Imperial College London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

End-to-End Neuro-Symbolic Learning of Logic-based Inference

Nuri Cingillioglu

May 14, 2022

Submitted in part fulfilment of the requirements for the degree of Doctor of Philosophy in Computing of Imperial College London and the Diploma of Imperial College London.

Declaration of Originality

I, Nuri Cingillioglu, 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.

Copyright

The copyright of this thesis rests with the author and its contents are made available under a Creative Commons Attribution Non-Commercial Share-Alike 4.0 International (CC BY-NC-SA 4.0) License. You may copy and redistribute the material in any medium or format. You may also remix, transform or build upon the material. In doing so, you must give appropriate credit to the author, provide a link to the license and indicate if any changes were made. If you remix, transform or build upon this material, you must redistribute your contributions under the same license. You may not use the material for commercial purposes.

Please seek permission from the copyright holder for uses of this work that are not included in the license mentioned above.

Abstract

Artificial Intelligence has long taken the human mind as a point of inspiration and research. One remarkable feat of the human brain is its ability to seamlessly reconcile low-level sensory inputs such as vision with high-level abstract reasoning using symbols related to objects and rules. Inspired by this, neuro-symbolic computing attempts to bring together advances in connectionist architectures like artificial neural networks with principled symbolic inference of logic-based systems. *How* this integration between the two branches of research can be achieved remains an open question. In this thesis, we tackle neuro-symbolic inference in an end-to-end differentiable fashion from three different aspects: learning to perform symbolic deduction and manipulation over logic programs, the ability to learn and leverage variables through unification across data points and finally the ability to induce symbolic rules directly from non-symbolic inputs such as images.

We first start by proposing a novel neural network model, Iterative Memory Attention (IMA), to ascertain the level of symbolic deduction and manipulation neural networks can achieve over logic programs of increased complexity. We demonstrate that our approach outperforms existing neural network models and analyse the vector representations learnt by our model. We observe that the principal components of the continuous real-valued embedding space align with the constructs of logic programs such as arity of predicates and types of rules.

We then focus on a key component of symbolic inference: variables. Humans leverage variables in everyday reasoning to construct high level abstract rules such as "if someone went somewhere then they are there" instead of mentioning specific people or places. We present a novel end-to-end differentiable neural network architecture called Unification Network that is capable of recognising which symbols can act as variables through the application of soft unification. The by-products of the model are invariants that capture some common underlying principle present in the dataset. Unification Networks exhibit better data efficiency and generalisation to unseen examples compared to models that do not utilise soft unification.

Finally, we redirect our attention to the question: How can a neural network learn symbolic rules directly from visual inputs in a coherent manner? We bridge the gap between continuous vector representations and discrete symbolic reasoning by presenting a fully differentiable layer in a deep learning architecture called the Semi-symbolic Layer. When stacked, the Semi-symbolic Layers within a larger model are able to learn complete logic programs along with continuous representations of image patches directly from pixel level input in an end-to-end fashion. The resulting model holistically learns objects, relations between them and logical rules. By pruning and thresholding the weights of the Semi-symbolic Layers, we can extract out the exact symbolic relations and rules used to reason about the tasks and verify them using symbolic inference engines. Using two datasets, we demonstrate that our approach scales better than existing state-of-the-art symbolic rule learning systems and outperforms previous deep relational neural network architectures.

Acknowledgements

Firstly, I would like to thank my parents for their wholehearted support and perpetual encouragement without whom this odyssey culminating in this thesis would not be possible. I would also like to thank my aunt and uncle who have always prioritised, reinforced and highlighted the importance of my education. Though, I doubt they expected me to still study and go to *school* at my age, a privilege few get to experience the full length and scale of higher education.

Secondly, I would like to express my profound gratitude to my supervisor Professor Alessandra Russo for her unyielding enthusiasm and belief in me throughout my PhD even when my research was repeatedly rejected for publications. Her guidance, patience and wisdom has helped to me grow and gain confidence not only academically but also in life.

I must also thank Professor Murray Shanahan for inspiring me towards the wonders of Artificial Intelligence in my second and final years of my undergraduate study and later sharing a part of my PhD journey. I still remember when he showed the video of the Shakey robot in class followed by his frolic commentary.

Finally, I would like to sincerely thank my colleagues from the SPIKE research group and friends for their companionship and solace throughout my degree.

Dedication

For my parents, Sema and Ali.

Contents

1 Introduction 12
- 1.1 Motivation . . . 13
- 1.2 Challenges . . . 15
- 1.3 Neuro-symbolic Spectrum . . . 16
- 1.4 Contributions . . . 18
- 1.5 Publications . . . 20

2 Background 21
- 2.1 Computational Logic . . . 21
  - 2.1.1 Propositional Logic . . . 22
  - 2.1.2 Classical First-Order Predicate Logic . . . 25
  - 2.1.3 Logic Programming . . . 27
  - 2.1.4 Goal-oriented Proof Systems . . . 29
  - 2.1.5 Answer Set Programming . . . 32
  - 2.1.6 Inductive Logic Programming . . . 33
- 2.2 Deep Learning . . . 35
  - 2.2.1 Artificial Neural Networks . . . 36
  - 2.2.2 Training . . . 39
  - 2.2.3 Convolutional Neural Networks . . . 43
  - 2.2.4 Recurrent Neural Networks . . . 46

3 Generating Symbolic Data 50
- 3.1 Stratified Logic Programs . . . 51
- 3.2 Structured Symbol Patterns . . . 55
- 3.3 Subgraph Set Isomorphism . . . 56

4 Learning Symbolic Deduction with Neural Reasoning Networks 59
- 4.1 Neural Reasoning Networks . . . 60
- 4.2 Datasets . . . 63
- 4.3 Experiments . . . 64
- 4.4 Analysis . . . 67
- 4.5 Related Work . . . 71
- 4.6 Discussion . . . 72

5 **Learning Invariants through Soft Unification** **74**
5.1 Unification Networks . . . 76
5.2 Datasets . . . 78
5.3 Instances of Unification Networks . . . 80
5.3.1 Unification MLP . . . 80
5.3.2 Unification CNN . . . 81
5.3.3 Unification RNN . . . 81
5.3.4 Unification Memory Network . . . 83
5.4 Experiments . . . 86
5.5 Analysis . . . 90
5.6 Related Work . . . 94
5.7 Discussion . . . 96

6 **Learning Symbolic Rules from Pixels** **98**
6.1 Semi-symbolic Layer . . . 100
6.2 Datasets . . . 102
6.3 DNF Layer & Model . . . 104
6.4 Experiments . . . 109
6.5 Analysis . . . 113
6.6 Related Work . . . 119
6.7 Discussion . . . 121

7 **Conclusion** **123**
7.1 Future Work . . . 124
7.2 Alternative Avenues . . . 126
7.3 Broader Impact . . . 129

Bibliography **131**

A **Soft Unification** **143**
A.1 Training Curves . . . 143
A.2 Further Results . . . 145

B **Pixels to Rules** **149**
B.1 Semi-symbolic Layer . . . 149
B.2 Dataset Details . . . 152
B.3 Hyper-parameters . . . 152
B.4 Training Curves . . . 152
B.5 Further Results . . . 162

List of Figures

1.1 Continuous versus discrete representations . . . 16
1.2 The neuro-symbolic spectrum . . . 17

2.1 An example structure $\mathscr{M}$ in first-order logic . . . 26
2.2 SLDNF procedure for a given query . . . 31
2.3 Multi-step pipeline of clingo . . . 33
2.4 Linear separation of conjunction and disjunction . . . 37
2.5 Graphical visualisation of a multi-layer perceptron . . . 38
2.6 Common activation functions and their derivatives . . . 39
2.7 Examples of common operations in convolutional networks . . . 44
2.8 Graphical overview of the convolution operation . . . 45
2.9 Graphical overview of of the recurrent neural network . . . 46
2.10 Graphical overview of the Long Short-Term Memory model . . . 48

3.1 Correspondence between a graph and its logic programming representation . . . 57

4.1 Graphical overview of the Iterative Attention Model . . . 61
4.2 Attention maps of the IMA model . . . 66
4.3 Dual attention maps of the IMA model . . . 67
4.4 PCA visualisation of IMA atom embeddings . . . 67
4.5 PCA visualisation of literal and rule embeddings . . . 68
4.6 Accuracy of IMA against increasing lengths . . . 69
4.7 Mean accuracy of IMA with different sizes and training regimes . . . 70

5.1 Example invariant learnt for bAbI task 16 . . . 75
5.2 Graphical overview of a Unification Network . . . 77
5.3 Graphical overview of Unification Memory Network (UMN) . . . 84
5.4 Test accuracies of Unification MLP and CNN . . . 87
5.5 Test accuracy of Unification RNN with an example prediction . . . 87
5.6 Results of Unification Networks with increasing number of invariants . . . 88
5.8 Example invariants learnt by Unification RNN . . . 91
5.7 Example variableness $\psi(s)$ values on sentiment analysis . . . 91
5.9 Example invariants learnt by Unification Networks . . . 92
5.10 Variable binding attention maps of Unification Networks . . . 93

5.11 Example mismatching invariants learnt by Unification Networks 94

6.1 Comparison of the product t-norm and the semi-symbolic layer 100
6.2 Graphical overview of the semi-symbolic layer 101
6.3 Samples from datasets used to evaluate the semi-symbolic layer 103
6.4 Graphical overview of the DNF model 104
6.5 An example graphical visualisation of unary and binary predicates 106
6.6 Sample object selection attention maps 114
6.7 Plots of the learnt predicates that contribute the most to the final prediction 117
6.8 Example image reconstructions of the DNF model 118

7.1 Membrane potential and spikes of Spiking Neural Networks 128

A.1 Training curves of Unification MLP, CNN and RNN with different learning rates 143
A.2 Training curves for UMN on the 1k bAbI dataset with strong supervision 144
A.3 Training curves for UMN on the 2k logic dataset with strong supervision 144
A.4 Different learning rates and invariants of Unification MLP, CNN and RNN 145
A.5 Example invariant learnt for bAbI task 6, yes or no questions 146
A.6 Invariants learnt on tasks 1, 2 and 11 of the logic dataset 146

B.1 Further samples from the Relations Game dataset 151
B.2 Training curves for the DNF layer on the subgraph set isomorphism dataset . . . 154
B.3 Training curves for the DNF model on the Relations Game dataset 155
B.4 Training curves for the DNF-h model 156
B.5 Training curves for the DNF-r model 157
B.6 Training curves for the DNF-i model 158
B.7 Training curves for the DNF-hi model 159
B.8 Training curves for the DNF-ri model 160
B.9 Training curves for PrediNet on Relations Game 161
B.10 Scatter plot of results of the DNF layer on the subgraph isomorphism task 162
B.11 Scatter plot of best results of the DNF layer on the subgraph set isomorphism task 163
B.12 Scatter plot of aggregated results of the DNF layer against symbolic solvers . . . 164
B.13 Scatter plot of all experiments on Relations Game 164
B.14 Attention maps of the DNF model on the All task 166
B.15 Further attention maps learnt by the DNF-hi model 166
B.16 Further image reconstruction example of the DNF-hi model 167

List of Tables

2.1 Truth table for common Boolean connectives 23
2.2 Example application of forward chaining 30
2.3 Example unifications between two atoms 31
2.4 Stable models of a simple program 32
2.5 Common data types and their encodings used with neural networks 36

3.1 Summary of generated datasets and tasks 50
3.2 Sample programs from tasks 1 to 5 52
3.3 Sample programs from tasks 6 to 8 53
3.4 Sample programs from tasks 8 to 12 54
3.5 Sample context, query and answer triples from the sequence and grid tasks 55
3.6 Training sizes for randomly generated fixed length and grid datasets 55
3.7 Different difficulty parameters for the subgraph set isomorphism dataset 58
3.8 Example rules for medium size subgraph set isomorphism 58

4.1 Recurrent components in the IMA model 62
4.2 Descriptive statistics of the predicate symbols for the fact matching task 63
4.3 Samples of different difficulties of the fact matching task 64
4.4 Results of the Neural Reasoning Networks 65

5.1 Dataset samples used by Unification Networks 78
5.2 Comparison of different Unification Networks and learnable components 80
5.3 Aggregate error rates on the bAbI dataset for UMN and baselines 89
5.4 Aggregate error rates on the logical reasoning dataset for UMN and IMA 90
5.5 Performance of exact invariant matches of Unification Networks 90

6.1 Example permutations of input facts given to the DNF layer 107
6.2 Median test results for the subgraph set isomorphism dataset 109
6.3 Median test accuracies for subgraph set isomorphism with input noise 110
6.4 Median test accuracy for the Relations Game tasks with full results in Appendix B.5 111
6.5 Aggregate median test accuracies for the image reconstruction experiments . . . 112
6.6 Example learnt continuous representations of objects 113
6.7 Example pruning and thresholding of semi-symbolic weights 114
6.8 Example pruning and thresholding of the full DNF model 115

A.1 Individual task error rates on the bAbI dataset 146
A.2 Comparison of Unification Networks and baselines on the bAbI dataset 147
A.3 Individual task error rates of UMN against IMA 147
A.4 Individual task error rates of UMN with extra configurations of the logic dataset . 148

B.1 Summary of symbols for the DNF layer and model formulation 150
B.2 All hyper-parameters used for training the DNF models 152
B.3 All hyper-parameters used for constructing DNF the models 153
B.4 Full results of the best runs of the DNF layer on the subgraph set isomorphism task 163
B.5 Median test accuracies of all DNF configurations 164
B.6 Median of test accuracies for all models across the Relations Game from Table 6.4 165
B.7 Median absolute deviation across the Relations Game, complements Table B.6 . . 165

Notation

The notation is based on the Deep Learning book [67]. Since neuro-symbolic research lies at the intersection of two disciplines, the notation can be overloaded. For example, while $p(a)$ represents a unary predicate within a logic program, it may instead refer to a probability distribution over a continuous variable. Where necessary, the context disambiguates the intended semantics.

Numbers and Arrays

a	A scalar (integer or real)
$\boldsymbol{a}$	A vector
$\boldsymbol{A}$	A matrix
A	A tensor
$\boldsymbol{I}_n$	Identity matrix with n rows and n columns
$\boldsymbol{I}$	Identity matrix with dimensionality implied by context
$\boldsymbol{e}^{(i)}$	Standard basis vector $[0,\dots,0,1,0,\dots,0]$ with a 1 at position i
$\text{diag}(\boldsymbol{a})$	A square, diagonal matrix with diagonal entries given by $\boldsymbol{a}$
a	A scalar random variable
$\mathbf{a}$	A vector-valued random variable
$\mathbf{A}$	A matrix-valued random variable

Sets and Graphs

$\mathbb{A}$	A set
$\mathbb{R}$	The set of real numbers
$\{0,1\}$	The set containing 0 and 1
$\{0,1,\ldots,n\}$	The set of all integers between 0 and n
$[a,b]$	The real interval including a and b
$(a,b]$	The real interval excluding a but including b
$\mathbb{A}\backslash\mathbb{B}$	Set subtraction, i.e., the set containing the elements of $\mathbb{A}$ that are not in $\mathbb{B}$
$\mathcal{G}$	A graph
$Pa_{\mathcal{G}}(\mathrm{x}_i)$	The parents of x_i in $\mathcal{G}$
$V(\mathcal{G})$	Vertices of graph $\mathcal{G}$
$E(\mathcal{G})$	Edges of graph $\mathcal{G}$

Boolean Algebra

$\top,\bot,U$	Truth, falsity and unknown symbols respectively
$p,p',p_0,p_1,\ldots,q,r,s,t$	Propositional atom
$A,A',B,A_0,\ldots$	Logical formulae
$\neg A$	Negation of a logical formula
$A_1 \wedge A_2 \wedge \ldots$	Conjunction (AND operator) of formulae
$A_1 \vee A_2 \vee \ldots$	Disjunction (OR operator) of formulae
$A \leftarrow B$	Material implication, $\neg B \vee A$
$A \leftrightarrow B$	A if and only if B, $A \leftarrow B \wedge B \leftarrow A$

First-order Logic

$p(t_1,\ldots,t_n)$	N-ary predicate with $t_1,\ldots,t_n$ terms as arguments, either constants $a,b,c,a_0,a_1,\ldots$ or variables $X,Y,Z,\ldots$
$\forall XA$	Universally quantified variable X in formula A
$\exists XA$	Existentially quantified variable X in formula A
$A_1,A_2,\ldots,A_n \vdash B$	Syntactic entailment of formulas
$A_1,A_2,\ldots,A_n \models B$	Semantic entailment of formulas

Logic Programming

$p(a).$	Fact, ground atom, i.e. no variables or conditions
not A	Negation as failure of formula A
$H \leftarrow B_1 \wedge \ldots \wedge B_n$	A rule with head atom H and body conditions B_i
$H\text{:-}B_1, \ldots, B_n$	Common logic programming notation of a rule

Indexing

a_i	Element i of vector $\boldsymbol{a}$, with indexing starting at 1
a_{-i}	All elements of vector $\boldsymbol{a}$ except for element i
$A_{i,j}$	Element i, j of matrix $\boldsymbol{A}$
$\boldsymbol{A}_{i,:}$	Row i of matrix $\boldsymbol{A}$
$\boldsymbol{A}_{:,j}$	Column i of matrix $\boldsymbol{A}$
$\mathsf{A}_{i,j,k}$	Element (i, j, k) of a 3-D tensor A
$\mathsf{A}_{:,:,k}$	2-D slice of a 3-D tensor

Linear Algebra Operations

$\boldsymbol{A}^\top$	Transpose of matrix $\boldsymbol{A}$
$\boldsymbol{A}^+$	Moore-Penrose pseudo-inverse of $\boldsymbol{A}$
$\boldsymbol{A} \odot \boldsymbol{B}$	Element-wise (Hadamard) product of $\boldsymbol{A}$ and $\boldsymbol{B}$
$\det(\boldsymbol{A})$	Determinant of $\boldsymbol{A}$
$[\boldsymbol{a}, \boldsymbol{b}]$	Concatenation of two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$

Calculus

$\frac{dy}{dx}$	Derivative of y with respect to x
$\frac{\partial y}{\partial x}$	Partial derivative of y with respect to x
$\nabla_{\boldsymbol{x}} y$	Gradient of y with respect to $\boldsymbol{x}$
$\nabla_{\boldsymbol{X}} y$	Matrix derivatives of y with respect to $\boldsymbol{X}$
$\nabla_{\mathsf{X}} y$	Tensor containing derivatives of y with respect to X
$\frac{\partial f}{\partial \boldsymbol{x}}$	Jacobian matrix $\boldsymbol{J} \in \mathbb{R}^{m \times n}$ of $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$
$\nabla_{\boldsymbol{x}}^2 f(\boldsymbol{x})$ or $\boldsymbol{H}(f)(\boldsymbol{x})$	The Hessian matrix of f at input point $\boldsymbol{x}$
$\int f(\boldsymbol{x}) d\boldsymbol{x}$	Definite integral over the entire domain of $\boldsymbol{x}$
$\int_{\mathbb{S}} f(\boldsymbol{x}) d\boldsymbol{x}$	Definite integral with respect to $\boldsymbol{x}$ over the set $\mathbb{S}$

Probability and Information Theory

$\mathrm{a} \bot \mathrm{b}$	The random variables a and b are independent
$\mathrm{a} \bot \mathrm{b} \mid \mathrm{c}$	They are conditionally independent given c
$P(\mathrm{a})$	A probability distribution over a discrete variable
$p(\mathrm{a})$	A probability distribution over a continuous variable, or over a variable whose type has not been specified
$\mathrm{a} \sim P$	Random variable a has distribution P
$\mathbb{E}_{\mathrm{x}\sim P}[f(x)]$ or $\mathbb{E}f(x)$	Expectation of $f(x)$ with respect to $P(\mathrm{x})$
$\mathrm{Var}(f(x))$	Variance of $f(x)$ under $P(\mathrm{x})$
$\mathrm{Cov}(f(x), g(x))$	Covariance of $f(x)$ and $g(x)$ under $P(\mathrm{x})$
$H(\mathrm{x})$	Shannon entropy of the random variable x
$D_{\mathrm{KL}}(P\|Q)$	Kullback-Leibler divergence of P and Q
$\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$	Gaussian distribution over $\mathbf{x}$ with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$

Functions

$f : \mathbb{A} \to \mathbb{B}$	The function f with domain $\mathbb{A}$ and range $\mathbb{B}$
$f \circ g$	Composition of the functions f and g
$f(\boldsymbol{x}; \boldsymbol{\theta})$	A function of $\boldsymbol{x}$ parametrized by $\boldsymbol{\theta}$.
$\log x$	Natural logarithm of x
$\sigma(x)$	Logistic sigmoid, $\dfrac{1}{1+\exp(-x)}$
$\lvert\lvert\boldsymbol{x}\rvert\rvert_p$	L^p norm of $\boldsymbol{x}$
$\lvert\lvert\boldsymbol{x}\rvert\rvert$	L^2 norm of $\boldsymbol{x}$
$\mathbf{1}_{\mathrm{condition}}$	is 1 if the condition is true, 0 otherwise

Sometimes we use a function f whose argument is a scalar but apply it to a vector, matrix, or tensor: $f(\boldsymbol{x})$, $f(\boldsymbol{X})$, or $f(\mathsf{X})$. This denotes the application of f to the array element-wise. For example, if $\mathsf{C} = \sigma(\mathsf{X})$, then $\mathsf{C}_{i,j,k} = \sigma(\mathsf{X}_{i,j,k})$ for all valid values of i, j and k.

Datasets and Distributions

p_{data}	The data generating distribution
$\hat{p}_{\mathrm{data}}$	The empirical distribution defined by the training set
$\mathbb{X}$	A set of training examples
$\boldsymbol{x}^{(i)}$	The i-th example (input) from a dataset
$y^{(i)}$ or $\boldsymbol{y}^{(i)}$	The target associated with $\boldsymbol{x}^{(i)}$ for supervised learning
$\boldsymbol{X}$	The $m \times n$ matrix with input example $\boldsymbol{x}^{(i)}$ in row $\boldsymbol{X}_{i,:}$

Chapter 1

Introduction

The human mind has always been a source of inspiration for Artificial Intelligence (AI) research. From a very young age, humans learn about their environment, reason about the objects and people in it. While the human brain's neuronal structure has promoted research in connectionist approaches to AI such as Artificial Neural Networks, the human brain's analytical inference capabilities have been closely studied for decades in the domain of computational logic [175]. Recent advances in Artificial Neural Networks, particularly under the umbrella of Deep Learning [67], have gained unprecedented achievements in a wide of range domains such as computer vision [109, 68], natural language processing [216, 45] and reinforcement learning [142, 80, 196, 197, 180]. While these methods can learn from continuous unstructured inputs such as images, they require vast amounts of data and computing power and lack the apparent principled inference that the logic-based methods provide. Logic-based methods often use a knowledge base together with formalised inference methods which apply rules, deduce facts, abduce explanations and induce further rules. Although symbolic methods were prominent in the early days of AI, especially around the 1970s and 80s in the form of expert systems [93], they struggle to learn directly from unstructured data which is not in the form of a knowledge base such as images, often require manually engineered features, and generalise with difficulty to previously unseen inputs. Yet, logic-based systems are data efficient and require less computing resources. Hence, we arrive at a point in which we desire coherent AI methods that bring together the best of both worlds: to efficiently learn and perform high-level principled inference directly from low-level inputs.

Neuro-symbolic computing [17, 24] attempts to bridge the gap between connectionist AI approaches and symbolic methods to yield models that are capable of learning high-level inference, i.e. symbolic rule-based deduction, abduction and induction, on top of low-level inputs such as images. As deep learning models become more and more embedded in critical applications such as self-driving cars [74], concerns regarding their trust, safety, interpretability and accountability have encouraged the need for integrating more well-founded inference methods into them [58]. Beyond practical applications, this need has also been promoted as a milestone for models to achieve human-level generalisation, i.e. to generalise beyond direct experiences in predictable and systematic ways [72]. In this context, the neural part often refers to a deep learning model while the symbolic component is a logic-based formalism such as first-order logic [175] implemented

within a logic programming paradigm. But, *how* we can fuse these two systems together remains an open question [94].

In this thesis, we present three novel neuro-symbolic approaches that outperform comparable state-of-the-art models in metrics such as accuracy, data efficiency and scalability across various datasets. The overarching goal of the contributions made in Chapters 4 to 6 is to explore how neural networks can be encouraged to bring about logic-based inference and analyse the effects of such architectural biases on the models' behaviour. While in each chapter we focus on a different improvement, we maintain an emphasis on learning from scratch in an end-to-end fashion solely from data to support a more holistic neuro-symbolic view buttressed by the human brain. The fundamental relationship between deep learning and symbolic reasoning that underpins the work in this thesis can be captured by the following question: *deep learning is great at learning patterns and human reasoning formalised by logic can be viewed as a collection of patterns governing symbols, thus can deep learning be used to train machines to perform logic-based inference?*

1.1 Motivation

While we provide more topic specific sources of inspiration in Chapters 4 to 6, in this section, we discuss the evidence surrounding human cognition that further substantiates an overall neuro-symbolic perspective on Artificial Intelligence. Human cognition as a reference for intelligence has been a crucial driving force for AI research. The most famous example is perhaps the Turing Test [214] in which an interrogator is tasked with trying to determine whether an interactive agent is human or not. Although the test itself is quite general, the idea of using human intelligence as a benchmark on tasks remains ubiquitous today, most notably in the domain of reinforcement learning by pitting human players against a machine [196]. Thus, it is natural to utilise principles of human cognition to design, build, alter and augment promising AI methods. For example, a very successful type of neural network for processing images, called the Convolutional Neural Network (CNN), is inspired by the workings of the visual cortex [121] (described further in Section 2.2.3). It is then not a surprise that the roots of neuro-symbolic computing can be traced to accounts of the human mind as a neuronal system capable of symbolic reasoning.

The language of thought hypothesis [144] describes the act of thinking as the manipulation of internal symbols. This language-like substrate, sometimes referred to as *mentalese*, promotes a compositional symbol processing system that underpins the cognitive capacity of humans. There have been many arguments for and against this conjecture, mainly from a connectionist perspective, debating the plausibility of a Turing-style symbol processing system realised in the neuronal structure of the human brain [166]. If one was to take the stance that our thoughts are structured, systematic in some rule-governed symbolic way, then one faces the question as to how this phenomenon can be implemented in the brain. Implementationist connectionism [161] holds this point of view and shines a spotlight onto neuro-symbolic approaches for both cognitive sciences and computing. Experiments with humans attempting to solve mathematical equations written in slightly different styles and formats demonstrate that the written format of the equations affects the performance of the participants despite probing the same mental tasks [113]. These experiments

show evidence for the inseparable nature of neural perception and symbolic thought in the human brain. The open question is whether the computational implementation of this phenomenon better suits a single task or multiple separate processing tasks.

Despite the various types of approaches on integrating neural networks with symbolic methods, none come close to the seamless nature of the human brain. Humans can learn from and combine low-level data such as intensities of light in an image with high-level inference in the form of objects, relations and rules. In his book *Thinking, Fast and Slow*, Kahneman [101] introduces a dichotomy between two modes of thought that bring about the neuro-symbolic account of human cognition: a more autonomous, fast, innate, intuitive, perceptive and associative *System 1* and a more recently evolved, slower, deliberative and logical *System 2*. These systems have evolved together and are complementary to each other as opposed to two distinct components. In other words, the level of symbolic deliberative thought of System 2 depends on the perceptive capacity of System 1 and vice versa. From this perspective, the human brain appears to be one complete and coherent neuro-symbolic system that has one adaptable neuronal substrate. This observation sits at the focal point of this thesis as we investigate the idea of end-to-end trainable neural networks that evoke some form of symbolic inference as the computational implementation of our neuro-symbolic systems. As a result, we direct our attention to methods that utilise a single substrate but aim to elicit symbolic behaviour through neural architectural biases.

Experimental hints for this dual-process accounts of reasoning [51] surrounding the interplay of Systems 1 and 2 can be found with the Cognitive Reflection Test (CRT) [54]. CRT tasks are designed to evoke an incorrect *gut* response from the participants who upon further reflection find the correct answer. Consider the CRT problem below:

A bat and a ball cost $1.10 in total.
The bat costs $1.00 more than the ball.
How much does the ball cost?

for which the intuitive, fast, gut response is often 10 cents, especially if the participants are under some time pressure or limited attention. However, upon further inspection, one realises the correct answer is in fact 5 cents. Modern state-of-the-art deep natural language models such as GPT-3 [25] also provide the quick response of 10 cents when asked the same question [149] hinting that modern deep learning may comprise the aforementioned System 1 but lack the logical System 2. Further work on the gap between human behaviour and deep neural networks provides similar insights concerning causality, use of intuitive physics and learning-to-learn [112]. In particular, the latter notion of learning-to-learn is being explored within the domain of meta-level reinforcement learning with fast and slow learning components [22], similar to the dual systems of Kahneman. One can also find influences of these dual systems in neural networks that attempt to combine bottom-up, immediately observed signals, with top-down, prior belief and expectation, signals to create a more robust perception pipeline [141]. In this case, the top-down signals are reminiscent of a System 2 that can override the immediate, potentially wrong, observations made by System 1.

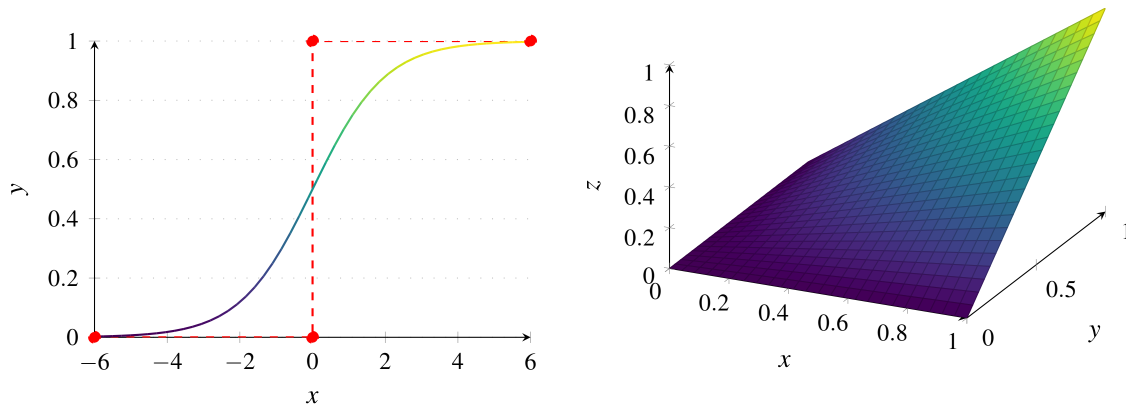
Perhaps the final piece of motivation is found in the answer to the question why. Why are we interested in building neuro-symbolic machines capable of robust perception and high-level reasoning?

This is the neuro-symbolic perspective of the original question posed by Alan Turing in his seminal work “Can machines think?” [214] if one were to regard human thought as the produce of a neuro-symbolic system with the aforementioned properties. The term Artificial General Intelligence (AGI) [66] has recently gained popularity to describe the original human-level intelligence goal that current AI methods have veered away from with more practical domain specific successes. As a result, the historical fascination with and potential of generally intelligent machines are still very much rife and await further exploration.

1.2 Challenges

Developing neuro-symbolic methods that try to combine deep learning with classical logic-based methods presents many challenges. The first challenge arises from the fundamental natures of the neural and symbolic domains. While the mathematics behind artificial neural networks utilise continuous values, e.g. neuron activations and connection weights, the symbolic systems are rooted in the inherently discrete Boolean algebra. In this context, continuous refers to values or functions that vary smoothly while discrete encompasses entities that can take on specific constants such as true or false but no other value in between. Another way to view the difference is that continuous values are often measured and discrete are counted. This continuousness is critical for the success of deep learning as it allows us to compute and utilise the gradients of the weights (parameters) of a neural network to optimise them with respect to some differentiable objective function [67]. The core idea is that the smooth, differentiable nature of the objective function with respect to the weights of the neural network permits small, incremental updates that ultimately allow the model to *learn*. This puts discrete symbolic representations in jeopardy as the derivative of a constant valued function is zero, i.e. there is no incremental change with respect to its input. Hence, in their original symbolic form, logic-based approaches are intrinsically incompatible with gradient based methods. The novel neuro-symbolic work in this thesis is no exception and the algorithms in Chapters 4 and 6 also face this conundrum.

The juxtaposition of continuous and discrete representations inevitably beckons a compromise from either neural network or symbolic methods. The resolution often presents two avenues: a continuous relaxation of discrete entities or a threshold of continuous values. Consider the representations of the constants true $\top$ and false $\bot$ from Boolean algebra with numeric values 1 and 0 respectively. Fig. 1.1a demonstrates two possible functions: in red dashed line a discrete threshold function $x > 0$ that yields the values $\{0,1\}$ and in varying colour the continuous sigmoid function $\sigma(x)$ from Eq. (2.3) with the output interval $[0,1]$. While the gradient of y with respect to x is zero for the threshold function, i.e. when x changes y stays constant except for a sudden and immediate jump at $x = 0$, we can see the smooth change from 0 to 1 of the sigmoid function. Thus, we take advantage of the approximation made by the sigmoid function to train neural networks with a binary true or false output and then threshold the output value (e.g. $y > 0.5$) to obtain a discrete prediction. This trend also extends to Boolean operators such as the logical AND gate $z = x \wedge y$. In probabilistic or fuzzy logic, logical operators are implemented using triangular norms (t-norms) [75] that work on the interval $[0,1]$ as opposed to the discrete true and false constants. Fig. 1.1b visualises the product t-norm for conjunction of two fuzzy Boolean variables $z = xy$.



(a) Continuous sigmoid function versus the threshold function to represent Boolean true and false values.

(b) Continuous relaxation of the AND gate using the product t-norm in the interval $[0,1]$.

Figure 1.1: The continuous against discrete representations of neural and symbolic systems proposes a challenge for neuro-symbolic approaches that attempt to combine them. It is common to find continuous approximations of discrete truth values as in probabilistic or fuzzy logic.

As both inputs x, y get close to 1 the output z also becomes 1 and if one input is 0, the output is 0 capturing the semantics of the logical AND gate in a continuous setting. In Chapter 6, we explore and propose a novel continuous relaxation of logical operators that overcomes some of the limitations of t-norms such as vanishing gradients. Despite the harmonious neuro-symbolic account of the human brain from Section 1.1, the mathematical implementation of continuous and discrete representations within a single substrate for applications of machine learning remains an open question and an active area of research.

