's performance, while far better than random, still falls quite short of a good classifier for these datasets. In particular, the accuracy for MNIST digits is well below that of our CNN classifier trained for Embed2Sym (see Table 5.2). This is not surprising, as BLIP-2 is a general model that was not trained on tasks involving MNIST digits, unlike our model. Specialised models will generally perform better than general ones on any given task. For neuro-symbolic tasks that involve many input images, this is particularly important, as the image classification error is amplified when converted to downstream error. Because of this, we wish to achieve a better classification accuracy than BLIP-2 offers. We therefore propose to use

BLIP-2 as a weak labeller, and use it to find the optimal permutation function.

For a latent concept $c_j = \left(n_{c_j}, V_{c_j}, \mathcal{Z}_{c_j}\right)$, where $V_{c_j} = \left\{v_1, \ldots, v_{|V_{c_j}|}\right\}$, we construct a dataset $\mathcal{D}_j$ of cluster indices and weak labels. Given some subset of the dataset $\mathcal{D}_2 \subseteq \mathcal{D}$, we have:

$$\mathcal{D}_j = \left\{ \left(\mathcal{C}_j \left(f_{i_j}\left(x_i\right)\right), WP_j\left(x_i\right)\right) \middle| x_i \text{ is a raw input in } \mathcal{D}_2 \text{ associated with } c_j \right\} \tag{6.7}$$

Every cluster index - weak label pair gives evidence that that particular cluster should be associated with that particular latent concept value. The more of these pairs found in the dataset, the more likely the optimal permutation function should map that cluster index to that latent concept value. There are $\left|V_{c_j}\right|$ clusters and $\left|V_{c_j}\right|$ latent concept values. We can construct a matrix M^j that counts the number of occurrences of each pair in the dataset $\mathcal{D}_j$:

$$M^j_{kl} = \left|\left\{(k, v_l) \in \mathcal{D}_j\right\}\right| \tag{6.8}$$

Then, an optimal permutation π_j maximizes the following sum:

$$\sum_{k=1}^{|V_{c_j}|} M^j_{k,\pi_j(k)} \tag{6.9}$$

Example 6.2.2. *Suppose a latent concept has four possible values. The perception function learned 4 clusters, which were discovered by K-Means. We construct a dataset by mapping images to their cluster and using BLIP-2 to receive a weak label. The dataset may look like:*

$$\left\{(1,1), (1,2), (2,1), (2,4), (3,3), (4,1), \ldots\right\} \tag{6.10}$$

We build the matrix M^j by counting entries in the dataset:

$$\begin{array}{c} \\ \textit{Cluster \#1} \\ \textit{Cluster \#2} \\ \textit{Cluster \#3} \\ \textit{Cluster \#4} \end{array} \begin{array}{c} \begin{array}{cccc} \textit{Label \#1} & \textit{Label \#2} & \textit{Label \#3} & \textit{Label \#4} \end{array} \\ \begin{bmatrix} 205 & 62 & 0 & 0 \\ 16 & 0 & 0 & 234 \\ 98 & 1 & 136 & 21 \\ 251 & 0 & 0 & 0 \end{bmatrix} \end{array} \tag{6.11}$$

the optimal permutation is the following:

$$\begin{aligned} &\textit{Cluster \#1} \longrightarrow \textit{Label \#2} \\ &\textit{Cluster \#2} \longrightarrow \textit{Label \#4} \\ &\textit{Cluster \#3} \longrightarrow \textit{Label \#3} \\ &\textit{Cluster \#4} \longrightarrow \textit{Label \#1} \end{aligned} \tag{6.12}$$

which has the sum:

$$M^j_{12} + M^j_{24} + M^j_{33} + M^j_{41} = 62 + 234 + 136 + 251 = 683 \tag{6.13}$$

Given a matrix M^j, finding the optimal permutation can be done efficiently using the Hungarian Algorithm (Kuhn, 1955). The Hungarian Algorithm solves assignment tasks between sets of the same size. Given two sets $A = \{a_1, \ldots, a_n\}$ and $B = \{b_1, \ldots, b_n\}$ and a cost matrix C where C_{ij} is the cost of assigning element a_i to element b_j, the Hungarian Algorithm finds an assignment with minimal cost. We employ the Hungarian Algorithm with A being the set

of cluster indices, B the set of latent concept values V_{c_j}. Since our matrix M^j is a "profit" matrix rather than a cost matrix (larger values in M^j are better), we construct a cost matrix $C = K - M^j$ where K is an integer larger than all entries in M^j. The time complexity of the algorithm is $O\left(\left|V_{c_j}\right|^3\right)$.

6.3 Symbolic Learning

In the previous section, we found the optimal permutation function according to a weak labeller. With this, our symbol grounding function can now be constructed, resulting in a classifier from raw images to latent concept values that uses our perception function as a backbone. For a latent concept c_j, we denote this classifier as P_j.

We can now turn our attention to the reasoning task. Namely, learning the set of rules to compute the downstream label. Our strategy is to use the classifiers P_j to construct a LAS task. Recall the LAS task is of the form (B, S_M, E^+, E^-). The background knowledge B and hypothesis space S_M are assumed to be given. What remains is to construct the sets of positive and negative examples.

Many neuro-symbolic learning tasks involve the classification of samples into a set number of classes. In this case, positive examples are sufficient and can be constructed as follows. For a sample $(\bigotimes_{i=1}^{N} x_i \otimes \bigotimes_{k=1}^{K} s_k, \bigotimes_{l=1}^{L} y_l) \in \mathcal{D}$, each y_l is encoded into the inclusion and exclusion set. The inclusion set encodes the ground truth label for each target concept:

$$e^{inc} = \{\text{holds}\left(n_{y_l}, t_{y_l}\right) | y_l = \left(n_{y_l}, T_{y_l}, t_{y_l}\right), 1 \leq l \leq L\} \tag{6.14}$$

The exclusion encodes all the incorrect labels for each target concept:

$$e^{exc} = \{\text{holds}\left(n_{y_l}, t\right) | y_l = \left(n_{y_l}, T_{y_l}, t_{y_l}\right), 1 \leq l \leq L, t \in T_l, t \neq t_{y_l}\} \tag{6.15}$$

The context encodes information about the features of the sample. Namely, the latent concept values and symbolic inputs:

$$\begin{aligned} e^{ctx} = &\{\text{holds}\left(n_{x_i}, P_j\left(x_i\right)\right) | 1 \leq i \leq N, x_i \text{ is associated with latent concept } c_j\} \\ &\cup \{\text{holds}\left(n_{s_k}, a_{s_k}\right) | s_k = \left(n_{s_k}, S_{s_k}, a_{s_k}\right), 1 \leq k \leq K\} \end{aligned} \tag{6.16}$$

In this way, we are encouraging ILASP to learn a hypothesis that, given the set of latent concept values and symbolic inputs, proves the correct target concept.

Some tasks, such as the Hitting Sets (see the evaluation section), do not offer direct symbolic classification into classes but rather classify in terms of the existence or nonexistence of a solution (realised as the existence or nonexistence of an answer set in ASP). In such cases, both positive and negative examples are used, where positive examples encode samples with a solution while negative examples encode those that do not have one. These samples have the same context as described above, but empty inclusion and exclusion sets (as the nature of the example already encodes the label).

Since the prediction of latent concepts is noisy, it is necessary to give a finite penalty to each example, to allow ILASP to optimise over examples with incorrect context. In this work, we use a uniform penalty for all examples, motivated by the fact that we have no prior information on which images are more difficult to classify than others. It is possible, however, that a non-

Algorithm 6 Embed2Rule Training

Input: $\mathcal{D}$, C, B, S_M
Output: f, $\mathcal{C}$, π, H

Train fully neural model $h = g \circ f$ on $\mathcal{D}$
Select $\mathcal{D}_2 \subseteq \mathcal{D}$
$\mathcal{C} \leftarrow \emptyset$
for $c_j \in C$ **do**
    $\mathcal{D}_j^1 \leftarrow \emptyset$
    for $(\bigotimes x_i \otimes \bigotimes s_k, \bigotimes y_l) \in \mathcal{D}_2$ **do**
        for x_i s.t. c_j is the latent concept of x_i **do**
            Add $f_j(x_i)$ to $\mathcal{D}_j^1$
    Train $\mathcal{C}_j$ on $\mathcal{D}_j^1$ using K-Means Clustering.
    Add $\mathcal{C}_j$ to $\mathcal{C}$
$\pi \leftarrow \emptyset$
for $c_j \in C$ **do**
    $\mathcal{D}_j^2 \leftarrow \emptyset$
    for $z_i \in \mathcal{D}_j^1$ **do**
        Add $\mathcal{C}_j(z_i)$ to $\mathcal{D}_j^2$
    Construct profit matrix M^j from $\mathcal{D}_j^2$.
    Apply the Hungarian Algorithm on $K - M^J$ for some large integer K to receive π_j.
    Add π_j to π
Encode ILASP task $T = (B, S_M, E^+, E^-)$ using $\mathcal{D}_2$, $\mathcal{C}$, π.
Solve T to receive optimal hypothesis H.
return f, $\mathcal{C}$, π, H.

uniform penalty based on some measure of confidence, such as proximity of the datapoint to the cluster centre, would be beneficial. We leave such an investigation to future work. We also note that the data-efficient nature of ILP means that in practice, only a small subset of $\mathcal{D}$ is needed to learn the correct hypothesis. The training algorithm for Embed2Rule is presented in Algorithm 6. The inference algorithm is identical to Algorithm 5.

6.4 Evaluation

We conducted several experiments on existing neuro-symbolic tasks to evaluate the performance and scalability of Embed2Rule[2]. In our experiments, we aim to answer the following questions:

1. Can Embed2Rule solve tasks involving both perception and reasoning while inducing missing symbolic knowledge?
2. How does Embed2Rule compare to both neuro-symbolic learning and reasoning systems in terms of the accuracy of prediction of latent concepts and downstream labels?
3. Can Embed2Rule solve tasks involving complex reasoning, such as NP-hard problems?
4. How well does Embed2Rule scale up to tasks involving a large number of image inputs, compared to other neuro-symbolic systems?
5. How well does Embed2Rule perform in the presence of a large number of latent concepts, compared to other neuro-symbolic systems?

To address these questions, we chose tasks that involve relatively large-scale complex reasoning:

[2]Implementation/data can be found at https://github.com/YanivAspis/Embed2Rule

1. **Hitting Sets** (Cunnington et al., 2023b): A visual version of the hitting set problem, which is NP-Hard (Karp, 1972). The program to be induced involves both choice and constraints and therefore cannot be represented without default negation. This task is used to answer questions 1, 2 & 3. We demonstrate that Embed2Rule can indeed solve this task.

2. **Visual Sudoku Classification** (Augustine et al., 2022): The task is to determine if a given complete Sudoku board is valid or invalid (according to the rules of Sudoku). This requires the correct identification of a large number of images (16 or 81, depending on the size of the board). This task is related to questions 1, 2 & 4. We show that Embed2Rule can successfully scale up to this task while learning the rules of Sudoku.

3. **Follow Suit Winner** (Cunnington et al., 2023a): This task involves a card game in which the winner must be determined. As all cards in the deck are used, the number of latent concept values is quite large (namely, 52 for a standard deck of cards). This task is related to questions 1, 2 & 5. We show that Embed2Rule can successfully deal with this large number of cards while inducing the correct rules of the game.

We compare Embed2Rule to two state-of-the-art neuro-symbolic baselines with existing public codebases that can be easily adapted to perform all three tasks. The Neuro-Symbolic Inductive Learner (NSIL; Cunnington et al., 2023b), to our knowledge, is the only existing system that can both train a perception component and learn general first-order answer set programs (see Section 7.3 for discussion on two other systems, MetaABD and NeuralFastLAS, that can learn first-order logic programs). We also tested against a neuro-symbolic reasoning system, SLASH (Skryagin et al., 2023), where the symbolic knowledge is fully given. Like Embed2Rule and NSIL, SLASH represents its symbolic knowledge in the form of answer set programs and has been shown to be more scalable than other neuro-symbolic reasoning systems, using its SAME method, in which gradients are computed for only a portion of the answer sets that seem most promising. We also compare against SLASH's exact method, which computes gradients for all answer sets.

An interesting ablation study would be to use BLIP-2 directly as the perception network and learning rules using the weak labels. We have already demonstrated in Table 6.1 that the latent accuracy of BLIP-2 is relatively low, so we do not expect this approach to yield better downstream results. In fact, this exact experiment was already performed in Cunnington et al. (2024) on two of the three tasks we consider, showing worse accuracy than we achieve using Embed2Rule.

We report results for downstream accuracy, latent accuracy and training time. We run each method 5 times and report the average to account for random initialisation. If a method did not complete any of the runs within 24 hours, we report a timeout. In all tasks, K-Means is trained with 1000 random samples from the training set, and another 1024 random samples are given to BLIP-2 for weak-labelling. Note that in ILP mode declarations must be tuned carefully as they can affect the performance of the learned hypothesis. This is a general drawback of ILP, not just of our method or ILASP. We employ the same mode declarations other authors used when evaluating on these tasks. In our experiments we used a system with the following specifications:

- **CPU:** Intel i7-12700K @ 3600 MHz.
- **RAM:** 32 GB.
- **GPU:** Nvidia GeForce RTX 3090.

6.4.1 Hitting Sets

In the Hitting Sets task, we are given a universe of elements U, and a set of sets S, where for each set $T \in S$, T is a subset of U. A hitting set for S is a set $H \subseteq U$ such that for every $T \in S$, $T \cap H \neq \emptyset$. Given an integer $k \geq 1$, the hitting set task is to determine if there exists a hitting set H with $|H| \leq k$.

Example 6.4.1. *Consider the universe* $U = \{0, 1, 2, 3, 4\}$ *and* $S = \{\{1\}, \{1, 3\}, \{3\}\}$. *For* $k \geq 2$ *the answer is yes as there exists a hitting set* $H = \{1, 3\}$. *For* $k = 1$ *the answer is no since there is no hitting set of one element.*

The visual version of this task replaces the elements of S with corresponding images, such as MNIST digits (LeCun et al., 2010b). Previously Cunnington et al. (2023b) considered a limited case where $|U| = 5$ and $|\bigcup S| = 4$ (that is, 4 images only). Here we relax this restriction and also consider the cases where $|U| = 10$ and $|\bigcup S| = 4$, 5 or 6. As in Cunnington *et al.*, we set $k = 2$. For $N = |\bigcup S|$, the task has the following specification:

- **Latent concept:** $\left(\text{digit}, U, \mathbb{R}^M\right)$, where either $U = \{0, \ldots, 4\}$ or $U = \{0, \ldots, 9\}$.
- **Raw inputs:** $x_i = \left(\text{element}\,(i), [0,1]^{28\times 28}, \boldsymbol{v}_i\right)$ for $i = 1, \ldots, N$.
- **Symbolic input:** $s = \left(\text{template}, \left\{0, 1, \ldots, 2^{|\cup S|-1}\right\}, a\right)$. "Template" describes the structure of S. That is, how the N images are divided into sets in S. For example, for $|\cup S| = 4$, we define:
  - $a = 0$ means $S = \{\{\cdot, \cdot, \cdot, \cdot\}\}$.
  - $a = 1$ means $S = \{\{\cdot\}, \{\cdot, \cdot, \cdot\}\}$.
  - $a = 2$ means $S = \{\{\cdot, \cdot\}, \{\cdot, \cdot\}\}$.
  - $a = 3$ means $S = \{\{\cdot\}, \{\cdot\}, \{\cdot, \cdot\}\}$.
  - $a = 4$ means $S = \{\{\cdot, \cdot, \cdot\}, \{\cdot\}\}$.
  - $a = 5$ means $S = \{\{\cdot\}, \{\cdot, \cdot\}, \{\cdot\}\}$.
  - $a = 6$ means $S = \{\{\cdot, \cdot\}, \{\cdot\}, \{\cdot\}\}$.
  - $a = 7$ means $S = \{\{\cdot\}, \{\cdot\}, \{\cdot\}, \{\cdot\}\}$.
- **Target concept:** $y = (\text{hitting_set}, \{\text{true}, \text{false}\}, t)$

The dataset For this task, we generated a dataset of 200K training samples, 1K validation samples, and 1K test samples for each case, using the following process: we randomly generated collections of sets of a given size (4-6) with a given number of elements (5 or 10) and checked if a hitting set of size 2 or less exists. We split the generated collections into training, validation and test sets with an equal number of positive and negative samples, ensuring there is no overlap between the sets. Then, we assigned each of the elements in a sample to a random image from MNIST with a label corresponding to that element, and from the appropriate split (train/validation/test). Note that MNIST images may repeat across different samples from the same split. The relatively large dataset size is introduced to account for Embed2Rule's first stage which involves end-to-end differentiable learning without prior knowledge assumed. Later stages of Embed2Rule are data efficient and can be learned using a relatively small sample. While Embed2Rule does not solve the data efficiency issue of neural networks, it does introduce other benefits of neuro-symbolic learning, such as interpretable and generalizable inference, while

```
num_elements(5). super_set_size(4). hitting_set_target_size(2).
element(0). element(1). element(2). element(3). element(4).
index(0..N) :- super_set_size(N+1).
set(S) :- index(S).
pow2(0, 1).
pow2(X, R) :- index(X), pow2(X-1, T), R = 2 * T.
value(0..M) :- super_set_size(N), pow2(N-1, M+1).
num_ones(0, 0).
num_ones(1, 1).
num_ones(X, N) :- value(X), num_ones(X / 2, M),
        num_ones(X \ 2, L), N = M+L.
num_ones_until(T, I, N) :- template(T), index(I), pow2(I, R),
        M = T \ R, num_ones(M, N).
set_assignment(I, S) :- index(I), template(T),
        num_ones_until(T, I, S).
template(T) :- holds(input_template, T).
super_set_entry(I, V) :- holds(input_element(I), V).
```

Listing 6.1: Background knowledge of the Hitting Sets 5-4 task.

Table 6.2: Average downstream test accuracy (%) on the Hitting Sets task. 5-4 means 5 elements in U and 4 images. T/O = "timeout", OOM = "Out of Memory".