The second challenge of neuro-symbolic research stems from the large amounts of computing resources required by deep learning. Despite their unprecedented success, modern deep learning methods require large amounts of data and computing power to train [132]. One could expect days of training on high-end graphics processing units (GPUs) to obtain state-of-the-art deep natural language understanding models [25, 45] costing thousands of pounds. As a result, only large corporations and industry research labs can realistically carry out research on internet scale datasets and utilise specialised hardware such as Tensor Processing Units (TPUs) [99]. Although this may initially disguise an engineering problem, a recent study of 1,058 research papers reveals that progress in five prominent deep learning research areas including image recognition and natural language processing rely strongly on increases in computing power and that the trend would be unsustainable [212]. This puts an overall limitation on, including the work carried out in this thesis, the types and scale of experiments that can be reasonably conducted. Hence, all experiments in Chapters 4 to 6 utilise smaller datasets and train models on the more readily available central processing units (CPUs). This also renders the research from this thesis more accessible to and reproducible by a larger audience with limited budgets.

1.3 Neuro-symbolic Spectrum

Modern state-of-the-art neuro-symbolic approaches can be categorised on how neural network or logic programming based they are. Fig. 1.2 shows the neuro-symbolic spectrum ranging from

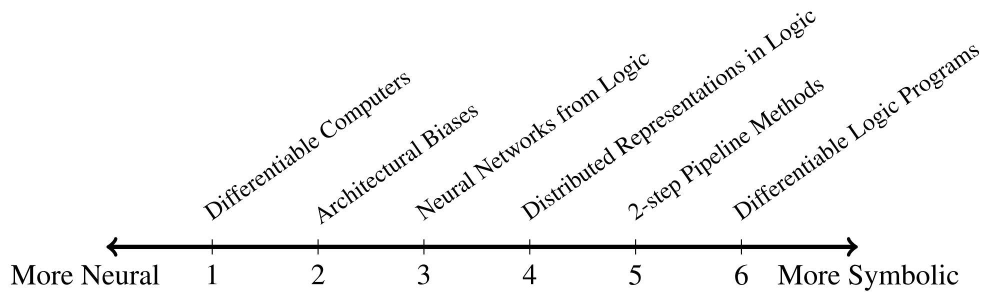


Figure 1.2: The spectrum of existing neuro-symbolic approaches range from how neural or symbolic they are. On the far left we have universal differentiable computers and on the right differentiable logic programs that are designed to leverage recent advances in gradient based optimization.

more neural to more symbolic methods based on their core characteristic. While the left extreme includes the most general neural networks such as the Multilayer Perceptron (MLP) [67], the right end of the spectrum leads to purely symbolic solvers like Prolog [37] and Answer Set Programming (Section 2.1.5). In between the two extremes we have neuro-symbolic methods that:

1. aim to learn universal computers purely using neural networks [70, 71, 228, 98]. These models are the most general form of neuro-symbolic computing since they attempt to mimic a full computer often with differentiable memory, controller and input-output components. In theory, they have the capacity to perform any computable function including symbolic manipulation and inference but in practice, it is not clear how they represent and operate on symbols when evaluated on simple algorithmic tasks such as copying or sorting numbers.

2. introduce architectural biases in order to constrain or refine the behaviour of neural networks towards some principled bottleneck [179, 151, 191, 69, 150, 123]. While these models demonstrate improvements in object-centric and relational learning tasks over plain neural networks, the extent and the type of biases that can be baked into neural networks to elicit well-founded reasoning remain an active area of research.

3. construct neural networks from logic programs [202, 186, 38, 169, 104]. The models in this category perform logical inference in a differentiable manner to enable gradient based optimization. However, they require a template or a starting point such as an existing knowledge base that dictates the connections in the neural networks which in return realise the logical inference steps like applying rules and deducing facts.

4. use real-valued vector representations of symbols within logic programs [171, 170, 138, 136, 133]. Inspired by the success of distributional semantics of words in natural language processing [134, 157, 27], these approaches take advantage of semantic similarities of constants such as “grandpa” and “grandfather” whilst still using a symbolic solver to carry out the inference steps. While they perform well on knowledge-base completion and link prediction tasks, they inherit the challenges of the symbolic solvers such as scalability.

5. work in two steps: first apply a neural network to parse an unstructured input to build a knowledge-base and then use a symbolic or neuro-symbolic solver [226, 89, 222, 149, 43, 140]. For example, an object-detecting neural network can parse an image to describe the

objects and their attributes. The resulting symbolic knowledge is readily utilised by existing symbolic methods to answer questions and deduce conclusions from what was originally an image. In effect, the pipeline methods glue together multiple steps of neural or symbolic systems but fall short of a holistic approach.

6. leverage advances in gradient based optimization [52, 154, 130, 44, 224]. Recent advances in deep learning have given rise to many tools and computing resources that focus on gradient based parameter optimization. The methods in this category utilise continuous relaxations of logical operators, often using triangular norms [75], to obtain a differentiable form of logical inference. The resulting models are capable of inducing rules through gradient based search as well as training upstream neural networks that provide probabilistic facts. However, these approaches often require manually engineered rules in the form of a given logic program or program templates in order to run.

Another way of categorising the existing approaches is how they reconcile the disparity between continuous and discrete representations mentioned in the previous Section 1.2. From this view, differentiable computers (category 1) to some extent are similar to distributed representations in logic (category 4) since they both use real-valued vectors to represent entities and objects. Neural networks from logic (category 3) offer similar continuous relaxations of Boolean operations as differentiable logic programs (category 6). Thus, the intention and the source of motivation driving these works are equally important in categorising them as done in Fig. 1.2 as opposed to just considering the technical resolutions they provide. Our source of motivation from Section 1.1, the complete and coherent neuro-symbolic account of the human mind, pulls our attention to the architectural biases of category 2 as plausible means of creating neural networks capable of symbolic reasoning. In the next section, we outline our novel contributions which mainly revolve around introducing new neural network architectures for end-to-end neuro-symbolic learning of logic-based inference.

1.4 Contributions

All machine learning models require data to train [19]. Despite the desire to obtain data efficient models, neuro-symbolic methods, particularly those on the more neural end of the spectrum in Fig. 1.2, still inherit the data hungry trait of deep learning [132]. This situation drives interest in novel datasets that capture various aspects of logical inference for neuro-symbolic research. The aim is to present a problem that involves some form of inference, e.g. deduction, such that after the model is trained and if it has successfully learnt a solution, we hope to analyse if and how the models exhibit the required logical reasoning. In Chapter 3, we describe three new datasets used alongside existing ones throughout the thesis to train, evaluate and analyse neuro-symbolic models. The datasets consist of numerous tasks that require deductive or inductive inference. We discuss the common mistakes one might encounter when generating synthetic datasets, how we generate the tasks and analyse their properties. All three datasets act as controlled environments to probe properties such as generalisation and scalability of the machine learning models presented in the subsequent chapters.

It is unknown to what extent neural networks can already learn symbolic deduction solely from examples. In Chapter 4, we propose a novel end-to-end trainable neural network to provide the first insight into how a neural network selects rules and deduces facts over multiple iterations in order to answer a given logic programming query as well as how it learns to represent logic programming constructs in a vector space. We train the neural network from scratch to perform deductive inference at the same time as learning real-valued vector representations of logical constructs such as predicates and rules. Evaluated against comparable state-of-the-art memory based neural networks, our approach yields higher predictive accuracies and better generalisation to previously unseen symbols across various testing conditions. When we analyse the high-dimensional representations of the logical constructs by projecting them to a lower dimensional space, typically 2 or 3, we visually observe a syntactic clustering of the entities such as rules and atoms based on their arity, use of negation and variables.

To perform multi-step deduction, neural reasoning networks from Chapter 4 require learning to identify variable symbols and unify them with constants. This usage is purely syntactic as if it was another special constant symbol but variables are primitive concepts per se, with the semantics that they can represent different entities. A neuro-symbolic system should be able to learn these abstractions in order to extrapolate general invariant principles implicit in the data beyond the confines of rigid logic programming syntax. Human reasoning involves the usage of variables everyday to recognise common underlying principles such as "if someone went somewhere then they are there" expressed using variables "someone" and "somewhere". This usage of variables allows for invariant abstractions about the environment to be learnt without mentioning specific people or places. Motivated by this, we shift our focus to the problem of learning to recognise and use variables by neural networks. In Chapter 5, we present an end-to-end differentiable neural network approach capable of lifting examples into invariants and using those invariants to solve a given task by identifying which symbols act as variables and assigning new values to them. The core characteristic of our contribution is a soft unification mechanism between data points that enables the network to generalise parts of the input into variables and use the resulting invariants to predict the output of previously unseen data points. We demonstrate that our method captures patterns in the data and can improve performance in terms of accuracy and data efficiency.

Despite the fact that the approach described in Chapter 5 can learn invariant patterns, these invariants lack logical structure and semantics. The goal is to have a single neural network model that can go from pixels to complete logic programs within a consistent neuronal substrate inspired by the seamless neuro-symbolic account of the human brain. In Chapter 6, we put forward a complete and coherent neuro-symbolic method for processing images into objects, learning relations and logical rules in an end-to-end fashion. To that end, we introduce a novel differentiable layer in a deep learning architecture from which symbolic relations and rules can be extracted by pruning and thresholding. When evaluated across two datasets, the resulting model scales beyond state-of-the-art symbolic rule learners and outperforms deep relational neural network architectures on accuracy and data efficiency. Crucially, this approach brings about a unifying view to connectionist and logic-based principles by realising logical operators for conjunction and disjunction within a neural network layer.

1.5 Publications

The following original workshop and conference papers have been produced based on the research presented in this thesis:

1. Nuri Cingillioglu and Alessandra Russo. "DeepLogic: Towards End-to-End Differentiable Logical Reasoning". In: *AAAI-MAKE* (Mar. 20, 2019). arXiv: 1805.07433 [cs.NE]
   https://github.com/nuric/deeplogic
2. Nuri Cingillioglu and Alessandra Russo. "Learning Invariants through Soft Unification". In: *NeurIPS* (Oct. 24, 2020). arXiv: 1909.07328 [cs.LG]
   https://github.com/nuric/softuni
3. Nuri Cingillioglu and Alessandra Russo. "pix2rule: End-to-end Neuro-symbolic Rule Learning". In: *IJCLR-NeSy* (June 14, 2021). arXiv: 2106.07487 [cs.LG]
   https://github.com/nuric/pix2rule

Chapter 2

Background

Neuro-symbolic computing lies at the intersection of two main branches of research: computational logic and artificial neural networks [175]. It is not a coincidence that these two domains provide the necessary underlying theories and tools to formalise symbolic reasoning over a neural substrate. Logic [82] is a branch of philosophy that deals with forms of reasoning and thinking, especially inference and the scientific method. Thus, the field of logic and its implementations within logic programs are naturally suitable for artificial intelligence models geared towards symbolic reasoning. By using logic, we mathematically define what we mean by reasoning, how we draw conclusions, the validity of such conclusions, how symbols can be manipulated and more. On the other hand, artificial neural networks act as the bedrock for machine learning inspired by the human brain. They can in theory act as universal function approximators [86] but in practice, as it is with symbolic learners, are often limited by how they are trained, the data available, computational resources and their architecture. Recently, by leveraging an increase in computing power, artificial neural networks have become a potent and established method for machine learning, particularly under the umbrella of deep learning [67] that aims to construct larger neural networks with more layers. Going back to the neuro-symbolic account of the human mind in Section 1.1, we now have the tools to firstly formalise then hopefully learn high-level reasoning.

In this chapter, we provide the principles of computational logic in Section 2.1 and deep learning in Section 2.2. We restrict our description of these domains to what is used within the scope of this thesis and leave further details for the reader to explore in the corresponding references.

2.1 Computational Logic

Logic [82] is a very large domain of interest and branches across many disciplines such as mathematics and philosophy. In the context of this thesis, it can be regarded as the *calculus of computing*: a mathematical foundation for dealing with symbols and truth. By truth, we often mean the veracity of arguments such as “All humans are mortal, Socrates is a human; therefore Socrates is mortal.” A logical system usually consists of three main ingredients: (i) syntax as in a formal language in which concepts are expressed, (ii) semantics to provide meaning for the aforementioned language and (iii) a proof theory to obtain or identify valid statements. These three components

may vary across different formalisms. In this section, we adhere to a formalism that is closer to modern logic programming used in neuro-symbolic literature.

2.1.1 Propositional Logic

At the most basic level, we are interested in expressing logical arguments and ascertaining their validity. Propositional logic, sometimes called zeroth-order logic, formalises an algebra for *propositions*, atomic statements that can be true or false. It serves as the foundation for many higher level logics such as the object and relation oriented first-order logic covered later in Section 2.1.2. In this section, we define the building blocks for a logical system including its syntax, semantics and a proof system following Hodkinson and Russo [83].

Syntax The syntax for propositional logic includes the following formal symbols:

- The Boolean symbols $\mathbb{B} = \{\top, \bot\}$ (read top and bottom) for truth and falsity respectively.
- Propositional atoms that attain truth values, denoted using lower case letters such as p, p', $p_0, p_1, \ldots$ and q, r, s, t.
- Boolean connectives $\neg$ (negation), $\wedge$ (and), $\vee$ (or), $\leftarrow, \rightarrow$ (implication) and $\leftrightarrow$ (equivalence), defined as n-ary functions $\mathbb{B}^n \rightarrow \mathbb{B}$ where n is the arity of the connective.

A logical *formula* is a tree in which the leaves are either propositional atoms or elements of $\mathbb{B}$ and the nodes are Boolean connectives, e.g. $(p \wedge q) \leftarrow (r \vee \neg q)$. We denote formulas using single upper case letters, such as $A, A', B, A_1, A_2, \ldots, A_n, P$. In the absence of explicit brackets, Boolean connectives in formulas have a binding strength; ranging from the strongest to the weakest, the order is $\neg, \wedge, \vee, \leftarrow, \leftrightarrow$.

Conjunction A formula of the form $A_1 \wedge A_2 \wedge \ldots A_n$ is called a conjunction and A_i are its conjuncts. It may also be represented as a set of conjuncts, written using the set notation $A_i \in P$ where P is the resulting conjunction. The set notation for conjunctions is used in Section 2.1.3 for logic programs where each formula can be considered a conjunct and the entire program the conjunction of the formulae it contains.

Disjunction A formula of the form $A_1 \vee A_2 \vee \ldots A_n$ is called a disjunction and A_i are its disjuncts.

Implication A formula of the form $A \rightarrow B$ or $B \leftarrow A$ is called an implication where A is the antecedent and B the consequent.

Literal A formula that is either an atom or a negated atom such as $p, \neg p$ is called a literal.

Clause A clause is a disjunction of one or more literals.

Disjunctive Normal Form is a disjunction of conjunctions of literals.

Conjunctive Normal Form is a conjunction of disjunctions of literals.

Semantics So far we have been creating a grammar over symbols and have not considered their meaning. For example, one can represent the phrase “if it rains, I take an umbrella” with $p \rightarrow q$ where p and q correspond to “it rains” and “I take an umbrella” respectively. The truth value of the

Table 2.1: Truth table for common Boolean connectives with $\top = 1$ and $\bot = 0$

p	q	$\neg q$	$p \wedge q$	$p \vee q$	$p \leftarrow q$	$p \leftrightarrow q$
0	0	1	0	0	1	1
0	1	0	0	1	0	0
1	0		0	1	1	0
1	1		1	1	1	1

consequent depends on whether it is raining in the current situation. In order to evaluate formulas, we must know the truth values of the propositional atoms and different situations may lead to different evaluations. Formally, a *situation* determines the truth value of every propositional atom. Symbols $\top$ and $\bot$ are always true and false respectively in every situation.

A proposition represents an atomic statement within the logical framework even if the meaning may consist of separable parts. A machine learning model in which propositional atoms are learnable as well, the semantics of the learnt atoms may indeed not be atomic as demonstrated and discussed in Chapters 4 and 6. For example, the statement "it is raining and it is cloudy" comprises of two parts but if we choose to set it as a propositional atom, we cannot probe it further.

The truth value of a formula is inductively defined following the truth values of its components and that of the Boolean connectives. Table 2.1 shows some common Boolean connectives and their truth tables using the $\top = 1, \bot = 0$ encoding. Whilst one can create further arbitrary Boolean functions, the sets $\{\neg, \wedge\}$ and $\{\neg, \vee\}$ are *functionally complete* which are sufficient to express all possible Boolean connectives [49]. As a result, when defining a new logical formalism, it is sufficient to implement a functionally complete set of connectives. Since a formula in DNF or CNF only involves the functionally complete sets $\{\neg, \wedge, \vee\}$, they are computationally efficient to implement and evaluate. In Chapter 6, we take this property into account when proposing a novel differentiable method for learning and evaluating DNF formulas using neural networks.

Given some formulas $A_1, A_2, \ldots, A_n$ and B, an argument of the form $A_1, A_2, \ldots, A_n \models B$ is *valid* if in every situation $A_1, A_2, \ldots, A_n$ are true, B is also true. The most famous example of a valid argument is perhaps $A, A \to B \models B$ also known as modus ponens, latin for method for affirming. In every situation where A and $A \to B$ hold, we can conclude B holds as well. The validity of modus ponens can be proved using the truth table of implication shown in Table 2.1. A valid formula A is one that is true in every possible situation, denoted as $\models A$ and valid propositional arguments are often called tautologies. A satisfiable formula is one that is true in at least one situation. Given two formulas A and B, they are logically equivalent $A \equiv B$ if they are both true in exactly the same situations. An equivalence worth noting is $A \to B \equiv \neg A \vee B$ which can be verified using Table 2.1. This equivalence is commonly used to capture the Boolean connective $\to$ using the functionally complete set $\{\neg, \vee\}$.

Although the semantics of propositional logic is defined for the discrete Boolean values $\top$ and $\bot$, one can extend it to the interval $[\top, \bot]$ using the encoding $\top = 1$ and $\bot = 0$ as part of fuzzy or many-valued logics [129]. We leave the formal definition of fuzzy logic as it is outside the scope of this thesis and instead highlight that it is common in neuro-symbolic literature to utilise

continuous relaxations of Boolean connectives in order to evaluate formulae in a differentiable manner [52, 169, 170] as alluded to in Section 1.2.

Definition 1 (Triangular Norm)**.** A triangular norm, or t-norm, is a function defined on the domain $f : [0,1] \times [0,1] \to [0,1]$ that satisfies the following properties [75]:

1. Commutativity: $f(p,q) = f(q,p)$
2. Associativity: $f(p, f(q,c)) = f(f(p,q),c)$
3. Monotonicity: $f(p,q) \leq f(c,d) \to p \leq c \wedge q \leq d$
4. 1 is identity element $f(p,1) = p$

The continuous relaxation of Boolean connectives is often achieved using triangular norms from Definition 1 and the most popular to model conjunction are: minimum $\min(p,q)$, product pq (shown in Fig. 1.1b) and Łukasiewicz t-norm $\max(0, p+q-1)$ which all correctly capture the logical $\wedge$ connective in Table 2.1. But one can assign other numeric values to $\top, \bot$ and create a different corresponding algebra. For example, in Chapter 6 we utilise the balanced ternary representation where $\top = 1$ and $\bot = -1$. Once the numeric values for $\top$ and $\bot$ are set, one can use the arithmetic mean to set the numeric encoding for unknown, $U = \frac{\top+\bot}{2}$ which has the desired property of being in the *middle* of true and false. The encoding for the truth values are purely for representational purposes and while one encoding might be easier for certain types of computation, the semantics of the Boolean connectives for the inputs $\top$ and $\bot$ are fundamentally equivalent as defined in Table 2.1.

Proof System A proof system is a collection of procedures that compute valid formulas [83]. It is a purely syntactical, procedural way of reasoning independently of what the symbols mean. Using a proof system a computer can determine the answer without evaluating whether every situation A and B can be true. A *theorem* is a provable formula and we write $A_1, A_2, \ldots, A_n \vdash B$ if a proof system arrives at some other formula B given some formulas $A_1, A_2, \ldots, A_n$. This is the syntactic counterpart of the logical argument $\models$.

Natural deduction by Gentzen [63] is a proof system for propositional logic. “A system of natural deduction can be thought of as a set of rules (of the *natural* kind) that determines the concept of deduction for some language or set of languages.” [162]. This means the syntactic rules proposed by natural deduction follow the semantics of the connectives closely. For example, if $A \wedge B$ holds, then we can naturally deduce both A and B hold independently. This kind of direct argument is used to generate logic programs later in Chapter 3. We leave the full extent of natural deduction outside the scope of this thesis as it is not required. The soundness and completeness properties of a proof system stipulate the equivalence between syntactic derivation $\vdash$ and logical arguments $\models$.

Definition 2 (Soundness)**.** For given formulas $A_1, A_2, \ldots, A_n$ and B, a proof system is sound if $A_1, A_2, \ldots, A_n \vdash B$ implies $A_1, A_2, \ldots, A_n \models B$.

Definition 3 (Completeness)**.** For given formulas $A_1, A_2, \ldots, A_n$ and B, a proof system is complete if $A_1, A_2, \ldots, A_n \models B$ implies $A_1, A_2, \ldots, A_n \vdash B$.

If we take $n = 0$ in Definitions 2 and 3, we can phrase soundness and completeness as any provable formula is valid and any valid formula is provable respectively. Before we use any automated method for proving arguments, we would like it to be both sound and complete. For example, in Chapter 4, we propose a novel method for learning a proof system using deep learning and investigate whether it is sound or complete. As for natural deduction, it can be shown that it is sound and complete for propositional logic [162].

Propositional logic, although intuitive and simple, is not very expressive. We cannot easily express statements such as "all bankers are rich" or "at least one person read this thesis". Such arguments require quantifying objects and capturing their properties and relationships. Given the environment we live in and how the human brain organises information into objects, relations and rules (refer back to Section 1.1), it is necessary to expand the class of logics to a more expressive one.

2.1.2 Classical First-Order Predicate Logic

First-order logic, also known as predicate logic, expands propositional logic to include quantified variables over objects. It effectively introduces the notion of non-logical entities into the formal logical system and allows us to reason about them by constructing arguments and again checking their validity. For example, we can express phrases of the form "if all humans are mortal and Alice is a human, then Alice is mortal" capturing what "all" and "Alice" is within the logic framework without resorting to just propositions. Hence, first-order logic seems adequate to represent real world tasks that humans reason about using objects, relations and rules as encouraged in Chapter 1.

Syntax The syntax for first-order logic extends that of propositional logic with the addition of the following formal symbols:

- Constant symbols denoting non-logical entities, using lower case letters such as a, a_1, b, c.
- Variable symbols that as act placeholders for entities, using uppercase letters X, Y, V_1 etc.
- Function symbols such as f, f_1, g that denote a n-ary mapping from constants to a constant.
- Predicate symbols with specified arities that denote Boolean relations among entities using lower case letters p, p', $p_0, p_1, \ldots, q, r, s, t$.
- Quantifiers $\forall$ (for all) and $\exists$ (there exists)

Constant, function and predicate symbols together form a *signature*, also called a language $\mathbb{L}$. Variable symbols are not part of a signature because they have no inherent meaning of their own, they merely act as placeholders. While constants may be interpreted as objects in the real world, predicates as relations between them, variables remain purely syntactic and ephemeral. For a given signature $\mathbb{L}$, any constant, application of functions and variable symbols are *terms*.

Atomic formulae of first-order logic are n-ary predicate symbols with terms as arguments such as $p(a), q(X,Y), r(f(a))$. Nullary predicates (predicate symbols with no arguments) are equivalent to propositional atoms which lets first-order logic subsume all of propositional logic. An atomic formula with no variables, is called ground or a *ground atom*. When a first-order formula consists only of ground atoms, it is equivalent to a propositional formula since every ground instance of

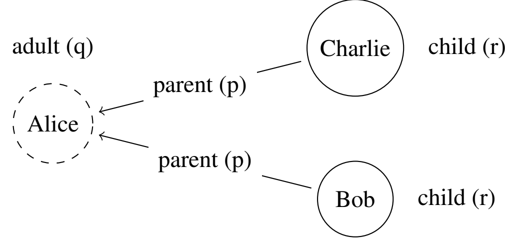


Figure 2.1: An example structure $\mathscr{M}$ consisting of three objects Alice, Bob and Charlie and an interpretation of a signature with binary predicate p and unary predicates q, r. Using this interpretation we can state that there is at least one child $\mathscr{M} \models \exists X r(X)$, that parents of children are adults $\mathscr{M} \models \forall X \forall Y (r(X) \land p(X,Y) \rightarrow q(Y))$ and that every child has a parent $\mathscr{M} \models \forall X (r(X) \rightarrow \exists Y p(X,Y))$.

an atom can be seen as a unique propositional atom. Hence, it is more often than not first-order predicate logic in neuro-symbolic literature colloquially refers to formulas that include at least one variable or incorporates the usage of variables at some point during inference.

We again inductively construct formulae using Boolean connectives with the addition of $\forall XA$ and $\exists XA$ for a given formula A. The quantifier symbols quantify occurrences of variable X in A and have the same binding strength as $\neg$. If a variable is quantified, we say it is *bound*, otherwise it is *free*. A *sentence* is a formula with no free variables, i.e. all variables are bound. Sentences play an important part in logic programming (described later in Section 2.1.3) as the variables that appear in the rules of a logic program are always implicitly universally quantified.

In this thesis we do not require function symbols; however, for completeness and potential future extensions of the neuro-symbolic work presented in Chapters 4 to 6, we cover them as part of the formalisation in this section.

Semantics Given a signature $\mathbb{L}$, a *structure*, sometimes loosely referred to as a model, is a tuple $\mathscr{M} = (\mathbb{M}, I)$ where $\mathbb{M}$ is the set of entities in the domain or universe of $\mathscr{M}$, and I is an interpretation that maps the symbols in $\mathbb{L}$ to objects and relations in $\mathbb{M}$. The interpretation (the meaning) of a constant symbol in $\mathbb{L}$ is an object *in* $\mathbb{M}$, a function symbol is a *mapping* from objects to an object in $\mathbb{M}$ and the interpretation of a predicate symbol with arity n is a relation *on* $\mathbb{M}^n$. Formally, a constant symbol c has the interpretation $I(c) = c_{\mathscr{M}} \in \mathbb{M}$, a function symbol $I(f) = f_{\mathscr{M}} : \mathbb{M}^n \rightarrow \mathbb{M}$ and the predicate symbol $I(p) = p_{\mathscr{M}} \subseteq \mathbb{M}^n$. Thus, an interpretation effectively specifies which ground atoms are true.

Structures in first-order predicate logic are akin to situations in propositional logic. They allow us to give *meaning* to symbols and evaluate first-order formulas. If some formula A holds under a structure $\mathscr{M}$, we write $\mathscr{M} \models A$. This notation is overloaded with its propositional counterpart since structures and situations both serve a similar purpose. Fig. 2.1 demonstrates a graphical elucidation of a structure with at most binary predicates in which nodes correspond to constants and edges to relations: $p \stackrel{\text{def}}{=}$ parent, $q \stackrel{\text{def}}{=}$ adult and $r \stackrel{\text{def}}{=}$ child.

Given a structure $\mathscr{M}$ and a formula A, an *assignment* is a mapping from the variable symbols that appear in A to the objects in the domain of the structure $\mathbb{M}$. We write $\mathscr{M} \models \forall XA$ if $\mathscr{M} \models A$ holds for *every* assignment of variable X and $\mathscr{M} \models \exists XA$ if $\mathscr{M} \models A$ holds for *some* assignment.

The process of assigning values to variables in formulas is sometimes called variable binding or grounding since if every variable is assigned a value, the formula becomes ground. The concept of variables and variable assignment are critical to symbolic reasoning which we further motivate and investigate in Chapter 5.

Proof System Natural deduction extended with procedures to handle the quantifiers is a complete and sound proof system for first-order logic [83]. All other aspects of natural deduction remain the same to its propositional counterpart. However, the general decision problem of checking whether a given formula A is valid in full first-order logic, i.e. $\models A$ where A may include function symbols, is undecidable [143]. In the next section, we focus on the computational implementations of logical frameworks, their semantics and proof systems.

2.1.3 Logic Programming

Logic programming [4] is a programming paradigm that is based on the semantics of logic described in the previous sections. We present the syntax and semantics of a common subset of the programming languages used by Prolog [37], a goal-oriented proof system, and clingo [61], an answer set solver. We base our presentation on Sergot [187].

Syntax A logic program P is a set, equivalently the conjunction, of rules of the form:

$$H \leftarrow L_1, L_2, \ldots, L_n$$

where $n \geq 0$ and L_i are literals. Literals are either positive or negated atoms of the form B_i and $\text{not } B_i$ respectively where 'not ' is the *negation as failure* operator [35]. A negated literal $\text{not } B_i$ succeeds when all attempts to prove B_i *fail* in finite time. H is called the *head* of the rule and literals $L_1, L_2, \ldots, L_n$ are called the *body*. All variables that appear in the head and in the body are implicitly universally quantified making them sentences. For example, the rule $p(X) \leftarrow q(X,Y)$ is equivalent to $\forall X \forall Y (p(X) \leftarrow q(X,Y))$. A rule with no body literals ($H \leftarrow$) is called a *fact*, and a rule without a head ($\bot \leftarrow \ldots$) is called a *constraint*. We restrict our definition of a rule to normal clauses which have at most one positive literal in the head. We leave disjunctive heads and extended logic programs that mix classical negation ($\neg$) and negation as failure (not), outside the scope of this thesis.

Negation as failure fundamentally differs from classical negation ($\neg$) by its association to the proof system of the logical framework and makes the underlying semantics *non-monotonic*. For first-order logic programs, a single predicate may be grounded with many objects such that it may be infeasible to explicitly assign a ground truth value for every predicate and constant combination to utilise classical negation. For example, we may want to express that "summer days are warm every year except in an abnormal year like 1816", known as the year without a summer. We can succinctly write down $\forall X \forall Y (warm(X,Y) \leftarrow day(X) \wedge summer(X) \wedge year(Y) \wedge \text{not } abnormal(Y))$ instead of enumerating the warmth of every possible day and potentially infinite number of years. Rules without negated body conditions ($m = 0$) are called *definite clauses*, otherwise they are called *normal clauses*. A *definite logic program* contains only definite clauses and a *normal logic program* additionally contains normal clauses.

Since characters such as $\leftarrow$ are not within the standard ASCII character encoding for computers, rules are written using the following punctuation:

$$p(X) \leftarrow q(X,Y) \wedge r(Y) \equiv p(X)\text{:-}q(X,Y), r(Y)$$

in which the characters $\leftarrow$ and $\wedge$ are replaced while the semantics remain the same. We utilise both notations depending on whether it is a logical framework or a logic programming context.

```
1 wings(X) :- fly(X).
2 fly(X) :- bird(X), not abnormal(X).
3 abnormal(X) :- wounded(X).
4 bird(john).
5 bird(mary).
6 wounded(john).
```

Listing 2.1: Example logic program about birds and rules about flying

An example normal logic program about birds and their properties is given in Listing 2.1. The program defines the relationships between birds, flying, wings and the abnormal cases, and mentions two constant terms `john` and `mary`. Another way of viewing logic programs is to regard them as knowledge bases since they contain explicit facts and rules to derive further information. A logic program is *stratified* if it can be partitioned into disjoint sets $P = \bigcup_{i=1}^{n} P_i$ such that all the rules defining a predicate symbol is contained in at most one partition, and the definitions of positive body conditions in P_i are contained in $\bigcup_{j \leq i} P_j$ while negative body conditions are contained in $\bigcup_{j<i} P_j$. $P_1, P_2, \ldots, P_n$ are called *strata* and there may be multiple such partitions. For example, Listing 2.1 can be stratified using lines $P_2 = \{1,2\}$ with P_1 as the remainder of the program.

Semantics For a given logic program P and formula A, we would like to establish whether A is semantically entailed by P, written as $P \models A$. A *model* of a logic program is a structure (as in first-order logic) in which every rule of P evaluates to true. We define $P \models A$ to mean that A is true in *every* model of P and refer to A as a *logical consequence* of P. Instead of dealing with all possible structures and models, we instead consider the *Herbrand structure* for a given signature $\mathbb{L}$. The Herbrand structure is a tuple $\mathscr{M} = (\mathbb{M}, I)$ in which the domain of objects $\mathbb{M}$ is the set of all ground terms in $\mathbb{L}$ called the *Herbrand universe*, and I is the *Herbrand interpretation* that maps every symbol to itself, i.e. an identity mapping. Formally, a constant symbol c in the program has the interpretation $I(c) = c \in \mathbb{L}$, a function symbol $I(f) = f : \mathbb{L}^n \to \mathbb{L}$ and predicate symbol $I(p) = p \subseteq \mathbb{L}^n$. It is then sufficient to provide which ground atoms are true to complete the structure in which formulas of P can be evaluated. The set of all possible ground atoms constructed using the Herbrand universe is called the *Herbrand base*. If a Herbrand structure $\mathscr{M}$ is a model of P, then it is called a *Herbrand model*.

A model, or a Herbrand model, represented as a set of ground atoms is *minimal* if there is no proper subset of it that is also a model and *supported* if every atom is the head of a ground rule in the program. Consider the following simple propositional logic program $\{p, q \leftarrow p, p \leftarrow r\}$. It has two Herbrand models: $\{p,q\}, \{p,q,r\}$ of which the former one is minimal and supported.

Since a model of a logic program must by definition make every rule evaluate to true, it must at least include all the facts that appear in the program. In the previous example, the atom p appears in all the models of the program since it is a fact and every model agrees with p being true. For Listing 2.1, the Herbrand universe is {`mary,john`}. The Herbrand base would be every predicate appearing in the program grounded with the two constants, i.e. {`wings(john), ...,` `wounded(mary)`}. A Herbrand interpretation would thus be any subset of the Herbrand base. For example, {`wounded(mary)`, bird(john)} is a Herbrand interpretation.

Why are Herbrand structures and models of interest? In general, it is sufficient to just consider the Herbrand structure and models. This is because a logic program P has a model if and only if it has a Herbrand model. It is clear that if P has a Herbrand model then it must have a model, and if there is some arbitrary structure $\mathscr{M}$ such that $\mathscr{M} \models P$ then one can construct a corresponding Herbrand structure which will be a model [187].

Proof Systems Finally, we require a procedure to determine whether a program P entails a query q. In the following sections we turn our attention to concrete implementations of computational proof systems that determine whether a query is provable $P \vdash q$.

2.1.4 Goal-oriented Proof Systems

Logic programs are versatile in the sort of information they can represent. For example, we can describe access control measures, patient treatment procedures and computer games using facts and rules. In these cases, we might want to query whether a principle has access to a certain resource or compute all the patients who receive a particular drug. Unless the information we seek is given as a set of facts, we must determine how to computationally apply the rules of a logic program and deduce whether it *entails* something or not. Goal-oriented proof systems apply the rules of a logic program with the aim of deriving further facts or searching for a particular goal. Starting from the facts and applying rules to see if a program contains the goal is called forward-chaining [52] while starting from a goal and going backwards towards facts is called backward-chaining [5]. Goals are often questions captured by first-order logic formulae such as does `mary` have `wings` and who can `fly` represented by `wings(mary)` and `fly(X)` respectively.

Definition 4 (Immediate Consequence)**.** Given a set of facts I and a logic program P, the immediate consequences of I using P is a set of facts that can be derived using the rules in P. Formally, it is a function from a set of ground atoms to another set of ground atoms such that

$$T_P(I) = \{\text{head}(r) \mid r \text{ is a ground instance of a rule in } P,\ \text{body}^+(r) \subseteq I \wedge \text{body}^-(r) \cap I = \emptyset\}$$

where body^+ and body^- are positive and negated atoms appearing in the body of the rule.

Fixed-point Semantics The T_P operator has important properties in capturing the logical consequences of logic programs. Given an interpretation I as a set of ground atoms, I is a *model* of P if and only if $T_P(I) \subseteq I$ and *supported* if and only if $I \subseteq T_P(I)$ [187]. Thus, any interpretation that is a fixed-point $T_P(I) = I$ is a supported model of P. Intuitively, if every fact we can derive is already part of our interpretation, then we have satisfied all the rules in the program, and if

Table 2.2: Forward chaining steps for the example logic program in Listing 2.1. At each iteration, immediate consequences of each rule are computed using the T_P operator. The program reaches a fix point at Step 3 after which $T_P(I) = I$ holds where I represents the atoms that are true.

Step 0	Step 1	Step 2	Step 3	Step 4
bird(john)	bird(john)	bird(john)	bird(john)	bird(john)
bird(mary)	bird(mary)	bird(mary)	bird(mary)	bird(mary)
wounded(john)	wounded(john)	wounded(john)	wounded(john)	wounded(john)
	abnormal(john)	abnormal(john)	abnormal(john)	abnormal(john)
	fly(john)			
		wings(john)		
	fly(mary)	fly(mary)	fly(mary)	fly(mary)
		wings(mary)	wings(mary)	wings(mary)

the interpretation is included in every derivable head of a ground rule, it must be supported. For definite logic programs, it can be shown that iteratively applying the T_P operator starting from an empty set yields the unique minimal supported Herbrand model, also known as the least fix point or the least Herbrand model [187]. However, this does not hold for normal logic programs such as $P = \{p \leftarrow \text{not } q,\ q \leftarrow \text{not } p\}$ where $T_P(\emptyset)$ loops forever between $\{p, q\}$ and $\emptyset$.

Forward Chaining Since a logical consequence must be true in all models of a program and that every fact must be in all models by definition, every fact is a logical consequence of its program. This observation serves as a starting point for forward-chaining algorithms which iteratively apply the rules to the accumulated facts in order to compute all the logical consequences of a program. Table 2.2 shows an example application of the T_P operator starting from the facts of the logic program in Listing 2.1. Since `abnormal(john)` is not a given fact, at Step 1, we erroneously derive `fly(john)` (highlighted in red) despite also deriving `abnormal(john)`. We reach a fix point at Step 3 since applying the T_P operator again at Step 4 yields the same ground atoms, i.e. $T_P(I) = I$. Since in every model `mary` is a `bird` and would not be `wounded` and therefore not `abnormal`, the logic program does entail `fly(mary)` and subsequently `wings(mary)` as highlighted in green. This method of starting with facts and deriving further ones is also referred to as a bottom-up approach and is the main proof system in Chapter 6 as a novel differentiable neural network layer.

Backward Chaining On the other hand, one can instead start with an atomic goal and apply the rules in the reverse direction, known as backward-chaining [5]. The popular programming language Prolog [37] works using this principle. The overall algorithm is similar to depth-first search through the rules of a logic program with facts as the leaves of the search tree. For example, in Chapter 3 we generate logic programs with tree-like search spaces for given atomic goals. Unlike the T_P operator from Definition 4, goal-driven proofs only bind variables that are needed and avoid grounding the entire rule.

Unification is a decision problem that involves determining whether given two atoms are syntactically equivalent. We call a successful set of substitutions between the arguments of the predicates that make them equivalent a *unifier* and the unifier with the least number of substitutions is called the most general unifier. Table 2.3 shows some example unifications between two atoms. We later revisit the idea of unification in Chapter 5 as part of a novel neuro-symbolic architecture in which we use semantic similarity between symbols as opposed to a syntactic match.

Table 2.3: Example unifications between two atoms. Unification is a purely syntactic check between two symbolic constructions of atoms.

Atom 1	Atom 2	Result	Unifier
p(a)	p(a)	✓	$\emptyset$
p(a)	p(b)	x	
p(a)	q(a)	x	
p(X)	p(b)	✓	{X/b}
p(a,Y)	p(X,b)	✓	{X/a, Y/b}
p(a,b)	p(X,X)	x	
p(X)	p(Y)	✓	{X/Y}

θ ={X/john}

fly(john) $\leftarrow$ ✓ — bird(john) $\leftarrow$ x — abnormal(john) $\leftarrow$ ✓ — wounded(john)

Figure 2.2: SLDNF procedure of the query `fly(john)` for the logic program in Listing 2.1. The negation as failure step is highlighted in red. The overall query fails since `john` is `wounded` failing the negated condition `not abnormal(X)`.

Using unification, the backward-chaining procedure is often realised using Selective Linear Definite with Negation as Failure (SLDNF) [5] algorithm, informally:

1. Given an atomic goal g, select a rule of which the head unifies with g.
2. For any positive body condition $\text{body}^+(r)$ of the selected rule, attempt to prove it by starting from Step 1 with the body literal as the new goal.
3. For any negative body condition $\text{body}^-(r)$ of the selected rule, attempt to disprove it again starting from Step 1 with the body literal as the new goal.
4. Repeat until either a goal fails or there are no more goals left to prove in which case the original query fails or succeeds respectively.

Fig. 2.2 shows the steps of SLDNF procedure for the goal `fly(john)`. Utilising the rule in line 2 of Listing 2.1, we create two sub-goals `bird(john)` and `not abnormal(john)`. The first sub-goal succeeds as it is a given fact. The second sub-goal on the other hand, highlighted in a red box, requires `abnormal(john)` to fail finitely. Repeating the procedure using the rule from line 3, and using the fact `wounded(john)`, the sub-goal succeeds and in return the original query fails. It is this process we attempt to learn later in Chapter 4 and analyse the properties of the resulting neural network. One visible advantage of goal-driven proof systems is their transparency.

We assume that given a logic program P, an atom p is false if it is not a logical consequence of that program, $P \not\models p$. We adhere to this *closed world assumption* throughout the thesis, for example when building the synthetic datasets in Chapter 3 and learning proof systems in Chapters 4 and 6. Consider the simple program $P = \{p, p \leftarrow q\}$, both $\{p\}$ and $\{p,q\}$ are models of P. Under the closed world assumption, we can then say P does not entail q, $P \not\models q$ since q is not true in all models of P. In the next section, we consider alternative semantics of which the proof systems overcome the limitation of programs with loops.

Table 2.4: Stable models of the program $p \leftarrow \text{not } q, q \leftarrow \text{not } p$

I	Reduct	Minimal Herbrand Model	Stable Model
{p, q}	$\emptyset$	$\emptyset$	x
{p}	$p \leftarrow$	$\{p\}$	✓
{q}	$q \leftarrow$	$\{q\}$	✓
$\emptyset$	$p \leftarrow, q \leftarrow$	$\{p,q\}$	x

2.1.5 Answer Set Programming

Consider the very simple program $P = \{p \leftarrow \text{not } q, q \leftarrow \text{not } p\}$. Starting from the query p and running SLDNF, we loop between p and q as sub-goals forever. Yet, both $\{p\}$ and $\{q\}$ are supported models of P, for example $T_P(\{p\}) = \{p\}$ using Definition 4. Stable model semantics [62] offer a declarative approach to solving normal logic programs instead of searching solutions to specific queries in a procedural manner.

Stable Model Semantics Given a *ground* normal logic program P and an interpretation I, we eliminate all instances of negation as failure by constructing the *reduct* of the program:

$$P^I = \{\text{head}(r) \leftarrow \text{body}^+(r) \mid r \text{ is a rule in P } \wedge \text{body}^-(r) \cap I = \emptyset\}$$

which removes any rule of which the negated body literal is known to be true. Then, the interpretation I is a *stable model*, also known as an answer set, if it satisfies the stability equation $I = M(P^I)$ where M is the unique minimal (least) Herbrand model of the reduct. Since the reduct P^I is a definite logic program, one can repeatedly apply the immediate consequence operator T_P from Definition 4 starting from the empty set $T_P(T_P(\dots T_P(\emptyset)))$ in order to obtain the unique minimal (least) Herbrand model and check the stability equation. It can be shown that every stable model of P is also a *minimal supported* Herbrand model despite the grounding and reduction steps [187]. The stability equation determines whether an interpretation is a stable model but does not indicate how one can obtain viable candidates to test. Modern answer set solvers utilise many optimisations that are outside of the scope of this thesis in order to avoid testing every possible interpretation and maintain an efficient search for answer sets [61].

One caveat is the requirement of a ground program which may not always be possible. If the signature is really big, for example one million predicates and constants, the grounding of a logic program may be too large for off-the-shelf computing hardware to effectively handle. We take this limitation into account when generating the combinatorial subgraph isomorphism dataset in Section 3.3 and evaluate how it affects symbolic learners in Chapter 6. There are many tricks and shortcuts modern answer set programming implementations employ in order to reduce the grounding size and speed up computation [61] which are outside the scope of this thesis.

The stable models for the example program at the beginning of the section are shown in Table 2.4. There are in fact two stable models: $\{p\}$ and $\{q\}$. Having two models perhaps intuitively represents the rules of the logic program, if we have p then we should not have q and vice versa but it does not specify which one. As a result, both are acceptable stable models and in general a logic

program may have many more answer sets. We say a formula A is bravely entailed by a logic program P if it is true in *at least one* stable model of P, written as $P \models_b A$ and cautiously entailed if it is true in *every* stable model, written as $P \models_c A$. The type of entailment used depends on the domain and the requirements of the task. Most of the normal logic programs handled in this thesis do not have cycles and often have a unique minimal supported Herbrand model for which both brave and cautious entailments agree with that of fixed-point semantics described in Section 2.1.4.

Answer Set Solvers The resulting programming paradigm built on the stable model semantics is called Answer Set Programming (ASP) [120]. Answer set programming is a form of declarative programming designed to handle difficult NP-hard search problems. The search problems are encoded in such a way that the answer sets of the corresponding logic program are the solutions to the search problem. Programs that find answer sets are called answer set solvers.

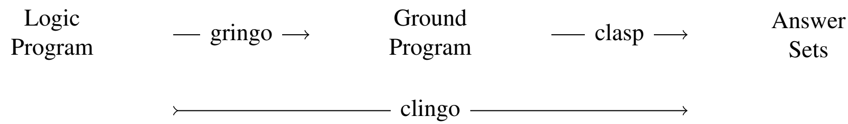


Figure 2.3: Clingo consists of two subprograms gringo and clasp.

One such modern popular answer set solver is clingo [61]. The clingo system comprises of a two stage pipeline as shown in Fig. 2.3: (i) first gringo grounds the given logic program then (ii) clasp finds answers sets. In this thesis, we are mainly interested in the stable model semantics of normal logic programs and do not utilise the full extent of answer set programming which includes constraints, choice rules, cardinality constraints and more. In Chapter 6, we utilise clingo as a tool to verify solutions and it also acts as the backbone of the state-of-the-art inductive logic programming systems described in the following section.

2.1.6 Inductive Logic Programming

Performing deduction to determine if a logic program entails a particular result does not incorporate any form of learning. From the perspective of symbolic artificial intelligence, it is crucial that a machine can not only apply the rules but also *learn* them from experience. The problem of inducing rules is studied by Inductive Logic Programming (ILP) [145] whereby an incomplete logic program is given along with positive and negative examples the program should and should not entail respectively. The problem is then to find candidate rules that complete the logic program.

Formally, given a known background knowledge B as a logic program, sets of positive examples E^+ and negative examples E^-, the objective is to find a correct hypothesis H such that:

$$
\begin{aligned}
B, H &\models e \ \forall e \in E^+ \\
B, H &\not\models e \ \forall e \in E^-
\end{aligned}
$$

as defined by Džeroski [48]. In this setting, it is common to have a logic program as background knowledge and ground atoms for positive and negative examples. It may very well be that the

background knowledge is empty if the task requires everything to be learnt as part of the hypothesis. Intuitively, the hypothesis can be seen as the missing set of rules that complete the background logic program B. If there are wrongly labelled examples in E^+ and E^- then an approximation may be made such as attempting to find an H that covers the most number of examples since it would be impossible to correctly satisfy the two conditions above.

From the lens of artificial intelligence, inductive logic programming fundamentally describes a search problem. Given examples, the system must find rules that together with B entail them. How this search for new rules is carried out can be as naive as a brute force attempt trying all possible candidate rules within the hypothesis space. State-of-the-art symbolic rule learners ILASP [114] and the more recent scalable version FastLAS [115] discover hypothesises by constructing a meta answer set program of which the optimal answer set corresponds to the rules that complete the program. Both of those methods are utilised as baselines in Chapter 6 but we leave further details of the symbolic rule learners for the readers to explore in their respective publications.

Given the recent meteoric rise of artificial neural networks in the last decade as the principal component of many machine learning models, the prospect of using them to learn symbolic rules acts as the motivating glue for many neuro-symbolic systems [58]. In the next section, we describe artificial neural networks under the umbrella of deep learning.

2.2 Deep Learning

Following an increase in accessible computing power over the last two decades, artificial neural networks became viable for practical machine learning despite having their roots in the late 50s and early 60s [174]. Building larger and deeper neural networks, powered by general purpose graphics processing units (GPUs), led to the advent of deep learning [67]. In 2012, a type of neural network architecture called convolutional neural network (described in Section 2.2.3) outperformed all existing image classification methods by solely training on labelled images with no prior feature engineering [109]. The success of deep learning extends to many other domains such as natural language processing [216, 20] and reinforcement learning [100, 126] and remains an active area of research. In this section, we describe what neural networks are, how they are trained and the common types of architectures. We build on the material from the books by Goodfellow, Bengio, and Courville [67] and Russell and Norvig [175].

As with any machine learning algorithm, the journey of neural networks also starts with data. Sampled from a data generating distribution p_{data}, we obtain a set of training examples $\mathbb{X}$ assumed to be independent and identically distributed (i.i.d.). The actual distribution induced by the training examples is called the empirical distribution $\hat{p}_{\text{data}}$. In this thesis, we follow the narrative of supervised learning whereby each training example $\boldsymbol{x}^{(i)}$ has a corresponding desired output label $y^{(i)}$. The structure and content of training examples vary from domain to domain such as images in image classification tasks, sentences or words in natural language processing and even full trajectories or episodes of games in reinforcement learning. It may sometimes be more convenient to curate all the training input examples into a matrix $\boldsymbol{X}$, called the design matrix, in which a row corresponds to a single input $\boldsymbol{X}_{i:} = \boldsymbol{x}^{(i)}$. The goal is then to learn a function, a mapping between the given inputs and desired outputs.

Definition 5 (One-hot Encoding)**.** Given a set of discrete values $\mathbb{S}$ and an arbitrary ordering, the one-hot encoding of the ith value is the ith row of the identity matrix, $\boldsymbol{I}_{i,:}$.

Unlike logic programs which operate on discrete symbols, neural networks require all inputs and outputs to be numerically encoded. This specification arises from their mathematical construction that involves neurons with weighted connections between them. Thus, it is necessary to encode given inputs and outputs into some numerical representation. There are two main types of data: discrete and continuous. Discrete, also called categorical, data is often one-hot encoded (Definition 5) that effectively assigns a unique vector to each value. On the other hand, continuous values are normalised to ranges such as $[0,1] \in \mathbb{R}$ or $[-1,1] \in \mathbb{R}$ to avoid large numbers that may cause numeric overflows. Encoding of data is not a standard process and may vary based on the individual tasks and properties of the domain. For example, one can construct a regression task in which a model is required to predict the salary of employees as continuous values; they can also turn the problem into a classification task by modelling salary ranges instead. It is up to the person to decide the most appropriate encoding for a given task, data and model.

Table 2.5 provides a summary of common data types and their encodings. For images, we often normalise pixel intensities and treat them as continuous values. One may also use the range $[-1,1] \in \mathbb{R}$ for images [125, 26]. For domains where the inputs consist of discrete *chunks* such

Table 2.5: Common data types and their encodings used with neural networks

Data	Encoding	Comment
Discrete	One-hot	Each value gets a corresponding unique vector
Continuous	Normalised	Continuous values are normalised to a reasonable range such as $[0,1]$
Image	$[0,1] \in \mathbb{R}$	Normalised pixel intensities
Natural Language	One-hot	Depending on the parsing level, either words or characters get a unique one-hot vector
Boolean	$\{0,1\}$	Truth values often follow the common numeric encodings of logic

as in natural language processing, one-hot encoding is used. The smallest chunk may depend on the level of parsing, for example whether it is word-level in which each word gets assigned a unique vector or character-level where each character is one-hot encoded. Consequently, a natural language sentence can then be represented as a sequence of one-hot encoded vectors. Finally, within the neuro-symbolic realm, it is common to deal with Boolean values using their numeric encodings as discussed in Section 2.1.1. Throughout Chapters 4 to 6, we utilise logic programs, natural language sentences and images as data along with different encoding schemes to make them compatible with novel neural network based neuro-symbolic architectures.

Many other machine learning practices, terminology and algorithms such as validation, accuracy, over-fitting and generalisation apply to neural networks and deep learning. We assume some familiarity with general machine learning topics and refer the reader to explore further in related citations [175, 19, 67].

2.2.1 Artificial Neural Networks

Inspired by the human brain, artificial neural networks present a connectionist machine learning model that incorporate neurons with weighted connections between them [175]. The connection weights are malleable and can be learnt to train a neural network to map the given inputs to desired outputs. Different numbers of neurons, different strategies of connections and methods for training all yield a wide variety of flexible architectures that make neural networks the most prominent and widely used machine learning model [67]. In this section, we provide a modern account of what neural networks are, balancing their mathematical description with technical implementation in deep learning libraries.

A *layer* in a deep learning model is an encapsulation of a mathematical function with learnable parameters, also known as weights. The notion of a layer is very flexible and state-of-the-art deep learning architectures are built using them in a modular fashion. A single layer encapsulates some operation with parameters that are going to be optimised, discussed later in Section 2.2.2. For example, a deep learning model could have attention based layers [30], memory [219], recurrent layers (Section 2.2.4), convolutional layers (Section 2.2.3) and more as architectures become more prolific and complex. In this context, the *architecture* of a model refers to the connection of such layers and often varies based on the task. In modern deep learning libraries such as TensorFlow [1]

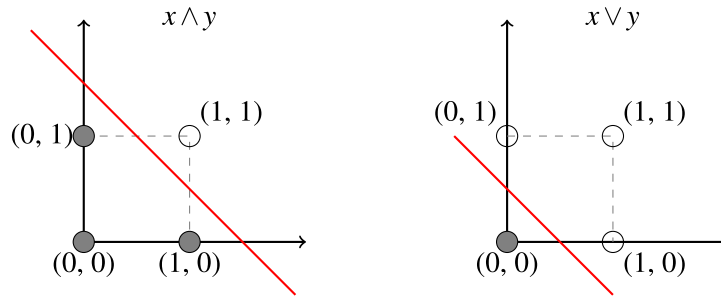


Figure 2.4: Both conjunction and disjunction are linearly separable functions. The grey and transparent circles indicate false and true respectively.

and PyTorch [152], layers may contain other layers creating a modular view of constructing neural networks through smaller learnable components.

Definition 6 (Linear Layer)**.** A linear layer has the form $\boldsymbol{y} = f(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$ where $\boldsymbol{W} \in \mathbb{R}^{M \times N}$ and $\boldsymbol{b} \in \mathbb{R}^M$ are learnable parameters, $\boldsymbol{x} \in \mathbb{R}^N$ is the layer input and f is a non-linear activation function. While $\boldsymbol{W}$ is called the weights of the layer, $\boldsymbol{b}$ is referred to as the bias. The linear layer is sometimes referenced as a dense layer due to the dense connections between inputs and outputs.

The linear layer is the most common building block of deep learning models. It gets its name from the linear transformation applied to its inputs $\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$ and extends the idea of a single perceptron [174, 175] to multiple outputs. A single linear layer can learn any linearly separable function by definition. Fig. 2.4 visually demonstrates how a line can separate the $\wedge$ and $\vee$ Boolean connectives. As a result, a single linear layer can model and potentially learn conjunction or disjunction. This observation underpins the novel differentiable layer introduced later in Chapter 6.

However, a function that is not linearly separable is outside the scope of what is learnable with a single linear layer. For example, the XOR Boolean connective, expressed as $(x \vee y) \wedge \neg(x \wedge y)$, is such a function. This limitation hampered progress on connectionist approaches and cast doubt on the perceptron algorithm in the early 70s [139]. This predicament motivates the use of multiple layers to learn intermediate transformations of the input data such that it eventually becomes linearly separable [67]. Consequently, we stack multiple linear layers with non-linear activation functions in order to learn more than just linear functions.

A *feed-forward neural network* consists of a sequence of linear layers connected to each other without any loops. The computation is carried out from the inputs to the outputs in a *forward* fashion. The layers in between the inputs and outputs are called hidden layers. If there is more than one output in any layer, it is sometimes referred to as a Multi-layer Perceptron (MLP). Although any neural network architecture without cycles may be referred to as a feed-forward neural network, in the context of this thesis, we follow the colloquial association of the terms multi-layer perceptron and feed-forward network and use them interchangeably. A multi-layer perceptron is often used as a baseline due to its conceptually simple architecture [67]. A visual diagram of a multi-layer perceptron with a single hidden layer is shown in Fig. 2.5. The number of layers in a neural network is referred to as the depth of the network and it is the trend of stacking more and more layers with more learnable parameters that gives rise to the term *deep* learning.

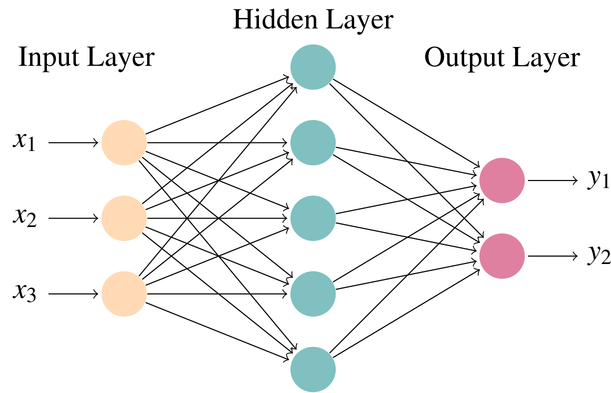


Figure 2.5: A multi-layer feed-forward neural network with 3 inputs, a single hidden layer with 5 neurons and 2 outputs. Every output is connected to every input of the subsequent layer.

After the network is constructed, one must initialise all the learnable weights. In practice, the weights of a linear layer are initialised using a normal distribution $\mathcal{N}(0,1)$ and the bias is set to 0. However, this may produce a large variance of the outputs of the layers as they are stacked in deeper networks. To overcome this issue, Glorot and Bengio [65] propose the following initialisation method referred to as Xavier uniform initialisation:

$$\boldsymbol{W} \sim U[-\frac{\sqrt{6}}{\sqrt{|\boldsymbol{x}|+|\boldsymbol{y}|}}, \frac{\sqrt{6}}{\sqrt{|\boldsymbol{x}|+|\boldsymbol{y}|}}] \tag{2.1}$$

where U is the uniform distribution and $|\boldsymbol{x}|, |\boldsymbol{y}|$ are the number of inputs and outputs of the layer. The intuition is to keep the variance of the outputs of the layers equal to 1 such that the intermediate computations do not cascade to produce very large or very small values. This leads to more stable training across various tasks. Unless otherwise stated, in our implementations we use the deep learning library's default weight initialisation method which is Xavier uniform for TensorFlow[1].

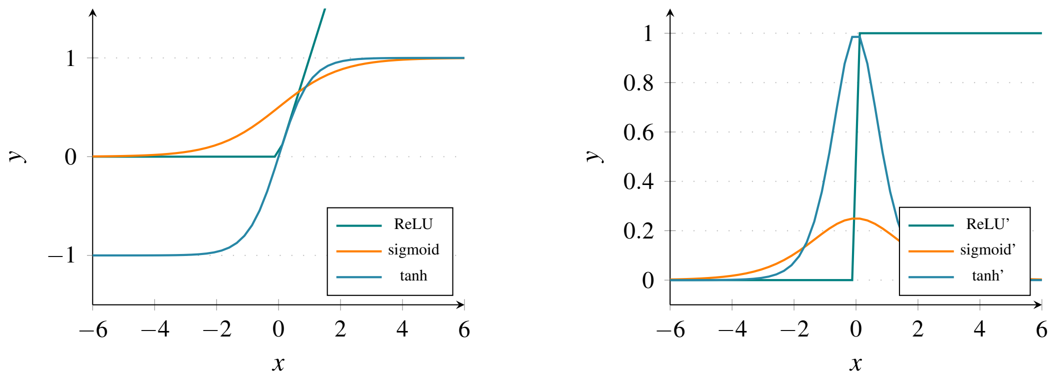
The final ingredient for completing the layer is the activation function. It is a *non-linear* function applied to the outputs of the layer. Without the activation functions, the whole feed-forward network becomes linear despite multiple layers. Consider two linear layers:

$$\boldsymbol{y} = \boldsymbol{W}^1(\boldsymbol{W}^2\boldsymbol{x}) = \boldsymbol{W}^*\boldsymbol{x} \text{ where } \boldsymbol{W}^* = \boldsymbol{W}^1\boldsymbol{W}^2 \tag{2.2}$$

without an activation function, the overall computation becomes equal to that of a single layer. Consequently, having some non-linear function between the layers is necessary to harness the full learning capacity of a neural network. Although any non-linear differentiable function may be used as an activation function, there are a few commonly used ones:

sigmoid compresses the output to be in the range $[0,1] \in \mathbb{R}$. It is also referred to as the logistic function. It can be interpreted as a soft version of the hard threshold function $x > 0$ and is

[1]Refer to https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense for more details.



(a) Plots of common activation functions.

(b) Derivatives of common activation functions.

Figure 2.6: Activation functions are non-linear functions that transform the output of neural network layers. They approximate in a differentiable manner whether a neuron is active or not.

used to model single probabilities as well as Boolean values.

$$\sigma(x) = \frac{1}{1 + (e^{-x})} \tag{2.3}$$

tanh is a scaled version of the sigmoid function that produces the range $[-1, 1] \in \mathbb{R}$.

$$\tanh(x) = 2\sigma(2x) - 1 = \frac{2}{1 + (e^{-2x})} - 1 \tag{2.4}$$

ReLU stands for Rectified Linear Unit and yields a *piece-wise linear* function. Although the components of the function are linear, the overall function is non-linear.

$$\text{ReLU}(x) = \max(0, x) \tag{2.5}$$

softmax produces an output vector such that each element is between 0 and 1 ($y_i \in [0, 1]$) and the sum is equal to 1 ($\sum y_i = 1$). It is commonly used to model the probability mass function of a categorical distribution. It can also be thought as a multi-dimensional sigmoid function.

$$\boldsymbol{y} = \text{softmax}(\boldsymbol{x}) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{2.6}$$

Fig. 2.6a plots the common scalar activation functions. It is common practice to use ReLU as the default activation function due to its desirable properties with respect to gradient descent [67]. However, depending on the task and the overall architecture, one may utilise different or a mixture of activation functions. In the following section , we describe how to train neural networks.

2.2.2 Training

The true power of neural networks lies in the accompanying effective training methods that iteratively optimise the weights to *learn* a desired mapping. This ability to learn places artificial neural networks at the centre of machine learning research and bolsters their various successes [67]. The

key ingredient is the usage of gradients in order to *nudge* the weights in the appropriate direction. This simple yet extremely powerful idea leverages the property that every weight in every layer is differentiable with respect to the output of the network. In this section, we cover how to train a neural network and some common practices involved.

Any neural network, whatever its architecture, may be viewed as a single function of its parameters and inputs $\hat{y} = f(x, \boldsymbol{\theta})$ where $\boldsymbol{\theta}$ is all the weights of all the layers. The goal then is to adjust $\boldsymbol{\theta}$ such that f starts to predict some given desired output $y \simeq \hat{y}$ where y is often called the target or labels of the corresponding input x. Consequently, we require a way of measuring how good or correct a prediction is.

Definition 7 (Loss function)**.** The loss function is a scalar performance function of the network parameters. It is also called the cost function or the objective function, denoted as $L(\boldsymbol{\theta})$.

The loss function depends solely on the parameters $\boldsymbol{\theta}$ since the inputs and desired outputs are fixed as part of the dataset $\mathbb{X}$. The formulation of the loss function depends on the task and what a *good* prediction means in that domain. For any given loss function and neural network, the objective is to find the optimal parameters:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) \tag{2.7}$$

which defines a search over the parameter space. This optimisation problem may be tackled using different methods but the most common approach is *gradient descent* which exploits the fully differentiable nature of the construction of neural networks. This assumes that the loss function is also differentiable with respect to the network output. Although having differentiable components may seem necessary, one may use alternative optimisation methods such as genetic algorithms in order to optimise neural networks in a gradient-free fashion [207, 178] which we revisit at the end of this thesis in Section 7.2.

Over-fitting occurs when the performance according to the given loss function is considerably lower for the training dataset compared to another unseen validation or test dataset. The point at which a neural network over-fits is usually when the training loss continues to improve (becomes smaller) while the test loss stagnates or worsens. As with any other machine learning algorithm, neural networks are also prone to over-fitting. While pursuing an optimal solution with respect to a fixed training dataset using Eq. (2.7), the model may converge to a solution that is not general across unseen test datasets. We encourage the reader to explore the problem of over-fitting and methods to mitigate it such as regularisation in the related citations as it is a large area of active research across many machine learning disciplines [67, 19].

A *hyper-parameter* is any value that is not optimised by the training procedure, i.e. not part of $\boldsymbol{\theta}$. Common hyper-parameters for almost every neural network are the number of layers, size of the layers as well as the overall network architecture. Hyper-parameters are set by the user often following previous works in a particular domain or task. The trainable parameters, on the other hand, are optimised using the gradient. The gradient of the loss function with respect to the

trainable parameters is all the partial derivatives of each parameter:

$$L'(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} L = \left(\frac{\partial L}{\partial \theta_1}, \frac{\partial L}{\partial \theta_2}, \ldots, \frac{\partial L}{\partial \theta_n} \right) \tag{2.8}$$

which would yield an optimal solution at $L'(\boldsymbol{\theta}) = 0$ if L is convex. However, in practice many real-world tasks such as image classification yield a non-convex search space. The choice of data, random initialisation of the parameters, regularisation and training regime all affect the performance of gradient descent methods which may converge to suboptimal *local minima* where $L'^{(}\boldsymbol{\theta}) = 0$ still holds. In practice, neural networks trained using gradient descent provide state-of-the-art solutions to complex problems in spite of potentially stagnating in local minima [67].

Since the entire network is built using differentiable operations, we can compute the derivative of the loss function with respect to every parameter utilising the chain rule:

$$\frac{\partial L}{\partial \theta^i} = \frac{\partial L}{\partial y^{i+1}} \frac{\partial y^{i+1}}{\partial y^i} \frac{\partial y^i}{\partial \theta^i} \tag{2.9}$$

where θ^i and y^i are the parameters and outputs of ith ($1 \leq i \leq n$) layer respectively. Due to the backward propagation of derivatives, the overall process is called the *back-propagation* algorithm. The back-propagation algorithm follows the connections of the layers, e.g. in a feed-forward network one layer connects to the next, and by caching intermediate results provides a uniform, efficient and practical method for computing the gradients of all the weights for any neural network architecture. Most modern deep learning libraries like TensorFlow [1] and PyTorch [152] further optimise the computation of gradients and training of neural networks for example by using vector instructions, distributed training and just-in-time compilation.

To propagate the gradient through activation functions, we multiply it with their derivatives following the chain rule, i.e. $\frac{\partial f(y)}{\partial x} = f'(y)\frac{\partial y}{\partial x}$. The derivatives of common activation functions are visually shown in Fig. 2.6b. The derivatives of both the sigmoid and tanh functions are strictly between 0 and 1. When many layers are stacked to create deeper neural networks, this property leads to *vanishing gradients* because at each encounter of sigmoid or tanh, the incoming gradients get multiplied by a number between 0 and 1. To overcome this unwanted effect, it is a common practice to use ReLU as the default activation function for modern deep learning architectures [67].

The *gradient descent* algorithm is an iterative procedure that gradually updates the parameters in the direction that minimises the loss function:

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t - \alpha \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) \tag{2.10}$$

where t is the iteration number and α is the learning rate. If updated sufficiently many times, the model converges to a solution with respect to the given loss function where the gradients come close to zero and the parameters are no longer updated. Note that this solution is with respect to $L'(\boldsymbol{\theta}) \simeq 0$ rather than the actual extrinsic task.

There are many variants that improve on the naive update of the network parameters in Eq. (2.10). The current, commonly used, state-of-the-art optimiser is called Adaptive Moment Estimation (Adam) [106]. Adam improves on the update equation by assigning an individual learning rate to each parameter and adapting it using the parameter's gradient information. The use of the gradient history generates *momentum* by increasing the learning rate along gradient directions with larger magnitude. Informally, the learning rate speeds up the steeper the gradient. The overall result is a faster, more robust parameter optimisation schedule at the cost of storing individual learning rates for every parameter. The efficacy of different optimisation routines may ultimately depend on the nature of the learning task as well as the network architecture. Unless otherwise stated, we utilise Adam to update our network weights in Chapters 4 to 6.