Task	5-4	10-4	10-5	10-6
NSIL	**99.6 ± 0.25**	T/O	T/O	T/O
SLASH (exact)	50.0 ± 0.00	OOM	OOM	OOM
SLASH (SAME)	50.0 ± 0.00	OOM	OOM	OOM
Embed2Rule	99.4 ± 0.20	**98.6 ± 1.00**	**96.8 ± 3.13**	**92.9 ± 1.18**

being scalable. As both NSIL and SLASH are more data efficient than Embed2Rule, we used smaller training sets of 10K samples to avoid artificially increasing training time. We found no gain from using additional data for training the baseline systems.

Training details We employ LeNet-5 (Lecun et al., 1998) as a perception network with the output layer changed to size 32 with ReLU activation. The output embeddings are concatenated and fed into the reasoning network, an MLP with residual connections. This MLP is comprised of a stack of residual blocks. Each block is a fully connected layer with an output size of 128, followed by a GELU activation, batch norm and finally adding the block input. We use 4 such residual blocks. The network is trained for 100 epochs (50 in the 5-4 case) with a learning rate of 0.0009 and a batch size of 256 using the Adam optimiser (Kingma and Ba, 2014). The text transformation function for BLIP-2 was simply the digit in numerical form. We set a uniform penalty of 100 for the ILASP examples and set a maximum rule length of 10 (to allow learning long choice rules). We provide ILASP with 50 examples from the training set. The background knowledge and mode declarations are provided in Listings 6.1 and 6.2, respectively. For NSIL, we follow the same hyperparameters used in the official implementation. NSIL is trained for 20 iterations using the SGD optimiser with a learning rate of 0.0008, momentum of 0.7643 and a batch size of 64. The λ value (that affects the weight of neural network confidence in symbolic learning) is set to 1. For SLASH, we report the results after 5 epochs of training.

Results We report results in Tables 6.2, 6.3 and 6.4. Both NSIL and Embed2Rule can solve the task in the case of 5 elements in U and 4 images, with comparable accuracy and correctly inducing the rules of the hitting set problem. However, Embed2Rule can solve the task even in the

```
#modeha(hitting_set_element(const(element))).
#constant(element, 0).
#constant(element, 1).
#constant(element, 2).
#constant(element, 3).
#constant(element, 4).
#minhl(5).
#maxhl(5).
#modeh(set_hit(var(set))).
#modeb(1, hitting_set_element(var(element))).
#modeb(1, super_set_entry(var(index), var(element))).
#modeb(1, set_assignment(var(index), var(set))).
#modeb(1, set_hit(var(set))).
#max_penalty(1000).
```

Listing 6.2: Mode declarations for the Hitting Sets 5-4 task.

Table 6.3: Average latent test accuracy (%) on the Hitting Sets task.

Task	5-4	10-4	10-5	10-6
NSIL	99.4 ± 0.12	T/O	T/O	T/O
SLASH (exact)	22.1 ± 0.00	OOM	OOM	OOM
SLASH (SAME)	22.1 ± 0.00	OOM	OOM	OOM
Embed2Rule	**99.5 ± 0.12**	**89.0 ± 6.50**	**83.9 ± 10.27**	**69.1 ± 6.63**

more challenging cases of 10 elements and 4 or more images, while learning the correct program (note that this program is also general enough to handle any number of raw inputs). NSIL, on the other hand, is unable to complete its bootstrap stage in these cases. The bootstrap stage involves encoding an ILASP task with an example for every possible set S, which is infeasible even for 10 elements and 4 digits. SLASH is not able to solve this task, even in the case of 5 elements, due to its inability to learn from negative samples. The loss function used by SLASH involves a sum over all answer sets that entail the downstream label. However, in the case of negative samples, there are no answer sets. Therefore, only positive samples can be passed to SLASH during training, which are insufficient to learn from, as the network quickly learns to predict the same digit for any given image, resulting in random guess results. Furthermore, SLASH does not scale to the task with 10 elements, as the publicly available implementation quickly runs out of memory when computing gradients in this case, not completing even a single iteration. The hypothesis learned by ILASP is given in Listing 6.3.

6.4.2 Visual Sudoku Classification

Augustine et al. (2022) introduced this version of the visual Sudoku task, inspired by a previous task introduced by Wang et al. (2019). In this task, complete Sudoku boards (all cells filled) are presented in visual form. Specifically, the board is given as a sequence of MNIST digits. The

Table 6.4: Average training time (sec) on the Hitting Sets task.

Task	5-4	10-4	10-5	10-6
NSIL	4117.7	T/O	T/O	T/O
SLASH (exact)	642.0	OOM	OOM	OOM
SLASH (SAME)	156.4	OOM	OOM	OOM
Embed2Rule	957.1	3147.4	6867.5	7754.1

```
:– set_assignment(V1,V2); not set_hit(V2).
set_hit(V1) :– hitting_set_element(V3); super_set_entry(V2,V3);
        set_assignment(V2,V1).
2{hitting_set_element(0); hitting_set_element(1);
        hitting_set_element(2); hitting_set_element(3);
        hitting_set_element(4) }2.
```

Listing 6.3: Induced rules for the Hitting Sets 5-4 task.

task is to classify these boards as valid (no repeats in rows, columns, or blocks) or invalid. We consider three versions of the task:

1. **4x4 Fixed:** Each sample is a 4x4 board with the same digits used across all samples, namely 0-3.

2. **4x4 Random:** Similar to 4x4 fixed, but instead of using the digits 0-3 only, we randomly select 4 digits from 0-9 per puzzle. Hence, the perception task is more challenging than the 4x4 Fixed case while the reasoning task remains the same.

3. **9x9 Fixed:** Similar to 4x4 Fixed but on a 9x9 board (with digits 0-8). This is by far the most challenging task as it involves 81 inputs that need to be classified correctly.

"Fixed" and "Random" correspond to the "Basic" and "PerPuzzle" cases from Augustine *et al.*. Note that this task is more challenging than the related task of visual Sudoku solving, as there is only a single binary label per sample, providing very little information on the contents of each cell. The three versions have the following specifications:

- **Latent concept:**
  - **4x4 Fixed:** $\left(\text{digit}, \{0,1,2,3\}, \mathbb{R}^M\right)$
  - **4x4 Random:** $\left(\text{digit}, \{0,1,\ldots,9\}, \mathbb{R}^M\right)$
  - **9x9 Fixed:** $\left(\text{digit}, \{0,1,\ldots,8\}, \mathbb{R}^M\right)$
- **Raw inputs:**
  - **4x4:** $x_{ij} = \left(\text{cell}(i,j), [0,1]^{28\times 28}, \boldsymbol{v}_{ij}\right)$ where $1 \leq i,j \leq 4$
  - **9x9:** $x_{ij} = \left(\text{cell}(i,j), [0,1]^{28\times 28}, \boldsymbol{v}_{ij}\right)$ where $1 \leq i,j \leq 9$
- **Target concept:** $y = (\text{valid}, \{\text{true}, \text{false}\}, t)$

The dataset For the 4x4 cases (Fixed and Random), we generated 50K training samples, 5K validation samples and 5K test samples. For the 9x9 case, we generated 200K training samples, 20K validation and 20K test samples. When generating samples, we followed a process similar to Augustine et al. (2022). We began by generating valid boards and then corrupted half to generate negative samples. There are two types of corruption: 1. Replacement - a cell is randomly chosen and its content is replaced with a different digit. 2. Substitution - two cells are randomly chosen and their contents are swapped. We apply at least one corruption (at random, with a 50% chance for each type) to each board and then flip a (biased) coin to decide if further corruptions are to be applied. If we get heads, we apply another random corruption and flip again until we get tails. We then checked to see if the multiple corruptions did not accidentally create a valid board. In the 4x4 cases, the coin has a bias of 0 (always landing on tails) so the board contains

Table 6.5: Average downstream test accuracy (%) on the Visual Sudoku Classification task.

Task	4x4 Fixed	4x4 Random	9x9 Fixed
NSIL	50.0 ± 0.00	50.0 ± 0.00	50.0 ± 0.00
SLASH (exact)	98.1 ± 0.37	OOM	OOM
SLASH (SAME)	98.0 ± 0.40	OOM	OOM
Embed2Rule	**99.1 ± 0.37**	**97.8 ± 0.09**	**78.3 ± 3.37**

a single corruption. For the 9x9 Fixed case, we used a bias of 0.75, so a board contains, on average, 4 corruptions. The generated (valid and invalid) puzzles are then split into a training, validation and test set, with an equal number of positive and negative samples, and ensuring no overlap. Assigning MNIST images to cells followed the same procedure as Hitting Sets. As both NSIL and SLASH are more data efficient than Embed2Rule, we used smaller training sets of 10K samples to avoid artificially increasing training time. We found no gain in accuracy, either latent or downstream, from using additional data for training the baseline systems, and therefore find using the smaller dataset to be a more fair reflection of their performance and training time.

Training details We utilise a CNN similar to that used in SATNet (Wang et al., 2019) for the perception network containing 2 convolutional and max-pooling layers, followed by 2 feed-forward layers with ReLU activation functions. In this task, we use a small transformer encoder (Vaswani et al., 2017) (400K parameters, can be trained on local hardware) as the reasoning network, which is particularly suited to model position-based correlations between a large number of image inputs. Such correlations are integral to determining Sudoku board validity. We set the number of attention heads to 4, the number of encoder blocks to 1, the feed-forward layer embedding dimensions to 512, the activation functions to GELU, and a dropout value of 0.1. The inputs and outputs of the transformer are embeddings of dimensions $batch \times seq \times emb$, where $batch$ is the batch size fixed to 512, seq is the sequence length set to 16 and 81 for the 4x4 and 9x9 Sudoku boards, respectively, and emb is the dimensions of the individual latent concept embeddings set to 128. We apply a single-layer classification head with a Sigmoid activation function to the layer-normalised outputs of the transformer to obtain the neural predictions. We train the network for 50 epochs using an AdamW optimiser with a linear warm-up phase from 0.0 to a base learning rate of 0.003 for 30% and 20% of the total number of iterations for the 4x4 and 9x9 boards, respectively, followed by a cosine annealing learning rate schedule between the base learning rate and 0.0001. We clip the gradient norm of each layer at 1.0. The prompt for BLIP-2 is again the digit in numerical form. We give ILASP 100 training examples with a uniform penalty of 1. The background knowledge and mode declarations are provided in Listings 6.4 and 6.5, respectively. We trained SLASH for 20 epochs using the Adam optimiser with a learning rate of 0.001 and a batch size of 100 (for 4x4 random and 9x9 we attempted to use batch sizes as small as 8 to overcome the memory issues, but found it did not help. Due to a bug in the implementation, batch sizes smaller than 8 are not possible.). For NSIL, no set of hyperparameters affected the results (we reported results when using 5 iterations).

Results We report results in Tables 6.5, 6.6 and 6.7. Note we do not use the original dataset of Augustine *et al.* as it contains a very small number of samples (200), which is not sufficient to train the fully neural model. For this reason, we do not directly compare our results to those previously published by other authors on this task.

Embed2Rule can be successfully trained in all cases to achieve high accuracy, even in the

```
element(0). element(1). element(2). element(3). element(4).
element(5). element(6). element(7). element(8). blockSize(3).
cellIdx(1..M) :- blockSize(N), M = N * N.
blockIdx(1..N) :- blockSize(N).
cell((X, Y)) :- cellIdx(X), cellIdx(Y).
block((BX, BY)) :- blockIdx(BX), blockIdx(BY).
sameRow((X, Y), (X, Z)) :- cell((X, Y)), cell((X, Z)), Y != Z.
sameCol((X, Y), (Z, Y)) :- cell((X, Y)), cell((Z, Y)), X != Z.
inBlock((X, Y), (BX, BY)) :- blockSize(N), cell((X, Y)),
        block((BX, BY)), BX = (X - 1) / N + 1,
        BY = (Y - 1) / N + 1.
sameBlock(C1, C2) :- inBlock(C1, B), inBlock(C2, B),
        C1 != C2.
valid_output(true) :- not valid_output(false).
cellVal((X, Y), V) :- holds(cell(X, Y), V).
```

Listing 6.4: Background knowledge of the Visual Sudoku 9x9 task.

```
#constant(valid, false).
#constant(valid, true).
#modeh(valid_output(const(valid))).
#modeb(1, sameRow(var(cell), var(cell))).
#modeb(1, sameCol(var(cell), var(cell))).
#modeb(1, sameBlock(var(cell), var(cell))).
#modeb(2, cellVal(var(cell), var(element))).
```

Listing 6.5: Mode declarations for the Visual Sudoku 9x9 task.

challenging 9x9 Fixed case (the downstream accuracy of 78.3% is the result of compounding errors that occur when predicting 81 latent labels with 99.2% accuracy). In the 4x4 cases, Embed2Rule learns the constraint on rows and columns of a Sudoku board, but not on blocks. This is due to the small number of cases where the restrictions on rows and columns are insufficient, and ILASP's preference for learning shorter programs. For the 9x9 case, ILASP usually learns either the restrictions on rows or columns, as the relatively large number of misclassified board cells means ILASP can cover fewer samples with additional restrictions, making it less preferable to induce them. Regardless, the downstream accuracy remains relatively high. Listings 6.6 and 6.7 provide examples of a hypothesis learned by ILASP for the 4x4 and 9x9 cases, respectively.

NSIL does not succeed in learning in this case. During the bootstrap stage, it learns a "perfectly wrong" hypothesis which assigns board as valid if all digits in a row or column are identical. The perception network then learns to predict all images as the same digits. The corrective examples do not help NSIL in this case, and it proceeds to relearn the same perfectly wrong hypothesis in an endless cycle. We consulted with the author who confirmed this can occur in binary tasks when a perfectly wrong hypothesis is learned. SLASH can solve the 4x4 Fixed case if provided only positive examples (as negative examples yield too many answer sets to be feasible). SLASH then achieves similar accuracy to our system, and the SAME algorithm

Table 6.6: Average latent test accuracy (%) on the Visual Sudoku Classification task.

Task	4x4 Fixed	4x4 Random	9x9 Fixed
NSIL	23.6 ± 0.00	10.1 ± 0.00	11.2 ± 0.00
SLASH (exact)	99.7 ± 0.04	OOM	OOM
SLASH (SAME)	99.7 ± 0.06	OOM	OOM
Embed2Rule	**99.9 ± 0.07**	**99.2 ± 0.07**	**99.3 ± 0.14**

Table 6.7: Average training time (sec) on the Visual Sudoku Classification task.

Task	4x4 Fixed	4x4 Random	9x9 Fixed
NSIL	2887.9	3384.1	45732.8
SLASH (exact)	8067.4	OOM	OOM
SLASH (SAME)	691.0	OOM	OOM
Embed2Rule	3654.4	3794.4	65208.6

```
valid_output(false) :— sameRow(V1,V2); cellVal(V1,V3);
        cellVal(V2,V3).
valid_output(false) :— sameCol(V1,V2); cellVal(V1,V3);
        cellVal(V2,V3).
```

Listing 6.6: Induced rules for the Visual Sudoku 4x4 cases. Only the constraints over rows and columns are learned.

even allows for faster training than Embed2Rule. However, neither NSIL nor SLASH can solve the 4x4 Random or 9x9 Fixed cases as the number of answer sets becomes too large, and the system runs out of memory. Note that even though SAME is more efficient than the exact method overall, it still requires enumerating all answer sets during the first few iterations, when all answer sets are equally likely, hence confronting the same scalability issues as the exact method.

6.4.3 Follow Suit Winner

In the Follow Suit Winner task, we are asked to predict the winner of a card game. The game consists of 4 players, numbered 1 to 4, each drawing a single card from the same deck (no repeats). The winner is the player with the highest-ranked card that has the same suit as the first player. For example, if the cards drawn (in order) are: $(3, \heartsuit)$,$(5, \spadesuit)$,$(J, \heartsuit)$,$(8, \heartsuit)$, the winner is player 3, since they have the same suit as player one ($\heartsuit$) and the highest ranking card (J). For the following cards: $(2, \clubsuit)$,$(9, \heartsuit)$,$(Q, \diamondsuit)$,$(10, \spadesuit)$ the winner is player 1 as no other player drew the same suit ($\clubsuit$). Each card corresponds to two latent concepts: rank and suit. Therefore this is an example of a composite latent concept as discussed in Section 4.1. Since all cards of a standard deck are used, there are 52 latent concepts to learn from only a winner label, making this task challenging. The task has the following specification:

- **Latent concept:** $\left(\text{card}, \{(R,S)|R \in \{2,3,\ldots,10,J,Q,K,A\}, S \in \{\clubsuit,\diamondsuit,\heartsuit,\spadesuit\}\}, \mathbb{R}^M\right)$.
- **Raw inputs:** $x_i = \left(\text{player_card}(i), [0,1]^{112\times112\times3}, \boldsymbol{v}_i\right)$ where $i = 1,2,3,4$ is the index of the player.
- **Target concept:** $y = (\text{winner}, \{1,2,3,4\}, t)$

The dataset We generated 200K training samples, 1K validation and 1K test samples. The generation process is done by continuously simulating games and recording the winner. The

```
valid_output(false) :— sameCol(V1,V2); cellVal(V1,V3);
        cellVal(V2,V3).
```

Listing 6.7: Induced rules for the Visual Sudoku 9x9 case. Only the constraints over columns is learned

```
rank(two). rank(three). rank(four). rank(five). rank(six).
rank(seven). rank(eight). rank(nine). rank(ten). rank(jack).
rank(queen). rank(king). rank(ace). rank_value(two,2).
rank_value(three,3). rank_value(four,4). rank_value(five,5).
rank_value(six,6). rank_value(seven,7). rank_value(eight,8).
rank_value(nine,9). rank_value(ten,10). rank_value(jack,11).
rank_value(queen,12). rank_value(king,13). rank_value(ace,14).
suit(diamonds).
suit(spades).
suit(hearts).
suit(clubs).
rank_higher(R1,R2) :- rank_value(R1,V1), rank_value(R2,V2),
        V1 > V2.
player(1..4).
priority_player(P1, P2) :- player(P1), player(P2), P1 < P2.
player_card(P, R, S) :- holds(player_card(P), R, S).
player_suit(P, S) :- player_card(P, _, S).
player_rank_higher(P1, P2) :- player_card(P1, R1, _),
        player_card(P2, R2, _), rank_higher(R1, R2).
maybe_winner(P) :- player(P), not loses(P).
actually_loses(P1) :- maybe_winner(P1), maybe_winner(P2),
        priority_player(P2, P1).
winner(P) :- maybe_winner(P), not actually_loses(P).
```

Listing 6.8: Background knowledge of the Follow Suit Winner task.

```
#constant(player, 1).
#constant(player, 2).
#constant(player, 3).
#constant(player, 4).
#modeh(loses(var(player))).
#modeb(1, var(suit) != var(suit)).
#modeb(2, player_suit(var(player), var(suit)), (positive)).
#modeb(1, player_suit(const(player), var(suit)), (positive)).
#modeb(1, player_rank_higher(var(player), var(player)),
        (positive)).
```

Listing 6.9: Mode declarations for the Follow Suit Winner task.

process of assigning images to cards is similar to the other tasks. As both NSIL and SLASH are more data efficient than Embed2Rule, we used smaller training sets of 10K samples to avoid artificially increasing training time. We found no gain from using additional data for the baseline systems.