We now give concrete examples for the loss function and their common use cases. We define the loss function in terms of a desired output y and the model prediction $\hat{y}$ such that $L(\boldsymbol{\theta}) = l(\mathbf{y}, \hat{\mathbf{y}})$ and $\hat{\mathbf{y}} = f(\boldsymbol{x}, \boldsymbol{\theta})$ where f is the entire neural network itself:

Mean Squared Error (MSE) is suitable for regression tasks where the output range is $[-\infty, \infty]$. It is also used for image reconstruction tasks in which the error is computed between the desired image and the image produced by the network [26, 125].

$$l(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \tag{2.11}$$

where N is the size of the output vector.

Binary Cross-entropy is used for supervising binary decisions such as true versus false, match versus no match. It is also known as binary negative log-likelihood loss function. For a given label $y \in \{0, 1\}$ and prediction $\hat{y} \in [0, 1]$, it is defined as:

$$l(y, \hat{y}) = -\left(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})\right) \tag{2.12}$$

Categorical cross-entropy extends the binary cross-entropy for multi-class classification tasks. It is also known as the negative log-likelihood loss. The cross-entropy loss is defined as:

$$l(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{k} y_k \log(\hat{y}_k) \tag{2.13}$$

where k is the number of classes.

The final ingredient is the utilisation of a concrete dataset to compute the gradients of the parameters so as to optimise them using Eq. (2.10). Modern machine learning datasets are larger in size than most commodity hardware memory, often measured in gigabytes [132]. This presents a challenge to compute predictions of the network and gradients of its parameters. In *Stochastic Gradient Descent* (SGD), the gradient of the parameters are computed based on a *random sample* of a given dataset. This random sample is called a *batch* and its size is referred to as the batch size. By randomly sampling data points, we reduce the amount of computation required and discourage the model from seeing the same data points grouped together which in return decorrelates

the data points and helps mitigate over-fitting. Stochastic gradient descent colloquially refers to *mini-batch gradient descent* which takes small batches of data points, often a power of two to align with computer hardware infrastructure such as 32, and averages the gradients from each data point in the batch. In this thesis, we follow that association and unless otherwise stated use mini-batch gradient descent with a fixed batch size.

Finally, to *train* a neural network, we:

1. Curate a dataset $\mathbb{X}$ that consists of many data points with inputs $\boldsymbol{x}^{(i)}$ and desired outputs $y^{(i)}$.
2. Sample a random mini-batch of data points with a fixed size such as 32.
3. Compute the predictions of the network for each data point in the batch. This step is sometimes referred to as the forward pass of the network as the output is computed from the inputs in a forward fashion.
4. Compute the gradient of the given loss function (Definition 7) with respect to the output of neural network. This would be the first term in the chain rule from Eq. (2.9).
5. Back-propagate to compute the gradient of the loss with respect to every parameter in the neural network. This step is referred to as the backward pass since the chain rule is applied backwards through the network's computation graph.
6. Update the parameters using Eq. (2.10) or an extension of it such as Adam.
7. Repeat steps 2 to 6 until a convergence criteria is reached.

The exact implementation and steps of the training procedure as well as the convergence criteria may vary from task to task. We follow the training routine above for all experiments in Chapters 4 to 6 implemented with the aid of deep learning libraries such as TensorFlow [1] and Chainer [213].

2.2.3 Convolutional Neural Networks

One of the most powerful features of artificial neural networks is their flexible architecture. One can construct different connection patterns without having to reinvent other components and training schedules described in Sections 2.2.1 and 2.2.2. As a result, neural networks are built with biases that exploit the characteristics of the problem they are trying to solve. Convolutional Neural Networks [67] are a class of neural networks which exploit translational invariance of the input data. The most prominent examples are images where visual features remain similar regardless of their position in the picture. In this context, an image is considered a matrix of pixel intensity values. Consider an image classification task, the identifying object such as a cat may appear anywhere in the image and consequently a regular feed-forward neural network would have to learn that cats at different locations are still cats.

Despite their existence since the 80s [116], with the advent of increased computing power over the last decade, convolutional neural networks were able to take centre stage. In 2012, Krizhevsky, Sutskever, and Hinton [109] used a deep convolutional neural network to outperform all existing methods on the image classification challenge ImageNet. This outstanding success rekindled the

Input			Kernel		Convolution	
1	2	3	1	2	23	33
2	3	4	3	4	33	43
3	4	5				

(a) Example convolution operation for given inputs and kernel with size 2, stride 1 and no padding.

Input			Max Pool	
1	2	3	3	4
2	3	4	4	5
3	4	5		

(b) Example max pooling operation for given inputs with a window of size 2×2 and stride of 1.

Figure 2.7: Examples of the two common operations found in convolutional neural networks.

interest in neural networks and their new found practical ability with modern computer hardware. Almost all successive entries to ImageNet became neural network based such as ResNet [76], a very deep convolutional neural network with over 100 layers that utilise residual connections across layers.

Unlike feed-forward networks in which every input-output connection is unique, convolutional neural networks are inspired by the receptive fields in the human-eye [121]. They mimic a connection pattern that incorporate convolving a small region of the input with a learnable weight, also known as a kernel. The same kernel *slides* over the input grid to produce an output at each position. Fig. 2.7a demonstrates this computation for a simple 3×3 input grid and a 2×2 kernel. The output at each position is the sum of the multiplication of the inputs with the kernel. One can view the convolution operation as repeated applications of the same linear layer (Definition 6) over windowed regions of the input. Just as with a feed-forward network, in between each convolution operation, we also use an activation function to ensure the overall network is non-linear.

The other operation found in convolutional neural networks is downsampling. Downsampling is often realised by pooling layers that reduce the size of the input by merging, i.e. pooling, the values in a small sliding window. Similar to the convolution operation, pooling produces an output for each input region. A common pooling operation is max pooling in which the maximum value of the input region is taken. Fig. 2.7b computes the output of max pooling over the same 3×3 input with a pooling window size of 2×2.

A convolutional neural network consists of many stacked convolutional layers followed by pooling layers for downsampling. The exact architecture, number of layers, their size and order is an active area of research and often depends on the task. For modern deep learning libraries, the following hyper-parameters are used in constructing convolutional layers:

Number of filters correspond to the number of distinct kernels that determine the output size of the convolution operation. It is akin to the number outputs of a linear layer. The term filter comes from traditional computer vision techniques where image filters are used.

Kernel size is the size of the input window such as 3×3. The total number of learnable weights in a convolutional layer is then the kernel size multiplied by the number of filters. For 32 filters and a kernel size of 3×3, the layer would have 288 parameters (weights).

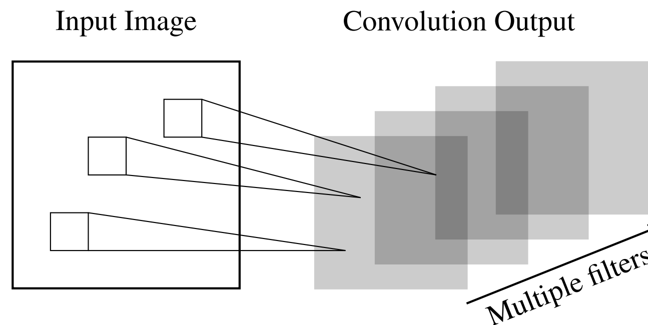


Figure 2.8: Graphical overview of the convolution operation with multiple kernels. For a single filter, also known as a kernel, each input region gets convolved using the same set of weights. This process is repeated with every filter to produce multiple output features.

Strides determine the number of cells to skip when sliding the window over the input grid. It reduces the output resolution in cases where the input does not vary too much.

Padding addresses the issue of what happens on the *edge* of the input grid. Either the input window is padded with zeros to match the kernel size or the edges are ignored.

The last three arguments also apply to pooling layers. An input window size, strides and padding must all be set by the user when building the convolutional neural network. The amount of expert design that a user must input to construct these networks, has also spawned research into automatic discovery of suitable neural network architectures, for example by using genetic algorithms [208] as well as a quest to find the ultimate neural network architecture [96, 95].

Fig. 2.8 visualises a convolutional layer with multiple kernels. This operation is often computationally optimised using graphic processing units and performed in a parallel fashion instead of multiplying each kernel with each input region sequentially. The resulting overall convolutional neural network yields the following desirable properties:

Local connectivity exploits spatial correlations of the input. As a result, convolutional networks are more apt at learning local features which is a natural fit for image processing.

Translational invariance is attained by design since the same learnable weights are applied across different input patches. Any important feature can be learnt regardless of their position.

Parameter sharing greatly reduces the number of learnable parameters required compared to a linear layer. For example, for a 3×3 input and a single output, a linear layer requires 9 weights (one for each input) while a kernel of size 2×2 only needs 4 as in Fig. 2.7a.

Due to their natural fit for image processing, we utilise convolutional neural networks in Chapter 6 as the first component of a larger neuro-symbolic architecture. Using convolutional layers to process images as part of more complex networks is a common trend found in many recently published works [179, 26, 125, 7, 191, 229, 92, 142, 102]. It highlights the flexibility of neural networks despite using the same building blocks from Sections 2.2.1 and 2.2.2.

Although inspired by the photoreceptive neuronal architecture of the human eye, convolutional

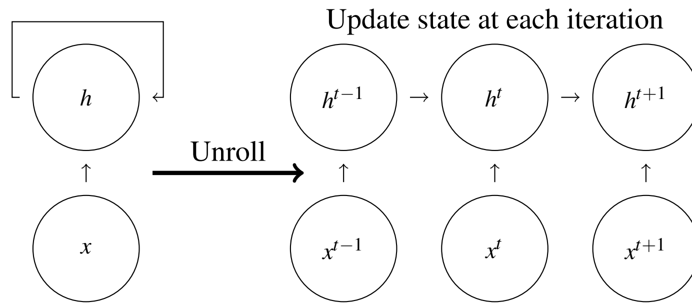


Figure 2.9: Conceptual overview of a recurrent neural network. The output of the network depends not only on the input but also its past hidden state h. When unrolled, it is applied to each time-step from the beginning of the input to the end.

networks are an abstraction of the biological visual processing pipeline. In the abstract of their work, Spoerer, McClure, and Kriegeskorte [204] highlight their lack of "the lateral and feedback connections, and the resulting recurrent neuronal dynamics, of the ventral visual pathway in the human and non-human primate brain". This recurrence in the computation, or lack of, leads us to another common architecture type that is designed to process temporal data.

2.2.4 Recurrent Neural Networks

While convolutional neural networks are best suited for spatially organised data, recurrent neural networks are designed to process sequences of inputs. Recurrent neural networks [67] are a class of neural networks that incorporate some form of loop in the computation graph, i.e. the output of a layer becomes the input for itself. In its most basic form, we extend the notion of a linear layer from Definition 6 to include a hidden recurrent state vector $\boldsymbol{h}$:

$$\boldsymbol{h}^t = f(\boldsymbol{h}^{t-1}, \boldsymbol{x}^t; \boldsymbol{\theta}) \tag{2.14}$$

where t is the input time-step. The initial hidden state $\boldsymbol{h}^0$ is often set to zeros or as another learnable weight. For example, if the input is a sequence of words, $\boldsymbol{x}^0$ would be the encoding of the first word, $\boldsymbol{x}^1$ the second and so on. Intuitively, the model has a way of capturing the history of inputs whereby $\boldsymbol{h}^t$ could represent features up to and including that time-point. By design, recurrent networks can handle inputs of varying length since there is no inherent limit on how many times Eq. (2.14) can be iterated. Another way of viewing the recurrence is by unrolling the network as shown in Fig. 2.9. In that case, it is akin to a feed-forward network with weight sharing across the temporal dimension. As a result, the methods for training feed-forward neural networks in Section 2.2.2 still apply to recurrent networks, and the gradients are computed based on the final unrolled network. A notable disadvantage of recurrent networks is their lack of parallel computation across the time-steps as each hidden state requires the previous state; hence, the computation must be carried out in a sequential fashion and cannot benefit from modern computer hardware geared towards parallelism such as graphics processing units.

Iterating the naive, also called vanilla, recurrent neural network from Eq. (2.14) becomes troublesome over longer sequences where vanishing gradients hamper it from learning long-distance

correlations. The length of the sequence can be viewed as the depth of the unrolled network and with a non-linear activation such as tanh applied at each time-step, the magnitude of the gradients get smaller and smaller (refer back to Section 2.2.2 for further details). To overcome this limitation, Hochreiter and Schmidhuber [81] have proposed the Long Short-Term Memory (LSTM) model which *learns* what to do at each time-step in a more fine-grained fashion. It is defined as:

$$
\begin{aligned}
\boldsymbol{f}^t &= \sigma(\boldsymbol{W}\left[\boldsymbol{x}^t, \boldsymbol{h}^{t-1}\right] + \boldsymbol{b}) && \text{(forget gate)} \\
\boldsymbol{i}^t &= \sigma(\boldsymbol{W}\left[\boldsymbol{x}^t, \boldsymbol{h}^{t-1}\right] + \boldsymbol{b}) && \text{(input gate)} \\
\boldsymbol{o}^t &= \sigma(\boldsymbol{W}\left[\boldsymbol{x}^t, \boldsymbol{h}^{t-1}\right] + \boldsymbol{b}) && \text{(output gate)} \\
\tilde{\boldsymbol{c}}^t &= \tanh(\boldsymbol{W}\left[\boldsymbol{x}^t, \boldsymbol{h}^{t-1}\right] + \boldsymbol{b}) && \text{(cell input gate)} \\
\boldsymbol{c}^t &= \boldsymbol{f}^t \odot \boldsymbol{c}^{t-1} + \boldsymbol{i}^t \odot \tilde{\boldsymbol{c}}^t && \text{(cell state)} \\
\boldsymbol{h}^t &= \boldsymbol{o}^t \odot \tanh(\boldsymbol{c}^t) && \text{(hidden state)}
\end{aligned}
$$

where each $\boldsymbol{W}$ and $\boldsymbol{b}$ are different parameters in each equation and $[,]$ is vector concatenation. The improvements over the vanilla neural network relies on the two internal states: a hidden state $\boldsymbol{h}$ and a memory cell $\boldsymbol{c}$. The core novelty lies in the penultimate equation that determines the next cell state. At a particular time-step, the forget gate allows the network to erase past information while the input gate determines how the new state should be incorporated. Intuitively, one can summarise the LSTM as a two-step process: (i) how does the past hidden state $\boldsymbol{h}^{t-1}$ and the current input $\boldsymbol{x}^t$ affect the previous memory $\boldsymbol{c}^{t-1}$ and (ii) how does the new memory state $\tilde{\boldsymbol{c}}^t$ translate to the next output $\boldsymbol{h}^t$. This dynamic control over the internal states allows the model to ignore irrelevant steps and learn correlations over longer sequences. Fig. 2.10 provides a graphical overview of the LSTM connection architecture. As with the vanilla recurrent neural network, the hidden state is used as the output of the LSTM at time-step t despite being referred to as the hidden state vector.

The LSTM model may incur an overhead due to the increased computation required at each time-step. In order to reduce the number of parameters while keeping the idea of an adaptive gating mechanism, the Gated Recurrent Unit [31] offers a simpler version with just one hidden state and two learnable gates:

$$
\begin{aligned}
\boldsymbol{z}^t &= \sigma(\boldsymbol{W}\left[\boldsymbol{x}^t, \boldsymbol{h}^{t-1}\right] + \boldsymbol{b}) && \text{(update gate)} \\
\boldsymbol{r}^t &= \sigma(\boldsymbol{W}\left[\boldsymbol{x}^t, \boldsymbol{h}^{t-1}\right] + \boldsymbol{b}) && \text{(reset gate)} \\
\tilde{\boldsymbol{h}}^t &= \tanh(\boldsymbol{W}\left[\boldsymbol{x}^t, \boldsymbol{r} \odot \boldsymbol{h}^{t-1}\right] + \boldsymbol{b}) && \text{(candidate hidden state)} \\
\boldsymbol{h}^t &= \boldsymbol{z}^t \odot \tilde{\boldsymbol{h}}^t + (1 - \boldsymbol{z}^t) \odot \boldsymbol{h}^{t-1} && \text{(hidden state)}
\end{aligned}
$$

where $\boldsymbol{W}$ and $\boldsymbol{b}$ in each equation are different learnable weights. The GRU model utilises the current input and the past hidden state to determine how to update the hidden state and whether

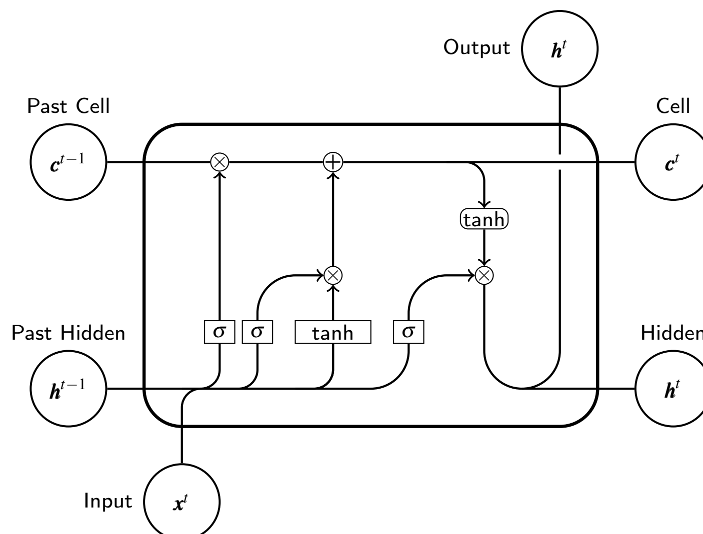


Figure 2.10: Graphical overview of the connections in a Long Short-Term Memory (LSTM) recurrent neural network. The model utilises the past hidden state and the current input in order to update the memory cell which in return determines the next output.

that candidate state should be the new hidden state. For example, for a simple task of counting the number of 1s in a binary sequence, the model may learn to set the update gate $\boldsymbol{z}$ to zeros when an input 0 is encountered. This way, it can carry over the previous sum potentially captured by $\boldsymbol{h}^{t-1}$. Although this may be an intuitive insight into the design of the model, in practice, there is no guarantee that any trained model will behave according to expectations. This is because, the data, the training routine and the random initialisation of the weights all contribute to the final solution learnt by the overall neural network. Building on the previous example of counting the number of 1s in binary sequences, if the dataset only consists of sequences with always four 1s, a constant function, then the model can easily predict four without having to utilise the mechanics of a recurrent neural network.

We refrain from detailing further variants and extensions of LSTMs and GRUs as they are outside the scope of this thesis. Regardless of how the recursion is modelled, a recurrent neural network holistically offers the following desirable properties:

Variable length sequences are naturally modelled. The recurrence in the network allows to process inputs of arbitrary length and often captures the semantics of the extrinsic time-series task. This property also makes them suitable for online learning in which there may be a stream of live input data.

Weight sharing again reduces the number of parameters needed to model sequences similar to how convolutional neural networks apply the same set of weights to each receptive field.

Due to their suitability for modelling sequences, we extensively use recurrent neural networks in Chapters 4 and 5 when dealing with logic programs and sequences of symbolic input. For example, a ground atom such as $p(a)$ can be considered a sequence of four characters. In Section 4.1, we take this insight further to learn dense vector embeddings of logical constructs whilst attempting to perform deductive inference.

The lack of inherit parallelism and the potential of vanishing gradients in recurrent neural networks have led to the invention of Transformers [216] to model sequences of inputs. They leverage self-attention in which every input symbol may attend to every other input symbol in stacked encoder layers involving position-wise feed-forward networks, normalisation layers and residual connections. They form the basis for many state-of-the-art deep natural language models [45, 25, 177, 20, 158]. We leave the reader to explore Transformers in the relevant citations as we do not utilise them in this thesis since for short (≤ 20 symbols) input sequences and the desired outcomes surrounding logic-based inference such as learning invariants in Chapter 5 and symbolic rules in Chapter 6, the aforementioned advantages of Transformers are not immediately relevant.

Going beyond sequential inputs, Graph Neural Networks (GNN) [14] operate on graph data structures and reflect the inductive bias of having objects (nodes) and relations (edges) in the architecture of a neural network. From this perspective, a recurrent neural network unrolled on a sequence of inputs x^{t-1}, x^t, x^{t+1} as shown in Fig. 2.9, can be considered a special case of a graph network in which the nodes are sequentially connected with directed edges. The computation is carried out using *message passing* iterations whereby each node of the GNN updates its internal hidden state based on its previous hidden state and incoming messages similar to Eq. (2.14). The new state of a node is then passed as input (message) to neighbouring nodes in the next iteration. GNNs have shown success in graph based problem domains such as social networks, physical systems and chemistry [230]. However, in this thesis we mainly focus on sequential data such as natural language sentences for which graph and recurrent networks are similar, and logic-based outcomes such as learning invariants in Chapter 5 and symbolic rules in Chapter 6 for which the benefits of modelling neural networks as graphs are not directly applicable. Hence, we do not use Graph Neural Networks in this thesis.

This concludes the technical background surrounding neural networks and deep learning. In the remainder of this thesis, we present our novel contributions starting with the new generated datasets in the next chapter.

Chapter 3

Generating Symbolic Data

A fully differentiable neural network based neuro-symbolic architecture inherits the data-hungry nature of deep learning [132]. Although the amount of data and training required may change depending on the specific task and architecture, it is nonetheless a pivotal component. In this chapter, we present three new automatically generated symbolic datasets: (i) a collection of stratified logic programs of increasing complexity in Section 3.1 to learn syntactic logical inference, (ii) symbol patterns in sequence and grid based structures in Section 3.2 to learn the notion of varying symbols and (iii) the scalable combinatorial problem of subgraph set isomorphism in Section 3.3 for scalable inductive logic programming. Table 3.1 summarises the tasks and the various testing conditions of the datasets. The publicly accessible source code of each dataset is available from their respective publications in Section 1.5.

The datasets in this chapter are all *synthetic*, i.e. they are created by an algorithm following a fixed data generating process. While real-world applications of neuro-symbolic models are central to assessing their benefits, real-world datasets may come with their intrinsic biases such as encouraging a specific type of inference or lack in size to reliably train a model in the first place. This property limits the ability to analyse the behaviour of novel architectures in a neutral fashion and may lead to equivocal conclusions along with difficulties in obtaining reproducible results. Synthetic datasets, on the other hand, give access to the data generating distribution p_{data}.

Table 3.1: Summary of generated datasets and tasks

Dataset	Tasks		Tests
Stratified Logic Programs	Facts N-Step Deduction Disjunction Negation as Failure	Unification Conjunction Transitivity	Unseen symbols Longer programs Irrelevant rules
Structured Symbol Patterns Sequences & Grids	Constant Head Tail Duplicate	Box Head (Grid) Centre Corner	Varying symbols Unseen inputs Correct patterns
Subgraph Set Isomorphism	Inductive Logic Programming		Scalability

Consequently, the exact parameters of the distribution may be adjusted to change the difficulty and the size of the dataset as well as reduce any biases. By fixing the random seed of any data generating distribution, the same dataset can be produced. All these aspects contribute to the creation of a *controlled environment* for experimentation, at least as a first step to probe a novel model. For example, in Section 3.1, the length of the logic programs, predicates and constants as well as the type of deductive reasoning can be modified to create various training, evaluation and test conditions. Another advantage of synthetic datasets, particularly those presented in this chapter, is their relatively small size and therefore computationally less demanding nature which allow rapid experimentation and prototyping in an accessible and reproducible manner. However, the assumptions made in and the simplicity of the synthetic datasets may not adequately reflect real-world problems and the ensuing complexity found therein, particularly for natural language processing [220, 198, 177, 36] and visual reasoning [225, 226, 26, 191] domains.

3.1 Stratified Logic Programs

One of the most prominent types of inference is deduction. Humans deduce from a wealth of information many logical conclusions on a day to day basis such as whether to take an umbrella before going outside. Deductive reasoning involves reaching a conclusion by applying known rules. A machine, on the other hand, has to be programmed in order to *know* how to apply such rules and perform deductive reasoning, for example the proof systems discussed in Sections 2.1.4 and 2.1.5. What if, instead, a neural network based neuro-symbolic architecture was to *learn* how to represent and apply rules as well as deduce facts. This task demands a large dataset capturing various aspects of deduction in order to not only train but also evaluate models. We later present a novel neuro-symbolic model capable of learning the steps of deductive inference in Chapter 4.

The stratified logic programs dataset is a synthetic, modular and scalable collection of logic programs that are procedurally generated. There are 12 classes of logic programs that exemplify increased levels of complexity of logical reasoning ranging from simple fact matching to negation as failure. It builds on the material surrounding classical first-order logic, logic programs and entailment discussed in Sections 2.1.2 to 2.1.4. The overall goal of the dataset is a decision problem: determine whether a given logic program P syntactically entails a ground query atom q, written as $P \vdash q$. To this end, we generate triples (P, q, y) where P is the context program, q the query and y the ground truth. The target label y is generated using direct argument to identify the relevant facts needed in order to prove the query and verified using the steps of backward-chaining described in Section 2.1.4. To create a balanced dataset, for each program a pair of query atoms (q, q') are generated such that $P \vdash q$ and $P \not\vdash q'$.

Unlike real-world logical reasoning datasets such as the crowdsourced Stanford Natural Language Inference dataset [23] which probes whether two English sentences entail or contradict each other, the semantics of the logic programs are formalised and unambiguous, The individual tasks in this dataset are explainable and verifiable with respect to the outcome, i.e. if $P \vdash q$ then there is a traceable sequence of operations that will substantiate it whereas for the aforementioned natural language task, researchers have resorted to crowdsourcing explanations to create

Table 3.2: Sample programs from tasks 1 to 5

1: Facts	2: Unification	3: 1 Step	4: 2 Steps	5: 3 Steps
e(l). i(u). n(d). v(h,y). p(n).	o(V,V). i(x,z). y(w,d). p(a,b). t(A,U).	g(L,S) :- x(S,L). x(a,m). y(X) :- r(X). p(h). s(t,v).	x(G,B) :- k(G,B). k(Z,V) :- g(Z,V). g(k,k). e(k,s). p(L,G) :- v(G,L).	p(P,R) :- b(R,P). b(A,L) :- a(A,L). a(W,F) :- v(F,W). v(t,i). l(D) :- t(D).
? e(l). 1 ? i(d). 0	? o(d,d). 1 ? o(b,d). 0	? g(m,a). 1 ? g(a,m). 0	? x(k,k). 1 ? x(k,s). 0	? p(t,i). 1 ? p(i,t). 0

the e-SNLI dataset [28]. The synthetic natural language reasoning datasets RuleTaker [36] and ProofWriter [210] generate facts and rules such as 'Bob is big.' and 'Big people are rough.' to teach deep language models logical entailment. However, the usage of natural language is unnecessary as the core deductive inference is easily captured in first-order logic, e.g. 'big(bob).' and 'rough(X) :- big(X).' offering a more compact, precise and verifiable representation. Finally, logic programming test suites such as the Prolog standard conformance tests [37] are limited in size insofar as training neural networks and contain programming concepts unrelated to logical inference such as input-output and exception handling. Hence, the dataset in this section serves as a practical, low-cost and flexible starting point for probing how neural networks may learn to perform deductive inference.

Rather than inundating a model with arbitrarily complex logic programs, we take inspiration from recent work surrounding prerequisite tasks for natural language reasoning [220]. We lay out 12 classes of normal logic programs that are of increasing complexity building up on previously seen concepts in a curriculum like fashion. The context logic programs contain the simplest possible rules for forcing out the required deductive step and fill the remainder with unrelated rules as noise. When viewed in isolation, each class is similar to a unit test for a proof system, a terminology used in software engineering to test the correctness of programs. Samples for each class of logic programs can be found in Tables 3.2 to 3.4.

Facts The simplest task consists only of facts. The query is successful when it appears in the context program but may fail in 3 different ways: (i) a constant argument might not match, (ii) the predicate might not match or (iii) the query might not be in the context at all. Except for the last failure case, this task reduces to the same decision problem of ground atom unification. These failures can cascade and a query may fail for multiple reasons with equal probability.

Unification These logic programs extend the decision problem surrounding unification from the previous task to include variables. They contain rules with empty bodies and an atom with variables in the head to emphasise the semantics of unification between different $p(X,Y)$ and identical variables $p(X,X)$. The query succeeds if the corresponding variables unify, and fail otherwise.

N-Step Deduction One prominent feature of deductive reasoning is the repeated application of rules to reason with causal chains, i.e. one thing leads to another. These classes of logic programs present a single positive atom in the body to create chains of arbitrary steps. For the default dataset, we include up to 3 steps of which samples are shown in Table 3.2. The query succeeds when the

Table 3.3: Sample programs from tasks 6 to 8

6: Logical AND	7: Logical OR	8: Transitivity
f(P,U) :- b(P) , p(U).	e(D,X) :- n(D,X).	f(A,W) :- q(A,P) , d(P,W).
b(x).	e(D,X) :- w(D,X).	q(h,t).
p(x).	e(w,y).	d(t,j).
p(y).	n(n,j).	q(d,m).
e(y,v).	w(t).	d(n,g).
e(x,v).	z(i,j).	s(S,F) :- x(S,A) , e(A,F).
? f(x,x). 1	? e(n,j). 1	? f(h,j). 1
? f(y,x). 0	? e(i,j). 0	? f(d,g). 0

body of the last rule in the chain is grounded with the same constant arguments of the query atom. We occasionally swap the variables $p(X,Y)$:-$q(Y,X)$. to emphasise the variable binding aspect of rules which can happen at any rule in the chain or not at all. The failure cases at the end of the rule chain are the same as those in the first task. In Chapter 4, we examine the behaviour of neural networks when faced with this task and in Chapter 5 propose a novel architecture that can learn the common pattern of multi-step deduction represented by these logic programs.

Logical AND Building upon the previous classes of logic programs, the conjunction task involves a rule with 2 body literals. The proof system now has to keep track of and prove both body literals to succeed. In backward-chaining systems such as Prolog [37], this is often done in a left-to-right fashion whilst storing an active queue of derived sub-goals whereas it is unknown as to how sub-goals may be represented in a neuro-symbolic architecture which we probe further in Chapter 4. A failure occurs when one randomly selected body literal fails for reasons similar to the first task.

Logical OR Having multiple rules that match the query captures the semantics of disjunction in logic programming. They create branches in the search space as several possible paths may lead to a successful proof. In this task, the normal logic programs branch the query predicate 3 ways: 2 rules and 1 fact. The query succeeds when *any* of the rules lead to a successful match and fail when *all* the matching rules fail.

Transitivity Based on the previous conjunction task, the transitive case introduces existential variable binding. The sub-goals derived from the body literals are now tied together by sharing an argument. The query succeeds when the inner variable unifies with the ground instances or fails otherwise. One would expect a proof system to be able to solve conjunctive and disjunctive tasks in order to tackle transitive rules since it involves both sub-goal derivation and branching.

N-Step Deduction with Negation These tasks introduce the concept of *negation as failure*. The body literal of the first matching rule is negated and a chain of arbitrary length is generated similar to the non-negated case. The query is successful when the negated body atom cannot be proved following the semantics of negation as failure. From the perspective of a backward-chaining proof system, negation as failure requires only 1 bit of extra information, i.e. whether the goal is negated or not. In Chapter 4, we analyse and visually demonstrate the effect of negation on the representations learnt by a neural network.

Logical AND with Negation We consider negation together with conjunction and randomly

Table 3.4: Sample programs from tasks 8 to 12

9: 1 Step NAF	10: 2 Step NAF	11: AND NAF	12: OR NAF
s(X,J) :- not p(J,X). p(e,x). v(V,Q) :- u(V,Q). o(N) :- not q(N). t(x,e). m(y,c).	r(C) :- not o(C). o(P) :- l(P). l(o). g(u). p(U,L) :- e(U,L). p(X,X).	b(G,B) :- not i(G) , u(B). i(w). g(a). u(a). f(t). l(W) :- a(W) , d(W).	y(Z) :- not e(Z). y(Z) :- b(Z). y(r). e(d). s(a). b(m).
? s(x,e). 0 ? s(e,x). 1	? r(u). 1 ? r(o). 0	? b(a,a). 1 ? b(w,a). 0	? y(a). 1 ? y(d). 0

negate one of the body literals as shown in Table 3.4. The query succeeds when the negated body atom fails and the other literal succeeds. The query can fail when either body literal fails similar to the non-negated conjunction task.

Logical OR with Negation Finally, we consider negation together with disjunction and negate the body literal of one of the branching rules. The query succeeds when any matching rule except the negated one succeeds and fails if the negated rule succeeds while other matching rules fail.

By default, the arity of the predicate symbols are randomly selected as 1 or 2 and are mixed within a single program. In addition to selecting the class of the logic program, the dataset generation can be customised using the following parameters:

Number of programs allows for different dataset sizes to be generated such that sufficient number of examples are obtained to train neural networks. The overall dataset is always balanced with respect to the target labels since pairs of queries are generated.

Number of noise rules adjusts the number of irrelevant extra rules that are added onto the minimal logic program corresponding to the task in order to gauge whether models learn to successfully ignore them.

Constant, predicate and variable lengths can be adjusted to utilise unique variable and predicate names so as to test generalisation to unseen symbols. Even with just two English characters, there are 26^2 unique symbols.

N-step task length may be modified to generate tasks involving multi-step deduction with longer rule chains which is useful for evaluating models on longer chains at test time.

Overall, this dataset yields a controlled, adjustable and balanced environment to train and evaluate neural networks on the topic of logical entailment represented as logic programs. The data generation process can be further extended by the community to include further tasks and evaluation regimes such as mixing different classes together of which examples are provided in the accompanying source code for Chapter 4 linked in Section 1.5. In the next section, we move away from logic programs and present simple structured symbolic patterns.

Table 3.5: Sample context, query and answer triples from the sequence and grid tasks

Sequence Dataset				Grid Dataset			
Task	Context	Query	Answer	Task	Context	Query	Answer
i	1488	constant	2	i	0 0 0 0 2 2 8 2 2	box	2
ii	6157	head	6	ii	4 0 0 0 7 0 8 0 1	head	4
iii	1837	tail	7	iii	0 6 0 1 7 2 0 3 0	centre	7
iv	3563	duplicate	3	iv	8 0 0 5 6 0 2 4 1	corner	2

3.2 Structured Symbol Patterns

Recognising patterns is a core subject of machine learning [19]. In order to facilitate neuro-symbolic research aimed at identifying varying symbols and their role in extracting common patterns in data, we create a collection of symbol pattern matching tasks that involve sequential and grid-like structures. Each data point consists of a triple (C, q, y) generated from a task specific pattern in which C is the context, q the query and y is the desired target symbol. The goal of the dataset is to identify the varying symbol in the input across both structures (sequences and grids) and uncover the 8 common patterns summarised in Table 3.5.

Fixed Length Sequences We generate sequences of a fixed length l from a set of n symbols represented as digits to predict (i) a constant symbol (digit 2), (ii) the head of the sequence, (iii) the tail and (iv) the duplicate symbol. For the duplicate symbol task, the sequence has unique symbols except for the duplicated one. Examples for $l = 4$ and $n = 8$ are shown in Table 3.5.

Grid To spatially organise symbols, we generate a grid of size $d \times d$ again from a set of n symbols. The grids contain one of (i) 2×2 box of identical symbols, (ii) a vertical, diagonal or horizontal sequence of length 3, (iii) a cross or a plus shape and (iv) a triangle with the remaining cells padded with zeros. In each task, the desired target symbol is (i) the identical symbol, (ii) the head of the sequence, (iii) the centre of the cross or plus and (iv) the corner of the triangle respectively. Samples with $d = 3$ and $n = 8$ are shown in Table 3.5.

We randomly generate 1000 triples for each structure type and discard any duplicate data points. This step ensures that any split of the dataset is disjoint and a model trained on one split will encounter previously unseen examples during evaluation and testing. The numbers of unique data points generated for each task with the default values from 5 runs are shown in Table 3.6. The reason for the smaller Grid task (i) size is that there are at most 32

Table 3.6: Training sizes for randomly generated sequences and grid tasks with 8 unique symbols.

Task	Sequence	Grid
i	704.7 ± 12.8	25.6 ± 1.8
ii	709.4 ± 13.8	623.7 ± 14.1
iii	709.7 ± 14.0	768.2 ± 12.5
iv	624.8 ± 12.4	795.2 ± 10.3

combinations of 2×2 boxes in a 3×3 grid with 8 unique symbols. The length of the sequences, the size of the grid, the set of unique symbols as well as the number of random samples can all be customised to create different controlled experiments. The particular instantiations of this dataset and its utilisation are described in Chapter 5.

Although the patterns themselves may look trivial, the objective of this dataset is to capture the notion of a *varying* symbol. Datasets designed to evaluate abstract reasoning and pattern recognition such as Raven-style progressive matrices [13] amalgamate many transformations and relations. In contrast, the tasks here are designed in such a way that only the target symbol varies consistently with the pattern requested by the query to provide a simple and computationally less demanding test bed for probing variable symbol recognition and pattern matching. Later in Chapter 5, we propose a novel neural network architecture that can not only correctly predict the varying symbol but also recognise and extract the common underlying patterns using variables.

The following section focuses on a specific type of combinatorial decision problem that yields an unbiased search for graphical patterns represented as logical rules.

3.3 Subgraph Set Isomorphism

One measure of scalability in Inductive Logic Programming (Section 2.1.6) is the number of possible rules that could be induced, called the hypothesis space. However, existing neuro-symbolic research concerning rule learning has focused on datasets with very short rules such as odd or even, smokers and friends, addition of digits, family trees and graph colouring [52, 186, 153]. Even with knowledge base completion tasks where there might be thousands of predicate symbols and constants, the length of the rules used in neuro-symbolic architectures is limited to less than three positive body atoms [170, 138, 136, 137]. What if a task demanded longer rules, included negation, and offered no way of exploiting contextual information to reduce the size of the relevant search space? Subgraph set isomorphism is a combinatorial NP-complete [39] decision problem that requires a model to determine whether any subgraph of a given graph is isomorphic to a set of other graphs. Formally, given a graph $\mathscr{G}$ and a set of graphs $\mathbb{H} = \{\mathscr{H}_1, \mathscr{H}_2, \dots, \mathscr{H}_n\}$, decide if the following target condition t holds:

$$t \leftrightarrow \exists \mathscr{L} \exists i \, (\mathscr{L} \subseteq \mathscr{G} \wedge \mathscr{L} \simeq \mathscr{H}_i) \tag{3.1}$$

where $\subseteq$ and $\simeq$ mean subgraph of and isomorphic to respectively. Given a dataset of graphs $\mathbb{X}$ with corresponding ground truth labels for t in Eq. (3.1) against a fixed hidden $\mathbb{H}$, the goal is to find the hidden set $\mathbb{H}$. This learning task can be rendered as an instance of Inductive Logic Programming where $\mathbb{H}$ is the hypothesis to learn with no background knowledge. The positive examples would be graphs for which t holds against an unknown but fixed $\mathbb{H}$ and negative examples would be those for which it does not. A successful learner would yield an $\mathbb{H}$ that satisfies all the data points in $\mathbb{X}$. Since we are interested in learning $\mathbb{H}$ as a set of symbolic rules rather than just subgraph isomorphism, recent advancements in deep graph neural networks [14, 118] and Transformers [216] are not suitable for extracting *what* $\mathbb{H}$ has been learnt, also known as the black-box problem [2].

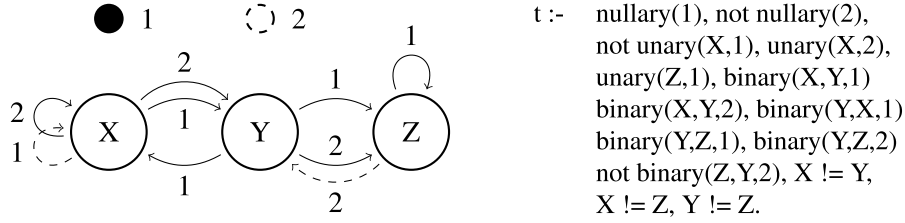


Figure 3.1: Correspondence between a graph and its representation in logic programming for the subgraph set isomorphism task. The uniqueness of variables is added to the rule since isomorphism is a bijection between graph nodes.

We encode the task in Answer Set Programming (Section 2.1.5) using the generic predicates nullary/1 for global graph properties, unary/2 and binary/3 for self-edges and directed edges between graph nodes respectively with the target t as the head of a rule. The last argument of each predicate symbol is a unique relation identifier. We use this particular signature since any logic program up to arity 2 can be converted to a program using these predicate symbols alone by assigning unique identifiers. Fig. 3.1 shows an example correspondence between a graph and its logic programming representation. Each node becomes a variable and each edge is a body literal. Thus, the decision problem in Eq. (3.1) can be concisely captured using the following steps:

1. The edges of the input graph $\mathscr{G}$ are provided as facts in the logic program using unique constants for each of its nodes.

2. Each candidate graph in $\mathbb{H}$ is represented as a rule similar to Fig. 3.1. Each node is assigned a unique variable name and uniqueness of variables is ensured since isomorphism is a bijection between graph nodes.

Querying the logic program constructed above as to whether t holds, is now equivalent to subgraph set isomorphism. The first part of the proof relies on the fact that the grounding of this logic program considers every possible constant to variable binding. Hence, each instantiation of the variables corresponds to a bijective mapping between the nodes of the given graph $\mathscr{G}$ (the constants of the program) with the nodes of a graph in $\mathbb{H}$ (the variables in a rule) by construction. As a result, when there are less or equal number of unique variables in a rule ($|V(\mathscr{H}_i)|$) than the number of constants present ($|V(\mathscr{G})|$), the grounding represents a subgraph. The second part involves the evaluation of the rules. The body of a rule with head t is only evaluated to true if and only if a corresponding grounding also holds. Since the predicate symbols represent the edges of the graphs, the overall evaluation can only be true if the graphs in question are isomorphic. And because t holds if *at least one* rule evaluates to true, there must exist at least one subgraph that is isomorphic capturing the existential quantifier in Eq. (3.1). As a result, if t holds then there must be a subgraph of $\mathscr{G}$ that is isomorphic to at least one graph in $\mathbb{H}$ and vice versa $\square$.

To generate the dataset, we uniformly sample unique graphs for $\mathbb{H}$ and then sample $\mathscr{G}$ of which subgraphs are isomorphic. The resulting data points represented as logic programs are checked using the answer set solver clingo [61]. In total we sample 10k graphs and take only the unique ones to ensure any partitioning of the data yields disjoint subsets. We then split into training, validation and test datasets with sizes 2k, 1k and 1k respectively.

Table 3.7: Different difficulty parameters for the subgraph set isomorphism dataset

Difficulty	$\lvert V(\mathscr{G})\rvert$	Nullary	Unary	Binary	$\lvert V(\mathscr{H}_i)\rvert$	Max. $\lvert\mathbb{H}\rvert$	Avg. $\lvert E(\mathscr{H}_i)\rvert$
Easy	3	2	2	2	2	3	7.29
Medium	4	4	5	6	3	4	37.27
Hard	4	6	7	8	3	5	50.62

Table 3.8: Example target rules generated for the medium dataset size. Note that the obj/1 predicate is added for ASP safe representation.

```
t :- nullary(0), nullary(2), not nullary(3), unary(X,0), unary(X,1), not unary(X,2),
     not unary(X,3), not unary(Y,0), not unary(Y,1), not unary(Y,2), not unary(Y,3),
     unary(Z,0), unary(Z,2), binary(X,Y,0), binary(X,Y,1), binary(X,Y,2),
     binary(X,Y,3), not binary(X,Y,4), not binary(X,Y,5), not binary(X,Z,0),
     binary(X,Z,2), not binary(X,Z,3), not binary(X,Z,5), binary(Y,X,1),
     not binary(Y,X,2), not binary(Y,X,3), not binary(Y,X,4), not binary(Y,X,5),
     binary(Y,Z,0), binary(Y,Z,3), not binary(Y,Z,4), binary(Z,X,0),
     not binary(Z,X,2), binary(Z,X,3), not binary(Z,X,4), not binary(Z,Y,1),
     not binary(Z,Y,3), not binary(Z,Y,4), not binary(Z,Y,5), obj(Z), Z != X,
     Z != Y, obj(X), X != Y, obj(Y).
t :- nullary(0), not nullary(1), not nullary(2), unary(X,0), unary(X,1), unary(X,3),
     unary(X,4), not unary(Y,0), not unary(Y,2), not unary(Y,4), not unary(Z,0),
     not unary(Z,1), unary(Z,2), not unary(Z,3), not unary(Z,4), not binary(X,Y,2),
     binary(X,Y,3), binary(X,Y,4), binary(X,Y,5), not binary(X,Z,1), binary(X,Z,2),
     binary(X,Z,3), not binary(X,Z,4), binary(X,Z,5), not binary(Y,X,2),
     binary(Y,X,3), not binary(Y,X,4), not binary(Y,X,5), not binary(Y,Z,4),
     not binary(Y,Z,5), not binary(Z,X,0), not binary(Z,X,1), not binary(Z,X,2),
     binary(Z,X,4), not binary(Z,Y,0), not binary(Z,Y,1), not binary(Z,Y,4),
     obj(Z), Z != X, Z != Y, obj(X), X != Y, obj(Y).
```

The subgraph set isomorphism task effectively creates an unbiased combinatorial search space for rule learning. Since each relation could be positively, negatively or be absent in the rule, the upper bound on the search space for possible rules is $3^{\lvert E(\mathscr{H}_i)\rvert}$. This ensures there is no bias in the rules or heuristics that can be used to prune the search space. As a result, any atom can appear in the rule equally likely. To adjust the difficulty of the dataset, as summarised in Table 3.7, we change the number of nodes $\lvert V(\mathscr{G})\rvert$, nullary, unary, binary relations, number of nodes in any $\mathscr{H}_i$ and the maximum size of $\lvert\mathbb{H}\rvert$ ensuring $\lvert V(\mathscr{H}_i)\rvert \leq \lvert V(\mathscr{G})\rvert$. We focus on these parameters to alter the number of edges to be learnt $\lvert E(\mathscr{H}_i)\rvert$ which in return corresponds to the length of the rules and increase the size of the possible hypothesis space.

An example set of rules $\mathbb{H}$ from the medium dataset size is shown in Table 3.8. State-of-the-art symbolic rule learners timeout on this size due to the long rules (empirical results are discussed in Chapter 6). Rules of this length are uncommon in many standard ILP datasets used in the neuro-symbolic literature [52]. In order for the rules to be safe in ASP, i.e. every variable appears in at least one positive atom, we also add the auxiliary obj/1 predicate for every variable which evaluates to true for every constant in the program. This is only done when evaluating with clingo [61] or using symbolic learners.

This concludes the symbolic dataset generation and in the remaining chapters, we present novel neuro-symbolic architectures that utilise these datasets along with others to learn aspects of logical inference using neural networks.

Chapter 4

Learning Symbolic Deduction with Neural Reasoning Networks

Existing neuro-symbolic architectures, particularly those on the more symbolic side of the spectrum in Fig. 1.2, directly implement and utilise proof procedures without actually learning them. A prime example is the Neural Theorem Prover [170] and its variants that implement Prolog's backward-chaining algorithm in order to learn predicate symbol embeddings for knowledge base completion tasks. Prolog is a hard-coded computer program written by human experts in order to parse and execute logic programs. Humans, on the other hand, can *learn* what a logic program is and its semantics. Even if someone has never seen first-order logic or a logic program before, they can learn how to parse them, manually apply rules to answer queries and even write new programs to solve previously unseen problems by reading Section 2.1. From the perspective of machine learning, it is unclear to what extent a neuro-symbolic model can learn to parse, represent and process queries without requiring any prior engineering or expert input. The driving motivation is to explore how neural networks may learn the steps of a proof system and potentially minimise the extent to which neural and symbolic paradigms have to be combined manually, e.g. as part of a larger network. Hence, in this chapter we provide a first insight into how neural networks learn to perform deductive inference over logic programs encoded at a character level.

There have been attempts to partially replace symbolic components of proof systems with neural networks such as Neural Prolog [46] which constructs networks through symbolic applications of logic rules. Another vital component, unification of variables has been tackled by Unification Neural Networks [107]. However, neither of these networks act as a complete inference engine that can be trained end-to-end solely using example logic programs and learn corresponding representations of logical constructs as well as reasoning.

In pursuit of a more holistic neural architecture capable of learning logical inference, we focus on neural networks that are (i) end-to-end differentiable so as to learn not only how to prove a given query but also the dense vector embeddings of logical constructs such as predicates and rules, and (ii) have a fixed architecture such that their connection patterns, size and computation graph do not change across different inputs in order to limit the amount of prior structural information

engineered into a network's design. These preferences rule out popular modern state-of-the-art deep neural networks such as tree based recurrent networks [211] and graph networks [14] since they construct a different network depending on the input; for example, the number of nodes and edges may vary according to the number of rules and facts in a program. This conversion also requires human knowledge, e.g. how a logic program can be converted into a graph, and breaks the end-to-end differentiability of the overall approach as only the final tree or graph like network can be trained excluding the input transformation. Similarly, Transformers [216] are built with a fixed number of encoder layers and cannot be iterated for an arbitrary number of time-steps which an inference algorithm may require.

We instead build on the prior work on iterative neural networks that learn algorithmic tasks such as addition of digits [70] and sequence processing [228]. A key component of deductive inference is multi-hop reasoning which requires an agent to process information over several steps to reach a conclusion. This paradigm of reasoning has been explored by neural networks in domains such as story comprehension with Dynamic Memory Networks [221], graph problems with Differentiable Neural Computers [71] and visual question answering with Memory Attention Control Networks [88]. Based on these works, we pose two questions: Can an iterative neural network learn to deduce queries solely from example logic programs, and if so how does it learn to represent logical constructs such as predicates and rules as real-valued dense vectors?

We present two novel contributions: (i) an iterative neural network architecture that outperforms existing state-of-the-art models on deductive inference tasks over logic programs and (ii) the analysis of how our network represents logic programs as well as its behaviour concerning multi-hop reasoning. The work in this chapter is based on "DeepLogic: Towards End-to-End Differentiable Logical Reasoning" [32] with the publicly available source code linked from the corresponding reference (see Section 1.5 for further details).

4.1 Neural Reasoning Networks

We call neural networks that learn the syntactic entailment function $\vdash$, Neural Reasoning Networks. Given a set of logic programs composed of a context P and a query atom q, the objective is to learn a function f such that: $f(P,q) = 1$ if $P \vdash q$, 0 otherwise and $f(P, \text{not } q) = 1 - f(P,q)$. We attempt to learn f because it neatly encapsulates many potentially complex steps of deductive reasoning in logic programs such as checking facts, applying rules, handling conjunctions and disjunctions. We take inspiration from Memory Networks [219], a neural network architecture that utilises a learnable memory matrix, and incorporate the end-to-end trainable iterative variants [209, 110] while following the steps of backward chaining summarised in Section 2.1.4 to propose a novel Neural Reasoning Network.

Design We can consider the context logic program a read-only memory and the proof state a writeable memory component. In a similar fashion to Prolog's backward chaining algorithm [37], we aim to have (i) a state to store information about the proof such as the query, (ii) a mechanism to select rules via attention and (iii) a component to update the state with respect to the selected rule. We introduce the Iterative Memory Attention (IMA) network that given a normal logic program

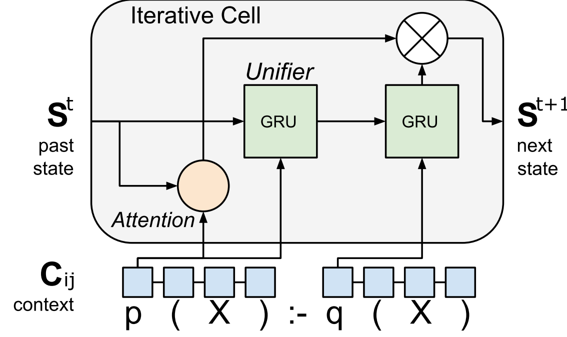


Figure 4.1: Overview of the Iterative Memory Attention (IMA) model. Using the rule's embedding, the unifier GRU produces the next state weighted by the attention value.

as context and a positive ground atom as query, embeds the literals in a high dimensional vector space, attends to rules using soft attention and updates the state using a recurrent network. IMA is a variant of Memory Networks that is architecturally biased to suit the logical reasoning process.

Literal Embedding The inputs to the network are two sequences of characters $c_1^P, c_2^P, \dots, c_n^P$ and $c_1^q, c_2^q, \dots, c_n^q$ for context and query respectively. The main motivation behind having character level inputs rather than symbol tokens is to constrain the network to learn sub-symbolic representations that could potentially extend to previously unseen literals. We pre-process the context program sequence into rules and literals to obtain an input tensor $P \in \mathbb{N}^{R \times L \times n}$ of characters encoded as positive integers where R is the number of rules, L the number of literals and n the length of the literals. Similarly, the query is a single ground positive atom encoded as a vector $\boldsymbol{q} \in \mathbb{N}^n$. This pre-processing allows the network to consider each literal independently when iterating over rules giving it finer control over the reasoning process. Each literal is embedded using a recurrent neural network that processes only the characters of that literal:

$$C_{i,j,:} = GRU(\text{onehot}(P_{i,j,:}), \mathbf{0}) \tag{4.1}$$

where $\text{onehot}(P_{i,j,:})$ is the one-hot encoding of the characters. We use a gated recurrent unit (GRU) (Section 2.2.4) starting with the zero vector $\mathbf{0}$ as the first hidden state. The characters are processed in reverse order to emphasise the predicate symbol and we take the final hidden state of the GRU to be the embedding of the literal. The context and query are embedded using the same network yielding a context program tensor $C \in \mathbb{R}^{R \times L \times d}$ where R is the number of rules, L the number of literals in a rule and d is the fixed embedding size; for the query we obtain a vector and use it as the first proof state of the network $\boldsymbol{s}^0 \in R^d$. By default, the embedding of a rule is the embedding of its head, $\boldsymbol{R}_{i,:} = C_{i,0,:}$ where $\boldsymbol{R} \in \mathbb{R}^{R \times d}$, reflecting what the rule can prove from a backward-chaining perspective. We also experiment with embedding rules using another GRU processing its literals and take the final hidden state as the representation of the rule, $\boldsymbol{R}_{i,:} = GRU(C_{i,:,:}, \mathbf{0})$. In this case, the rule embedding may contain information about its body conditions.

Iteration The iterative step consists of attending to the rules, computing a new state using each rule and updating the old state. The network is iterated for t^* time-steps, fixed in advance, starting with the initial state $\boldsymbol{s}^0$ which is the embedding of the query atom. At each time-step t, we curate

Table 4.1: Summary of recurrent components used in the Iterative Memory Attention (IMA) network. RtL and LtR stands for right to left and left to right respectively.

GRU	Direction	Comment
Literal Embedding	RtL	Emphasise the predicate symbol
Rule Embedding	RtL	Emphasise the body of the rule
Unifier	RtR or LtR	Either mimic backward-chaining or forward-chaining

a feature matrix $\boldsymbol{A}^t \in \mathbb{R}^{R\times 5d}$ and compute state to rule attention scores $\alpha^t \in \mathbb{R}^R$:

$$\boldsymbol{A}^t_{i,:} = \left[\boldsymbol{s}^t, \boldsymbol{s}^0, \boldsymbol{R}_{i,:}, (\boldsymbol{s}^t - \boldsymbol{R}_{i,:})^2, \boldsymbol{s}^t \odot \boldsymbol{R}_{i,:}\right] \tag{4.2}$$

$$\alpha^t_i = \sigma(\boldsymbol{W}^{1\times\frac{d}{2}} \tanh(\boldsymbol{W}^{\frac{d}{2}\times d}\boldsymbol{A}^t_{i,:} + \boldsymbol{b}^{\frac{d}{2}}) + b^1) \tag{4.3}$$

where $[,]$ is the concatenation operator and $\boldsymbol{W}, \boldsymbol{b}, b$ are independent learnable weights. To compute the attention vector α^t, we use a two layer feed-forward neural network in Eq. (4.3) where σ is the sigmoid function as defined in Eq. (2.3). We also experiment with the softmax formulation of the attention vector α^t by replacing sigmoid, which induces a categorical distribution over the rules. To *apply* a rule, we use another recurrent neural network that processes every literal of every rule starting with the current proof state $\boldsymbol{s}^t$ as the initial hidden state:

$$\boldsymbol{U}^t_{i,:} = GRU(C_i, \boldsymbol{s}^t) \tag{4.4}$$

$$\boldsymbol{s}^{t+1} = \sum_i^R \alpha^t_i \boldsymbol{U}^t_{i,:} \tag{4.5}$$

and produces new candidate proof states $\boldsymbol{U}^t \in \mathbb{R}^{R\times d}$. Then a weighted sum of the attention scores yields the next proof state and the model iterates. We call this inner GRU *unifier* as it needs to learn unification between variables and constants as well as how each rule interacts with the current state. For example, if $\boldsymbol{s}^t$ is some representation of $p(a)$ and the applied rule is $p(X) : -q(X)$, the unifier needs to produce a candidate state representation for $q(a)$, unifying and propagating the representation of symbols. A graphical overview is shown in Fig. 4.1 in which the proof state vector is updated using the rule $p(X) : -q(X)$. The direction of the GRUs may influence which part of their input sequence is emphasised. This is because of the diminishing gradients at the tail end of a sequence processed by a recurrent neural network as mentioned in Section 2.2.4. As a result, one can change the direction of the unifier GRU to potentially alter the behaviour of the proof system. For example, although to mimic backward-chaining it may be obvious to iterate the unifier left to right starting from the head of the rule, one can change the direction to right to left to elicit forward-chaining like behaviour, as summarised in Table 4.1. We revisit this configurable aspect of the unifier GRU and its effects in Section 4.3.

Table 4.2: Descriptive statistics of the predicate symbols for the fact matching task

Difficulty	Total	Unique	Max Length	Mean Length	Std
Training	70140	702	2	1.96	0.19
Easy	89883	29074	4	3.58	0.54
Medium	129997	78932	8	5.62	1.67
Hard	170077	121835	12	7.63	2.82

The use of a weighted sum to update the proof state $\boldsymbol{s}^{t+1}$ in Eq. (4.5) is a common feature of attention based models. An attention mechanism focuses to certain parts of the input often for the purpose of selecting a relevant portion [12, 127, 30, 110]. Since they are also differentiable, what the neural network attends to can be jointly learnt within a task. In this case, instead of saturating the proof state with all possible rule updates, the network may learn to select a rule. As with any architectural bias, there is no guarantee that the rule selection learnt by the model would follow an expected pattern. In order for the network to propagate the state unchanged, we append a *null sentinel* $\phi = \mathbf{0}$ to the context tensor C which allows the network to ignore the current reasoning step by carrying over the state if it selects the null sentinel. We also append a *blank rule* `()` that acts as a learnable parameter and is often attended to when no other rule is suitable.

The final prediction of the model is then based on the final proof state $\boldsymbol{s}^{t^*}$:

$$\hat{y} = \sigma(\boldsymbol{W}\boldsymbol{s}^{t^*} + b) \tag{4.6}$$

where $\boldsymbol{W}$ and b are learnable parameters. This final linear layer projects the proof state into a successful or unsuccessful scalar in the range $[0,1]$. When trained in an end-to-end fashion, the model receives only whether this prediction should be 1 or 0 with no other information. All the other components such as literal embeddings from Eq. (4.1), attention scores from Eq. (4.3) and proof state updates from Eq. (4.5) that ultimately lead to $\boldsymbol{s}^{t^*}$ must *all* be learnt in a joint fashion.

4.2 Datasets

To train neural reasoning networks in an end-to-end fashion, we use the stratified logic programs dataset from Section 3.1. The dataset consists of 12 classes of normal logic programs capturing various aspects of deductive reasoning. The logic programs are generated by sampling a random generative process. As a concrete instantiation of the dataset, we first generate a training set of 20k logic programs per task (class of program). For constants and predicates we use the lower case English alphabet and for variables upper case. The length of the character sequences that make up predicates and constants are at most 2 characters and the arities of the atoms are selected randomly between 1 and 2. For every logic program, the generation process also appends irrelevant rules as *noise* that always have different predicates and random structure while still preserving the desired semantics of the task. Each logic program is parsed into rules and literals to obtain the input tensor P described in the previous section.

To evaluate the models, we generate 4 test sets of increasing difficulty: validation, easy, medium

Table 4.3: Samples for different generated difficulties of the first fact matching task from Section 3.1. All remaining 11 classes of logic programs are generated in a similar fashion.

Training	Easy	Medium	Hard
ap(vq,u).	gmsza(gtp).	vxk(aglcurma).	pwuwknime(bklalabjr).
ap(vq,x).	xmsza(l).	qsmg(xljfk).	xlmhivunhw(lfke,pirwjhvl).
oz(vq,u).	ymez(gtp).	vcn(gc,ecdmgiv).	exqcbepjhjx(giowqnul,es).
tn(g,lw).	pj(tk).	bryvl(zwzhaqu).	uaydje(vswjsd,yojmjkxiwul).
a(tx,l).	szpt(ij,umcv).	ydofigq(lxiilprr).	u(njc,lafpdm).
sn(o,gb).	sbf(ni).	j(urupcnh,dey).	jpegyifve(lrxf,wdaqxgbpzheh).
	sbvmv(nx,e).	glkkyizi(f,qltwxjrz).	wlkemav(edvitzq,xhkxblecjstw).
	sksgb(fceq,h).	rbzs(co).	jfrlthn(xscsu,swkoeqjs).
		vkmx(nzdsnsg,mhpks).	ukktm(mxp,jtzyjjvldle).
		i(lispqtaq,xlo).	fjbkeandihp(nb).
			yg(arkthcdwoc,v).
			gzrztdniska(sxymmwhuplkb,lxe).
			ahmcbh(ofryxbxc).
? ap(vq,u). 1	? msza(gtp). 1	? vxk(aglcurma). 1	? pwuwknime(bklalabjr). 1
? jm(g,lw). 0	? vv(tk). 0	? mouy(xljfk). 0	? womjgn(tourwsvjew,ihiro). 0

and hard which have up to 2, 4, 8 and 12 characters for predicates and constants as well as added number of irrelevant rules respectively. While the validation set mirrors the configuration used in the training set, the other test sets offer previously unseen predicate and constant symbols. We focus on these configurable parameters to assess a model's ability to systematically generalise to longer symbols. Each test set consists of 10k generated logic programs per task and the statistics for the predicate symbols in the fact matching task are shown in Table 4.2. The length and the number of unique predicate symbols increase as more characters are used to construct them. The constant symbols have similar statistics as they are generated using the same process.

Table 4.3 shows sample logic programs for the fact matching task from different difficulties. Since an exact match is required for a successful entailment, a model is expected to represent longer sequences correctly whilst performing the required deductive reasoning of the corresponding task. This may become more and more difficult in later tasks such as multi-step deduction where the constants need to be carried over multiple iterations.

4.3 Experiments

We experiment with two combinations of variations of the Iterative Attention Memory model: sigmoid or softmax based attention in Eq. (4.3) and just literal or literal with rule embeddings in Eq. (4.1). As a baseline, we use a LSTM (Section 2.2.4) to process the query and then the context, to predict the answer. We compare our model against the Dynamic Memory Network (DMN) [110] and the Memory Attention Control (MAC) [88] network which both incorporate iterative components achieving competitive results in visual question answering datasets. With DMN and MAC, the context is separated into rules and the entire rule is embedded using a GRU at the character level. Unlike DMN which uses another GRU to accumulate information over the rule embeddings and the current state, our model IMA processes literal embeddings using a GRU to compute the new state as a weighted sum over each rule following Eq. (4.5).

Table 4.4: Accuracy against 10k test programs per task of the best single training run out of 3 with the mean accuracy for each difficulty. The embedding dimension is $d = 64$ and sm = softmax.

Training		Multi-task					Curriculum				
Model	LSTM	MAC	DMN	IMA			MAC	DMN	IMA		
Embedding	-	rule	rule	literal		lit+rule	rule	rule	literal		lit+rule
Attention	-	sm	σ	σ	sm	sm	sm	σ	σ	sm	sm
Facts	0.61	0.84	1.00	1.00	1.00	0.98	0.89	1.00	1.00	0.99	0.94
Unification	0.53	0.86	0.87	0.90	0.87	0.85	0.83	0.85	0.88	0.88	0.86
1 Step	0.57	0.90	0.74	0.98	0.94	0.95	0.77	0.62	0.96	0.93	0.92
2 Steps	0.56	0.81	0.67	0.95	0.95	0.94	0.70	0.58	0.95	0.91	0.89
3 Steps	0.57	0.78	0.77	0.94	0.94	0.94	0.64	0.64	0.93	0.86	0.87
AND	0.65	0.84	0.80	0.95	0.94	0.85	0.81	0.70	0.80	0.78	0.83
OR	0.62	0.85	0.87	0.97	0.96	0.93	0.75	0.75	0.96	0.93	0.90
Transitivity	0.50	0.50	0.50	0.50	0.52	0.52	0.50	0.50	0.50	0.50	0.50
1 Step NAF	0.58	0.92	0.79	0.98	0.94	0.95	0.65	0.58	0.96	0.91	0.92
2 Steps NAF	0.57	0.83	0.85	0.96	0.93	0.95	0.57	0.73	0.95	0.90	0.90
AND NAF	0.55	0.82	0.84	0.92	0.93	0.85	0.61	0.61	0.71	0.77	0.83
OR NAF	0.53	0.74	0.75	0.86	0.86	0.86	0.59	0.63	0.86	0.83	0.84
Easy Mean	0.57	0.81	0.79	**0.91**	0.90	0.88	0.69	0.68	0.87	0.85	0.85
Medium Mean	0.52	0.70	0.70	**0.86**	0.81	0.79	0.60	0.61	0.81	0.76	0.74
Hard Mean	0.51	0.63	0.66	**0.83**	0.75	0.72	0.55	0.58	0.76	0.70	0.68

We use two training regimes: curriculum learning [16] and multi-task. With curriculum learning the model is trained in an incremental fashion starting with tasks 1 and 2 with only $t^* = 1$ iteration. Then tasks accumulate with increasing number of iterations: tasks 3, 7, 9, 12 are added with the model running for $t^* = 2$ iterations and then tasks 4, 6, 8, 11 using $t^* = 3$ iterations. We determine the minimum number of iterations required for a task based on the number of sub-goal derivations Prolog performs. Finally all tasks are introduced with a maximum of $t^* = 4$ iterations. The multi-task approach trains on the entire dataset with 4 iterations fixed in advance. Models are trained via back-propagation using Adam (Section 2.2.2) for 120 epochs (set from 10 epochs per task) with binary cross-entropy loss (Eq. (2.12)) and a mini-batch size of 32. We ensure a mini-batch contains at least 1 sample from each training task to avoid any biases towards a particular task. Logic programs are shuffled after each epoch and rules within a context are also shuffled with every mini-batch to mitigate any memorisation. Since we have access to the data generating distribution, we do not use any regularisation in any of the models, and increase the training dataset size accordingly to avoid over-fitting [67].

The results for the best single training run out of 3 for each model with embedding size $d = 64$ on the easy set are shown in Table 4.4. We observe that the mean performance (bottom 3 rows) of all iterative models are better than the LSTM baseline except for the Transitivity task which all models fail at. We speculate the reason is that the models have not seen existential variable binding for more than 2 character constants and fail to generalise in this particular case. We note that the mean accuracies of the curriculum training regime is lower than that of multi-task for all the models, most likely because models with prior training have no advantage when new tasks with an extra iteration is introduced. For example, solving the OR task in 2 iterations does not improve a solution for AND in 3 iterations. Embedding literals seems to provide an advantage over embedding rules since all IMA models outperform both DMN and MAC when we consider the mean accuracy over every test set. We postulate literal embeddings give a finer view and allow

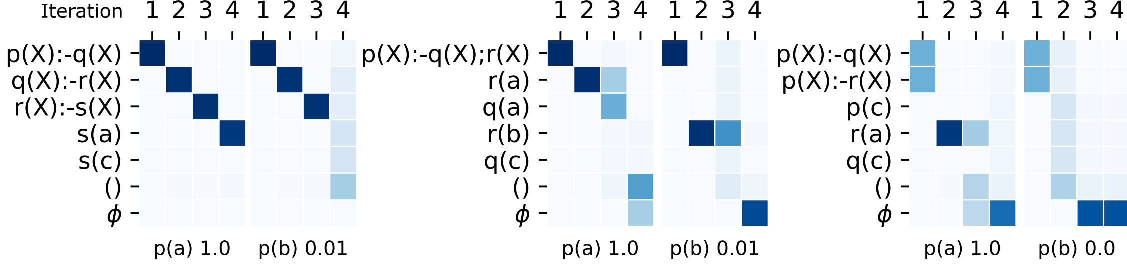


Figure 4.2: Attention maps produced for queries $p(a)$ and $p(b)$ for IMA with softmax attention. Darker shades of blue indicate higher attention scores and each column sums up to 1.

for better generalisation over increasing lengths since embedding rules with literals (lit+rule) also degrades the performance. Although our variant IMA performs the best on all the test sets, all models quickly degrade in mean accuracy as the difficulty increases. We speculate that this is because a fixed sized proof state vector can only store limited amount of information about unseen unique sequences of increasing lengths and analyse this behaviour further in the next section.