Training details We use a ResNet-18 (He et al., 2016) as the perception network, with an output layer of size 128. As a reasoning network, we use a similar MLP architecture to the Hitting Sets task, with 4 residual blocks and a hidden size of 256. We train for 50 epochs with a learning rate of 0.0001 and a batch size of 128. For BLIP-2 we use a prompt of the format "The playing card *rank* of *suit*" where *rank* and *suit* are replaced with their appropriate values. We provide ILASP with 100 examples with a uniform penalty of 1. The background knowledge and mode declarations are provided in Listings 6.8 and 6.9, respectively.

Results We report results in Table 6.8. Embed2Rule was able to solve this task while the baseline methods failed to do so. In addition, it correctly induced the rules of the game. To investigate the source of errors in the model's latent prediction, we inspected the clusters formed

Table 6.8: Results for Follow Suit Winner. Training time is in seconds.

	Downstream (%)	Latent (%)	Train time
NSIL	OOM	OOM	OOM
SLASH (exact)	OOM	OOM	OOM
SLASH (SAME)	OOM	OOM	OOM
Embed2Rule	**86.2 ± 2.16**	**81.5 ± 2.53**	13219.1

```
loses(V1) :- V2 != V3; player_suit(V1,V2); player_suit(1,V3).
loses(V1) :- player_suit(V1,V3); player_suit(V2,V3);
        player_rank_higher(V2,V1).
```

Listing 6.10: Induced rules for the Follow Suit Winner task.

in latent space and discovered the model was able to produce 52 distinct clusters, i.e. the perception network learned to discriminate card images correctly. However, BLIP-2 had a high source of errors in its weak labelling, resulting in lower latent accuracy when labelling the clusters and consequently affecting downstream performance, despite the correctly learned rules. It is quite likely, however, that future work on vision-language models would produce better weak labellers that could perform better at this task. The hypothesis learned by ILASP is given in Listing 6.10.

As for the baselines, the large number of latent concepts results in a large number of answer sets that is infeasible for those systems. SLASH does not complete even a single iteration before running out of memory. NSIL can perform the bootstrap stage (which does not result in the correct hypothesis), but as it relies on NeurASP (Shakarian et al., 2023) to train its perception component, it encounters the same scalability issues as SLASH. No set of hyperparameters that we tried made any difference on the results.

6.4.4 Discussion

Regarding the questions raised at the beginning of the section, we can now provide the following answers:

1. Embed2Rule can successfully solve tasks involving perception and reasoning, and induce missing symbolic knowledge. In the Hitting Sets task and the Follow Suit winner tasks, we learned the correct ground truth hypothesis, while in the Visual Sudoku task we learned a hypothesis that was optimal with respect to the given dataset.

2. Embed2Rule achieved similar or better accuracy as state-of-the-art neuro-symbolic learning and neuro-symbolic reasoning systems, both in terms of latent concept and downstream prediction, as seen in all three tasks.

3. Embed2Rule can learn complex symbolic knowledge, as seen in the Hitting Sets task, an NP-hard problem. It is worth noting that SLASH, a state-of-the-art neuro-symbolic reasoning system, could not solve the task as its loss function cannot handle cases without answer sets. This suggests there is a wide class of problems that many neuro-symbolic systems may not be able to solve.

4. Embed2Rule can scale to a large number of raw inputs, far beyond baseline systems, as seen in the Visual Sudoku Classification task, where we were able to learn in the presence of 81 raw inputs and achieve good results.

5. Embed2Rule can also scale to a large number of latent concepts. We see this in the Follow Suit Winner task that has 52 latent concepts. Embed2Rule successfully learned to solve the task, while baseline systems could not scale.

Embed2Rule therefore presents a novel and promising direction to neuro-symbolic learning. Embed2Rule builds on the insight of Embed2Sym by pretraining the perception component using a fully neural architecture, rather than interleaving perception training and symbolic learning, as other systems do. This pretraining stage allows for scaling up to a large number of raw inputs and latent concepts. The usage of a weak labeller helps this process further, as it allows the symbolic learner to receive inputs after they have been symbolically grounded, simplifying its learning process. The result is a scalable neuro-symbolic learning system that can take full advantage of the capabilities of a symbolic learner, in our case ILASP. In addition, the modular nature of Embed2Rule allows it to benefit from improvements in any of its components, such as the VLM or the symbolic learner.

While Embed2Rule demonstrates good progress in neuro-symbolic learning, it also has some limitations. The assumption that an image is associated with a single latent concept somewhat limits the scope of the tasks addressable. Although this is mitigated through the notion of a composite latent concept, a system that naturally represents multiple latent concepts per image would be preferable. The approach is not as data-efficient as other neuro-symbolic systems as it relies on the pre-training of a reasoning network, although this can be mitigated with data augmentation and regularisation. The weak-labeller may also underperform on tasks that require highly specific domain knowledge, which can only be overcome using a relatively expensive fine-tuning process. However, fine-tuning on a small number of samples labelled by an expert may be sufficient for our purposes, as Embed2Rule assumes the VLM is a weak-labeller that can underperform on the perception task. In the Future Work section (8.1) of Chapter 8, we will propose some methods for addressing some of these shortcomings.

6.5 Summary

In this chapter, we introduced Embed2Rule, a neuro-symbolic learning system that supports ILP over raw data in a scalable manner. Our approach builds on our earlier work with Embed2Sym by also making use of a neural pre-trainining stage and a clustering stage. We explored briefly a method for jointly learning the symbol grounding process and the symbolic rules, and discussed how such an approach would not be feasible in practice. Instead, we showed a method to learn these two parts sequentially. We make use of a state-of-the-art vision-language model as a weak-labeller to assign cluster labels efficiently, and a state-of-the-art symbolic learner that can robustly deal with noise. Empirical results show that Embed2Rule can solve existing benchmark tasks while learning complex knowledge, and scaling up with respect to the number of raw inputs and latent concepts, infeasible for existing approaches. We then briefly discussed the benefits of the Embed2Rule method and its shortcomings, which we will address in the future.

Chapter 7

Related Work

In this chapter, we provide an overview of work by others that is closely related to our own. We begin by listing works related to Chapter 3 that explore the representation of logical formulas using tensors, and the computation of stable models in vector spaces. We then discuss works related to neuro-symbolic AI. We first go over the neuro-symbolic reasoning case, which is related to Logic Program Layers (Chapter 4) and Embed2Sym (Chapter 5). Next, we cover work related to neuro-symbolic learning, which is related to learning Logic Program Layers (Chapter 4) and Embed2Rule (Chapter 6). We conclude the chapter by listing some additional related work.

7.1 Tensor-Based Semantics for Logic Programs

Historically, a matrix representation of logical operations has been considered as far back as 1948 (Copilowish, 1948). In that work, binary relations are represented as a 0-1 matrix over Boolean Algebra. Logical operations are implemented as their boolean counterpart over matrices. Mizraji (1992) describes matrix representations of certain logical operations, including negation, conjunction and disjunction. This is again visited by Grefenstette (2013), who uses a similar concept to represent predicate calculus. Sato (2017b) extends this even further to support function-free first-order logical formulas in vector spaces. Unlike our approach, these representations do not allow the computation of models for an arbitrary logic program, but Sato showed that for a certain class of Datalog programs this is possible by translating relations to matrix equations (Sato, 2017a). Later work by the author considered abduction in vector spaces (Sato et al., 2018) and implementation and evaluation (Kojima and Sato, 2019), but not supported or stable reasoning.

In Chapter 3, we introduce a novel matrix representation of program reducts, inspired by the work of Sakama et al. (2017). In their original work, the authors described a matrix representation of definite programs, which in our work is a special case of the program reduct representation. They also described a third-order tensor representation of normal programs, but that tensor is exponential in size. Later reformulation of their approach replaced this tensor by a polynomial-sized matrix (Nguyen et al., 2018), but interpretations were instead represented as a matrix of exponential order, although certain optimisation methods have also been evaluated (Sakama et al., 2018; Nguyen et al., 2021a,b). While these algorithms are exact and therefore have a 100% success rate, they also have an exponential time complexity. In contrast, our algorithm is polynomial, but does not guarantee retrieving a stable model, instead we use random restarts until one is found.

Computation of supported models in vector spaces was also considered by Sato et al. (2020),

although the representation is discrete and part of the computation is performed on graphs, unlike our approach which is entirely in vector space. Tensor-based logical abduction has also been realised in Horn propositional programs (Aspis et al., 2018; Quoc et al., 2021; Nguyen et al., 2023), using exponential-sized tensors. Closely related to our work is the work of (Sato and Kojima, 2019), which considers a gradient-based method for logical inference. Their work focuses on abduction, SAT solving and probabilistic inference. Computing supported models is discussed briefly for a restricted class of Datalog programs but the approach is not fully demonstrated or proven. After the publication of the work presented in Chapter 3, several authors continued to explore our idea of computing supported models as cost-minimisation of a differentiable function. One approach defines a different matrix representation of a general normal program to compute supported models using Newton's method (Sato et al., 2023). Another proposes a differentiable SAT solver that proposes a matrix representation of SAT in CNF form, finding solutions to the SAT using Newton's method as well (Sato and Kojima, 2021).

More generally, gradient-based algorithms have been considered for SAT solving using novel neural network architectures (Selsam et al., 2019; Amizadeh et al., 2019; Wang et al., 2019; Shi et al., 2023). These models must be trained on datasets of SAT problems, unlike our approach which requires no training. An answer set solving technique has also been developed which uses gradient-based optimisation to guide the branching of the CDNL algorithm (Nickles, 2018, 2021), the traditional symbolic algorithm for computing stable models. We instead take a completely different approach by computing stable models completely in vector spaces.

Answer Set Networks (Skryagin et al., 2024) maps Answer Set Programs to computational graphs. A message passing algorithm is used to generate potential stable models, with nodes receiving truth values from their neighbours using logical operators, with post-filtering of interpretations once message passing is complete. Aggregates such as choice rules are supported by generating multiple versions of the same program. As programs grow in complexity, so will the graph, returning to exponential memory requirements, but the operation can be parallelised on GPU. To date, Answer Set Networks has only been benchmarked on neuro-symbolic reasoning tasks that are not complex computationally (no positive or negative loops). In contrast, our approach does not require exponential memory, and has been shown to work on tasks involving both positive and negative loops.

7.2 Neuro-Symbolic Reasoning

There is much research on combining neural networks and symbolic reasoning, and approaches vary greatly, see surveys (Besold et al., 2017; Marra et al., 2024; Bhuyan et al., 2024).

As mentioned in the introduction, the difficulty in neuro-symbolic AI is in bridging the continuous representations learned by neural systems and the inherently discrete nature of symbolic representation and processing. The most straightforward approach is to disentangle the symbolic reasoning from the neural perception (Yi et al., 2018). Such an approach pipelines the neural components to the symbolic ones, with minimal interaction. For example, one can sample from the probability distribution predicted by the neural network to feed into the symbolic component (Cunnington et al., 2023a) or use predictions as inputs directly (Surís et al., 2023). While these approaches offer the benefit of simple implementation and flexibility, they also rely on a neural network pretrained directly from latent concept labels, which are often expensive to obtain, especially in specialised domains. Our work does not assume that any latent concept labels are available.

7.2.1 Neural approximation of logical operations

To allow training from downstream labels, many approaches choose to approximate the symbolic reasoning in continuous space. One possibility is to train a neural network to approximate the logical operators.

For example, Lifted Relational Neural Networks (Šourek et al., 2018) learns relations in structured data expressed as First Order Rules. The rules are grounded according to a given example, and a network is constructed for it. There are neurons that represent atoms, facts and rules, and disjunction over rule heads. Facts and rules have learnable weights. Weights are shared for networks across different examples. Neural Logic Networks (Shi et al., 2019) represents propositional atoms as learned embedding vectors, and logical operations as neural modules that are trained to satisfy truth values. Given a propositional formula, a network is dynamically constructed using the learned operator modules.

Neural Theorem Prover (Minervini et al., 2018) is built on a differentiable version of Prolog's backward chaining algorithm to perform differentiable inference in Datalog Knowledge bases. Constant terms are represented as learnable vector embeddings and a soft unification operation is introduced to allow terms and predicates with similar vector representations to match. It builds a computational graph of all possible proof paths (up to a maximum depth), and differentiably computes a score for the proof. The system is trained to prove known facts and not to prove corrupted facts. It also supports the induction of rules with learnable predicate embeddings from templates. To improve scalability, a greedy version of Neural Theorem Prover was introduced (Minervini et al., 2020a). Instead of computing score over all proof paths, it uses the K-nearest neighbours to select the closest facts, and the most likely rules based on how similar the head atom is to the goal. It also allows proofs over natural language by mapping token embeddings to the same space as predicates. Rule induction is also made more scalable by using an attention mechanism.

Recurrent Transformer (Yang et al., 2023b) is an architecture introduced to solve constraint satisfaction problems. The recurrent transformer takes textual or image inputs (after an embedding layer) and passes them through the transformer to predict the values of other variables while satisfying constraints. These constraints are encoded into the loss function used to train the recurrent transformer (in addition to the regular cross-entropy constraint, which is also employed). The same transformer is used recurrently to allow the network to correct predictions at every iteration, improving accuracy.

Approximating logical reasoning using neural networks is indeed a straightforward and scalable approach, and we employ such an approach as the first stage of Embed2Sym (Chapter 5) and Embed2Rule (Chapter 6). However, such approximations come at the cost of interpretability, one of the main advantages of neuro-symbolic AI. The rules implicitly learned by the network are not easy to identify and extract post-training. As such, one cannot verify the correctness of the network or modify its behaviour to correct mistakes. Unlike these works, we only use the approximation as a first step towards a true neuro-symbolic model in which reasoning is completely interpretable.

7.2.2 Fuzzy Approximations of Logic Operators

Instead of learning an approximation of logical operators, one can use fuzzy logic, which has well-known semantics (Zadeh, 1988), resulting in an interpretable reasoning component that is nonetheless fully differentiable.

A prominent example is Logic Tensor Networks (LTN; Badreddine et al., 2022a), which

introduces Real Logic, a continuous formulation of First-Order Logic. Constants are embedded as tensors. Functions and predicates are implemented as neural functions. Logical operators are implemented using fuzzy counterparts (in more recent formulations, these are modified slightly to prevent vanishing/exploding gradients), and quantifiers using differentiable aggregator functions. logLTN (Badreddine et al., 2023b) improves the training stability of LTN by introducing new t-norm and t-conorm operations that operate in log space. It also employs LogMeanExp to increase the smoothness of the disjunction operator along with other numerical stability improvements, resulting in improved performance compared to LTN. A more recent study showed that a focal loss can improve training stability further when classes are unbalanced (Piano et al., 2024). Interval LTN (Badreddine et al., 2023a) formalized Interval Real Logic - a real-valued framework for reasoning about trajectories and events over time, where events are represented as a trapezoid (a four tuple (a, b, c, d) s.t. $a < b < c < d$ with the truth value increasing between a and b, staying constant between b and c and dropping between c and d). Interval LTN is trained to predict the truth value of an event and its trapezoid.

Knowledge Enhanced Neural Networks (KENN; Daniele and Serafini, 2019) employ a *Knowledge Enhancement* layer. This layer boosts the truth value of the output of a linear layer in order to better comply with clauses from a given propositional knowledge base. The clauses too are weighted so their effect can be modified depending on data fit. Satisfaction of the logical formula is computed using fuzzy logic to ensure differentiability. The boosting effect occurs pre-activation so that outputs remain in the interval $[0, 1]$. KENN has been extended to support binary predicates for relational reasoning (Daniele and Serafini, 2022). An additional matrix is constructed to represent the truth values of all pairs in the domain for all predicates. Since now there is a dependency between the truth values of entries in the matrix, the Knowledge Enhancement layer needs to be applied multiple times in a row. This is similar to our stacking of Logic Program Layers in Chapter 4. While the method supports a restricted form of First-Order Logic, it places learnable weights on rules so inference is approximated.

Another approach for adjusting the output of a neural network layer to satisfy a logical formula was proposed by (Daniele et al., 2023b). They analytically derive "refinement functions" that minimally alter the inputs of a fuzzy operation to achieve a given output truth value. An iterative local refinement algorithm computes the minimal changes to the output of a neural layer using a forward and backward pass on the fuzzy logical formula.

Logical Neural Networks (Riegel et al., 2020) dynamically construct a network architecture from a logical formula. Rather than using a fixed fuzzy version of AND and OR, it learns a weighted version with a tailored activation function. The learnable gates output an upper and lower bound on truth values. An Upward-Downward algorithm trains the network end-to-end, so both forward and backwards inference are supported.

Like fuzzy logic, Logic Program Layers does forward reasoning in continuous space. However, we maintain two-valued semantics with guarantees when values are lower than $\gamma^{\perp}$ and higher than $\gamma^{\top}$, rather than using infinite-valued semantics. We are thus much closer to exact inference. In Embed2Sym and Embed2Rule we go even further and replace the differentiable reasoning with an exact symbolic solver with guarantees. In addition, unlike many of these approaches, Logic Program Layer does not require any training from data to realise reasoning in continuous space.