To get a visual representation of the behaviour of the model, we plot the attention scores α from Eq. (4.3) with softmax in Fig. 4.2. The heat maps show the rules that the model attends to over 4 iterations starting from queries $p(a)$ or $p(b)$ as well as the final prediction for tasks 5 to 7. In all cases, the model correctly predicts the answer and follows an approximation of backward-chaining. For the 3-step reasoning task (left most program), the model attends to the rules in the chain including the final fact if the query matches. If the final fact is missing, it seems to utilise the blank rule `()`. However, it is difficult to ascertain the role of the blank rule since the attention scores are smeared across multiple rules in the last iteration. The smearing of attention scores is also observed for the conjunction and disjunction tasks (middle and right most programs). Unlike a symbolic solver that utilises a queue of sub-goals, the model attends to multiple rules at the same time. We suspect this behaviour arises from a lack of *backtracking*, i.e. when a solver such as Prolog revisits a previous proof state to explore other proof paths. For example, in the disjunctive case (right most program), the model could have used 2 iterations to explore the first matching rule $p(X) : -q(X)$ and possibly backtrack to explore $p(X) : -r(X)$; however, it instead attempts to follow multiple paths simultaneously. An advantage of superimposing representations is that a neuro-symbolic model such as IMA may prove a query in fewer iterations than Prolog. However, with more matching rules, larger programs and more iterations, we speculate a degradation in performance and analyse it further in the next section. For example, going from a 2-way disjunction as shown in Fig. 4.2 to a n-way disjunction where n is large, the attention scores may spread across even further. An explicit supervision of the attention mechanism at each iteration, similar to the original DMN [110] work, or a hierarchical architecture might encourage a form of iterative backtracking resembling that of Prolog.

Although models by default converge to backward chaining, by (i) reversing the direction of the unifier GRU (see Table 4.1), and (ii) skipping task 2, we can create a bias towards ground facts that have a matching constant. This approach encourages our model IMA with rule embeddings (lit+rule) to converge to a *forward* chaining approach as shown in Fig. 4.3 along side the backward-

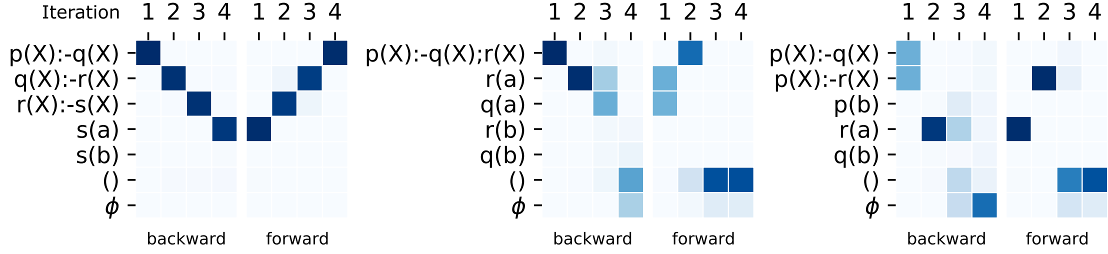
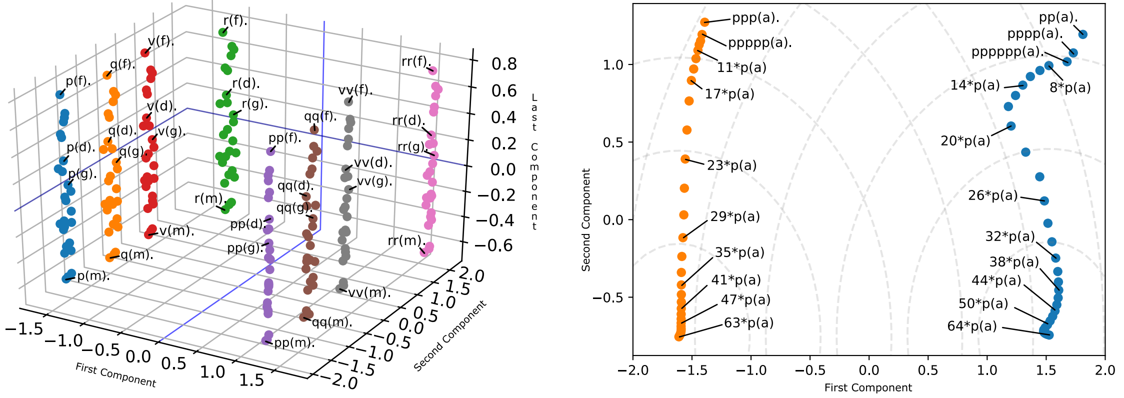


Figure 4.3: Attention maps produced for query $p(a)$ for IMA with softmax attention performing *backward* chaining in the left column and IMA with literal + rule embedding performing *forward* chaining in the right column on tasks 5 to 7.



(a) First, second and last principal components of embeddings of single character literals form a lattice like structure with predicates clustered in vertical columns and constants on horizontal surfaces, from IMA with softmax attention.

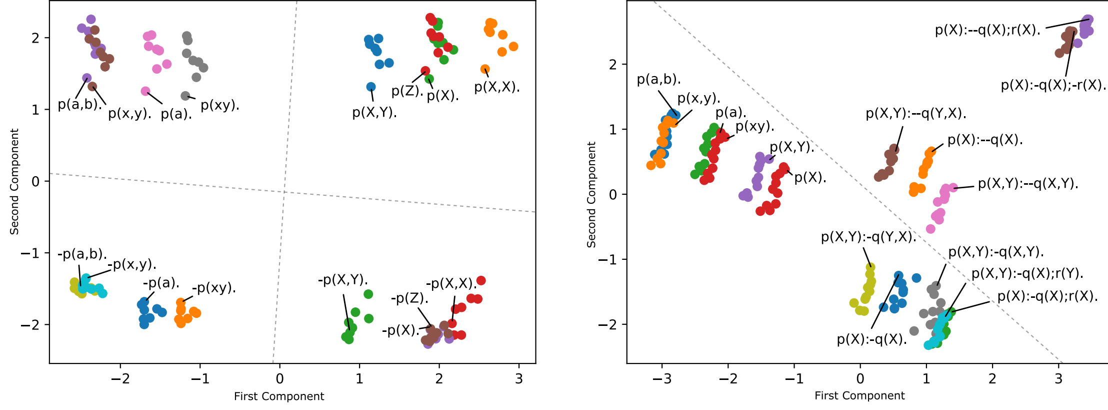
(b) Predicate symbols with increasing number of repeating characters saturate the embedding vector and converge to their respective points based on parity. Equidistant lines are plotted in grey, from IMA with softmax attention.

Figure 4.4: Principal component analysis of positive literal embeddings from Eq. (4.1).

chaining attention heat maps. We skip task 2 since it biases the network to match queries with the head of the rules, i.e. matching $p(a)$ with $p(X)$ in task 2 leads to a bias towards matching the head of a rule $p(X) : -q(X)$ as opposed to a body condition as in $q(X) : -p(X)$. Eliciting this behaviour requires more training time and therefore the results for forward chaining are not included in Table 4.4. This dual behaviour emphasises the fact that despite having a fixed network architecture, our model is flexible enough to learn two different solutions given the right incentives.

4.4 Analysis

Since models are trained in an end-to-end fashion where only the final label is provided, the representations for the logical constructs such as literals and rules must also be learned. In this section, we scrutinise these learnt representations from the embedding GRU in Eq. (4.1) for IMA with softmax attention and the rule embeddings of DMN. Each literal and rule in this case is a dense vector of real numbers with a fixed size of $d = 64$. Since we cannot directly visualise 64 dimensions, we utilise Principal Component Analysis (PCA) [67] to project the vectors down to 2 or 3 dimensions and plot the resulting low-dimensional approximation.



(a) Literals with different constructions first cluster by whether they are negated or grounded then by arity (grey lines are added as visual aids). Obtained from IMA with softmax attention.

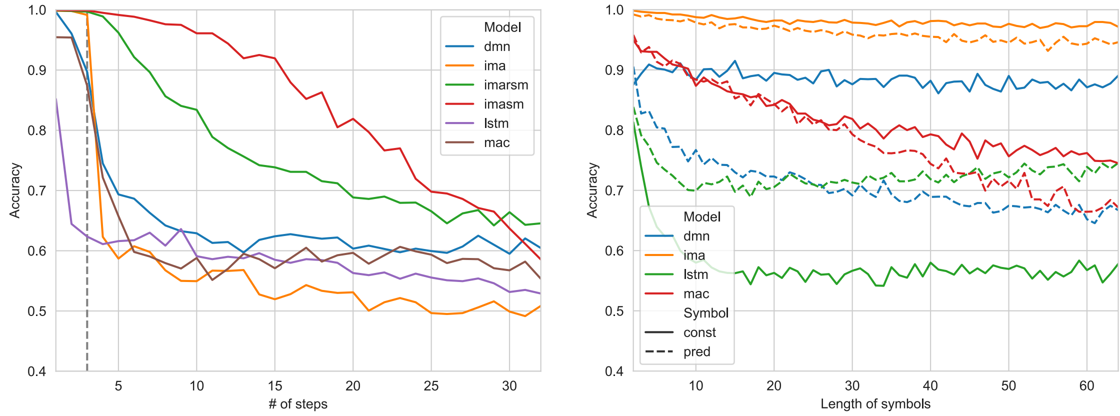
(b) Rule embeddings form clusters based on their structure with a distinction between negated and non-negated rules (grey line are added as a visual aid). Obtained from DMN.

Figure 4.5: Principal component analysis (PCA) of learnt literal and rule representations.

An important feature of logic programs in the dataset is that every predicate and constant combination is unique. Unification of ground atoms only succeed if they are an exact character by character match. We expect this property to create an embedding space that would allow every atom to be differentiated. To assess whether this is true in practice, we plot the positive literals for the range of the English alphabet in Fig. 4.4a reduced to 3 dimensions using PCA. We take the first, second and last principal components to visualise the embedding features that vary the most against the least. The embeddings of single and double character atoms form a lattice like structure: the first two components select the predicate symbol and the final the constant to uniquely identify the literal with a clear distinction between single and double character predicates. We skip certain letters in the alphabet to reduce cluttering in the visualisation and the missing columns in between letters portray a similar structure. This arrangement is in contrast with distributional representations that exploit similarities between entities such as in natural language word embeddings [134] and their corresponding usage in neuro-symbolic architectures [170, 136]. In this low-dimensional projection, `p(a)` might be more *similar* to `p(b)` than to `pp(a)` although all are deemed unique. The predicate symbol taking precedence might be encouraged by the right to left direction of the literal embedding GRU as mentioned in Table 4.1.

As the size of the embeddings is fixed ($d = 64$ in this case), we can also expect a saturation in the embedding space as the length of the literals increases, i.e. a finite representational capacity. We visually capture this phenomenon by repeating the character p 64 times and observe a converging pattern in Fig. 4.4b. As opposed to a single point of convergence, odd and even length predicates saturate to their own respective points suggesting that the embeddings produced by the GRU has learnt parity. Based on this low dimensional approximation, we postulate that the model would be able to differentiate between `63*p(a)` and `64*p(a)` despite having never seen such long predicate symbols and the saturated embedding space.

If we take literals with different constructions, we observe a clustering based on whether they are negated or grounded and then their arity as shown in Fig. 4.5a. We believe this clustering captures



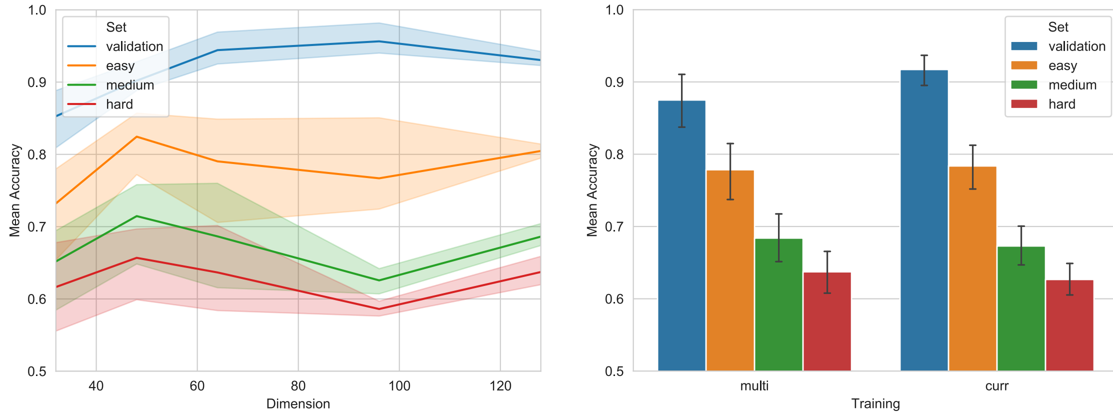
(a) When models are iterated beyond the training number of steps (grey line) to perform increasing steps of deduction, the accuracy degrades.

(b) The models can cope, in particular IMA with literal embeddings, when predicate and constant symbols of increasing length are randomly generated.

Figure 4.6: Accuracy over increasing number of iterations and symbol lengths.

the semantics of literals available in the dataset within the 4 major clusters highlighted using the dashed grey lines. Visually, `p(x,y)` is closer to `p(a,b)` than `p(xy)` reflecting the syntactic differences between predicates of arity 1 and 2. Furthermore, when we plot the rule embeddings learnt by DMN in Fig. 4.5b, we notice a similar clustering based on the rule structure with facts, negation, arity and number of body literals as learnt distinguishable features. The multiple points within a cluster correspond to different predicate symbols (letters of the alphabet) with the same construction. For both plots in Fig. 4.5, the positions of predicate symbols in the clusters seem to be preserved. For example, rules with the predicate symbol `p` in the head are often the upper most point of the clusters in Fig. 4.5b since the model would need to differentiate between similarly structured rules with different predicate symbols.

To evaluate multi-hop reasoning, we generate logic programs that require increasing number of steps of deduction using the n-step task described in Section 3.1. While the training data contains deductive chains of length up to $n = 3$, we generate up to $n = 32$ steps and the models are iterated for $t = n + 1$ iterations. When we plot the accuracy against the number of deductive steps in Fig. 4.6a, we observe a steady decrease in performance beyond the training length highlighted with a grey dashed line. The models `imasm` and `imarsm` refer to IMA with softmax attention and rule embedding respectively. We speculate that with each iteration the representation of the proof state accumulates an error that eventually leads to the loss of the necessary information akin to some form of additive noise. If we consider the unifier GRU from Eq. (4.4) to produce a new candidate state with some noise such that the next state becomes $\boldsymbol{s}^{t+1} = \boldsymbol{s}^{*t+1} + \varepsilon$ where $\boldsymbol{s}^*$ is the ground truth state and ε is some Gaussian noise, then the models may accumulate error with each iteration $\boldsymbol{s}^{t+1} = ((\ldots + \varepsilon) + \varepsilon) + \varepsilon$ to the extent that the desired label can no longer be correctly predicted. This gradual drop in performance parallels other works using recurrent neural networks to solve algorithmic tasks such as learning to reverse a sequence of numbers then testing on longer sequences [228]. Despite a drop in performance for all the networks, our IMA model with softmax attention (red line) maintains the highest accuracy most likely due to the sparsity of rule selection encouraged by the softmax operation (refer back to Eq. (2.6)).



(a) Mean accuracy over all tasks against increasing embedding dimension d shows no clear increase performance beyond $d = 64$.

(b) Mean accuracy of training regimes applied to IMAsm against test sets across all dimensions show no advantage of curriculum learning.

Figure 4.7: Mean accuracy with respect to embedding dimension and training regime.

To evaluate generalisation to unseen symbols, we take the 1 step deductive reasoning task and generate random predicate and constant symbols of increasing character lengths up to 64, well beyond the training dataset length of 2. Although in Fig. 4.4b we observe that literal embeddings may saturate, our IMA model maintains > 0.9 accuracy against longer randomly generated predicate or constant symbols as shown in Fig. 4.6b because, we speculate that, looking at only a few characters can determine uniqueness. This phenomenon reflects our intuition that looking at portions of sequences might be enough to determine equality rather than memorising them entirely.

In order to understand how the embedding and state dimension d affects the model performance, we experiment with sizes 32, 48, 64, 96 and 128 repeating each curriculum training session 3 times for our IMA model. Fig. 4.7a shows that increasing the dimension size does not contribute to a clear increase in mean accuracy over all the tasks and the drop in accuracy across easy, medium and hard test sets follow a similar pattern for every dimension. Despite the initial increase beyond $d = 32$, we get the highest upper bound in accuracy with $d = 64$, for which the individual results are shown in Table 4.4. Beyond a performance boost, another important consequence of larger hidden state and embedding size d is more learnable weights in every equation from Section 4.1 which may require more computational resources to train and evaluate.

The tasks are designed to capture increasing difficulty, building on previously seen problems such as unification before multi-step deduction. As a result, it is natural to expect models trained using a curriculum scheme to generalise better as the network iterations would increase gradually as described in Section 4.3. However, when we plot the mean accuracy for all dimensions, for all 3 runs of IMA with softmax attention across the test sets, we do not discern any advantage for curriculum training over multi-task training beyond the validation set. Fig. 4.7b shows a similar decrease in performance across the test sets for both training regimes. We believe this result stems from introducing new tasks with each iteration and models not having any incentive to abstract subtasks until the next set of unseen tasks are incorporated into the training dataset. For example, models which solve tasks 1, 2 and 3 (1-step deduction) have no advantage on solving task 4 (2-step deduction) because the final state representation from task 3 specifically looks for a ground

instance such as `p(a)` to complete the task. However, in task 4 the model needs to match another rule `p(X):-` and iterate once more. This lack of generalisation observed in curriculum learning originates from the data-driven learning of neural networks as described in Section 2.2.2: if the training examples do not require to match another rule at the final iteration, then the models have no incentive to do so. As a result, we do not compare neural reasoning networks on existing rigorous logic programming benchmarks designed to test symbolic solvers.

4.5 Related Work

Learning dense vector representations of symbols has been popularised by natural language word embeddings [134, 157] because of their ability to capture similarities in an effective, learnable manner. Applying this ability to learn similarities between symbols using vectors, Neural Theorem Provers [170] and its variants [138, 136] employ the backward-chaining algorithm over a knowledge base encoded as a logic program in order to determine which symbols should be similar using the radial basis function. The vector representations of predicates such as "grandfather" and "grandpa" become closer during training if used in similar contexts within a knowledge base completion task. In contrast, our deep recurrent neural network approach not only learns dense vector representations of logical constructs as described in Section 4.4 but also learns to approximate the backward-chaining algorithm solely from example logic programs.

The construction of a computation graph that serves as the backbone of a neural network in order to perform logical inference is shared by many related neuro-symbolic approaches. Lifted Relational Neural Networks [202] construct a set of weighted definite clauses and ground them turning the logic programs in question into propositional formulae. We, on the other hand, refrain from grounding programs and attempt to learn unification from training examples. TensorLog [38] produces *factor graphs* from logic programming rules in order to build the neural network with the appropriate connections. The resulting network then performs inference over one-hot encoded constant symbols. The literal embedding learnt from characters in Eq. (4.1) overcomes the need for one-hot encoding every constant and allows the network to generalise to previously unseen predicate and constant symbols as shown in Fig. 4.6b. Logic Tensor Networks [186] only tackle the knowledge base completion task "on a simple but representative example" also grounding every rule in the program prior to performing any reasoning. All the related works mentioned so far build neural networks by parsing logic programs and this yields different connection patterns to represent the rules. On the other, our model IMA described in Section 4.1 has a fixed architecture whilst being able to perform both backward and forward-chaining as demonstrated in Fig. 4.3.

Learning similarities between constants can be used to perform link-prediction tasks [148] and knowledge base completion [200] but creates unwanted inferences when similar constants should indeed be unique [171]. Although the dataset we use (see Section 4.2) requires the uniqueness of each symbol, we expect embeddings of similar constants to cluster during training if the data entails the same conclusions. For example, if $p(a) \vdash p(b)$ then our model has the capacity to adjust the representations of those atoms so as to achieve the desired entailment. Switching from a syntactic entailment $\vdash$ to that of a semantic one $\models$ (refer back to Definitions 2 and 3), Possible World

Net [53] provides a different approach to learning valid propositional arguments using randomly generated worlds, also known as situations described in Section 2.1.1. However, they construct the neural network in a tree like manner by parsing the propositional formulae in which nodes correspond to Boolean connectives. Although all related, the works mentioned so far are designed for either deductive databases, relation learning, link prediction, knowledge base completion or propositional programs; thus our task of learning deductive reasoning and embeddings of normal logic programs using a fixed RNN-based iterative network is inherently different making a direct empirical comparison unreasonable. Consequently, we focus on iterative memory based neural networks such as MAC for empirical comparison in Section 4.3.

Neural Reasoning Networks can be seen as interpreters of logic programs as rules can act like instructions. This perspective is related to systems such as Neural Program-Interpreters [164] and Neural Symbolic Machines [119]. These approaches contain discrete operations that allow more complex actions to be performed overcoming the problem of a degrading state representation as alluded to by Fig. 4.6a. However, they require a reinforcement learning setting to train. We believe a reinforcement learning approach applied on our dataset using the set of actions governing SLDNF described in Section 2.1.4 such as derive a new sub-goal, substitute a variable and check symbol equality, would learn a similar algorithm to that of Prolog [37] since the example logic programs are generated using a similar process, see Section 3.1 for further details. For a more end-to-end trainable approach, the differentiable Forth ($\partial 4$) abstract machine is a differentiable implementation of the stack-based Forth programming language [21]. It is trained on program *sketches* that consist of state transitions with learnable neural network based transitions inside a larger program together with desired input and output states. Unlike our approach, $\partial 4$ benefits from pre-implemented differentiable transitions such as how to duplicate a stack state as opposed to learning what each symbol may mean solely from execution examples.

4.6 Discussion

The logic programs dataset from Section 4.2 is basic from the perspective of existing symbolic proof systems such as Prolog (Section 2.1.4) and clingo (Section 2.1.5) since they can solve *all* the tasks without any errors and in every test condition. On the other hand, we struggle to observe a comparable performance from neural networks. Their fragility against increasing symbol lengths, number of deduction steps and program complexity discussed in Section 4.4 renders an unfavourable outcome for the use of continuous representations for rigid symbolic reasoning. To answer the first question in the introductory section of this chapter, it seems possible to teach a neural network to deduce queries solely from examples in an end-to-end fashion albeit with limitations. This result highlights the difficulty in learning both the proof system and the representations of logical symbols in tandem.

The analysis of the predicate and rule embeddings in Section 4.4 suggests a consistent clustering based on the syntactic properties of the logical constructs. Hence, the neural networks learn to represent logic programs within a dense vector embedding space based on their syntactic building blocks. The unique advantage of this end-to-end representation learning is that it can adapt to

different logic programming syntaxes and formalisms. For example, if the letter cases for constants and variables were swapped or even used inconsistently, neural reasoning networks have the capability to learn new representations as opposed to static parsing used by symbolic solvers. At the expense of interpretability, these representations can also pave the way for learning novel deductive inference algorithms such as using two rules simultaneously as shown in Fig. 4.3. Yet, this flexibility and learning comes at the cost of a large training dataset and computation time.

Referring back to the overarching theme of using neural architectural bias as the means of eliciting neuro-symbolic behaviour from Chapter 1, we highlight how all memory based networks outperform the simpler LSTM in Table 4.4. From this perspective, the better performance of our model IMA over MAC and DMN might be attributed to the fact that it more closely follows the steps of a goal-oriented proof system: start with a goal state, select rules, derive sub-goals and iterate. This observation further substantiates the idea that the closer the architecture of a network is to the domain, the better suited it is for learning. Similar to how convolutional networks (Section 2.2.3) are suited for visual processing and recurrent networks (Section 2.2.4) for temporal data, the architectural bias of neural reasoning networks makes them more viable for deductive inference over logic programs encoded as sequences of characters.

Creating differentiable relaxations of deductive inference is often the first step of inducing rules using neural networks since it opens up the possibility of gradient-based optimisation [52, 170, 38, 186]. However, if we take a fully trained IMA model as a proof system, it is neither complete nor sound (see Definitions 2 and 3) since not all test examples are successfully solved. This lack of robust deductive reasoning undermines the potential of using neural reasoning networks for inductive logic programming because the rule learning is fundamentally bounded by how the rules are used by the underlying proof system (Section 2.1.6). We explore how differentiable rule learning over dense representations may be achieved later in Chapter 6 using a novel neural network layer.

One could argue that the principal benefit of logic-based inference stems from the usage and creation of rules. In this chapter, the former is realised through various tasks of increasing difficulty involving different rule structures as shown in Fig. 4.5. What differentiates a rule from the rest of the program are *variables*. Instead of enumerating potentially an infinite number of factual relationships, rules concisely generalise to many objects through their use of variables. For example, instead of a relationship of the form $mortal(alice) \leftarrow human(alice)$, we consider the general rule $\forall X(mortal(X) \leftarrow human(X))$. In the next chapter, we take this observation and the core idea of using variables further to propose a novel deep learning architecture capable of recognising and using variable symbols outside the confines of logic programming.

Chapter 5

Learning Invariants through Soft Unification

Logical inference primarily revolves around rules [82]: deduction is concerned with the application of rules to derive further facts, abductive reasoning attempts to provide explanations by searching plausible premises of rules and inductive inference aims to learn rules. Consider the rule $\forall X(mortal(X) \leftarrow human(X))$, given $human(alice)$ we can *deduce* $mortal(alice)$ while given $mortal(alice)$ we can *abduce* $human(alice)$. From instances of humans and their mortality we can also *induce* the original rule. Contrary to propositional rules, first-order rules capture general statements about objects and their relations through the use of *variables*. Thus, the power of such rule-based inference seems to align with the employment and operation of variables that make those rules general across specific instances. Yet, outside of a manually engineered symbolic solver, particularly within an artificial neural network, it is not clear how variables can be incorporated to leverage the aforementioned rule-like reasoning. In this chapter, we explore whether machines can also learn and use variables solely from data.

The motivation for a neuro-symbolic model capable of learning rule-like structures can be traced back to the human brain. While humans have the ability to process symbolic knowledge and maintain symbolic thought [215], they do not require combinatorial enumeration of examples but instead utilise invariant patterns where specific entities are replaced with placeholders. Symbolic cognitive models [117] embrace this perspective with the human mind seen as an information processing system operating on formal symbols such as reading a stream of tokens in natural language. The language of thought hypothesis [144] frames human thought as a structural construct with varying sub-components such as “X went to Y”. By recognising what varies across examples, humans are capable of lifting examples into invariant principles that account for other instances. Furthermore, this symbolic thought with variables is learned at a young age through symbolic play [160]. For instance, a child learns that a sword can be substituted with a stick [56] and engages in pretend play.

While plausible biological neural models for the neuro-symbolic nature of the human brain have been proposed such as for handling triplets of symbols in question answering [90], there is no

$\boldsymbol{V}$:**bernhard** is a $\boldsymbol{V}$:**frog**
$\boldsymbol{V}$:**lily** is a $\boldsymbol{V}$:**frog**
$\boldsymbol{V}$:**lily** is $\boldsymbol{V}$:**green**

what colour is $\boldsymbol{V}$:**bernhard**

green

Figure 5.1: Invariant learned for bAbI task 16, basic induction, where $\boldsymbol{V}$:**bernhard** denotes a variable. This invariant accounts for all the training, validation and test examples of this task.

evidence that the *rules* in our brain adhere to the formal logic syntax or semantics as defined in Section 2.1. This case relaxes the constraints on what constitutes a potential rule beyond the common Horn clause form. A natural candidate for a rule-like template with varying symbols is the input itself avoiding any conversion into a logic program. Fig. 5.1 shows an example invariant learned by our approach: *if someone is the same thing as someone else then they have the same colour*. With this invariant, our approach solves *all* training and test examples in task 16 of the natural language reasoning dataset bAbI [220] (see Section 5.2). Thus, being able to recognise which symbols could act as variables and therefore take on different values is crucial for lifting examples into general principles that are invariant across multiple instances.

We address the question of whether a machine can learn and use the notion of a *variable*, i.e. a symbol that can take on different values as a proxy for learning invariant rule-like patterns. For instance, given an example of the form "bernhard is a frog" the machine would learn that the token "bernhard" could be *someone* else and the token "frog" could be *something* else. When the machine learns that a symbol is a variable, assigning a value can be reframed as attending to an appropriate symbol. Attention models [12, 172, 127, 30] allow neural networks to focus, attend to certain parts of the input often for the purpose of selecting a relevant portion. This perspective motivates our idea of an unification mechanism for learning variables across examples that utilises attention and is fully differentiable. Recall that unification in logic from Section 2.1.4 captures two key aspects: (i) whether two inputs match and if so (ii) what values the variables are assigned. As a result, we refer to this approach as *soft unification* that can jointly learn which symbols can act as variables and how to assign values to them.

Existing work such as Neural Theorem Provers [170, 136], TensorLog [38], Logic Tensor Networks [186] and Logical Neural Networks [169] do not learn what symbols can act as variables and utilise them during pre-processing of logic programs, e.g. when grounding or symbolically unrolling a proof whilst deep neural networks such as Relation Networks [179], Graph Neural Networks [14] and Transformers [216] do not yield extractable invariants with variables. Hence, we propose an end-to-end differentiable neural network approach for learning and utilising the notion of variables in order to lift examples into invariants, which can then be used by the network to solve given tasks. The main contribution of this chapter is a novel architecture capable of learning and using variables through the application of soft unification. We present the empirical results of our approach in a controlled environment using four synthetic datasets and a real-world dataset along with the analysis of the learned invariants that capture the underlying patterns present in the tasks. The work in this chapter is based on "Learning Invariants through Soft Unification" [33] and the link to the publicly available implementation using Chainer [213] is given in Section 1.5.

5.1 Unification Networks

When learning solely from data, reasoning with variables involves first of all identifying what variables are in a given context as well as defining the process by which they are assigned values. The intuition is that when the varying components, i.e. variables, of an example are identified, the example can be lifted into an invariant that captures its structure but with variables replacing its varying components. This is equivalent to identifying which symbols stay constant as the model would make a binary prediction whether a symbol varies or not. In this section, we first present our novel, architecture independent approach to learning and utilising variables in Algorithm 1 and then provide concrete implementations as part of various neural networks in Section 5.3. We refer to neural networks that employ soft unification as *Unification Networks*.

Step 1 (Pick Invariant Example)**.** We start from an example data point to generalise from. By generalise, we mean to recognise which symbols act as variables and in turn generate a rule-like template that may allow the prediction of other data points of the same structure. If we assume that the training examples are drawn from a data generating distribution p_{data} but with a *single* common pattern, we can consider each input an instance of that underlying pattern. Therefore, we randomly pick any example within a task from the dataset as our invariant example G which is a full data point consisting of a context, query and an answer as shown in Table 5.1.

Algorithm 1: Unification Networks

Input: Invariant I consisting of example G and variableness network ψ, example K, features network ϕ, unifying features network ϕ_U, downstream predictor network f
Output: Predicted label for example K

```
1  begin ▷ Unification Network
2      return f∘g(I,K)                        ▷ Predictions using Soft Unification g
3  begin ▷ Soft Unification function g
4      foreach symbol s in G do
5          A_{s,:} ← φ(s)                     ▷ Features of G, A ∈ ℝ^{|G|×d}
6          B_{s,:} ← φ_U(s)                   ▷ Unifying features of G, B ∈ ℝ^{|G|×d}
7      foreach symbol s in K do
8          C_{s,:} ← φ(s)                     ▷ Features of K, C ∈ ℝ^{|K|×d}
9          D_{s,:} ← φ_U(s)                   ▷ Unifying features of K, D ∈ ℝ^{|K|×d}
10     Let α = softmax(BD^T)                  ▷ Attention map over symbols, α ∈ ℝ^{|G|×|K|}
11     Let H = αC                             ▷ Attended representations of G, H ∈ ℝ^{|G|×d}
12     foreach symbol s in G do
13         U_{s,:} ← ψ(s)H_{s,:} + (1−ψ(s))A_{s,:}   ▷ Unified representation of I, U ∈ ℝ^{|G|×d}
14     return U
```

Step 2 (Lift Invariant Example)**.** In order for the invariant example to predict other examples correctly, certain symbols might need to *vary* such as ***V*:bernard** in Fig. 5.1. We capture this degree of *variableness* with a function $\psi : \mathbb{S} \to [0,1]$ for every symbol appearing in G where $\mathbb{S}$ is the set of all symbols. When ψ is a learnable function, the model learns to identify the variables and convert the data point into an invariant, i.e. learning invariants. For a certain threshold $\psi(s) \geq t$, we visualise them in bold with a $\boldsymbol{V}$ prefix. An invariant is thus a pair $I = (G, \psi)$, the example data point and the variableness of each symbol.

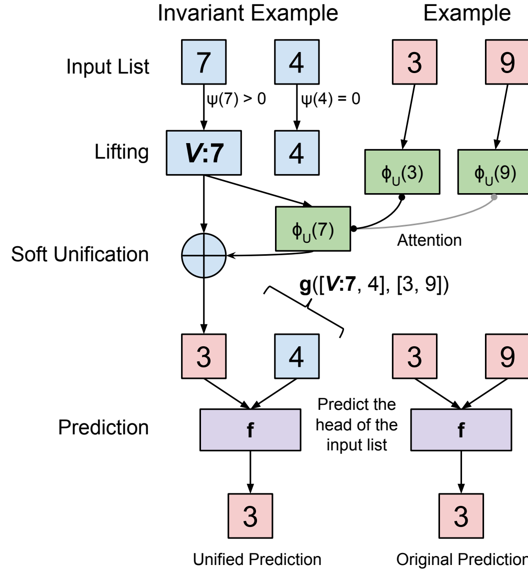


Figure 5.2: Graphical overview of predicting the head of a sequence using an invariant. The objective is to predict the head of the two digit sequence 3 9 using the invariant example ***V*:7** 4. The model has learnt that the head of the invariant sequence is a variable and the soft unification g between the two examples outputs the new sequence 3 4. The downstream predictor f correctly predicts the head of the sequence as 3 similar to how it would have for the original sequence.

Step 3 (Compute Unifying Features)**.** Suppose we are now given a new data point K that we would like to unify with our invariant I from the previous step. K might start with a question "what colour is lily" and our invariant "what colour is ***V*:bernard**". We would like to match bernard with lily. However, if we were to just use d-dimensional representations of symbols $\phi : \mathbb{S} \to \mathbb{R}^d$, the representations of bernard and lily would need to be similar which might confound a downstream predictor network, e.g. when lily and bernard appear in the same story it would be difficult to distinguish them with similar representations. To resolve this issue, we learn *unifying features* $\phi_U : \mathbb{S} \to \mathbb{R}^d$ that intuitively capture some common meaning of two otherwise distinct symbols. For example, $\phi(\text{bernhard})$ and $\phi(\text{lily})$ could represent specific people whereas $\phi_U(\text{bernhard}) \approx \phi_U(\text{lily})$ the notion of *someone*. Similarly, in Fig. 5.2 $\phi_U(7) \approx \phi_U(3)$ may capture the notion of the head of a sequence. We use the original symbol bernhard when computing the representation of ***V*:bernhard** to capture the variable's bound *meaning* following the idea of referents [55].

Step 4 (Soft Unification)**.** Using the learnt unifying features, for every variable in I we can now find a corresponding symbol in K. This process of unification, i.e. replacing variables with symbol values, is captured by the function g that given an invariant and another example it updates the variables with appropriate values. To achieve this, we compute a soft attention for each symbol s in the invariant using the unifying features $\phi_U(s)$ (Line 10 in Algorithm 1) and interpolate between its own and its variable value (Line 13 in Algorithm 1). Since g is differentiable and ψ, ϕ and ϕ_U are learnable, we refer to this step as *soft unification*. In Fig. 5.2, the variable ***V*:7** is changed towards the symbol 3, having learnt that the unifiable feature is the head of the sequence.

Step 5 (Predict)**.** So far we have constructed a unified data point of I with the new example K of which we would like to predict the answer. Since the unified input has the same structure as any other example with a shared feature space ϕ, we use another, potentially downstream task specific,

Table 5.1: Sample context, query and answer triples and their training sizes *per task*. For the distribution of generated number of examples per task on Sequence and Grid dat, refer to Section 3.2.

Dataset	Context	Query	Answer	Training Size
Sequence	8384	duplicate	8	$\leq$ 1k, $\leq$ 50
Grid	0 0 3 0 1 6 8 5 7	corner	7	$\leq$ 1k, $\leq$ 50
Sentiment A.	easily one of the best films	Sentiment	Positive	1k, 50
bAbI	Mary went to the kitchen. Sandra journeyed to the garden.	Where is Mary?	kitchen	1k
Logic	p(X) ← q(X). q(a).	p(a).	True	2k

network f that tries to predict the answer based on our unified input. In question answering, f could be a memory network, or when working with grid like inputs, a CNN. By predicting on the output of our unification, we expect that $f \circ g(I,K) \simeq f(K) \simeq y$ where y is the desired label. If f is differentiable, we can learn how to unify while solving the downstream task. We focus on g and use standard networks for f to understand which invariants are learned and the interaction of $f \circ g$ instead of the raw predictive performance of f.

The overall unification network recipe does not impose any constraints on the learnable components: variableness network ψ, features network ϕ, unification features network ϕ_U and the downstream predictor network f. This allows unification networks to be customised for the required task and leverage the existing beneficial architectural biases available for a particular domain such as using convolutional networks (Section 2.2.3) to process spatial inputs. Consequently, we present four concrete architectures to model $f \circ g$ and demonstrate the flexibility of our approach towards different input data structures in Sections 5.2 and 5.3.

5.2 Datasets

In order to demonstrate that our approach of recognising and utilising variables is structure agnostic, we use five datasets over different domains consisting of a context, a query and an answer (C,q,y). Samples of each dataset are shown in Table 5.1 with varying input structures: fixed or varying length sequences, grids and nested sequences (e.g. stories). While the overall goal in each dataset is to correctly predict the desired label y given the corresponding context and query, we also focus on *how* the unification networks predict for further analysis of the symbols that are recognised as variables and the invariant patterns they yield in Section 5.5. As a result, we mainly use synthetic datasets (4 out of the 5 datasets) of which the data generating distributions are known to evaluate not only the quantitative predictive performance but also the quality of the invariants learned by our approach. In contrast, the real-world dataset does not contain clear patterns and offers a noisy learning environment. Furthermore, to evaluate whether learning and utilising invariants through our novel soft unification mechanism provides better training data efficiency over models without soft unification, we use a full training set and a smaller data constrained training set as summarised in the last column of Table 5.1.

Fixed Length Sequences and **Grid** datasets from Section 3.2 provide a controlled environment to evaluate the emergence of a single symbol as a variable. In each task, the desired target symbol y consistently varies with the input pattern such as the head of the sequence in Fig. 5.2. Sequences of length $l = 4$ and grids of size 3×3 are sampled from a fixed set of 8 digits with unique data points partitioned into training and test datasets. This process ensures that a model trained on the training split will encounter previously unseen examples during evaluation. The final training sets contain either at most 1k or 50 unique data points to evaluate data efficiency. For further details, please refer back to Section 3.2.

bAbI The bAbI dataset consists of 20 synthetically generated natural language reasoning tasks including deduction, path finding and induction, refer to "Towards AI-Complete Question Answering: A Set of Prerequisite Toy Tasks" [220] for task details and examples. We take the 1k English set and use 0.1 of the training set as validation. The sentences are parsed and tokenised at a word level. Each token is then lower cased and considered a unique symbol. Following previous works [185, 209], we take multiple word answers such as 'milk,football' from the lists and sets task (Task 8), to also be a single unique symbol. To initially form the repository of invariants, we use the bag-of-words representation of the questions and find the most dissimilar ones based on their cosine similarity as a heuristic to obtain varied examples.

Logical Reasoning To distinguish our notion of a variable from that used in logic-based formalisms, we utilise the logic programs from Section 3.1 to learn character level patterns within logic programs using unification networks. The tasks involve learning whether a context logic program syntactically entails a ground query atom, $C \vdash q$, over 12 classes of logic programs. We generate 1k logic programs per task for training with 0.1 as validation and another 1k for testing. Each logic program has one positive and one negative prediction giving a total of 2k training data points. Similar to Chapter 4, we first parse the context logic programs into rules but then process them directly as a sequence of characters in which each character is considered a unique symbol. This renders the structure of this dataset similar to that of natural language stories; the logic program is akin to a story whereby each rule is like a sentence and each character token is a word. For example, each of the 4 characters of the atom $p(a)$ is a potential variable symbol from the perspective of unification networks. For further details on how the dataset is generated and the tasks involved, refer back to Section 3.1.

Sentiment Analysis To evaluate on a noisy real-world dataset, we take the movie review sentiment analysis task from "Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank" [201] and prune sentences to a maximum length of 20 words. We prune the sentences to keep the computational resources required low and the experiments more widely reproducible as alluded to in Section 1.2. Unlike the synthetic natural language stories of the bAbI dataset, movie reviews written by humans do not adhere to specific patterns and therefore provide a noisy training environment for unification networks to learn invariants from. We threshold the scores ≤ 0.1 and ≥ 0.9 for negative and positive labels respectively to ensure unification cannot yield a neutral score, i.e. the model is forced to learn either a positive or a negative label. We then take 1k or 50 training examples *per label* for the full and data constrained settings respectively and use the remaining ≈ 676 data points as unseen test examples.

Table 5.2: Comparison of different Unification Networks and learnable components

Model	Data Structure	Variableness ψ	Features ϕ	Unifying Features ϕ_U	Predictor f
UMLP	Fixed length	$\sigma(w_s)$	$\boldsymbol{E}_{s,:}$	bi-GRU $\circ\ \phi$	MLP
UCNN	Grid	$\sigma(w_s)$	$\boldsymbol{E}_{s,:}$	CNN $\circ\ \phi$	CNN
URNN	Varied length	$\sigma(\boldsymbol{W}\phi(s)+b)$	$\boldsymbol{E}_{s,:}^{\text{ConceptNet}}$	MLP $\circ\ \phi$	LSTM
UMN	Nested lists	$\sigma(w_s)$	$\boldsymbol{E}_{s,:}$	bi-GRU $\circ\ \phi$	MemNN

5.3 Instances of Unification Networks

Constructing neural networks with architectural biases that align with that of the problem domain often yields better performance [67]. In order to benefit from different network architectures and demonstrate the flexibility of our novel unification network approach from Section 5.1, we instantiate four unification networks to process the appropriate input structures: feed-forward (MLP) for fixed length sequences, convolutional neural network for spatial inputs, recurrent neural network for variable length natural language sentences and memory network for nested story like sequence of tokens. In each case, we choose suitable neural networks to model the 4 learnable components that are passed as inputs to Algorithm 1 and assign names based on the architecture of the predictor network as summarised in Table 5.2.

5.3.1 Unification MLP

The first input structure we consider is a sequence of symbols with a fixed length l, for example a sequence of digits 4234 such as phone numbers, postcodes and serial numbers of products. To model the variableness of symbols from Step 2, we assign a learnable weight w_s to each symbol $s \in \mathbb{S}$ and set $\psi(s) = \sigma(w_s)$ where $\boldsymbol{w} \in \mathbb{R}^{|\mathbb{S}|}$. This formulation checks if each symbol is a variable independently and does not consider the symbol's context. At this stage, we leave more advanced ways of implementing ψ as future work. The features of the symbols are obtained using the d-dimensional rows of a learnable embedding matrix $\boldsymbol{E}$ such that $\phi(s) = \boldsymbol{E}_{s,:}$ with $\boldsymbol{E} \in \mathbb{R}^{|\mathbb{S}|\times d}$ randomly initialised by $\mathcal{N}(0,1)$. We compute the unifying features of symbols using a bi-directional GRU [31] (see Section 2.2.4 for further details) over the embedded features of the symbols:

$$\phi_U(s) = \boldsymbol{W}\,\text{bi-GRU}(\phi(s)) \tag{5.1}$$

where bi-GRU $\in \mathbb{R}^{2d}$ is the concatenation of the hidden states of the GRU at symbol s and $\boldsymbol{W}$ is a learnable weight matrix. By using a recurrent neural network, the symbol's context can be taken into account when computing its unifying features such as whether it is at the head or tail of the sequence. For example, in the sequence 2342, the unifying features ϕ_U of the digit 2 may be different due to their position in the sequence despite having the same embedding ϕ.

To predict the desired symbol y, the downstream network f is a feed-forward network consisting of 2 hidden layers with tanh non-linearity of sizes $2d$ and d respectively and an output layer of size d. Given some embedded input $\boldsymbol{X} \in \mathbb{R}^{l\times d}$, obtained either as $\boldsymbol{X} = \phi(K)$ or through soft unification $\boldsymbol{X} = g(I,K)$ as in Algorithm 1, we flatten it to obtain an input vector $\boldsymbol{x}$ and concatenate the one-hot

encoded query. Concretely for the fixed length sequences task, we take symbol embeddings of size $d = 16$ and flatten the sequence of length $l = 4$ to obtain an input vector $\boldsymbol{x} \in \mathbb{R}^{64+4}$ with the one-hot encoded query (4 in total) appended. The final prediction is obtained via $f(\boldsymbol{x}) = \text{softmax}(\boldsymbol{h}\boldsymbol{E}^T)$ where $\boldsymbol{h} \in \mathbb{R}^d$ is the output of the last layer in the network.

5.3.2 Unification CNN

Convolutional neural networks by design are apt at exploiting the spatial information available in the input space and are widely used for grid-like or visual inputs (refer back to Section 2.2.3 for further details). To adapt and build an unification network geared towards grid based inputs, we utilise convolutional networks to compute the unifying features ϕ_U and the final predictions through f. While the variableness and the features of symbols are computed as in the Unification MLP network, the unifying features are obtained by convolving the symbol embeddings:

$$\phi_U(s) = c_2(\text{ReLU}(c_1(\phi(s)))) \tag{5.2}$$

where c_1, c_2 are the d-dimensional outputs of the convolutional kernels at symbol s in the input grid. This formulation allows the unifying features to incorporate the spatial information of symbols such as whether they are surrounded by other symbols or they are at the corner of the grid. The grid is padded with 0s to obtain the same grid size after each convolution such that every symbol has a unifying feature. The padding itself is masked out for the unification stage at Step 4 in order to avoid assigning null values to variables.

Given a grid of embedded symbols as input $X \in \mathbb{R}^{w \times h \times d}$, again either as $X = \phi(K)$ or through soft unification $X = g(I, K)$, where w and h are the width and height respectively, the final prediction utilises another convolutional network such that:

$$f(X) = \text{softmax}((\boldsymbol{W}\boldsymbol{h} + \boldsymbol{b})\boldsymbol{E}^T) \tag{5.3}$$

$$\boldsymbol{h} = \max_{w,h} \text{ReLU}(c_2^f(\text{ReLU}(c_1^f(X)))) \tag{5.4}$$

where $\max_{w,h}$ is the global max-pooling operation, c_1^f, c_2^f the convolutional layers of the network applied to the entire input grid and $\boldsymbol{W}, \boldsymbol{b}$ are learnable weights. For the specific 3×3 grid dataset from Section 5.2, we start with an input tensor $X \in \mathbb{R}^{3 \times 3 \times (32+4)}$ with a symbol embedding of size $d = 32$ concatenated with the one-hot query id (the grid dataset has 4 tasks). For all the convolutional layers, c_1, c_2, c_1^f, c_2^f we use $d = 32$ filters, kernel size of 3 and a stride of 1. Despite the convolutional networks used for ϕ_U and f are architecturally identical, they serve different purposes and highlight the modular nature of the overall unification network approach.

5.3.3 Unification RNN

In order to process varying length sequences such as natural language sentences, we construct unification networks using recurrent components. To learn patterns over and predict the sentiment

of the movie reviews dataset described in Section 5.2, we take each natural language word as an unique symbol and utilise the ConceptNet word embeddings [203] as the origin of the symbol features $\boldsymbol{E}^{\text{ConceptNet}}$. Word embeddings trained on a large natural language corpus capture semantic similarities between words and serve as a starting point for downstream tasks [134, 157]. To leverage this rich information offered by pre-trained ConceptNet word embeddings, we build the remaining learnable components using it:

$$\phi(s) = \boldsymbol{W}\boldsymbol{E}^{\text{ConceptNet}}_{s,:} + \boldsymbol{b} \tag{5.5}$$

$$\psi(s) = \sigma(\boldsymbol{W}\phi(s) + b) \tag{5.6}$$

$$\phi_U(s) = \boldsymbol{W}\phi(s) + \boldsymbol{b} \tag{5.7}$$

where $\boldsymbol{W}, \boldsymbol{b}, b$ are separate independent learnable weights for each equation. The benefit of tying the features to the word embeddings is that if a word aids in predicting a sentiment and emerges as a variable, any semantically similar word would inherit the same properties. For example, consider the word 'amazing', by the virtue of its potential semantic similarity to the word 'great', if one is recognised as a variable so might the other $\psi(\text{amazing}) \simeq \psi(\text{great})$ since $\phi(\text{amazing}) \simeq \phi(\text{great})$. This might encourage our unification approach to learn sentiment patterns more consistently across different phrases and generalise better to previously unseen words.

Given an embedded input in the form of a matrix $\boldsymbol{X} \in \mathbb{R}^{l \times d}$ where l is the length of the sentence and d the embedding size, the final sentiment prediction is obtained using an LSTM [81] (refer back to Section 2.2.4 for more details):

$$f(\boldsymbol{X}) = \sigma(\boldsymbol{W}\,\text{LSTM}(\boldsymbol{X}) + b) \tag{5.8}$$

where $\text{LSTM}(\boldsymbol{X}) \in \mathbb{R}^d$ is the final hidden state of the LSTM over the input $\boldsymbol{X}$ and $\boldsymbol{W}, b$ are learnable parameters. We apply a dropout of 0.5 to the output of the LSTM hidden state to reduce overfitting [205] and set the initial states of the LSTM as zero vectors following standard practice. Although the original ConceptNet word embeddings have a size of 300, we project them down to $d = 16$ dimensions to reduce the computational resources required to run the experiments.

Given the variety of ways one could express positive and negative movie reviews in natural language, we extend the idea of using a single invariant in Step 1 to include a set of invariant examples $\mathbb{I}$. For example, 'easily one of the best films' may be deemed a different pattern to 'a film you will never forget' despite potentially denoting positive sentiment. To let the models learn and benefit from different invariants, we can pick multiple examples to generalise from and aggregate the predictions from each invariant. One simple approach is to sum the predictions of the invariants $\sum_{I \in \mathbb{I}} f \circ g(I, K)$ as used in Unification MLP, CNN and RNN.

5.3.4 Unification Memory Network

Soft unification does not need to happen prior to f in a $f \circ g$ fashion but can also be incorporated at any intermediate stage multiple times. To demonstrate this ability, we unify the symbols at different memory locations at each iteration of a Memory Network [219]. Memory Networks utilise an iterative procedure to interact with a memory component using attention mechanisms. We take a list of lists as input such as a tokenised story as shown in Fig. 5.3; the outer list consists of sentences and each sentence consists of a sequence of word tokens. First we describe the predictor memory network f and then focus on how at each iteration soft unification is applied over the tokens of the selected memory slots between an invariant and another input example.

As the natural language story or the logic program represented by the memory component does not change throughout the execution of the neural network, we compute a *read-only* memory matrix $\boldsymbol{M} \in \mathbb{R}^{N \times d}$ where N is the number of memory slots (fixed by the length of the story or number of rules of a logic program) and d is some embedding size:

$$\phi(s) = \boldsymbol{E}_{s,:} \tag{5.9}$$

$$\boldsymbol{M}_{i,:} = \text{bi-GRU}(\phi(C_i)) \tag{5.10}$$

where $\boldsymbol{E} \in \mathbb{R}^{|\mathbb{S}| \times d}$ is a learnable embedding matrix, $\phi(C_i)$ is an element-wise application over the sequence of tokens at the ith context and bi-GRU $\in \mathbb{R}^d$ is the mean of the last hidden states of a bidirectional GRU. The context sentence or rule C_i is parsed as a sequence of either natural language words like in Unification RNN (Section 5.3.3) or characters of each rule from a logic program. The same procedure is applied to the query of the input q to obtain an embedded query representation $\boldsymbol{q} \in \mathbb{R}^d$. Starting with the query representation as the initial state $\boldsymbol{h}^0 = \boldsymbol{q}$, the predictor memory network f iterates a fixed number of times and updates the hidden state of the network. First, an attention map over the memory slots is computed:

$$\boldsymbol{A}^t_{i,:} = \tanh(\boldsymbol{W}\rho(\boldsymbol{M}_{i,:}, \boldsymbol{h}^t) + \boldsymbol{b}) \tag{5.11}$$

$$\alpha^t = \text{softmax}(\boldsymbol{W}\,\text{bi-GRU}(\boldsymbol{A}^t) + b) \tag{5.12}$$

where $\boldsymbol{A}^t \in \mathbb{R}^{N \times d}$ is an intermediate slot affinity matrix, bi-GRU $\in \mathbb{R}^{N \times d}$ denotes the outputs of another recurrent neural network as a matrix of all the hidden states (the recurrent network is unrolled over the rows of $\boldsymbol{A}$), $\boldsymbol{W}, \boldsymbol{b}$ are learnable weights and $\rho(\boldsymbol{x}, \boldsymbol{y}) = [\boldsymbol{x}; \boldsymbol{y}; \boldsymbol{x} \odot \boldsymbol{y}; (\boldsymbol{x} - \boldsymbol{y})^2]$. The two equations effectively allow the model to learn which memory slot is the most relevant given the current state of the network captured by $\alpha^t \in \mathbb{R}^N$. The bidirectional GRU allows modelling temporal relationships between memory slots. For example if someone goes to multiple places in a single story, the network needs to differentiate the *last* known location in order to answer the question where that person is. This temporal aspect, however, is not needed for working with logic programs as the rules are permutation invariant, i.e. changing the order of the rules does

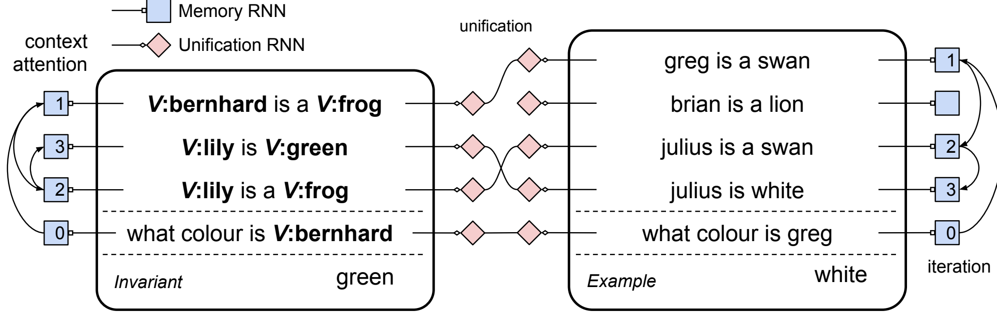


Figure 5.3: Graphical overview of soft unification within a memory network (UMN). Each sentence is processed by two bi-directional RNNs for memory and unification. At each iteration the context attention selects which sentences to unify.

not change the outcome of a query. After successful training, the model often learns to select the most relevant memory slot using the induced categorical distribution α over the entire memory $\boldsymbol{M}$ as shown using blue squares in Fig. 5.3. Using this attention vector with another set of learnable weights $\boldsymbol{W}, \boldsymbol{b}$ (per equation), the next state $\boldsymbol{h}^{t+1}$ and the final prediction $f(C,q)$ are computed as:

$$\boldsymbol{h}^{t+1} = \sum_i \alpha_i^t \tanh(\boldsymbol{W}\rho(\boldsymbol{M}_{i,:,:}, \boldsymbol{h}^t) + \boldsymbol{b}) \tag{5.13}$$

$$f(C,q) = \text{softmax}(\boldsymbol{W}\boldsymbol{h}^{t^*} + \boldsymbol{b}) \tag{5.14}$$

with t^* denoting the last iteration. This resembles the iterative update used by the Iterative Memory Attention (IMA) network, another memory-based neural network introduced in Section 4.1. At each iteration, the state of the network is updated based on the selected memory slot. For a natural language story this may be the sentence involving the entity in the query such as 'greg is a swan' following on from the question 'what colour is greg' as shown for the story in the right-hand side of Fig. 5.3. For a logic program, the hidden state $\boldsymbol{h}^t$ may represent the current derived sub-goal after attending a relevant rule as previously shown in Fig. 4.3.

Incorporation of soft unification into a memory-based neural network involves matching and unifying the memory slots of the invariant I and input example K. We set the variableness of symbols as $\psi(s) = \sigma(w_s)$ where $\boldsymbol{w} \in \mathbb{R}^{|\mathbb{S}|}$ is a learnable weight and compute the unification features using another bidirectional GRU over the symbol features $\phi_U(s) = \boldsymbol{W}\,\text{bi-GRU}(\phi(s))$ similar to the Unification MLP. The application of soft unification for the queries yields $g(q^I, q^K) = \boldsymbol{q}^U \in \mathbb{R}^{n \times d}$ and for the contexts a tensor $g(C^I, C^K) = U \in \mathbb{R}^{N \times n \times M \times d}$ where N, M are the lengths of the outer sequences and n, m the inner ones for I and K respectively. For example, N could be number of sentences of a story and n the number of words in each sentence. Sequences of different lengths are padded with zeros to obtain a uniform tensor shape and the recurrent components skip over this padding in practice. U effectively contains all the pairwise $N \times M$ unifications of varying length sequences at each memory slot of the invariant and the given example. At each iteration, the state update becomes a reduction of the unified representations using the corresponding attention maps:

$$\boldsymbol{h}^{U,0} = \text{bi-GRU}(\boldsymbol{q}^U) \tag{5.15}$$

$$\boldsymbol{M}^{U,t} = \text{bi-GRU}\left(\sum_k \alpha_k^{U,t} U_{:,:,k,:}\right) \tag{5.16}$$

$$\boldsymbol{h}^{U,t+1} = \sum_i \alpha_i^{I,t} \tanh(\boldsymbol{W}\rho(\boldsymbol{M}_{i,:}^{U,t}, \boldsymbol{h}^{U,t}) + \boldsymbol{b}) \tag{5.17}$$

where bi-GRU is the same one as Eq. (5.10), $\alpha^{U,t} \in \mathbb{R}^M$ is the attention map obtained using $\boldsymbol{h}^{U,t} \in \mathbb{R}^d$ and $\boldsymbol{M}^K \in \mathbb{R}^{M\times d}$ following Eq. (5.12), $\boldsymbol{M}^{U,t} \in \mathbb{R}^{N\times d}$ is an intermediate unified memory and $\alpha^{I,t} \in \mathbb{R}^N$ is the attention map for the invariant. Since the invariant is fixed in advance, the attention scores $\alpha^{I,t}$ are the same for any unified input K. For example, at iteration 1 in Fig. 5.3, '***V*:bernhard** is a ***V*:frog**' is matched with 'greg is a swan' and one would expect the variables are replaced with their corresponding counterparts: ***V*:bernhard** with greg and ***V*:frog** with swan, yielding the ground sentence 'greg is a swan'. This ground sentence is then encoded using the bidirectional GRU to obtain the unified representation for the memory slot. As a result, Steps 2 to 4 are interleaved at each iteration of the predictor network although in practice the common computation such as the unification of variables is done once to reduce training time, i.e. all pairwise unifications U are computed in advance. For the bAbI and logical reasoning datasets from Section 5.2, we set $d = 32$ as the size of the embeddings and hidden vectors used through the network and iterate for a fixed number of times as set by the tasks.

To let the model potentially differentiate queries such as 'Where is X?' and 'Why did X go to Y?' within the same task, we gather multiple invariant examples $\mathbb{I}$ in Step 1 similar to Unification MLP, CNN and RNN. However, to produce a single hidden state at each iteration and allow the network to learn which invariant to use, we employ a bilinear attention $\eta \in \mathbb{R}^{|\mathbb{I}|}$ over the hidden states obtained from unifying each invariant with the training example:

$$\eta = \text{softmax}\left(\boldsymbol{Q}\boldsymbol{W}\boldsymbol{q}^K\right) \tag{5.18}$$

$$\boldsymbol{h}^{U,t} = \sum_i \eta_i \boldsymbol{H}_{i,:}^{U,t} \tag{5.19}$$

where $\boldsymbol{Q} \in \mathbb{R}^{|\mathbb{I}|\times d}$ and $\boldsymbol{q}^K \in \mathbb{R}^d$ are the representations of the queries, $\boldsymbol{W}$ is a learnable weight and $\boldsymbol{H}^{U,t} \in \mathbb{R}^{|\mathbb{I}|\times d}$ is the hidden unified states obtained from each invariant. By checking the similarity of the queries, the model has the capacity to select invariant examples that better match the given input although in practice this may create an overhead and hamper training as discussed further in Section 5.4. Note that if there is a single invariant $|\mathbb{I}| = 1$, then the attention mechanism becomes the identity function since the softmax of a single element always yields $\eta = [1]$ (refer back to Eq. (2.6) for the definition of softmax).

The overall effect of unification in a memory network can be summarised by how one can transform the memory of the invariant I such that it yields enough information to predict the desired

outcome of the training input K. Since the unified memory $\boldsymbol{M}^{U,t}$ from Eq. (5.16) utilises the context attention of the invariant example $\alpha^{I,t}$, this setup requires pre-training f such that the context attentions match the correct pairs of sentences to unify which limits the performance of the combined network $f \circ g$ by how well f starts to perform. In the next section, we focus on how f and $f \circ g$ can be trained in an end-to-end fashion to not only provide competitive quantitative results but also yield invariant patterns through the use of soft unification.

5.4 Experiments

We probe three aspects of soft unification: the impact of unification on performance over unseen data, the effect of multiple invariants and data efficiency. To that end, we train Unification MLP, CNN and RNN (Sections 5.3.1 to 5.3.3) with and without unification over different training sizes, and Unification Memory Network (Section 5.3.4) with pre-training while varying the number of invariants. We first describe the experiments involving Unification MLP, CNN and RNN trained on the fixed length sequences, grid and movie sentiment datasets from Section 5.2 respectively.

The loss function balances the predictor network f with the soft unification function g:

$$L = \overbrace{\lambda_K l_{\text{nll}}(y, f(K))}^{\text{Original output}} + \lambda_I \left(\overbrace{l_{\text{nll}}(y, f \circ g(I, K))}^{\text{Unification output}} + \overbrace{\tau \sum_{s \in \mathbb{S}} \psi(s)}^{\text{Sparsity}} \right) \tag{5.20}$$

where l_{nll} is the negative log-likelihood loss (Eq. (2.13)) with sparsity regularisation over ψ at $\tau = 0.1$ to discourage the models from utilising spurious number of variables. Firstly, we add the sparsity constraint over the variableness of symbols $\psi(s)$ to avoid the trivial solution in which every symbol is a variable and G is completely replaced by K in Line 13 of Algorithm 1 still allowing f to predict correctly. Hence, we would like the *minimal* transformation of G towards K to expose the common underlying pattern. Secondly, we adjust λ_K and λ_I in order to enable or disable unification. For Unification MLP and CNN, we set $\lambda_K = 0$, $\lambda_I = 1$ for training just the unified output and the converse for the non-unifying versions. For Unification RNN, we set $\lambda_K = \lambda_I = 1$ to train the unified output and set $\lambda_I = 0$ for non-unifying version. In the case of Unification RNN, training f whilst also training $f \circ g$ yields better performance as opposed to training solely through $f \circ g$ which we speculate is due to the noisy nature of the movie reviews dataset as discussed in Section 5.2. This flexibility in the objective function is achieved by the end-to-end differentiable nature of our novel unification approach.

Every model is trained via back-propagation using Adam [106] with a learning rate of 0.001 and batch size of 64 on an Intel Core i7-6700 CPU and evaluated with 5-fold cross-validation. The models are trained for 2000 iterations where one iteration consists of a single batch update saving the training and test accuracies every 10 iterations. Further results such as training with different learning rates are available in Appendix A.

Fig. 5.4 portrays how soft unification (orange lines) generalises better to unseen examples by achieving higher test accuracies against models without soft unification (blue lines) on the syn-

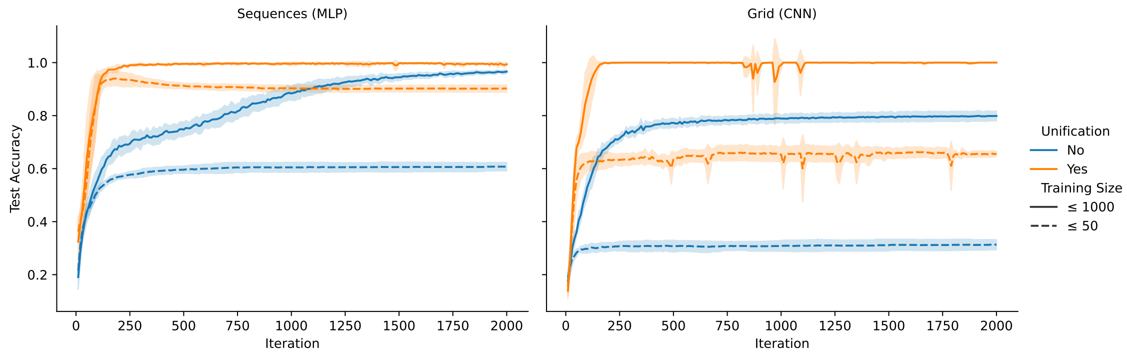
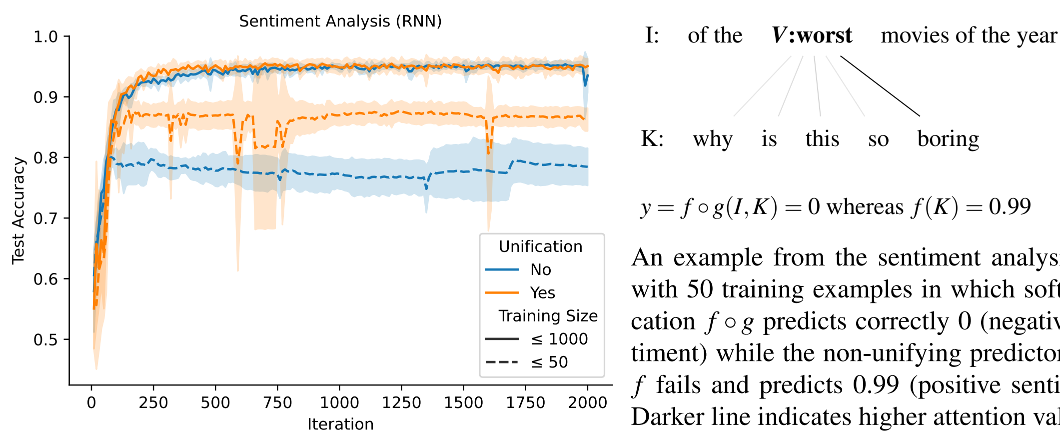


Figure 5.4: Test accuracies of Unification MLP and CNN on the sequences and grid datasets respectively with 1 invariant. Soft unification is more data efficient at generalising to unseen examples than non-unifying version for both training sizes (orange versus blue lines).



$y = f \circ g(I, K) = 0$ whereas $f(K) = 0.99$

An example from the sentiment analysis task with 50 training examples in which soft unification $f \circ g$ predicts correctly 0 (negative sentiment) while the non-unifying predictor RNN f fails and predicts 0.99 (positive sentiment). Darker line indicates higher attention values.