7.2.3 Other Approximations of Differentiable Operations

Besides fuzzy logic, one can also use probabilistic logic as a differentiable form of reasoning. NeuPSL (Pryor et al., 2023) extends Probabilistic Soft Logic to integrate with perception neural

networks by using a deep hinge-loss Markov random field (HL-MRF): a probabilistic graph model that incorporates weighted logical formulas, probabilistic symbolic inputs and neural network predictions. An energy function is defined to train all the model parameters, both neural and symbolic, using gradient descent. DeepPSL (Dasaratha et al., 2023) is a similar approach, which differs in the training algorithm by using a linear approximation of the HL-MRF. It alternates between MAP optimisation of the HL-MRF parameters and a gradient update of the neural weights.

Mixed Integer Linear Programming (MILP) was successfully employed for temporal event detection (Apriceno et al., 2022), that is, recognising a complex event from a sequence of atomic events. The network was trained to recognise both atomic and structured (complex) events from a sequence of images depicting sports events. MILP was used to enforce given temporal constraints between atomic events and a structured event.

Unlike the works above, we focus on representing normal logic programs in neuro-symbolic architectures. A similar approach to our Logic Program Layer was proposed by Takemura and Inoue (2024), several years after the work discussed in Chapter 4. They encode a normal program into a pair of matrices to realise the Immediate Consequence operator. The first matrix performs conjunction and the second disjunction, with a max operation as activation. They formulate two losses using this matrix, one for implication rules, and the second for constraints, which are added to create a joint loss. The authors do not consider programs that require multi-hop reasoning, unlike our own work. They evaluated the approach on MNIST arithmetic tasks only, achieving good results as we did. However, it is likely that results on more complex datasets will encounter some issues, as discussed in Section 4.2.3.

7.2.4 Solver-based Neuro-Symbolic Reasoning

A popular approach for neuro-symbolic reasoning involves using a symbolic solver in order to compute gradients for the parameters of the neural network. Such approaches have the advantage of maintaining the full semantic capability of the symbolic language they choose, while performing exact inference.

One example is DeepProbLog (Manhaeve et al., 2021a), an extension of ProbLog (De Raedt et al., 2007b) to deep learning. ProbLog is a language and solver for probabilistic logic programs. ProbLog allows for the computation of the probability that a formula is true in a given probabilistic program. One component of ProbLog involves the computation of a gradient of a probabilistic formula with respect to a probabilistic atom, using a gradient semi-ring. DeepProbLog leverages this capability to allow the interface of neural networks with probabilistic programs that supports end-to-end training. To do so, DeepProbLog introduces a construct called "Neural Annotated Disjunction" that states that one of n given atoms needs to be true, defining a simplex, which corresponds to the softmax probabilities of a neural network's last layer. The probabilistic logic program would then define how the label is derived from these atoms. By considering all possible proof paths of the ground truth label and using the gradient semi-ring, error information can be transferred to the neural network, and backpropagation proceeds as normal from there.

DeepProbLogA* (Manhaeve et al., 2021b) offers a speedup improvement over DeepProbLog. Instead of considering all possible proof paths, an A* search is performed for the most likely proof path according to a heuristic function. The paper considers several heuristic functions, including one that is based on the geometric mean and another which is implemented by a neural network. To avoid training getting stuck in a suboptimal local minimum, an exploration strategy called

Upper Confidence Bound is employed to encourage diversity in explored proof paths. Results show 2-3x speedup in training compared to DeepProbLog. DeepStochLog (Winters et al., 2022) is an analogous framework that uses Stochastic Definite Clause Grammars rather than probabilistic logic programs, with some speedup.

Scallop (Huang et al., 2021a) introduces a probabilistic version of Datalog. Rather than considering all proof paths, it computes only the top k-proofs (based on probabilities). Then, like DeepProbLog, it performs weighted model counting and computes gradients using a provenance semiring. This allows it to integrate with neural predicates for end-to-end training. It demonstrated greater scalability than DeepProbLog. DeepSoftLog (Maene and Raedt, 2023) is another neuro-symbolic reasoning system based on ProbLog. Unlike DeepProbLog, it represents terms as embedding vectors and uses soft-unification to compute the probabilities of ground atoms. Like DeepProbLog, it also converts formulas to arithmetic circuits for inference, and uses a gradient semiring to compute gradients. In experiments, DeepSoftLog was able to scale up on the MNIST addition task up to 100 digits (this was achieved more than a year after the publication of our work on Embed2Sym, the first neuro-symbolic system to scale up on this task). This is due to a more efficient encoding of the task, which computes each digit of the sum conditioned on the digits in the same position and a carry, which are represented as embeddings. This allows the gradient computation to scale linearly. Nevertheless, this is a unique property of the addition task, which does not apply in general, so we do not expect DeepSoftLog to be scalable in general.

DPNL is another probabilistic neuro-symbolic reasoning system. Rather than computing all possible proof paths as DeepProbLog does, the paper introduces the DPNL algorithm, which is inspired by the DPLL algorithm from SAT solving. The algorithm assumes access to an oracle that when given a partial assignment of raw inputs to latent concepts, can tell which completions will give the desired output. This significantly reduces the number of proof paths that need to be considered, allowing DPNL to scale up to 4 digits in MNIST Addition. An approximate variance of DPNL with bounded errors is also offered.

DeepProbLog was employed for event recognition from a sequence of raw inputs (Apriceno et al., 2021). The system learns from labels of high-level observations over a sequence of data points. A neural component extracts low-level observations at each time point and rules define how low-level observations map to high-level observations over time. Training was performed using DeepProbLog.

Many other approaches choose to employ the Semantic Loss (Xu et al., 2018). This is a loss function that encodes logical constraints. The loss penalises outputs of a neural network if they break any of these constraints, thus encouraging the network not to break them. This can help in regularising the network. However, it does require iterating over all interpretations that break the constraint, so this loss function may not always be tractable.

One system that uses a variation of the Semantic Loss is NeurASP (Shakarian et al., 2023). NeurASP is built over Answer Set Programming, and uses Clingo Gebser et al. (2017) as its symbolic solver. NeurASP, similarly to DeepProbLog, introduces a neural predicate that can take one of n values. This neural predicate is translated into a choice rule in ASP, defining a large number of answer sets (models) of the program. Clingo is used to compute the probability of each such model, with Semantic Loss used to train the neural network.

SLASH (Skryagin et al., 2023) extends NeurASP by integrating probabilistic circuits into neural predicates, allowing for joint probability estimations. It improves scalability by introducing the SAME method for computing gradients, which subsamples answer sets as the perception component becomes more confident, but still requires the enumeration of answer sets at the beginning of training, which is computationally infeasible in large-scale tasks.

Related to these is dPASP (Geh et al., 2024), a library for combining ASP with probabilistic logic and deep learning. Besides supporting stable semantics, it also supports maximum entropy semantics for non-deterministic programs, probabilistic programs with credal semantics among others. Neural integration is done through an implementation of NeurASP.

Abductive Learning (Dai et al., 2019) looks to bridge logic programs and neural networks by means of abduction. The authors formulate the mapping of high-level concepts to latent ones as an abductive task, and use non-differentiable optimisation to identify the minimal changes to the neural network's prediction required to correctly predict the ground truth label. More recently, abductive learning has been extended into ABL-sim that uses vector embedding of latent concepts to guide the training of the neural network (Huang et al., 2021c), which is trained using an additional loss term which encourages clustering for embedding vectors belonging to the same class. Like ABL-Sim, Embed2Sym considers the representation of latent concepts in embedding space. However, in our approach, these embeddings are not used as tools for guiding the training of the perception component, and hence we do not require any additional loss terms. Instead, these embeddings arise naturally when training the neural network, and are used to extract symbolic concepts out of the trained network for downstream symbolic computations.

While Abductive Learning attempts to find the best abductive explanation for the downstream concept, NeuroLog (Tsamoura and Michael, 2020) has gone further and considered *all* abductive solutions to the task, retrieved by a symbolic solver. The set of all abductive solutions are then considered as different models that can be aggregated using Semantic Loss.

Interestingly, it is known that for certain knowledge bases, abductive learning will fail to learn. This has been studied by (Tao et al., 2024), which offered a sufficient condition over the knowledge base for abductive learning to succeed. In particular, it studies tasks where the input is a sequence of images where the location (or index) of the image in a sequence can give a hint to what its contents might be. It considers a matrix of all location-based probabilities of the latent concepts. If the matrix has full rank, the task can be learned reliably.

We consider Embed2Sym to be closely related to the above approaches. Similar to these approaches, Embed2Sym uses an exact symbolic solver for neuro-symbolic inference. We also divide a task into perception and reasoning sub-tasks and enhance a symbolic paradigm, in our case Answer Set Programming, by providing it with a means to reason over unstructured data. Unlike these approaches, we do not use the symbolic component to guide the training of the network. Instead, we use it to extract and label the latent concepts discovered by the network through end-to-end training, unlocking greater scalability.

7.2.5 Neural Approximation in Training with Solver-Based Inference

The approach we take in the latter half of our work, namely, to pre-train an approximate reasoning neural network and use an exact solver at inference, has gained more popularity in recent years, especially following our work.

An earlier example is Neuroplex (Xing et al., 2020). It performs complex temporal reasoning over raw data received from sensors. It also separates the architecture into a perception component (neural network) and a reasoning component. Symbolic reasoning is done through a state machine with constraints, given in advance by a user. A reasoning neural network is pretrained on synthetic data to simulate the symbolic reasoning. During inference, a symbolic solver is used instead.

Yang et al. (2022) propose to replace symbolic computation of gradients by a solver with an encoding of a propositional CNF formula as a matrix. The outputs of a perception neural network

are binarised before being fed into the matrix to guarantee correctness in logical inference. To keep the architecture differentiable, straight-through estimates are employed, as is typical in binary neural networks. This allows significant speedup in training on several neuro-symbolic reasoning benchmark tasks. Our work considers more general logical formulas.

A-Nesi (van Krieken et al., 2023) tackles the scalability issue in weighted model counting, an important aspect of systems like DeepProbLog. Like Neuroplex, they train a neural reasoning component to approximate a given knowledge base. To train this component, they generate samples that map latent concepts to downstream concepts using the given rules and a prior belief over the latent concepts. This approximate reasoner is then used to train the perception component end-to-end. The result is a far more scalable system. They built on many of the insights of Embed2Sym, especially for the encoding of the multi-digit MNIST Addition task.

As mentioned, these systems are closely related to Embed2Sym. However, we do not train the reasoning component on synthetic data. Rather, we let the perception component and reasoning component train together, which allows for latent concepts to emerge naturally. This, for example, can allow the system to discover latent concepts that do not necessarily correspond to the ones pre-specified by the user. Another advantage of our approach is the fact that the rules of the task need not be known in advance before training, an important aspect that allows us to use the Embed2Sym pretraining approach in Embed2Rule to realise neuro-symbolic learning.

Another training strategy has been suggested (Daniele et al., 2024) that builds on the insights of Embed2Sym. Rather than using a clustering algorithm directly, they introduce a neural mapping function for mapping the latent embeddings to latent concept values. In the first phase, they train the system end-to-end similarly to Embed2Sym, with a reasoning network replacing the symbolic solver. In the second phase, they freeze the weights of the perception network, and only train the mapping function, using a neuro-symbolic reasoning system such as DeepProbLog or NeurASP. In this way, training of the neuro-symbolic reasoning system becomes easier.

7.3 Neuro-Symbolic Learning

Most neuro-symbolic works assume the symbolic knowledge is fully known in advance. Among those that learn interpretable rules, most restrict themselves to the propositional case. Some works choose to focus instead on learning rules differentiably (Differentiable ILP). These will typically learn from symbolic data or otherwise assume the perception component is pre-trained. Only a few learn the perception component together with First-Order rules.

7.3.1 Neuro-Symbolic Learning: Propositional

Deep Symbolic Learning (Daniele et al., 2023a) learns a perception component and symbolic propositional rules jointly from downstream labels. The perception network outputs confidence in latent concept values, which are then selected using a policy, in the style of reinforcement learning. It represents rules as a tensor over all combinations of truth values of the latent concepts and the target concept. The final prediction for the downstream concept is computed by taking the vector of confidence values for a given input latent concept, then selecting the value using the policy (greedy or ϵ-greedy). The entire architecture is differentiable and can be trained end-to-end.

E-PNLL (Defresne et al., 2023) combines neural networks with discrete Graphical Models to learn constraints for NP-hard tasks. The network outputs truth values for the discrete variables

(propositions) and the weights are used to represent the constraint. At inference time, a symbolic solver can be used to compute the optimal values of the rest of the variables. For training, an E-PNLL loss is introduced that randomly mutes some of the constraints, improving the ability of the model to learn. E-PNLL was successfully employed to solve Visual Sudoku boards[1].

Li et al. (2023c) introduced a system for learning cardinality constraints together with perception neural networks. As these can be represented as arithmetic constraints, they can be optimised using standard approaches after continuous relaxation. The perception component is trained simultaneously with the constraints using SGD. An annealing process is used at the end to encourage the constraint to become boolean. At inference time, an SMT solver is used for the boolean constraints. The system was shown to be effective in several neuro-symbolic tasks, including Visual Sudoku solving.

SATNet (Wang et al., 2019) uses a layer that represents a MAXSAT task, through which one can backpropagate using a differentiable MAXSAT solver. SATNet can learn the MAXSat problem from data which is either symbolic or raw. For raw data, a perception component can be trained jointly. The parameters of the SATNet layer, however, are not easily interpretable. The original paper reported results on the Visual Sudoku task. However, later it was shown (Li et al., 2023b) that the task contained a *label leakage* problem — the labels from the solved board leaked through the network directly to the perception network, providing a direct supervised signal on latent concepts. When these labels are removed, SATNet failed to solve the Visual Sudoku problem. Topan et al. (2021) addresses the symbol grounding problem in SATNet for tasks like Visual Sudoku Solving. They pre-train the perception component using a clustering objective. A permutation matrix is used to map from clusters to latent concept values, which is trained together with the differentiable MAXSAT layer. This allows SATNet to solve the Visual Sudoku task without intermediate labels. While our approach also looks at clusters in latent space, we do not employ unsupervised training, as it is unlikely to converge to clusters for most datasets.

Li et al. (2023b) address the lack of interpretability of rules learned by SATNet. They reformulate the problem as learning a sparse version of a similarity matrix between variables, using the Iterative Hard Thresholding algorithm. Background knowledge can also be encoded either as symbolic constraints or using an additive weight. They introduce a method for verifying decoded rules against a ground truth SAT formula.

(Wang et al., 2021) propose a framework for learning propositional rules for classification from both categorical and continuous inputs. The architecture is hierarchical with three types of layers: a binarisation layer for discretising continuous inputs, a logical layer that does weighted conjunction and disjunction operations and a linear layer for outputting an average output based on rule importance. To improve training, the authors propose a novel gradient update algorithm and an improved activation function.

Deep Concept Reasoner (Barbiero et al., 2023) represents symbolic concepts as embedding vectors. It generates rules from these concept embeddings using two trained networks. Logical operators are implemented using their fuzzy counterparts. Shorter rules are encouraged during training by introducing special activation functions that put concepts in competition with one another, to only keep the most relevant concepts.

Neural to Circuit (NTOC; Liu et al., 2023) learns probabilistic propositional rules, represented as a logic circuit, together with a perception component. The perception component

[1]Note that this is the Visual Sudoku Solving task, where the goal is to solve a partially filled board. This is different from the Visual Sudoku Classification task we considered in Chapter 6, where the task is to classify if a complete board is valid.

is first pre-trained with a small set of labelled latent concepts. The symbolic reasoner is then trained jointly with the perception network. During training, the logic circuit is analysed with some rules being pruned according to the data. The weights are then adjusted together with the network weights.

Neurosymbolic Decision Trees (Möller et al., 2025) extend Probabilistic (Boolean) Decision Trees with neural networks. The trees are learned using a new algorithm, NeuID3. In addition to the standard tests used for selecting a feature for splitting, NeuID3 can create a neural test by training a neural network to predict the class of a sample conditioned on some of the features. In this way, a decision tree can learn to classify using raw data as features. Note that the same neural network needs to be trained every time it is considered as a test for splitting, resulting in high computational costs. Neurosymbolic Decision Trees can be seen as learning a propositional formula for classification from raw data. It does support the incorporation of First-Order rules as engineered background knowledge, using "neural rule" tests.

While these work induce rules, they restrict themselves to the propositional case. They are therefore more closely related to our approach for learning Logic Program Layers (which are propositional), while Embed2Rule can induce First-Order Programs. Most of these works do not support negation, either, which our work does. While our work on Logic Program Layer predates most of these works, it is unique in representing and learning rules using an interpretable tensor representation, which naturally integrates inside a neural network architecture. Since our work on Logic Program Layer, a closely related approach has been proposed (Sato and Inoue, 2023). The authors build on the matrix representation of normal logic programs (Nguyen et al., 2018) to learn propositional rules, specifically to represent Boolean networks. Normal rules are represented as a pair of binary matrices, one for conjunction and the other for disjunction, with ReLU activation. The matrices are learned from input-output interpretation vectors. It was applied to several symbolic tasks for learning Boolean networks. There are several technical differences between this approach and Logic Program Layer that could warrant further comparison in the future.

More recently, Phua and Inoue (2024) introduced a new version of $\delta LFIT2$ which trains a neural network to predict rules from state transitions in a Boolean network (represented as normal rules). The network is a set transformer. Previous issues are addressed by handling variable permutations better and an improved loss function can handle rules of different body lengths more efficiently.

7.3.2 Differentiable ILP

Recently, Inductive Logic Programming has been realised using differentiable methods.

Neural Logic Machines (NLMs; Dong et al., 2019) encoded ground predicates of any arity in tensors. They use expand and reduce operations to stack tensors of different arity together. These tensors are fed into neural networks that are trained to approximate logical operators. NLMs have been designed to approximate a partial set of Horn clauses. NLMs are not interpretable due to their neural nature, and it is not simple to extract any learned rules. Li et al. (2024) offers an improved training process for Neural Logic Machines with faster convergence.

Differentiable Logic Machines (DLMs; Zimmer et al., 2021) improve upon NLMs by implementing the logical operators using fuzzy logic rather than a trained neural network. By placing weights on predicates rather than rules, DLMs can learn first-order programs that can be extracted at the end of training. Training is done using either supervised learning, reinforcement learning or incremental learning.

∂ILP (Evans and Grefenstette, 2017) is a system for inductive logic programs trained using an end-to-end neural network. The system generates a hypothesis space of definite programs out of a program template and assigns weights to the rules. The optimisation of these weights essentially corresponds to selecting a subset of rules in the hypothesis space. Because even a small program template can result in a rather large hypothesis space, the system has memory-related scaling issues. Inference is made differentiable using a matrix representation of all the rules, which is different from our own.

dNL-ILP (Payani and Fekri, 2019) improves upon ∂ILP by replacing the per-rule weights with boolean flags that are assigned to each body atom per head, similarly to our approach. This reduces the number of parameters significantly compared to ∂ILP, although the complexity of forward chaining still grows exponentially in the number of variables appearing in body predicates. Purgal et al. (2022) improved upon ∂ILP in several ways. They placed weights on literals instead of the rules, greatly decreasing the number of weights, and used generic templates instead of task-specific ones, supporting predicate invention. They also used a different fuzzy operation for forward computation and batch probability.

Hierarchical Rule Induction (HRI; Glanois et al., 2022) is another differentiable ILP system. It introduces proto-rules, an extension of meta-rules for a more compact representation. It can learn Horn clauses with unary or binary predicates. It has a multi-layered architecture with input predicates at layer 0, the target predicate at the end, and auxiliary predicates in the middle. Predicates and meta rules are represented as embedding vectors, with soft unification. It can be trained end-to-end using either supervised learning or reinforcement learning.

NLIL (Yang and Song, 2020) learns definite First-Order rules differentiably, with support for binary predicates in the head. It uses a transformer architecture to learn rules for a given target predicate. The architecture works in three levels: the first level deals with variable assignment in predicates, the second with choosing relevant predicates, the third with combining predicates using logical operators. The attention weights of the transformer are trained from the downstream truth value of the target predicate. These attention weights can be used post-training to extract symbolic rules that correspond to the soft rules the network learned.

αILP (Shindo et al., 2023) can learn First-Order definite rules with binary predicates from images. It takes a similar approach to δILP for learning clauses with some improvements, such as filtering the hypothesis space pre-training using a top-K beam search. Ground facts about image content are given by a pre-trained perception component that extracts objects in an image and their properties. These are then converted to truth-value probabilities using "neural predicate" functions (a fully connected layer). Forward reasoning is done through a tensor representation of the possible clauses, which are weighted. These weights are updated during training to select a final set of clauses. αILP was evaluated on visual reasoning tasks such as CLEVR (Yi et al., 2020) and Kadinsky Patterns (Müller and Holzinger, 2021).

NEUMANN (Shindo et al., 2024) improves upon previous approaches to δILP and αILP by representing logic programs as bi-partite graphs. There are ground atom nodes and conjunction nodes, with edges going from argument ground atoms to their conjunction node, and from the conjunction node to the head atom. Therefore, NEUMANN still only learns definite clauses, but forward chaining inference is now more memory efficient using a message-passing algorithm. The message passing is implemented using fuzzy operations. Like αILP, ground facts can be initialised from a pre-trained perception network. NEUMANN scales better than other differentiable ILP systems, and can scale to more examples compared to symbolic ILP systems.

Conditional Theorem Prover (CTP; Minervini et al., 2020b), unlike Neural Theorem Prover,

learns rules to prove goals rather than from a pre-given knowledge base. The motivation is that it will then be able to learn a minimal set of rules for learning the goal, rather than having to search through a large knowledge base. CTP introduced a select mechanism for choosing among the generated rules in order to prove the goal. The select mechanism is completely differentiable and can be trained end-to-end. ACTP (Morris et al., 2022) extends CTP by allowing a policy to learn which subgoals to choose when trying to prove a goal. This reduces the number of generated rules being considered, improving scalability. The policy is learned through REINFORCE.

Sen et al. (2022) extend Logical Neural Networks to support ILP. This version supports learning definite clauses with binary predicates supporting a given template. From the template and the knowledge base, they generate all the possible grounded predicates that could potentially be proven true. From a target atom, they then construct a neural network with the possible ways to prove it, and place weights on the predicates and logical operators. They enforce the weights to maintain fuzzy logical semantics by adding mathematical constraints using the Minkowski-Weyl theorem. The network is then trained end-to-end to select relevant predicates and proof paths, thus inducing rules.

DFORL (Gao et al., 2024) builds upon $\delta LFIT$ to support learning First-Order Horn clauses with binary predicates. DFORL learns "same-head logic programs", that is, programs where all rule heads are the same predicate. However, it does support some predicate invention. It encodes same-head rules and auxiliary rules (for predicate invention) in tensors, and does forward reasoning in a similar fashion to $\delta LFIT$, through tensor multiplication. In addition to a "semantic loss" that compares output vectors to ground truth values, it also has a "syntactic loss" that looks at the variable placement and their diversity of the learned rule, and it implements curriculum learning. It was able to learn almost all tasks introduced in the ∂ILP paper and could also learn from knowledge graphs.

The motivation behind differentiable ILP works is to increase tolerance to noise and provide better scalability, although usually at the cost of losing optimality and convergence guarantees, and more limited expressibility. In addition, it has been shown that ILASP can tolerate noise as well as differentiable ILP systems Law et al. (2018), which is why we choose to use it in Embed2Rule. Still, the potential of differentiable ILP methods can make them suitable candidates for integration with Embed2Rule in the future. None of these works learn the perception component jointly with the rules, unlike Embed2Rule.

7.3.3 Neuro-Symbolic Learning: First-Order

Few works have jointly trained a perception network while inducing first-order logic programs. MetaABD (Dai and Muggleton, 2021; Huang et al., 2023b,a) extends Meta Interpretive Learning (Muggleton et al., 2015) to learn datalog programs from raw images using an abduction-induction procedure. It performs a search for an optimal hypothesis together with optimal latent concept values for the current neural network predictions and downstream labels. The abduced latent concept values are used as pseudo-labels to train the network. The expressivity of the datalog rules that can be induced by MetaABD does not permit it to learn tasks that require default negation, such as the Hitting Sets task. In addition, the search phase encounters combinatorial difficulties that prevent its application to large-scale tasks. In contrast, our Embed2Rule approach can learn programs that have the full expressivity of ASP and scales up well.

AbdGen (Peng et al., 2024) learns rules from a generative task. It trains a generative encoder-decoder model to generate a sequence of images while inducing First-Order (definite) rules that describe how the images transform at a high level. To support this, the latent factors are

represented as a codebook which can be used to map images to symbolic latent concepts. A contrastive version of MetaAbd (that can learn from both positive and negative examples) is used to induce rules which then predict grounded facts for the next image. These are mapped to a vector representation that is concatenated to the latent vector representation that was given by the encoder. The decoder then transforms these back into an image. The training uses some amount of images labelled with latent concepts to get a more informative encoder. AbdGen learns rules from a sequence of images that MetaAbd was not successful in. (Jin et al., 2025) improve upon AbdGen by introducing a pre-training phase where a policy for selecting a smaller subset of meta rules is learned. This phase trains a neural network from purely symbolic data and labelled latent concept data using Proximal Policy Optimisation. The network consists of self-attention on samples, embeddings for meta rules and cross-attention between them. Metagol is used to evaluate the selected meta-rule to compute a reward. The approach showed improved training times compared to AbdGen and comparable accuracy to handcrafted meta rules. Our work focuses on the discriminative case instead, aiming for much greater scalability. In the future, we may explore the generative case as well.

NSIL (Cunnington et al., 2023b) learns Answer Set Programs from raw data, using either ILASP or FastLAS (Law et al., 2021). It runs a symbolic bootstrap task to generate an initial hypothesis to begin training the perception network using NeurASP. After every epoch, NSIL generates *corrective examples* for ILASP, based on downstream accuracy and network confidence, enabling ILASP to revise the hypothesis if required before training proceeds. NSIL inherits the combinatorial difficulties of NeurASP as well as potential scalability issues in the bootstrap stage and the construction of corrective examples. Embed2Rule avoids these difficulties by adopting the pre-training strategy of Embed2Sym and exploiting a weak-labeller to learn the latent concept values.

NeuralFastLAS (Charalambous et al., 2023) is another neuro-symbolic learning approach that adapts FastLAS2 to support the joint learning of symbolic knowledge and a perception network. However, it inherits the limitations of the FastLAS2 system. Namely, it partially supports non-observational predicate learning and does not support recursive programs or predicate invention. In particular, it cannot learn the Hitting Set rules. As such, we chose not to directly compare to NeuralFastLAS. Embed2Rule uses ILASP, which supports all of these features.

7.4 Additional Related Work

In this section, we list work either related to our own or that is otherwise of interest, and that does not clearly fit in the previous categories.

Concept Bottleneck Models (Koh et al., 2020) break the task of image classification into two steps: concept extraction and concept-based prediction. The idea is that if concepts can be predicted, it not only improves the interpretability of the model but also allows for concept intervention - the user can change the value of a concept and see how it affects the downstream prediction. The authors consider three training methods: training each network separately, training them jointly (loses concept interpretability) or sequentially. It found that joint training performs best but loses the ability to intervene.

Concept Embedding Models (Espinosa Zarlenga et al., 2022) is a newer type of Concept Bottleneck Models that improves accuracy without sacrificing interpretability. Each concept has two vector embeddings: an active and an inactive state. A network predicts the probability that the concept is active and the concept's vector is computed as a weighted average of its two state

vectors. The concatenation of all the concept embedding vectors is passed to a downstream function. This model achieved better accuracy compared to baseline CBM and even neural networks without concepts.

Language Model Guided Concept Bottlenecks (LaBo; Yang et al., 2023a) is able to train concept bottleneck models on images without pre-specifying the concepts or having them labelled. It uses GPT-3 to generate a set of concepts per class. These are further filtered to select the most promising classes. CLIP (Radford et al., 2021) is used as a concept detector in images and a linear function is used to predict the downstream label. LaBo achieved similar accuracy to blackbox models.

Label-Free Concept Bottleneck Models (Oikarinen et al., 2023) is similar to LaBo. It generates concepts using GPT-3 and filters them per class. It computes concept confidence scores with a similarity function between embeddings extracted by a backbone model and by CLIP (essentially using CLIP-dissect on the backbone in order to extract the concepts). It learns a sparse linear matrix for predicting classes from concepts.

In general, concept bottleneck models and derived works do not look to express the downstream function using logical rules. Hence, they are naturally less interpretable than our work.

Recently, it has been pointed out that Neuro-Symbolic works that learn from raw data with a given symbolic knowledge suffer from a "Reasoning Shortcuts" problem (Marconato et al., 2023). It is possible for the perception model to learn wrong labelling of latent concepts despite being entirely consistent with the knowledge base. This is because abduction can have multiple solutions that are consistent, while only one of them applies to any given sample. It is therefore possible to assign wrong latent concepts but still reach a good minimum in the neuro-symbolic loss function. The paper offers some mitigation strategies: altering the knowledge base, multi-task learning, labelling some latent concepts directly, unsupervised learning, changing the architecture of the perception component to predict independent concepts separately, among others.

Bortolotti et al. (2025) explores reasoning shortcuts in neuro-symbolic settings further. It looks at Joint Reasoning Shortcuts - situations where the model does well on downstream predictions but not on latent concept predictions. It explores mitigation strategies in many types of systems - truth tables, probabilistic logic, fuzzy logic, concept bottleneck models and neural. It finds that all proposed mitigation strategies have very little effect, except for labelling the latent concepts.

Given these results and the prevalence of reasoning shortcuts in neuro-symbolic approaches, it is likely that Embed2Sym and Embed2Rule would suffer from a similar problem. In the future, we wish to investigate what effects reasoning shortcuts have on our systems.

NeSyGPT (Cunnington et al., 2024) argues for separating the training of the perception component and the learning of the reasoning module. It takes a pre-trained vision language model (BLIP) and fine-tunes it on a small amount of data labelled with the latent concepts. It then takes the predictions of the fine-tuned BLIP model as inputs into a symbolic learning from answer sets tasks, and uses ILASP to solve it. The results show increased scalability and performance compared to neuro-symbolic systems that train end-to-end, arguing that this simple approach is preferable.

AIMLE (Minervini et al., 2023) is a method for estimating gradients for discrete latent variables when direct computation is intractable. This is interesting for our purposes, because one can estimate the output of a combinatorial solver as such discrete latent variables. This can allow for the computation of gradients through a symbolic solver, suggesting another method for integrating a neural network with symbolic AI.

The apperception engine (Evans et al., 2021; Evans, 2022) learns a causal theory to explain sensory data. It uses a new type of binary neural network, which is implemented as an Answer Set Program. It searches for a minimal Answer Set program to explain the data. The approach has been evaluated on data that is either symbolic or can very easily be represented symbolically. It is unlikely to scale to raw data involving more complex images.

ULLER (van Krieken et al., 2024) is a unfied language for neuro-symbolic AI. It supports various types of semantics: classical, probabilistic, fuzzy and more. It allows designers of neuro-symbolic systems to express knowledge once and apply it across many systems. This reduces the barrier to entering neuro-symbolic research and could potentially facilitate a faster experimentation cycle.

Chapter 8

Conclusion

In this work, we presented a novel approach for realising neuro-symbolic reasoning and learning at scale. In neuro-symbolic tasks, we combine neural networks to extract low-level *latent* symbolic concepts from raw data, and use symbolic modules to perform high-level reasoning. While in the neuro-symbolic *reasoning* setting, the symbolic knowledge is assumed to be fully specified in advance, in neuro-symbolic *learning*, we also induce symbolic knowledge jointly with the neural network parameters. Existing approaches would typically involve approximating logic-based reasoning while sacrificing some of the expressibility of the logic programs or guarantees of correctness. Alternatively, they may maintain full symbolic capabilities at the cost of limited scalability. In our work, we developed scalable approaches that support the full semantics of a declarative symbolic language. We chose to focus on Answer Set Programming as our symbolic language, as it supports constructs useful to express complex reasoning tasks as well as commonsense knowledge. However, we believe that the approaches we developed can easily be adapted to other symbolic paradigms.

We began our work by exploring how normal logic programs can be embedded in vector spaces. To this end, we developed a linear algebraic representation of logic programs and their semantics (Chapter 3). We provided a one-to-one mapping of program interpretations to one-hot vectors. We also showed how normal programs can be represented as matrices using the notion of a program reduct. This matrix representation allowed us to perform logical deduction using linear algebraic operations and simple element-wise non-linearities. We provided a detailed proof of the correctness of the mapping, demonstrating that it maintains exact logical deduction. To move towards neuro-symbolic computations, we replaced the non-differentiable non-linearity with a differentiable one, and allowed the interpretation vectors to hold continuous values. We then showed how to maintain two-valued semantics by enforcing certain mathematical conditions on the approximation and proved the results. We used our differentiable version of the Immediate Consequence operation to introduce a completely novel method for commuting supported and stable models. We define a differentiable function with roots corresponding to supported models, and use a gradient-based method, namely the Newton method, to solve for the root. We performed an initial evaluation to explore and demonstrate the feasibility of our approach, showing that it can compute stable models even for programs that contain positive and negative loops.

We then turned our attention to realising neuro-symbolic computations, using our differentiable approximation of the Immediate Consequence operator (Chapter 4). We introduced some modifications to our matrix representation of normal logic programs to relax some of the conditions imposed in Chapter 3. We proved that our logical deduction is still sound even after these

modifications. We then showed how this matrix representation can be integrated into a neural network architecture as a new type of layer, which we call "Logic Program Layer". A Logic Program Layer represents one step of logical deduction. By stacking these layers, we showed that we can encode any hierarchical normal logic program into a neural network. By stacking these Logic Program Layers on top of a perception network, we can train the architecture end-to-end on neuro-symbolic reasoning tasks. We first evaluated the approach on the Multi-Digit MNIST Addition, showing greater scalability compared to state-of-the-art approaches at the time. We then evaluated the method on the Syllogism task, showing that successful training is dependent on the specific form of the rules. We discussed the implications of these results, suggesting that while the method may work well for some tasks like MNIST Arithmetic, it is unlikely to scale up on others. We also began exploring an "Embedding" approach to neuro-symbolic reasoning tasks, which played a large role in our later work.

We also explored a method for learning normal logic programs using the matrix representation in the Logic Program Layer. We modified an existing learning framework, "Learning from Interpretation Transition", to allow for the differentiable learning of the logic program matrix from vector transitions. We developed a new loss function that can learn normal rules while encouraging a sparse matrix with the shortest rules possible. We discussed the issue of "rule-packing" and steps for mitigation. We performed an initial evaluation to assess the feasibility on several toy tasks, showing that the method successfully learns minimal programs, including rules with default negation. We also evaluated on the Zoo task for animal classification, demonstrating that the method achieves similar accuracy to exact symbolic learners. We discussed the benefits of the approach but also some of its limitations, especially in terms of the expressibility of the programs learned.

Seeking a more scalable approach to neuro-symbolic AI than the matrix-based methods, we began exploring the idea of pre-training a perception network from downstream labels. We begin with the neuro-symbolic reasoning case. The system we developed, Embed2Sym (Chapter 5), constructs a neuro-symbolic model in multiple stages to solve a given task. In the first stage, we build a two-step neural network. The first step is a perception network on raw inputs, while the second performs approximate reasoning to solve the downstream task. We showed that a cluster structure emerges after training that corresponds to the latent concepts of the task. We discussed using K-Means to discover these clusters, and how labelling the clusters corresponds to finding an optimal permutation function on cluster indices. We described an optimisation task for solving for the optimal permutation, and how it can be encoded as an Answer Set Program. Finally, we formally presented the algorithm for training in Embed2Sym and Neuro-Symbolic Inference. Crucially, at inference time we can apply rules for a different downstream task, achieving the generalisation capabilities of neuro-symbolic systems. We evaluated Embed2Sym on several neuro-symbolic benchmarks to show its performance and scalability over existing systems. On the Multi-Digit MNIST Addition task, we were able to scale up to 15 digits, the first neuro-symbolic system that succeeded to learn at this scale. We demonstrated that Embed2Sym can learn with more complex perception tasks with CIFAR-10 Addition. On the Member task, we showed the advantage of using ASP optimisation, as the task cannot be expressed using linear constraints only. Finally, we show that Embed2Sym is not limited only to image data using the Fort Sort task, while also demonstrating that it can induce missing information.

In the final chapter of our work, we shifted focus to the neuro-symbolic learning case. While Embed2Sym achieved great scalability, it still assumed the symbolic knowledge is fully given in advance. We discussed which aspects of Embed2Sym will not work when this assumption is removed. Namely, the optimal permutation function cannot be computed as some of the

rules of the ASP program will be missing. We briefly considered the option of using a symbolic learner, such as ILASP 4, to both learn the rules of the downstream task along with inducing the permutation function, explaining why such an approach will not be practical. We instead chose to employ a Vision-Language Model (VLM) as a weak-labeller of images. BLIP-2 was chosen as the VLM, as it was state-of-the-art at the time, open source and with publicly available weights. The weak labels were used to find the optimal permutation by means of the Hungarian Algorithm. This allowed us to induce an optimal hypothesis for the downstream task using ILASP 4, an ILP system that can handle noise in the data. We evaluated our system, Embed2Rule, on several well-known neuro-symbolic benchmarks. We were able to show that Embed2Rule learned the correct rules and achieved comparable or better accuracy to state-of-the-art neuro-symbolic learning and reasoning systems. In addition, it could do so while scaling up to far more difficult tasks. It was able to handle learning complex knowledge in the Hitting Sets task, scale to at least 81 raw inputs in the Visual Sudoku Classification task, and at least 52 latent concepts in the Follow Suit Winner task, all infeasible for existing systems.

Our work signifies an important step towards neuro-symbolic reasoning and learning that can be used in practice. The scalability achieved by Embed2Sym for neuro-symbolic reasoning was unprecedented at the time. At the time of writing, Embed2Rule still outperforms existing neuro-symbolic learning systems for inducing Answer Set Programs from raw data, to the best of our knowledge. Perhaps their greatest strength is in combining the scalability of neural training with the strong correctness guarantees and interpretability of exact symbolic solvers. The approach they represent, that of pre-training a neural network and lifting the model to neuro-symbolic post-training, shows great potential in carrying neuro-symbolic research forward, as we attempt to construct systems that can learn and reason, can generalise beyond training, while remaining human-interpretable and trustworthy.

8.1 Future Work

Future work on scalable neuro-symbolic reasoning and learning can proceed in several directions.

Extensions of Embed2Sym/Embed2Rule: The first stage of Embed2Sym & Embed2Rule involved training an approximate reasoner. In most experiments, we employed an MLP for this task. This simple network is unlikely to do well in general, and probably contributes to the data inefficiency of the approach. In contrast, we saw that the transformer architecture helps in modelling tasks with sequential constraints, such as Sudoku. It will be beneficial to evaluate the neural training stage on a variety of tasks using different network architectures, such as Graph Neural Networks or Recurrent Neural Networks, to better understand how these architectures affect performance, depending on the task. To improve data efficiency, we can also consider using neuro-symbolic architectures that approximate reasoning, for example Logic Tensor Networks Badreddine et al. (2023b) or the Neural Theorem Prover (Minervini et al., 2020a).

One of the main drawbacks of our approach is the assumption of a single latent concept per raw input. In practice, a raw input may contain multiple objects of interest, and we may want to model several properties for each object as well as relations between objects. The CLEVR dataset Johnson et al. (2016) would be one such example. One approach is to apply some pre-processing to extract objects, such as using object detectors. This would typically require latent concepts to be labelled, however. Several approaches can be considered to solve the problem more directly. First, it is likely that information on multiple latent concepts is still encoded in the latent space in some interpretable way, since this information is required by the reasoning

network. The perception network may use half-planes to store this information, as seen in Figure 5.4, or a projection may give this information. Alternatively, even relational information may be stored in clusters, as seen in figure 4.4. More sophisticated clustering methods may be helpful here too (Ester et al., 1996; Guo et al., 2021). In the future, we would like to explore these possibilities for extracting latent concept information post-training.

When inducing rules in Embed2Rule, we used a uniform penalty on all examples. This may be suboptimal, as the network may be more confident in some of its predictions compared to others. Non-uniform penalties have been explored before in a neuro-symbolic setting (Cunnington et al., 2023a), and we can attempt to adapt such approaches to our case too.

Interpreting Neural Networks: Embed2Sym and Embed2Rule's approach of identifying latent concepts post-training, can be seen as related to the topic of *neural network interpretability*. Methods for post-training interpretations attempt to extract concepts detected by the network on a given input. For example, CLIP-Dissect (Oikarinen and Weng, 2023) employs the pre-trained CLIP model to compare the predictions of various neurons of a network to probe, against a given set of concepts. Mechanistic Interpretability (Bereska and Gavves, 2024) is a field of study that attempts to reverse engineer a neural network, understanding it as a collection of simple computer programs that operate on human-understandable concepts. As these approaches develop, they can greatly benefit our own, as latent concepts become simpler to extract, allowing the application of exact reasoning methods with the flexibility and scalability of neural networks.

Lifting Foundation Models with Symbolic Reasoning Recently, Large Language Models have shown some limited reasoning capabilities. Like most neural approximations, they come with no guarantees and do not generalise out-of-distribution. However, their reasoning capabilities do improve with chain-of-thought prompting (Wei et al., 2022) and with the inclusion of external tools (Shen, 2024). This suggests external symbolic reasoners can be beneficial for LLMs on complex reasoning tasks. A recent line of work enhances LLM reasoning on natural language tasks by extracting symbolic concepts and employing an exact solver (Surís et al., 2023; Dutta et al., 2023; Ishay et al., 2023; Rajasekharan et al., 2023). Applying such an approach in a systematic way to neuro-symbolic tasks, especially for those involving image data, has not yet been explored sufficiently. In Embed2Rule, we have taken some first steps towards this by employing a Vision-Language model as a weak labeller. In the future, we will continue research along these lines, especially as foundation models improve.

Lifting Symbolic Reasoning with Foundation Models Finally, most neuro-symbolic works consider how symbolic methods can improve neural computations. Less common is the opposite view: how can neural networks improve symbolic computations? One of the common drawbacks of Symbolic AI is the need for a large amount of background information, which traditionally needed to be manually engineered. Large Language Models, on the other hand, having been trained on large amounts of information, can be a vast source of information for symbolic systems. Another drawback is the scalability issues of combinatorial search. For example, Inductive Logic Programming systems need a hypothesis space to be specified in order to prune down the search space and make it feasible. The attention mechanism that underlies LLMs can be used as a pruning mechanism as well. It can either select more promising search branches, or specify the most likely search space to begin with (for example, by generating mode declarations). It is likely, however, that some fine-tuning of these models would be required. As an alternative, directly generating answer set programs with LLMs (Coppolillo et al., 2024) may

circumvent the need for symbolic learning if these can be verified to be optimal. One possible research direction, inspired by the recent success of LLMs in inducing programs for reasoning tasks (Chollet, 2025), is to ask the LLM to generate many ASP candidates and evaluate against a given dataset (Cunnington et al., 2024). Although still approximate, this may lead to the generation of symbolic, interpretable knowledge from a few examples, one of the main goals of ILP.

Bibliography

S. F. Ahmed, M. S. B. Alam, M. Hassan, M. R. Rozbu, T. Ishtiak, N. Rafa, M. Mofijur, A. B. M. Shawkat Ali, and A. H. Gandomi. Deep learning modelling techniques: current progress, applications, advantages, and challenges. *Artificial Intelligence Review*, 56(11):13521–13617, Nov 2023. ISSN 1573-7462. doi: 10.1007/s10462-023-10466-8. URL https://doi.org/10.1007/s10462-023-10466-8.

J.-B. Alayrac, J. Donahue, P. Luc, A. Miech, I. Barr, Y. Hasson, K. Lenc, A. Mensch, K. Millicah, M. Reynolds, R. Ring, E. Rutherford, S. Cabi, T. Han, Z. Gong, S. Samangooei, M. Monteiro, J. Menick, S. Borgeaud, A. Brock, A. Nematzadeh, S. Sharifzadeh, M. Binkowski, R. Barreira, O. Vinyals, A. Zisserman, and K. Simonyan. Flamingo: a visual language model for few-shot learning. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc. ISBN 9781713871088.

S. Amizadeh, S. Matusevych, and M. Weimer. Pdp: A general neural framework for learning constraint satisfaction solvers, 2019.

G. Apriceno, A. Passerini, L. Serafini, et al. A neuro-symbolic approach to structured event recognition. *LEIBNIZ INTERNATIONAL PROCEEDINGS IN INFORMATICS*, 206:1101–1114, 2021.

G. Apriceno, A. Passerini, and L. Serafini. A Neuro-Symbolic Approach for Real-World Event Recognition from Weak Supervision. In A. Artikis, R. Posenato, and S. Tonetta, editors, *29th International Symposium on Temporal Representation and Reasoning (TIME 2022)*, volume 247 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 12:1–12:19, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-262-4. doi: 10.4230/LIPIcs.TIME.2022.12. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.TIME.2022.12.

K. R. Apt, H. A. Blair, and A. Walker. Towards a theory of declarative knowledge. In *Foundations of deductive databases and logic programming*, pages 89–148. Elsevier, 1988.

Y. Aspis, K. Broda, and A. Russo. Tensor-based abduction in horn propositional programs. In *ILP Up-and-Coming / Short Papers*, 2018. URL https://api.semanticscholar.org/CorpusID:52910196.

Y. Aspis, K. Broda, A. Russo, and J. Lobo. Stable and Supported Semantics in Continuous Vector Spaces. In *Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning*, pages 59–68, 9 2020. doi: 10.24963/kr.2020/7. URL https://doi.org/10.24963/kr.2020/7.

Y. Aspis, K. Broda, J. Lobo, and A. Russo. Embed2Sym - Scalable Neuro-Symbolic Reasoning via Clustered Embeddings. In *Proceedings of the 19th International Conference on Principles of Knowledge Representation and Reasoning*, pages 421–431, 8 2022. doi: 10.24963/kr.2022/44. URL https://doi.org/10.24963/kr.2022/44.

Y. Aspis, M. Albinhassan, J. Lobo, and A. Russo. *Embed2Rule Scalable Neuro-Symbolic Learning via Latent Space Weak-Labelling*, pages 195–218. 09 2024. ISBN 978-3-031-71166-4. doi: 10.1007/978-3-031-71167-1_11.

S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives. Dbpedia: A nucleus for a web of open data. In K. Aberer, K.-S. Choi, N. Noy, D. Allemang, K.-I. Lee, L. Nixon, J. Golbeck, P. Mika, D. Maynard, R. Mizoguchi, G. Schreiber, and P. Cudré-Mauroux, editors, *The Semantic Web*, pages 722–735, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-76298-0.

E. Augustine, C. Pryor, C. Dickens, J. Pujara, W. Y. Wang, and L. Getoor. Visual sudoku puzzle classification: A suite of collective neuro-symbolic tasks. In A. S. d'Avila Garcez and E. Jiménez-Ruiz, editors, *Proceedings of the 16th International Workshop on Neural-Symbolic Learning and Reasoning as part of the 2nd International Joint Conference on Learning & Reasoning (IJCLR 2022), Cumberland Lodge, Windsor Great Park, UK, September 28-30, 2022*, volume 3212 of *CEUR Workshop Proceedings*, pages 15–29. CEUR-WS.org, 2022. URL https://ceur-ws.org/Vol-3212/paper2.pdf.

S. Badreddine, A. d'Avila Garcez, L. Serafini, and M. Spranger. Logic tensor networks. *Artificial Intelligence*, 303:103649, 2022a. ISSN 0004-3702. doi: https://doi.org/10.1016/j.artint.2021.103649. URL https://www.sciencedirect.com/science/article/pii/S0004370221002009.

S. Badreddine, A. d'Avila Garcez, L. Serafini, and M. Spranger. Logic tensor networks. *Artificial Intelligence*, 303:103649, 2022b. ISSN 0004-3702. doi: https://doi.org/10.1016/j.artint.2021.103649. URL https://www.sciencedirect.com/science/article/pii/S0004370221002009.

S. Badreddine, G. Apriceno, A. Passerini, and L. Serafini. Interval logic tensor networks, 2023a. URL https://arxiv.org/abs/2303.17892.

S. Badreddine, L. Serafini, and M. Spranger. logltn: Differentiable fuzzy logic in the logarithm space, 06 2023b.

P. Barbiero, G. Ciravegna, F. Giannini, M. Espinosa Zarlenga, L. C. Magister, A. Tonda, P. Lio, F. Precioso, M. Jamnik, and G. Marra. Interpretable neural-symbolic concept reasoning. In A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 1801–1825. PMLR, 23–29 Jul 2023. URL https://proceedings.mlr.press/v202/barbiero23a.html.

L. Bereska and S. Gavves. Mechanistic interpretability for AI safety - a review. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856. URL https://openreview.net/forum?id=ePUVetPKu6. Survey Certification, Expert Certification.

T. R. Besold, A. d'Avila Garcez, S. Bader, H. Bowman, P. Domingos, P. Hitzler, K.-U. Kuehnberger, L. C. Lamb, D. Lowd, P. M. V. Lima, L. de Penning, G. Pinkas, H. Poon, and G. Zaverucha. Neural-symbolic learning and reasoning: A survey and interpretation, 2017.

B. P. Bhuyan, A. Ramdane-Cherif, R. Tomar, and T. P. Singh. Neuro-symbolic artificial intelligence: a survey. *Neural Computing and Applications*, 36(21):12809–12844, Jul 2024. ISSN 1433-3058. doi: 10.1007/s00521-024-09960-z. URL https://doi.org/10.1007/s00521-024-09960-z.

C. M. Bishop and N. M. Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

S. Bortolotti, E. Marconato, P. Morettin, A. Passerini, and S. Teso. Shortcuts and identifiability in concept-based models from a neuro-symbolic lens, 2025. URL https://arxiv.org/abs/2502.11245.

M. Bošnjak, T. Rocktäschel, J. Naradowsky, and S. Riedel. Programming with a differentiable forth interpreter. In D. Precup and Y. W. Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 547–556. PMLR, 06–11 Aug 2017. URL https://proceedings.mlr.press/v70/bosnjak17a.html.

R. Bracewell. *The Fourier Transform and Its Applications*. Electrical engineering series. McGraw Hill, 2000. ISBN 9780073039381. URL https://books.google.co.uk/books?id=ZNQQAQAAIAAJ.

R. Calegari, G. Ciatto, E. Denti, and A. Omicini. Logic-based technologies for intelligent systems: State of the art and perspectives. *Information*, 11(3):167, 2020.

T. Charalambous, Y. Aspis, and A. Russo. Neuralfastlas: Fast logic-based learning from raw data, 2023.

K. Cho, B. van Merriënboer, D. Bahdanau, and Y. Bengio. On the properties of neural machine translation: Encoder–decoder approaches. In D. Wu, M. Carpuat, X. Carreras, and E. M. Vecchi, editors, *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, pages 103–111, Doha, Qatar, Oct. 2014. Association for Computational Linguistics. doi: 10.3115/v1/W14-4012. URL https://aclanthology.org/W14-4012/.

F. Chollet. Openai o3 breakthrough high score on arc-agi-pub. https://arcprize.org/blog/oai-o3-pub-breakthrough, 2025. [Accessed 19-04-2025].

K. L. Clark. *Negation as Failure*, pages 293–322. Springer US, Boston, MA, 1978. ISBN 978-1-4684-3384-5. doi: 10.1007/978-1-4684-3384-5_11. URL https://doi.org/10.1007/978-1-4684-3384-5_11.

I. M. Copilowish. Matrix development of the calculus of relations. *The Journal of Symbolic Logic*, 13(4):193–203, 1948. ISSN 00224812. URL http://www.jstor.org/stable/2267134.

E. Coppolillo, F. Calimeri, G. Manco, S. Perri, and F. Ricca. Llasp: Fine-tuning large language models for answer set programming, 2024. URL https://arxiv.org/abs/2407.18723.

A. Cropper and R. Morel. Learning programs by learning from failures. *Machine Learning*, 110(4):801–856, Apr 2021. ISSN 1573-0565. doi: 10.1007/s10994-020-05934-z. URL https://doi.org/10.1007/s10994-020-05934-z.

D. Cunnington, M. Law, J. Lobo, and A. Russo. Ffnsl: feed-forward neural-symbolic learner. *Machine Learning*, 112(2):515–569, 2023a.

D. Cunnington, M. Law, J. Lobo, and A. Russo. Neuro-symbolic learning of answer set programs from raw data. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence*, pages 3586–3596. International Joint Conferences on Artificial Intelligence Organization, 8 2023b.

D. Cunnington, M. Law, J. Lobo, and A. Russo. The role of foundation models in neuro-symbolic learning and reasoning. In T. R. Besold, A. d'Avila Garcez, E. Jimenez-Ruiz, R. Confalonieri, P. Madhyastha, and B. Wagner, editors, *Neural-Symbolic Learning and Reasoning*, pages 84–100, Cham, 2024. Springer Nature Switzerland. ISBN 978-3-031-71167-1.

W.-Z. Dai and S. Muggleton. Abductive knowledge induction from raw data. In Z.-H. Zhou, editor, *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, pages 1845–1851. International Joint Conferences on Artificial Intelligence Organization, 8 2021. doi: 10.24963/ijcai.2021/254. URL https://doi.org/10.24963/ijcai.2021/254. Main Track.

W.-Z. Dai, Q. Xu, Y. Yu, and Z.-H. Zhou. Bridging machine learning and logical reasoning by abductive learning. *Advances in Neural Information Processing Systems*, 32, 2019.

A. Daniele and L. Serafini. Knowledge enhanced neural networks. In A. C. Nayak and A. Sharma, editors, *PRICAI 2019: Trends in Artificial Intelligence*, pages 542–554, Cham, 2019. Springer International Publishing. ISBN 978-3-030-29908-8.

A. Daniele and L. Serafini. Knowledge enhanced neural networks for relational domains. In *International Conference of the Italian Association for Artificial Intelligence*, pages 91–109. Springer, 2022.

A. Daniele, T. Campari, S. Malhotra, and L. Serafini. Deep symbolic learning: Discovering symbols and rules from perceptions. In E. Elkind, editor, *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI-23*, pages 3597–3605. International Joint Conferences on Artificial Intelligence Organization, 8 2023a. doi: 10.24963/ijcai.2023/400. URL https://doi.org/10.24963/ijcai.2023/400. Main Track.

A. Daniele, E. van Krieken, L. Serafini, and F. van Harmelen. Refining neural network predictions using background knowledge. *Machine Learning*, 112(9):3293–3331, Sep 2023b. ISSN 1573-0565. doi: 10.1007/s10994-023-06310-3. URL https://doi.org/10.1007/s10994-023-06310-3.

A. Daniele, T. Campari, S. Malhotra, and L. Serafini. Simple and effective transfer learning for neuro-symbolic integration, 2024. URL https://arxiv.org/abs/2402.14047.

S. Dasaratha, S. A. Puranam, K. S. Phogat, S. R. Tiyyagura, and N. P. Duffy. Deeppsl: End-to-end perception and reasoning. In E. Elkind, editor, *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI-23*, pages 3606–3614. International Joint Conferences on Artificial Intelligence Organization, 8 2023. doi: 10.24963/ijcai.2023/401. URL https://doi.org/10.24963/ijcai.2023/401. Main Track.

L. De Raedt, A. Kimmig, and H. Toivonen. Problog: A probabilistic prolog and its application in link discovery. In *IJCAI*, volume 7, pages 2462–2467. Hyderabad, 2007a.

L. De Raedt, A. Kimmig, and H. Toivonen. Problog: A probabilistic prolog and its application in link discovery. In *IJCAI*, volume 7, pages 2462–2467. Hyderabad, 2007b.

M. Defresne, S. Barbe, and T. Schiex. Scalable coupling of deep learning with logical reasoning. In E. Elkind, editor, *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI-23*, pages 3615–3623. International Joint Conferences on Artificial Intelligence Organization, 8 2023. doi: 10.24963/ijcai.2023/402. URL https://doi.org/10.24963/ijcai.2023/402. Main Track.

H. Dong, J. Mao, T. Lin, C. Wang, L. Li, and D. Zhou. Neural logic machines, 2019.

A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=YicbFdNTTy.

D. Dua and C. Graff. UCI machine learning repository, 2017. URL http://archive.ics.uci.edu/ml.

S. Dutta, J. Singh, I. Pandey, S. Manchanda, S. Chakrabarti, and T. Chakraborty. Frugal lms trained to invoke symbolic solvers achieve parameter-efficient arithmetic reasoning, 2023. URL https://arxiv.org/abs/2312.05571.

M. Espinosa Zarlenga, P. Barbiero, G. Ciravegna, G. Marra, F. Giannini, M. Diligenti, Z. Shams, F. Precioso, S. Melacci, A. Weller, P. Lió, and M. Jamnik. Concept embedding models: Beyond the accuracy-explainability trade-off. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 21400–21413. Curran Associates, Inc., 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/867c06823281e506e8059f5c13a57f75-Paper-Conference.pdf.

M. Ester, H.-P. Kriegel, J. Sander, X. Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, volume 96, pages 226–231, 1996.

R. Evans. The apperception engine. In H. Kim and D. Schönecker, editors, *Kant and Artificial Intelligence*, pages 39–104. De Gruyter, 2022.

R. Evans and E. Grefenstette. Learning explanatory rules from noisy data. *Journal of Artificial Intelligence Research*, 61, 11 2017. doi: 10.1613/jair.5477.

R. Evans, M. Bošnjak, L. Buesing, K. Ellis, D. Pfau, P. Kohli, and M. Sergot. Making sense of raw input. *Artificial Intelligence*, 299(C):103521, 2021. doi: 10.1016/j.artint.2021.103521.

B. Everitt, S. Landau, M. Leese, D. Stahl, and a. O. M. C. Safari. *Cluster Analysis, 5th Edition*. John Wiley & Sons, 2011. URL https://books.google.co.uk/books?id=3RFPzQEACAAJ.

F. Fages. *A new fixpoint semantics for general logic programs compared with the well-founded and the stable model semantics*, page 442–458. MIT Press, Cambridge, MA, USA, 1990. ISBN 0262730901.

A. Falkner, G. Friedrich, K. Schekotihin, R. Comploi-Taupe, and E. Teppan. Industrial applications of answer set programming. *KI - Künstliche Intelligenz*, 32:1–12, 06 2018. doi: 10.1007/s13218-018-0548-6.

C. Fellbaum, editor. *WordNet: An Electronic Lexical Database*. Language, Speech, and Communication. MIT Press, Cambridge, MA, 1998. ISBN 978-0-262-06197-1.

D. Fierens, G. Van den Broeck, J. Renkens, D. Shterionov, B. Gutmann, I. Thon, G. Janssens, and L. De Raedt. Inference and learning in probabilistic logic programs using weighted boolean formulas. *Theory and Practice of Logic Programming*, 15(3):358–401, 2015.

K. Fukushima and S. Miyake. Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In S.-i. Amari and M. A. Arbib, editors, *Competition and Cooperation in Neural Nets*, pages 267–285, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg. ISBN 978-3-642-46466-9.

K. Gao, K. Inoue, Y. Cao, and H. Wang. A differentiable first-order rule learner for inductive logic programming. *Artificial Intelligence*, 331:104108, 2024. ISSN 0004-3702. doi: https://doi.org/10.1016/j.artint.2024.104108. URL `https://www.sciencedirect.com/science/article/pii/S0004370224000444`.

A. d. Garcez and L. C. Lamb. Neurosymbolic ai: the 3rd wave. *Artificial Intelligence Review*, 56(11):12387–12406, Nov 2023. ISSN 1573-7462. doi: 10.1007/s10462-023-10448-w. URL `https://doi.org/10.1007/s10462-023-10448-w`.

A. S. d. Garcez, D. M. Gabbay, and K. B. Broda. *Neural-Symbolic Learning System: Foundations and Applications*. Springer-Verlag, Berlin, Heidelberg, 2002. ISBN 1852335122.

M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.

M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. Multi-shot ASP solving with clingo. *CoRR*, abs/1705.09811, 2017.

R. L. Geh, J. Gonçalves, I. C. Silveira, D. D. Mauá, and F. G. Cozman. dpasp: a probabilistic logic programming environment for neurosymbolic learning and reasoning. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, volume 21, pages 731–742, 2024.

M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski, Bowen, and Kenneth, editors, *Proceedings of International Logic Programming Conference and Symposium*, pages 1070–1080. MIT Press, 1988. URL `http://www.cs.utexas.edu/users/ai-lab?gel88`.

C. Glanois, Z. Jiang, X. Feng, P. Weng, M. Zimmer, D. Li, W. Liu, and J. Hao. Neuro-symbolic hierarchical rule induction. In K. Chaudhuri, S. Jegelka, L. Song, C. Szepesvari, G. Niu, and S. Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 7583–7615. PMLR, 17–23 Jul 2022. URL `https://proceedings.mlr.press/v162/glanois22a.html`.

I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

E. Grefenstette. Towards a formal distributional semantics: Simulating logical calculi with tensors. In *Second Joint Conference on Lexical and Computational Semantics (*SEM), Volume 1: Proceedings of the Main Conference and the Shared Task: Semantic Textual Similarity*,

pages 1–10, Atlanta, Georgia, USA, June 2013. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/S13-1001.

W. Guo, K. Lin, and W. Ye. Deep embedded k-means clustering. In *2021 International Conference on Data Mining Workshops (ICDMW)*, pages 686–694, Los Alamitos, CA, USA, dec 2021. IEEE Computer Society. doi: 10.1109/ICDMW53433.2021.00090. URL https://doi.ieeecomputersociety.org/10.1109/ICDMW53433.2021.00090.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9:1735–1780, 11 1997. doi: 10.1162/neco.1997.9.8.1735.

C. J. Hogger. *Essentials of Logic Programming*. Oxford University Press, 12 1990. ISBN 9780198538202. doi: 10.1093/oso/9780198538202.001.0001. URL https://doi.org/10.1093/oso/9780198538202.001.0001.

K. Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4 (2):251–257, 1991.

J. Huang, Z. Li, B. Chen, K. Samel, M. Naik, L. Song, and X. Si. Scallop: From probabilistic deductive databases to scalable differentiable reasoning. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 25134–25145. Curran Associates, Inc., 2021a. URL https://proceedings.neurips.cc/paper_files/paper/2021/file/d367eef13f90793bd8121e2f675f0dc2-Paper.pdf.

Y.-X. Huang, W.-Z. Dai, L.-W. Cai, S. Muggleton, and Y. Jiang. Fast abductive learning by similarity-based consistency optimization. *Advances in Neural Information Processing Systems*, 34, 2021b.

Y.-X. Huang, W.-Z. Dai, L.-W. Cai, S. H. Muggleton, and Y. Jiang. Fast abductive learning by similarity-based consistency optimization. *Advances in Neural Information Processing Systems*, 34:26574–26584, 2021c.

Y.-X. Huang, W.-Z. Dai, Y. Jiang, and Z. Zhou. Enabling knowledge refinement upon new concepts in abductive learning. In *AAAI Conference on Artificial Intelligence*, 2023a. URL https://api.semanticscholar.org/CorpusID:259731271.

Y.-X. Huang, Z. Sun, G. Li, X. Tian, W.-Z. Dai, W. Hu, Y. Jiang, and Z.-H. Zhou. Enabling abductive learning to exploit knowledge graph. In E. Elkind, editor, *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI-23*, pages 3839–3847. International Joint Conferences on Artificial Intelligence Organization, 8 2023b. doi: 10.24963/ijcai.2023/427. URL https://doi.org/10.24963/ijcai.2023/427. Main Track.

K. Inoue, T. Ribeiro, and C. Sakama. Learning from interpretation transition. *Machine Learning*, 94, 01 2014. doi: 10.1007/s10994-013-5353-8.

A. Ishay, Z. Yang, and J. Lee. Leveraging large language models to generate answer set programs. In *Proceedings of the 20th International Conference on Principles of Knowledge Representation and Reasoning*, KR '23, 2023. ISBN 978-1-956792-02-7. doi: 10.24963/kr.2023/37. URL https://doi.org/10.24963/kr.2023/37.

Y. Jin, J. Liu, Z. Luo, Y. Peng, Z. Qin, W.-Z. Dai, Y.-X. Ding, and K. Zhou. Pre-training meta-rule selection policy for visual generative abductive learning, 2025. URL https://arxiv.org/abs/2503.06427.

J. Johnson, B. Hariharan, L. van der Maaten, L. Fei-Fei, C. L. Zitnick, and R. Girshick. Clevr: A diagnostic dataset for compositional language and elementary visual reasoning, 2016.

R. M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972. ISBN 978-1-4684-2001-2. doi: 10.1007/978-1-4684-2001-2_9. URL https://doi.org/10.1007/978-1-4684-2001-2_9.

D. Kingma and J. Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 12 2014.

P. W. Koh, T. Nguyen, Y. S. Tang, S. Mussmann, E. Pierson, B. Kim, and P. Liang. Concept bottleneck models. In *International Conference on Machine Learning*, pages 5338–5348. PMLR, 2020.

R. Kojima and T. Sato. A tensorized logic programming language for large-scale data. *arXiv e-prints*, art. arXiv:1901.08548, Jan. 2019.

R. Krishna, Y. Zhu, O. Groth, J. Johnson, K. Hata, J. Kravitz, S. Chen, Y. Kalantidis, L.-J. Li, D. A. Shamma, M. S. Bernstein, and L. Fei-Fei. Visual genome: Connecting language and vision using crowdsourced dense image annotations. *International Journal of Computer Vision*, 123(1):32–73, May 2017. ISSN 1573-1405. doi: 10.1007/s11263-016-0981-7. URL https://doi.org/10.1007/s11263-016-0981-7.

A. Krizhevsky and G. Hinton. Learning multiple layers of features from tiny images. Technical report, University of Toronto, Toronto, Ontario, 2009.

A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017. ISSN 0001-0782. doi: 10.1145/3065386. URL https://doi.org/10.1145/3065386.

H. W. Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

G. Lample and F. Charton. Deep learning for symbolic mathematics. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL https://openreview.net/forum?id=S1eZYeHFDS.

M. Law. Conflict-driven inductive logic programming. *Theory and Practice of Logic Programming*, 23(2):387–414, 2023.

M. Law, A. Russo, and K. Broda. Inductive learning of answer set programs. In E. Fermé and J. Leite, editors, *Logics in Artificial Intelligence*, pages 311–325, Cham, 2014. Springer International Publishing. ISBN 978-3-319-11558-0.

M. Law, A. Russo, and K. Broda. Inductive learning of answer set programs from noisy examples. *arXiv preprint arXiv:1808.08441*, 2018.

M. Law, A. Russo, and K. Broda. The ilasp system for inductive learning of answer set programs. *arXiv preprint arXiv:2005.00904*, 2020.

M. Law, A. Russo, K. Broda, and E. Bertino. Scalable non-observational predicate learning in asp. In Z.-H. Zhou, editor, *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, pages 1936–1943. International Joint Conferences on Artificial Intelligence Organization, 8 2021. doi: 10.24963/ijcai.2021/267. URL https://doi.org/10.24963/ijcai.2021/267. Main Track.

M. Law, K. Broda, and A. Russo. Search space expansion for efficient incremental inductive logic programming from streamed data. In L. D. Raedt, editor, *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, pages 2697–2704. International Joint Conferences on Artificial Intelligence Organization, 7 2022. doi: 10.24963/ijcai.2022/374. URL https://doi.org/10.24963/ijcai.2022/374. Main Track.

Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. doi: 10.1109/5.726791.

Y. LeCun, C. Cortes, and C. Burges. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010a.

Y. LeCun, C. Cortes, and C. Burges. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010b.

H. Li, Z. Xu, G. Taylor, C. Studer, and T. Goldstein. Visualizing the loss landscape of neural nets. *Advances in neural information processing systems*, 31, 2018.

J. Li, D. Li, S. Savarese, and S. Hoi. BLIP-2: bootstrapping language-image pre-training with frozen image encoders and large language models. In *ICML*, 2023a.

Z. Li, J. Guo, Y. Jiang, and X. Si. Learning reliable logical rules with satnet. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 14837–14847. Curran Associates, Inc., 2023b. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/2ff46d83d1dcc063e075058b29d55efe-Paper-Conference.pdf.

Z. Li, Y. Huang, Z. Li, Y. Yao, J. Xu, T. Chen, X. Ma, and J. Lu. Neuro-symbolic learning yielding logical constraints. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023c.

Z. Li, Y. Cao, Y. ZHENG, X. Liu, B. Wu, T. Li, X. Xu, J. Jiang, Y. S. Teo, S.-W. Lin, and Y. Liu. Improving neural logic machines via failure reflection. In *Forty-first International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=JObct1zyTb.

T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. Microsoft coco: Common objects in context. In D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, editors, *Computer Vision – ECCV 2014*, pages 740–755, Cham, 2014. Springer International Publishing. ISBN 978-3-319-10602-1.

A. Liu, H. Xu, G. Van den Broeck, and Y. Liang. Out-of-distribution generalization by neural-symbolic joint training. *Proceedings of the AAAI Conference on Artificial Intelligence*, 37 (10):12252–12259, Jun. 2023. doi: 10.1609/aaai.v37i10.26444. URL https://ojs.aaai.org/index.php/AAAI/article/view/26444.

J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, volume 5, pages 281–298. University of California press, 1967.

J. Maene and L. D. Raedt. Soft-unification in deep probabilistic logic. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=s86M8naPSv.

R. Manhaeve, S. Dumančić, A. Kimmig, T. Demeester, and L. D. Raedt. Neural probabilistic logic programming in deepproblog, 2019.

R. Manhaeve, S. Dumančić, A. Kimmig, T. Demeester, and L. De Raedt. Neural probabilistic logic programming in deepproblog. *Artificial Intelligence*, 298:103504, 2021a. ISSN 0004-3702. doi: https://doi.org/10.1016/j.artint.2021.103504. URL https://www.sciencedirect.com/science/article/pii/S0004370221000552.

R. Manhaeve, G. Marra, and L. De Raedt. Approximate inference for neural probabilistic logic programming. In *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning*, pages 475–486. IJCAI Organization, 2021b.

E. Marconato, S. Teso, A. Vergari, and A. Passerini. Not all neuro-symbolic concepts are created equal: Analysis and mitigation of reasoning shortcuts. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 72507–72539. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/e560202b6e779a82478edb46c6f8f4dd-Paper-Conference.pdf.

G. Marra, S. Dumančić, R. Manhaeve, and L. De Raedt. From statistical relational to neurosymbolic artificial intelligence: A survey. *Artificial Intelligence*, 328:104062, 2024. ISSN 0004-3702. doi: https://doi.org/10.1016/j.artint.2023.104062. URL https://www.sciencedirect.com/science/article/pii/S0004370223002084.

P. Minervini, M. Bosnjak, T. Rocktäschel, and S. Riedel. Towards neural theorem proving at scale. *arXiv preprint arXiv:1807.08204*, 2018.

P. Minervini, M. Bošnjak, T. Rocktäschel, S. Riedel, and E. Grefenstette. Differentiable reasoning on large knowledge bases and natural language. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 5182–5190, 2020a.

P. Minervini, S. Riedel, P. Stenetorp, E. Grefenstette, and T. Rocktäschel. Learning reasoning strategies in end-to-end differentiable proving. In *International Conference on Machine Learning*, pages 6938–6949. PMLR, 2020b.

P. Minervini, L. Franceschi, and M. Niepert. Adaptive perturbation-based gradient estimation for discrete latent variable models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 9200–9208, 2023.

E. Mizraji. Vector logics: The matrix-vector representation of logical calculus. *Fuzzy Sets and Systems*, 50(2):179 – 185, 1992. ISSN 0165-0114. doi: https://doi.org/10.1016/0165-0114(92)90216-Q. URL http://www.sciencedirect.com/science/article/pii/016501149290216Q.

M. Morris, P. Minervini, and P. Blunsom. Learning proof path selection policies in neural theorem proving. 2022. URL http://ceur-ws.org/Vol-3212/paper5.pdf.

S. Muggleton. Inductive logic programming. *New generation computing*, 8:295–318, 1991.

S. Muggleton. Inverse entailment and progol. *New Generation Computing*, 13(3):245–286, Dec 1995. ISSN 1882-7055. doi: 10.1007/BF03037227. URL https://doi.org/10.1007/BF03037227.

S. H. Muggleton, D. Lin, N. Pahlavi, and A. Tamaddoni-Nezhad. Meta-interpretive learning: application to grammatical inference. *Machine learning*, 94:25–49, 2014.

S. H. Muggleton, D. Lin, and A. Tamaddoni-Nezhad. Meta-interpretive learning of higher-order dyadic datalog: Predicate invention revisited. *Machine Learning*, 100(1):49–73, 2015.

M. Möller, A. Norlander, P. Z. D. Martires, and L. D. Raedt. Neurosymbolic decision trees, 2025. URL https://arxiv.org/abs/2503.08762.

H. Müller and A. Holzinger. Kandinsky patterns. *Artificial Intelligence*, 300:103546, 2021. ISSN 0004-3702. doi: https://doi.org/10.1016/j.artint.2021.103546. URL https://www.sciencedirect.com/science/article/pii/S0004370221000977.

H. D. Nguyen, C. Sakama, T. Sato, and K. Inoue. Computing logic programming semantics in linear algebra. In M. Kaenampornpan, R. Malaka, D. D. Nguyen, and N. Schwind, editors, *Multi-disciplinary Trends in Artificial Intelligence*, pages 32–48, Cham, 2018. Springer International Publishing.

H. D. Nguyen, C. Sakama, T. Sato, and K. Inoue. An efficient reasoning method on logic programming using partial evaluation in vector spaces. *Journal of Logic and Computation*, 03 2021a. ISSN 0955-792X. doi: 10.1093/logcom/exab010. URL https://doi.org/10.1093/logcom/exab010. exab010.

T. Nguyen, K. Inoue, and C. Sakama. Enhancing linear algebraic computation of logic programs using sparse representation. *New Generation Computing*, 40, 12 2021b. doi: 10.1007/s00354-021-00142-2.

T. Nguyen, K. Inoue, and C. Sakama. Linear algebraic abduction with partial evaluation. In *Practical Aspects of Declarative Languages: 25th International Symposium, PADL 2023, Boston, MA, USA, January 16–17, 2023, Proceedings*, page 197–215, Berlin, Heidelberg, 2023. Springer-Verlag. ISBN 978-3-031-24840-5. doi: 10.1007/978-3-031-24841-2_13. URL https://doi.org/10.1007/978-3-031-24841-2_13.

M. Nickles. Differentiable satisfiability and differentiable answer set programming for sampling-based multi-model optimization, 2018.

M. Nickles. diff-sat – a software for sampling and probabilistic reasoning for sat and answer set programming, 2021. URL https://arxiv.org/abs/2101.00589.

T. Oikarinen and T.-W. Weng. CLIP-dissect: Automatic description of neuron representations in deep vision networks. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=iPWiwWHc1V.

T. Oikarinen, S. Das, L. M. Nguyen, and T.-W. Weng. Label-free concept bottleneck models, 2023. URL https://arxiv.org/abs/2304.06129.

A. Payani and F. Fekri. Inductive logic programming via differentiable deep neural logic networks, 2019.

Y. Peng, Y. Jin, Z. Luo, Y.-X. Ding, W.-Z. Dai, Z. Ren, and K. Zhou. Generating by understanding: Neural visual generation with logical symbol groundings, 2024. URL https://arxiv.org/abs/2310.17451.

Y. J. Phua and K. Inoue. Variable assignment invariant neural networks for learning logic programs. In T. R. Besold, A. d'Avila Garcez, E. Jimenez-Ruiz, R. Confalonieri, P. Madhyastha, and B. Wagner, editors, *Neural-Symbolic Learning and Reasoning*, pages 47–61, Cham, 2024. Springer Nature Switzerland. ISBN 978-3-031-71167-1.

L. Piano, F. Manigrasso, A. Russo, and L. Morra. Enhancing neuro-symbolic integration with focal loss: A study on logic tensor networks. In *Neural-Symbolic Learning and Reasoning: 18th International Conference, NeSy 2024, Barcelona, Spain, September 9–12, 2024, Proceedings, Part II*, page 14–23, Berlin, Heidelberg, 2024. Springer-Verlag. ISBN 978-3-031-71169-5. doi: 10.1007/978-3-031-71170-1_2. URL https://doi.org/10.1007/978-3-031-71170-1_2.

S. Pouyanfar, S. Sadiq, Y. Yan, H. Tian, Y. Tao, M. P. Reyes, M.-L. Shyu, S.-C. Chen, and S. S. Iyengar. A survey on deep learning: Algorithms, techniques, and applications. *ACM Comput. Surv.*, 51(5), Sept. 2018. ISSN 0360-0300. doi: 10.1145/3234150. URL https://doi.org/10.1145/3234150.

C. Pryor, C. Dickens, E. Augustine, A. Albalak, W. Y. Wang, and L. Getoor. Neupsl: Neural probabilistic soft logic. In E. Elkind, editor, *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI-23*, pages 4145–4153. International Joint Conferences on Artificial Intelligence Organization, 8 2023. doi: 10.24963/ijcai.2023/461. URL https://doi.org/10.24963/ijcai.2023/461. Main Track.

S. J. Purgal, D. M. Cerna, and C. Kaliszyk. Differentiable inductive logic programming in high-dimensional space. *CoRR*, abs/2208.06652, 2022. doi: 10.48550/ARXIV.2208.06652. URL https://doi.org/10.48550/arXiv.2208.06652.

J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3):239–266, Aug 1990. ISSN 1573-0565. doi: 10.1007/BF00117105. URL https://doi.org/10.1007/BF00117105.

T. N. Quoc, K. Inoue, and C. Sakama. Linear algebraic computation of propositional horn abduction. In *2021 IEEE 33rd International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 240–247, 2021. doi: 10.1109/ICTAI52525.2021.00040.

A. Radford, K. Narasimhan, T. Salimans, I. Sutskever, et al. Improving language understanding by generative pre-training. 2018.

A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, et al. Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pages 8748–8763. PmLR, 2021.

A. Rajasekharan, Y. Zeng, P. Padalkar, and G. Gupta. Reliable natural language understanding with large language models and answer set programming. *arXiv preprint arXiv:2302.03780*, 2023.

R. Riegel, A. Gray, F. Luus, N. Khan, N. Makondo, I. Y. Akhalwaya, H. Qian, R. Fagin, F. Barahona, U. Sharma, et al. Logical neural networks. *arXiv preprint arXiv:2006.13155*, 2020.

D. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

C. Sakama, K. Inoue, and T. Sato. Linear algebraic characterization of logic programs. In G. Li, Y. Ge, Z. Zhang, Z. Jin, and M. Blumenstein, editors, *Knowledge Science, Engineering and Management*, page 520–533, Cham, 2017. Springer International Publishing. ISBN 978-3-319-63558-3.

C. Sakama, H. D. Nguyen, T. Sato, and K. Inoue. Partial evaluation of logic programs in vector spaces. *arXiv preprint arXiv:1811.11435*, 2018.

T. Sato. A linear algebraic approach to datalog evaluation. *Theory and Practice of Logic Programming*, 17(3):244–265, 2017a. doi: 10.1017/S1471068417000023.

T. Sato. Embedding tarskian semantics in vector spaces, 2017b.

T. Sato and K. Inoue. Differentiable learning of matricized dnfs and its application to boolean networks. *Mach. Learn.*, 112(8):2821–2843, June 2023. ISSN 0885-6125. doi: 10.1007/s10994-023-06346-5. URL https://doi.org/10.1007/s10994-023-06346-5.

T. Sato and R. Kojima. Logical inference as cost minimization in vector spaces. In *Proceedings of the Fourth International Workshop on Declarative Learning Based Programming (DeLBP'19), https://delbp.github.io/*, 2019.

T. Sato and R. Kojima. Matsat: a matrix-based differentiable sat solver, 2021. URL https://arxiv.org/abs/2108.06481.

T. Sato, K. Inoue, and C. Sakama. Abducing relations in continuous spaces. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 1956–1962. International Joint Conferences on Artificial Intelligence Organization, 7 2018. doi: 10.24963/ijcai.2018/270. URL https://doi.org/10.24963/ijcai.2018/270.

T. Sato, C. Sakama, and K. Inoue. From 3-valued semantics to supported model computation for logic programs in vector spaces. In *Proceedings of the 12th International Conference on Agents and Artificial Intelligence (ICAART 2020; Valletta, Malta, 22-24 February 2020)*, pages 758–765, 01 2020. doi: 10.5220/0009093407580765.

T. Sato, A. Takemura, and K. Inoue. Towards end-to-end asp computation, 2023. URL https://arxiv.org/abs/2306.06821.

D. Selsam, M. Lamm, B. Bünz, P. Liang, L. de Moura, and D. L. Dill. Learning a sat solver from single-bit supervision, 2019.

P. Sen, B. W. de Carvalho, R. Riegel, and A. Gray. Neuro-symbolic inductive logic programming with logical neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 8212–8219, 2022.

P. Shakarian, C. Baral, G. I. Simari, B. Xi, and L. Pokala. *NeurASP*, pages 63–74. Springer Nature Switzerland, Cham, 2023. ISBN 978-3-031-39179-8. doi: 10.1007/978-3-031-39179-8_7. URL https://doi.org/10.1007/978-3-031-39179-8_7.

Z. Shen. Llm with tools: A survey, 2024. URL https://arxiv.org/abs/2409.18807.

S. Shi, H. Chen, M. Zhang, and Y. Zhang. Neural logic networks, 2019.

Z. Shi, M. Li, Y. Liu, S. Khan, J. Huang, H.-L. Zhen, M. Yuan, and Q. Xu. Satformer: Transformer-based unsat core learning. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, Oct. 2023. doi: 10.1109/iccad57390.2023.10323731. URL http://dx.doi.org/10.1109/ICCAD57390.2023.10323731.

H. Shindo, V. Pfanschilling, D. S. Dhami, and K. Kersting. α ilp: thinking visual scenes as differentiable logic programs. *Machine Learning*, 112(5):1465–1497, 2023.

H. Shindo, V. Pfanschilling, D. S. Dhami, and K. Kersting. Learning differentiable logic programs for abstract visual reasoning. *Mach. Learn.*, 113(11):8533–8584, Oct. 2024. ISSN 0885-6125. doi: 10.1007/s10994-024-06610-2. URL https://doi.org/10.1007/s10994-024-06610-2.

A. Skryagin, D. Ochs, D. S. Dhami, and K. Kersting. Scalable neural-probabilistic answer set programming. *J. Artif. Int. Res.*, 78, dec 2023. ISSN 1076-9757. doi: 10.1613/jair.1.15027. URL https://doi.org/10.1613/jair.1.15027.

A. Skryagin, D. Ochs, P. Deibert, S. Kohaut, D. S. Dhami, and K. Kersting. Answer set networks: Casting answer set programming into deep learning, 2024. URL https://arxiv.org/abs/2412.14814.

R. Speer, J. Chin, and C. Havasi. Conceptnet 5.5: An open multilingual graph of general knowledge. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI'17, page 4444–4451. AAAI Press, 2017.

R. K. Srivastava, K. Greff, and J. Schmidhuber. Highway networks, 2015. URL https://arxiv.org/abs/1505.00387.

Stanford CS231n. Convolutional Neural Networks (CNNs / ConvNets). https://cs231n.github.io/convolutional-networks/, n.d. Accessed: 30 May 2026.

H.-I. Suk. An introduction to neural networks and deep learning. In *Deep Learning for Medical Image Analysis*, pages 3–24. Elsevier, 2017.

D. Surís, S. Menon, and C. Vondrick. Vipergpt: Visual inference via python execution for reasoning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 11888–11898, October 2023.

A. Takemura and K. Inoue. Differentiable logic programming for distant supervision, 2024. URL https://arxiv.org/abs/2408.12591.

L. Tao, Y.-X. Huang, W.-Z. Dai, and Y. Jiang. Deciphering raw data in neuro-symbolic learning with provable guarantees. In *Proceedings of the Thirty-Eighth AAAI Conference on Artificial Intelligence and Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence and Fourteenth Symposium on Educational Advances in Artificial Intelligence*, AAAI'24/IAAI'24/EAAI'24. AAAI Press, 2024. ISBN 978-1-57735-887-9. doi: 10.1609/aaai.v38i14.29455. URL https://doi.org/10.1609/aaai.v38i14.29455.

S. Topan, D. Rolnick, and X. Si. Techniques for symbol grounding with satnet. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 20733–20744. Curran

Associates, Inc., 2021. URL https://proceedings.neurips.cc/paper_files/paper/2021/file/ad7ed5d47b9baceb12045a929e7e2f66-Paper.pdf.

E. Tsamoura and L. Michael. Neural-symbolic integration: A compositional perspective. *arXiv preprint arXiv:2010.11926*, 2020.

E. Tsamoura, T. Hospedales, and L. Michael. Neural-symbolic integration: A compositional perspective. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(6):5051–5060, May 2021. URL https://ojs.aaai.org/index.php/AAAI/article/view/16639.

L. Van der Maaten and G. Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

M. van Emden and R. Kowalski. The semantics of predicate logic as a programming language. *J. ACM*, 23:733–742, 01 1976.

A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38(3):619–649, July 1991. ISSN 0004-5411. doi: 10.1145/116825.116838. URL https://doi.org/10.1145/116825.116838.

E. van Krieken, E. Acar, and F. van Harmelen. Analyzing differentiable fuzzy implications, 2020.

E. van Krieken, T. Thanapalasingam, J. Tomczak, F. Van Harmelen, and A. Ten Teije. A-nesi: A scalable approximate method for probabilistic neurosymbolic inference. *Advances in Neural Information Processing Systems*, 36:24586–24609, 2023.

E. van Krieken, S. Badreddine, R. Manhaeve, and E. Giunchiglia. Uller: A unified language for learning and reasoning. In *International Conference on Neural-Symbolic Learning and Reasoning*, pages 219–239. Springer, 2024.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc. ISBN 9781510860964.

G. Šourek, V. Aschenbrenner, F. Železný, S. Schockaert, and O. Kuželka. Lifted relational neural networks: efficient learning of latent relational structures. *J. Artif. Int. Res.*, 62(1):69–100, May 2018. ISSN 1076-9757. doi: 10.1613/jair.1.11203. URL https://doi.org/10.1613/jair.1.11203.

D. Wang, M. Jamnik, and P. Liò. Investigating diagrammatic reasoning with deep neural networks. In P. Chapman, G. Stapleton, A. Moktefi, S. Perez-Kriz, and F. Bellucci, editors, *Diagrammatic Representation and Inference*, pages 390–398, Cham, 2018a. Springer International Publishing.

D. Wang, M. Jamnik, and P. Liò. Investigating diagrammatic reasoning with deep neural networks. In *Diagrams*, 2018b.

P. Wang, P. L. Donti, B. Wilder, and J. Z. Kolter. Satnet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver. In K. Chaudhuri and R. Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 6545–6554. PMLR, 2019. URL http://proceedings.mlr.press/v97/wang19e.html.

Z. Wang, W. Zhang, N. Liu, and J. Wang. Scalable rule-based representation learning for interpretable classification. In *Proceedings of the 35th International Conference on Neural Information Processing Systems*, NIPS '21, Red Hook, NY, USA, 2021. Curran Associates Inc. ISBN 9781713845393.

J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc. ISBN 9781713871088.

R. J. Williams and D. Zipser. *Gradient-based learning algorithms for recurrent connectionist networks*. Citeseer, 1990.

T. Winters, G. Marra, R. Manhaeve, and L. D. Raedt. Deepstochlog: Neural stochastic logic programming. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36(9):10090–10100, Jun. 2022. doi: 10.1609/aaai.v36i9.21248. URL https://ojs.aaai.org/index.php/AAAI/article/view/21248.

T. Xing, L. Garcia, M. R. Vilamala, F. Cerutti, L. Kaplan, A. Preece, and M. Srivastava. Neuroplex: learning to detect complex events in sensor networks through knowledge injection. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, SenSys '20, page 489–502, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375900. doi: 10.1145/3384419.3431158. URL https://doi.org/10.1145/3384419.3431158.

J. Xu, Z. Zhang, T. Friedman, Y. Liang, and G. V. den Broeck. A semantic loss function for deep learning with symbolic knowledge, 2018.

Y. Yang and L. Song. Learn to explain efficiently via neural logic inductive learning. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=SJlh8CEYDB.

Y. Yang, A. Panagopoulou, S. Zhou, D. Jin, C. Callison-Burch, and M. Yatskar. Language in a bottle: Language model guided concept bottlenecks for interpretable image classification. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 19187–19197, June 2023a.

Z. Yang, A. Ishay, and J. Lee. Neurasp: Embracing neural networks into answer set programming. In C. Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, pages 1755–1762. International Joint Conferences on Artificial Intelligence Organization, 7 2020a. doi: 10.24963/ijcai.2020/243. URL https://doi.org/10.24963/ijcai.2020/243. Main track.

Z. Yang, A. Ishay, and J. Lee. Neurasp: Embracing neural networks into answer set programming. In C. Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, pages 1755–1762. International Joint Conferences on Artificial Intelligence Organization, 7 2020b. doi: 10.24963/ijcai.2020/243. URL https://doi.org/10.24963/ijcai.2020/243. Main track.

Z. Yang, J. Lee, and C. Park. Injecting logical constraints into neural networks via straight-through estimators. In K. Chaudhuri, S. Jegelka, L. Song, C. Szepesvari, G. Niu, and S. Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of

Proceedings of Machine Learning Research, pages 25096–25122. PMLR, 17–23 Jul 2022. URL https://proceedings.mlr.press/v162/yang22h.html.

Z. Yang, A. Ishay, and J. Lee. Learning to solve constraint satisfaction problems with recurrent transformer, 2023b. URL https://arxiv.org/abs/2307.04895.

K. Yi, J. Wu, C. Gan, A. Torralba, P. Kohli, and J. Tenenbaum. Neural-symbolic vqa: Disentangling reasoning from vision and language understanding. *Advances in neural information processing systems*, 31, 2018.

K. Yi, C. Gan, Y. Li, P. Kohli, J. Wu, A. Torralba, and J. B. Tenenbaum. Clevrer: Collision events for video representation and reasoning, 2020.

L. A. Zadeh. Fuzzy logic. *Computer*, 21(4):83–93, 1988.

J. Zhang, C.-Y. Hsieh, Y. Yu, C. Zhang, and A. Ratner. A survey on programmatic weak supervision, 2022. URL https://arxiv.org/abs/2202.05433.

M. Zimmer, F. Xuening, G. Claire, J. Zhaohui, Z. Jianyi, P. Weng, J. Hao, D. Li, and W. Liu. Differentiable logic machines, 02 2021.