Figure 5.5: Test accuracy of Unification RNN with 1 invariant versus no unification on the sentiment analysis dataset. Soft unification generalises better to unseen movie reviews by ≈ 0.1 with less training data ≤ 50 (dashed orange versus blue lines).

thetic sequences and grid datasets. Models with unification start to consistently outperform those without in as few as 150 iterations (batch updates) for both datasets and training sizes. Despite $f \circ g$ has more trainable parameters than f alone, this data efficiency is more visible when trained with only ≤ 50 examples per task (dashed lines) with an average improvement of ≈ 0.3 across both datasets. The fluctuations in accuracy around iterations 750 to 1000 in Unification CNN are caused by penalising ψ which forces soft unification to use less variables half way through training and subsequently requires the downstream predictor network f to adjust as well. These results are similar when using different learning rates which are shown in the appendix Fig. A.1.

In the real-world dataset of sentiment analysis, Unification RNN (orange lines) performs as good as if not better than its non-unifying version (blue lines) in Fig. 5.5. When using only ≤ 50 training examples, soft unification generalises better to unseen movie reviews in a more data efficient manner and yields an improvement in predictive test accuracy of ≈ 0.1 in as few as 250 iterations (batch updates) against its non-unifying counterpart (dashed orange versus blue lines). In contrast to the unambiguous patterns available in the synthetic datasets used in Fig. 5.4, movie reviews do not contain clear patterns which we believe leads to the noisier, fluctuating test accuracy curves compared to the steadier lines of Unification MLP and CNN. The large fluctuation in test accuracy

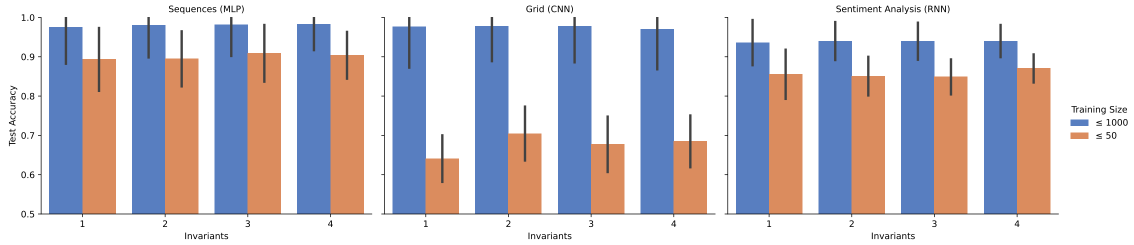


Figure 5.6: Results of Unification MLP, CNN and RNN on increasing number of invariants. There is no impact on performance when more invariants per task are given. We speculate that the models converge to using 1 invariant because the datasets can be solved with a single invariant and the sparsity constraint applied on ψ zeroing out unnecessary invariants.

for Unification RNN at around iteration 700 (dashed orange line) is due to the sparsity constraint on ψ in Eq. (5.20) which causes the invariant to use less and less variables as it is training and deteriorates the predictive performance before recovering again around iteration 750. We believe the improvements in data efficiency and predictive performance obtained via soft unification are linked to the fact that it architecturally biases the models towards learning unifying features that are common across examples, therefore, potentially also common to unseen examples as shown on the right hand side of Fig. 5.5 for the sentiment analysis task. In this example, the full soft unification network $f \circ g(I,K)$ correctly predicts the desired negative sentiment $y = 0$ while the predictor recurrent neural network $f(K)$ on its own incorrectly outputs a positive sentiment (0.99). We speculate that the similarity between the words 'worst' and 'boring', as picked up by the attention map (darker line), produces a unified representation of a similar negative sentiment which the downstream recurrent network predicts correctly. We present further examples in Section 5.5.

Results with multiple invariants are very similar to using a single one as shown in Fig. 5.6. The models seem to learn to ignore the extra invariants which we speculate is due to the sparsity constraint applied on ψ zeroing out unnecessary invariants. This behaviour highlights the level of abstraction we can achieve through ψ. One might consider "2 $\boldsymbol{V}$ 7 $\boldsymbol{V}$" and "$\boldsymbol{V}$ 8 4 $\boldsymbol{V}$" to be different patterns or invariants but at a higher level of abstraction they can both represent the concept of a repeated symbol irrespective of the position of the repeating item. Thus, extra invariants can be ignored in favour of one that accounts for different examples through the learnable unifying features ϕ_U, which might look for the notion of a *repeated symbol* regardless of its position in the sequence. Given the capacity to learn unifying features and the pressure to use the least amount of variables, the models in these cases can optimise to learn and use a single invariant. Training with different learning rates overall paint a similar picture as shown in the appendix Fig. A.4.

Unification Memory Networks are trained using the bAbI and logical reasoning datasets described in Section 5.2. Unlike Unification MLP, CNN and RNN, the interleaved nature of soft unification within a memory network allows us to integrate further supervision into L from Eq. (5.20):

$$L' = L + \frac{\lambda_I}{t^*} \overbrace{\sum_t \left(\boldsymbol{h}^{U,t} - \boldsymbol{h}^{K,t}\right)^2}^{\text{State consistency}} + \mathbf{1}_{\text{strong}} \sum_t \overbrace{\lambda_I l_{\text{nll}}(a^t, \alpha^{U,t}) + l_{\text{nll}}(a^t, \alpha^{K,t})}^{\text{Attention supervision}} \tag{5.21}$$

where we first add the mean squared error between the unified hidden state $\boldsymbol{h}^{U,t}$ and the training

Table 5.3: Aggregate error rates (%) on bAbI 1k for UMN and baselines N2N [209], GN2N [124], EntNet [79], QRN [185] and MemNN [219] respectively. Full comparison and individual task results are available in Appendix A.2.

Supervision	Weak		Strong		Weak				Strong
# Invs / Model	1	3	1	3	N2N	GN2N	EntNet	QRN	MemNN
Mean	18.8	19.0	**5.1**	6.6	13.9	12.7	29.6	11.3	6.7
# > 5%	10	9	3	3	11	10	15	5	4

example hidden state $\boldsymbol{h}^{K,t}$ at each iteration. This term encourages the hidden states between the unified memory and the original memory to be consistent across iterations as opposed to waiting for the final prediction. Recall that we already compute $f(K)$ in Eq. (5.20); hence, the state consistency loss does not incur additional computation. The second added term is referred to as *strong supervision* in the bAbI and logical reasoning datasets whereby the model is supervised as to which memory slot, called the supporting fact, it should select at each iteration. The desired memory slot a^t, either a sentence in a story or a rule of a logic program, is given as part of the input. When the $\mathbf{1}_{\text{strong}}$ condition is false, it is referred to as *weak supervision* and requires the model to learn which memory slots to attend to. Weak supervision is a relatively harder training regime, particularly for tasks that require many iterations since the correct learning signal is only presented at the final iteration.

To enable pre-training of the predictor memory network f, we start with $\lambda_K = 1,\ \lambda_I = 0$ for 40 epochs and then set $\lambda_I = 1$ to jointly train the unified output using Eq. (5.21). We again use a batch size of 64 and the Adam optimizer with a learning rate of 0.001. A dropout of 0.1 is applied to all recurrent neural network components used within the memory network to reduce over-fitting [205]. Only for the bAbI dataset, we also apply a weight decay of 0.001 to further counteract over-fitting following previous work [209].

For iterative reasoning datasets, Tables 5.3 and 5.4 aggregate the results for our approach against comparable baseline memory networks which are selected based on whether they are also built on Memory Networks (MemNN) [219] and predict by iteratively updating a hidden state. For example, End-to-End Memory Networks (N2N) [209] update a hidden state vector after each iteration by attending to a single context sentence similar to our Unification Memory Network architecture described in Section 5.3.4. We observe that strong supervision yields lower error rates, e.g. 5.1 in column 3 versus 18.8 in column 1 in Table 5.3, which is consistent with previous work reflecting how $f \circ g$ can be bounded by the efficacy of f modelled as a memory network. In a weak supervision setting, i.e. when sentence selection α is not supervised in Eq. (5.21), our model attempts to unify arbitrary sentences often failing to follow the iterative reasoning chain. Learning which context memory slots to attend to whilst learning their representations as well as how to unify them may be too complex of a task to tackle in an end-to-end fashion. As a result, only in the supervised case we observe a minor improvement over MemNN by 1.6 in Table 5.3 and over IMA by 4.1 in Table 5.4 (column 3 versus the last column in both tables).

This dependency on f reflects on the holistic nature of our unification approach such that despite the architectural biases to learn and utilise variables through soft unification, the process is funda-

Table 5.4: Aggregate task error rates (%) on the logical reasoning dataset (2k training examples) for UMN and baseline IMA [32]. Refer to Table A.3 for individual task results.

Model	UMN				IMA	
Supervision	Weak		Strong		Weak	Strong
# Invs	1	3	1	3	-	
Mean	37.7	37.6	**27.4**	29.0	38.8	31.5
$\# > 5\%$	10	10	10	11	11	11

Table 5.5: Number of *exact* matches of the learnt invariants with expected ones in synthetic datasets where the invariant is known. For sequences and grid datasets, there are 5 folds each with 4 tasks giving 20 and the logic dataset has 12 tasks with 3 runs giving 36 invariants.

Model	UMLP		UCNN		UMN	
Dataset	Sequences		Grid		Logic	
Train Size	$\leq 1k$	≤ 50	$\leq 1k$	≤ 50	2k	
Supervision			-		Weak	Strong
Correct / Total	18/20	18/20	13/20	14/20	7/36	23/36
Accuracy (%)	90.0	90.0	65.0	70.0	19.4	63.9

mentally limited by f as the gradients must back-propagate through it. If the downstream predictor network cannot solve the task, then soft unification falters too. We speculate that the slight increase in error rates with 3 invariants in Table 5.3 (columns 2 and 4 versus columns 1 and 3) is also related to this phenomenon and stems from having more parameters and more pathways in the network, rendering training more difficult and slower. While f may limit how soft unification performs, we do not observe the converse across all the datasets in Figs. 5.4 and 5.5 and Tables 5.3 and 5.4. The overall unification networks $f \circ g$ perform comparatively if not better against baselines. In these cases, our approach is still able to learn invariant patterns which we analyse further in Section 5.5.

For the synthetic sequences, grid and logic datasets in which we know exactly what the invariants can be, Table 5.5 shows how often our approach captures *exactly* the expected invariant. We threshold ψ from Step 2 as explained in Section 5.5 and check for an exact match; for example for predicting the head of a sequence with Unification MLP, we compare the learnt invariant against the pattern '***V***_ _ _'. We observe that the accuracy of recognising the desired invariants drops from 0.9 down to 0.63 with increasing dataset complexity ranging from simple sequences to logic programs (left to right). However, the models may still solve the tasks despite having learnt a different invariant. In these cases, they may use extra or more interestingly fewer variables which we analyse and discuss in the next section.

5.5 Analysis

In addition to the quantitative predictive performance of unification networks discussed so far, in this section we redirect our attention to the qualitative analysis of the learnt invariants. In particular, we are interested in visualising which symbols emerge as variables and how the unification networks assign new values to them in Steps 2 and 4 of the soft unification algorithm presented in Section 5.1 respectively.

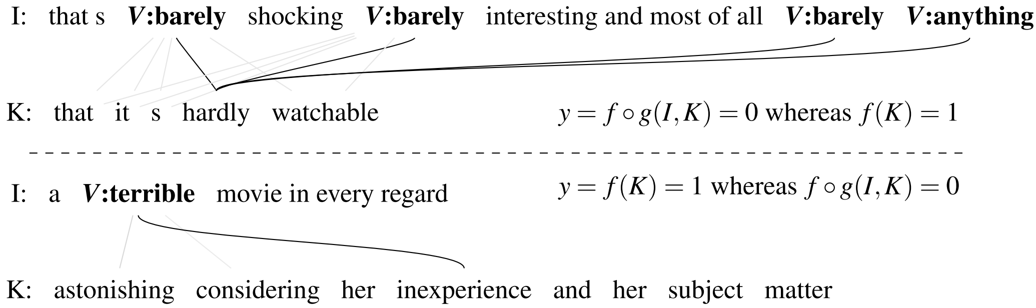


Figure 5.8: Samples from the sentiment analysis task with 50 training examples showing soft unification $f \circ g$ success and failure. Darker lines are higher attention values.

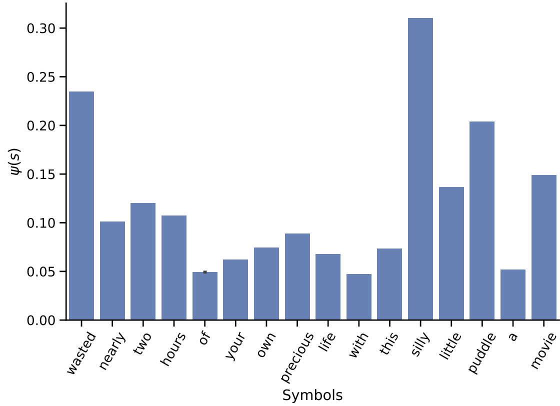


Figure 5.7: The variableness $\psi(s)$ of an invariant for sentiment analysis with words such as 'silly' emerging as variables. The symbols on the x axis represent a single data point (movie review).

Fig. 5.7 shows an invariant for sentiment analysis in which sentiment related words in the sentence such as 'silly' and 'wasted' have relatively higher variableness values ψ compared to most of the other words like 'hours' and 'this'. Intuitively, if one replaces 'silly' with the adjective 'great', the sentiment might change. This replacement or variable binding, however, is not a hard value assignment but an interpolation as in Line 13 from Algorithm 1 which produces a new intermediate representation from G towards K different enough to allow f to predict correctly. From this perspective, soft unification can be seen as a transformation between examples that is dependent on the downstream predictor network f. Since we penalise the magnitude of ψ in Eq. (5.20), we expect these values to be as low as possible. Thus, the final converged solution, if any, is a *minimal* interpolation from the invariant example G towards K which is key to unearth symbols that vary versus those that do not.

We threshold $\psi(s) > t$ to extract the learned invariants and set t to be the mean of the variable symbols $t = \frac{1}{|\mathbb{S}|} \sum_s \psi(s)$ as a heuristic except for the bAbI dataset where we use $t = 0.1$. The magnitude of t depends on the amount of regularisation τ applied in Eq. (5.20), the number of iterations and the batch size. Sample invariants shown in Figs. 5.8 and 5.9 describe the patterns present in the tasks with parts that contribute towards the final answer becoming variables. Since soft unification is dependent on f, the symbols that are recognised as variables seem to be consistent with those that affect the final prediction. Extra symbols such as 'is' or 'travelled' do not emerge as variables, as shown in Fig. 5.9a; we attribute this behaviour to the fact that changing the token 'travelled' to 'went' does not influence the prediction but changing the action, the value of ***V*:left** to 'picked' does. Pre-training f as done in UMN seems to produce more robust and consistent invariants since, we speculate, a pre-trained f encourages more $g(I,K) \approx K$. As movie reviews written by humans lack clear patterns, the invariants for the sentiment analysis dataset shown in Fig. 5.8 are not as consistent as those from the synthetic datasets. One might reasonably expect in the top

***V*:john** travelled to the ***V*:office**
***V*:john** ***V*:left** the ***V*:football**
where is the ***V*:football**
office

(a) bAbI task 2, two supporting facts. The model also learns ***V*:left** since people can also drop or pick up objects potentially affecting the answer.

this ***V*:morning** ***V*:bill** went to the ***V*:school**
yesterday ***V*:bill** journeyed to the ***V*:park**
where was ***V*:bill** before the ***V*:school**
park

(b) bAbI task 14, time reasoning. ***V*:bill** and ***V*:school** are recognised as variables alongside ***V*:morning** capturing *when* someone went.

5 8 6 4	const	2
***V*:8** 3 3 1	head	8
8 3 1 ***V*:5**	tail	5
***V*:1** 4 3 ***V*:1**	dup	1

(c) Successful invariants learned with UMLP using 50 training examples only shown as (C, q, y).

0 ***V*** ***V***	0 1 0	0 0 1
0 ***V*** ***V***	6 ***V*** 8	0 5 4
0 0 0	0 7 0	7 8 ***V***
box	centre	corner

(d) Successful invariants learned with UCNN. Variable default symbols are omitted for clarity.

***V*:x** (K) ← ***V*:n** (K),
***V*:l** (U) ← ***V*:x** (U),
***V*:i** (T) ← ***V*:l** (T),
***V*:n** (***V*:o**) ⊢ ***V*:i** (***V*:o**)

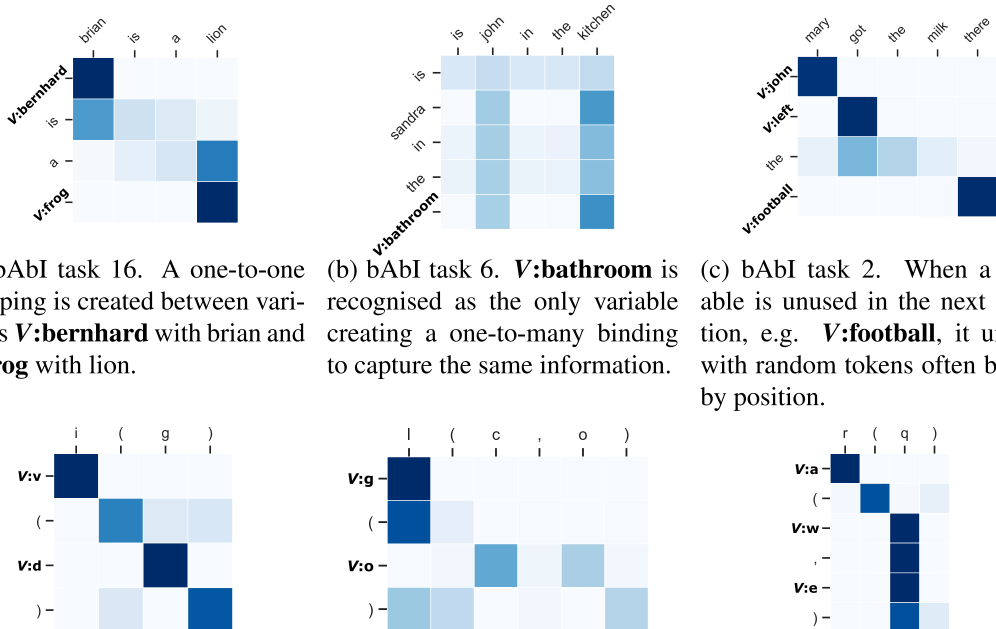
(e) Logical reasoning task 5 with arity 1. The model captures how ***V*:n** could entail ***V*:i** in a chain.

Figure 5.9: Invariants learned across the four datasets using the three architectures. For iterative reasoning datasets, bAbI and logical reasoning, they are taken from strongly supervised UMN.

most invariant that the word ‘shocking’ would be recognised as a variable but the unification network $f \circ g$ utilises ***V*:barely** instead to correctly predict the sentiment while the predictor network f fails. In contrast, the invariant on the bottom of Fig. 5.8 matches our intuition of recognising ***V*:terrible** but makes an incorrect prediction. Further examples are shown in Appendix A.2.

A desired property of interpretable models is transparency [122]. A novel outcome of the learned invariants in our approach is that they provide an *approximation* of the underlying general principle that may be present in the data. Fig. 5.9e captures the structure of multi-hop reasoning in which a predicate ***V*:n** (the fact before ⊢) can entail another ***V*:i** (atom after the ⊢) with a matching constant ***V*:o** if there is a chain of rules (first 3 rows) that connect the two predicates. This invariant captures what multi-hop rule application is at a character level (refer back to Section 3.1 for a detailed task description of n-step deduction). However, certain aspects regarding the ability of the model such as how it performs temporal reasoning, are still hidden inside f. In Fig. 5.9b, although we observe ***V*:morning** as a variable, the overall learned invariant captures nothing about how changing the value of ***V*:morning** alters the behaviour of f, i.e. how f uses the interpolated representations produced by g. The downstream model f might look *before* or *after* a certain time point ***V*:bill** went somewhere depending what ***V*:morning** binds to. Whilst having invariants is a step closer to transparency in an otherwise black-box nature of deep learning models, soft unification is limited to offering an insight only at the input symbol level and not inside the predictor network where arguably the core reasoning is carried out.

Attention maps from Line 10 in Algorithm 1 reveal three main patterns: one-to-one, one-to-many or many-to-one bindings as shown in Fig. 5.10. Figs. 5.10a and 5.10d capture what one might expect unification to look like where variables unify with their corresponding counterparts, e.g. ***V*:bernhard** with brian and ***V*:frog** with lion. However, occasionally the model can optimise to use less variables and *squeeze* the required information into a single variable, for example by binding ***V*:bathroom** to john and kitchen in Fig. 5.10b and binding a single argument ***V*:d** to



(a) bAbI task 16. A one-to-one mapping is created between variables ***V*:bernhard** with brian and ***V*:frog** with lion.

(b) bAbI task 6. ***V*:bathroom** is recognised as the only variable creating a one-to-many binding to capture the same information.

(c) bAbI task 2. When a variable is unused in the next iteration, e.g. ***V*:football**, it unifies with random tokens often biased by position.

(d) Logical reasoning task 1. A one-to-one alignment is created between predicate and constant symbols.

(e) Logical reasoning task 3. An arity 1 atom is forced to bind with an arity 2 atom creating a one-to-many mapping.

(f) Logical reasoning task 1. An arity 2 predicate is forced to bind with an arity 1 atom creating a many-to-one binding.

Figure 5.10: Variable bindings from Line 10 in Algorithm 1. Darker cells indicate higher values.

two constants in Fig. 5.10e. We believe this occurs due to the sparsity constraint on $\psi(s)$ in the loss function from Eq. (5.20) encouraging the model to be as conservative as possible. Since the downstream network f is also trained, it has the capacity to compensate for condensed or, what one might consider, malformed unified representations; a possible option could be to freeze the downstream predictor network f while learning to unify in order to mitigate the effect of the sparsity constraint.

If a symbol is no longer needed at a particular iteration but is recognised a variable, it may unify with an arbitrary symbol often biased by its position. Fig. 5.10c demonstrates this phenomenon at iteration 1 for bAbI task 2 which requires a model to find where an object is after someone has picked it up or dropped it. The variable ***V*:football** is actually no longer needed to predict the answer to the question ‘Where is the football?’ at iteration 1 since the symbol ‘football’ is used to find who carries it and then the task is concerned with where that person is. Hence, what ***V*:football** binds to no longer affects the final prediction after iteration 1. This situation can be remedied by modelling the variableness of symbols ψ in a context sensitive fashion in Section 5.3.4 since whilst the symbol ‘football’ may be recognised as a variable in the question, it may not be for its other appearances within the context of the story. Alternatively, the model could be penalised through a change in the task by making it impossible to predict correctly unless it unifies ‘football’ with ‘milk’. Finally, if there are more variables than needed as in Fig. 5.10f, we observe a many-to-one binding with ***V*:w** and ***V*:e** mapping to the same constant q. This behaviour begs the question how does the model differentiate between $p(q)$ and $p(q,q)$. We speculate the model uses the magnitude of $\psi(w) = 0.037$ and $\psi(e) = 0.042$ to encode the difference despite both variables unifying with the same constant.

3	**V:4**	7	**V:8**	head
7	4	**V:2**	**V:6**	head
V:4	3	1	**V:5**	tail
V:3	**V:3**	5	**V:6**	duplicate

(a) Invariants with extra variables learned with Unification MLP (Section 5.3.1).

V:4	0	0	0	**V:4**	0	6	4	**V:3**
0	1	0	1	**V:6**	8	0	**V:2**	8
0	0	**V:3**	0	2	0	0	0	7
	head			centre			corner	

(b) Mismatching invariants learned with Unification CNN (Section 5.3.2).

Figure 5.11: Invariants learned that do not match the data generating distribution from Unification MLP and CNN using ≤ 1000 examples to train. In these cases, soft unification still assigns the correct symbols in order to predict the desired answer.

Without the regularising term on $\psi(s)$, we initially noticed the models using extra symbols as variables and binding them to the same value occasionally producing unifications such as "bathroom bathroom to the bathroom" and still f predicting, unsurprisingly, the correct answer as bathroom. Even with the regularisation, our approach may learn what one might call spurious or incorrect variables as shown in Fig. 5.11 where the invariants do not match the data generating patterns of the tasks. However, in terms of predicting the correct symbol using these invariants, either the downstream predictor network f or the unifying features ϕ_U can compensate for the extra or mismatching variables. That is, the overall model $f \circ g$ correctly predicts the answer. For example, the second invariant '7 4 **V:2** **V:6**' from Fig. 5.11a could predict the head of another example by unifying the tail with the head using ϕ_U of those symbols from Step 3 and let f predict the tail symbol of the invariant **V:6** effectively creating a warped version of the task but ultimately compensating for the representations it has learnt. While this demonstrates the capacity and the flexibility of our approach, it may well be regarded as a limitation under an end-to-end training regime since there is no guarantee that the desired invariants will be learnt. Hence, regularising ψ with the correct amount τ to reduce the capacity of unification as well as training f along with $f \circ g$ in Eq. (5.20) seems critical in extracting not just any invariant but one that represents the common structure.

5.6 Related Work

Learning an underlying general principle in the form of an invariant is often the means for arguing for generalisation in neural networks. For example, neural networks designed to mimic computers such as Neural Turing Machines [70] and Differentiable Neural Computers [71] are deemed to have generalised to previously unseen test examples by learning simple algorithms and patterns from training data. In Chapter 4, we test Neural Reasoning Networks on longer and larger logic programs to evaluate whether the models might have captured the underlying pattern of syntactic entailment. However, in all these cases, the learnt pattern or algorithm in question is *hidden* from the user. In fact for Memory Networks, Weston, Chopra, and Bordes [219] claim "MemNNs can discover simple linguistic patterns based on verbal forms such as (X, dropped, Y), (X, took, Y) or (X, journeyed to, Y) and can successfully generalise the meaning of their instantiations." However, this claim is based on the output of the predictor memory network f, and it is unknown whether the model has truly learned such representations or is indeed utilising them. Our approach sheds light on this ambiguity and presents these linguistic patterns as well as syntactic entailment at a character level (as shown in Fig. 5.9) explicitly as invariants, ensuring their utility through the soft unification function g without solely analysing the output of f on previously unseen inputs.

Although we associate these invariants with our existing understanding of the task to perhaps mistakenly anthropomorphise the machine, for example by thinking it has learned ***V*:mary** as *someone*, it is important to acknowledge that these are just symbolic patterns. The invariants only indicate what symbols *might* need to be altered in order for the downstream network f to predict correctly. They do not make our approach, in particular the downstream predictor network f, more interpretable in terms of how these invariants are used or what they mean to the model. More recently, RuleBERT [177] leverages a very large pre-trained language model called BERT [45] to fine-tune and teach natural language based rules such as 'Two persons living together are married.' so as to perform deductive reasoning. Although the quantitative performance over reasoning tasks show an increase, how natural language rules are represented or utilised is unclear. In all these cases, our interpretations may not necessarily correspond to any understanding of the machine, relating to the Chinese room argument [182] since they operate at a level of symbolic recognition and manipulation potentially devoid of any task related semantics.

Learning invariants by lifting ground examples is related to the bottom-up approach of least common generalisation [168]. In this case, inductive inference is performed on facts [194] such as generalising *went(mary,kitchen)* and *went(john,garden)* to *went(X,Y)*, i.e. going from ground instances to more general statements that utilise variables. Unlike in a predicate logic setting, our approach allows for soft alignment between variables and constants and therefore generalisation between varying length sequences. Existing neuro-symbolic systems [24] focus on inducing rules that adhere to *given* logical semantics of what variables and rules are. For example, δ-ILP [52] constructs a network by rigidly following the given semantics of first-order logic (Section 2.1.2) and searches viable rules through gradient-based optimisation. Similarly, Lifted Relational Neural Networks [202] ground first-order logic rules into a neural network while Neural Theorem Provers [170] build neural networks using backward-chaining [175] on a given background knowledge base with templates. This architectural approach for combining logical variables is also observed with TensorLog [38] and Logic Tensor Networks [186]. Additionally, grounding first-order logical rules can also be used as regularisation [87]. These approaches, summarised under the Neural Networks from Logic category in Fig. 1.2 from Section 1.3, exploit the notion of a variable inside a pre-defined logical formalism with a focus on presenting a practical approach to solving certain problems rather than learning it. On the other hand, our motivation stems from a cognitive perspective as encouraged in Section 1.1.

As highlighted in Section 5.5, the invariants capture patterns that potentially approximate the data generating distribution but we still do not know *how* the predictor network f uses them downstream. Thus, from the perspective of explainable artificial intelligence (XAI) [2], learning invariants or interpreting them does not constitute an explanation of the reasoning model f even though the invariant 'if *someone* goes *somewhere* then they are there' might look like one. Instead, it can be perceived as causal attribution [135] in which someone being somewhere is attributed to them going there. This perspective also relates to gradient based model explanation methods such as Layer-Wise Relevance Propagation [11] and Grad-CAM [183, 29] that attempt to unearth the importance of certain pixels in an input image for a specific classification. Consequently, a possible view on ψ from Step 2 in Section 5.1, is a gradient based usefulness measure such that a

symbol utilised downstream by f to determine the answer becomes a variable similar to how a group of pixels in an image contribute more to its classification. However, gradient based saliency methods have shown to be unreliable if based solely on visual assessment [3], also emphasised as a potential limitation of our approach in Fig. 5.11.

Finally, one can argue that unification networks maintain a form of counterfactual thinking [173] in which soft unification g creates counterfactuals on the invariant example to alter the output of f towards the desired answer in Step 5. The question *where Mary would have been if Mary had gone to the garden instead of the kitchen* is the process by which an invariant is learned through multiple examples during training; the invariant example G is transformed to predict many other training examples as in Eq. (5.20). This view relates to methods of causal inference [156, 84] in which counterfactuals are vital as demonstrated in structured models [155].

5.7 Discussion

Can machines learn from data the notion of variables and utilise them? With our novel approach of amalgamating two input streams using soft unification, the resulting neural unification networks identify and assign new values to varying symbols which reveal common underlying patterns in the form of invariants as outlined and discussed in Sections 5.4 and 5.5. Whilst using an architecturally flexible, end-to-end trainable neural network enhanced with variables may yield data efficient results and learn invariants such as in Fig. 5.9, it may also produce undesired or unexpected solutions similar to ones highlighted in Fig. 5.11. The success and failure cases both stem from the plug-and-play nature of differentiable components within deep learning. On one hand, it provides the power to construct networks for and learn invariants over different input structures as in Section 5.3; on the other hand its capacity to learn finds solutions outside the originally intended hypothesis space. Thus, this brings forward a dilemma as to how one can balance the desired behaviour of a network without severely limiting its fundamental ability to *learn* as it may not learn what is desired. While the active area of neural network regularisation research offers practical solutions for common over-fitting scenarios by limiting their capacity [67], learning the unexpected invariant is not necessarily a product of over-fitting as the models may continue to perform well against previously unseen test examples.

While the varying symbols organically emerge during training based on the available data, the idea that symbols may vary is baked into the architecture of unification networks. Steps 2 to 5 provide a novel recipe on how one might incorporate variables into a neural network but leave certain steps learnable. From the perspective of utilising architectural biases to leverage some form of symbolic reasoning over a neural substrate, one may question whether Section 5.1 provides the *right* bias. Is there an ultimate neuro-symbolic bias of the human mind that allows it to employ variables in everyday reasoning? How much bias is too much or too little? These questions kindle an active area of research as to how far we can go with designing networks to exploit or elicit certain behaviours such as going from the simpler multi-layer perceptron in Section 2.2.1 to more adapted convolutional networks in Section 2.2.3 and here, to highly customised unification networks. This alludes to a search problem of its own: an architectural search for neural networks whether they

are general purpose or specialised. We revisit and discuss this line of thought further in Chapter 7.

So far we worked with discrete sequences of symbolic input such as digits, words and characters as described in Section 5.2 but harnessed continuous real-valued vector representations of such symbols (e.g. ϕ and ϕ_U in Section 5.1) to incorporate the logic inspired idea of variables. Yet, the neuro-symbolic account of the human mind accommodates discrete entities from continuous input such as processing an image into objects and relations to reason with. Furthermore, the learnt invariants shown in Fig. 5.9 lack a logical foundation, and the core problem solving is hidden inside the upstream network f. These two arguments push towards a more coherent neuro-symbolic system capable of processing continuous inputs into a formalised proof system in an end-to-end fashion. In the next chapter, we propose a novel algorithm for learning discrete logic-based relations and rules directly from images.

Chapter 6

Learning Symbolic Rules from Pixels

In contrast to Chapters 4 and 5 in which we attempt to learn real-valued vector representations of symbolic input, the world abounds with continuous and unstructured stimuli from which the human mind renders discrete and structured symbolic thought. For example, light as an input starts its journey as a wave with continuous frequency and amplitude values that trigger an electric response from the photoreceptor cells in the human eye [85]. From this humble beginning, the light that reflects off this screen or paper transforms into characters and words as we read. The human brain readily accommodates this journey from continuous to discrete realms everyday for visual and audio processing and thereafter high-level reasoning. Despite being surrounded by continuous input, humans have evolved to recognise, process and maintain symbolic thought that seems to have co-evolved with the use of a symbolic natural language [215]. The result is a coherent information processing system which can integrate low-level signals with high-level abstract reasoning so well that symbolic cognitive models [117] suggest the human mind operates on formal symbols. Similarly, physical symbol systems [147] characterise cognition as not only symbol recognition but also manipulation and combination. This symbolic thought on top of signal processing within a connectionist architecture is learnt at a young age with children building an understanding of objects, their relations and rules in their environment [160]. Furthermore, neurobiological mechanisms in the human brain have been proposed for handling triplets of symbols, e.g. in the form of subject, verb and object [90]. Yet, this level of harmony between neural and symbolic domains remains a mystery for machine learning.

The juxtaposition of continuous and discrete representations mentioned in Section 1.2 lies at the heart of neuro-symbolic methods where high-level reasoning formalised by logic operates on discrete symbols such as predicates, constants and rules whereas neural networks are concerned with continuous values such as neuron activations, weights and gradients. In this chapter, we pose the question: *how can a neural network learn not only clear and but also verifiable discrete logic-based computation atop continuous input such as images?* By clear, we mean a direct correspondence between the calculations carried out by the network and that of Boolean algebra and by verifiable, we mean whether the same computation can be carried out by a different program such as a symbolic solver. Previous works in the category of using logic-inspired architectural biases fall short of realising logic programs within neural networks and offer only relations [179, 151],

predicates [191] and soft rule applications [69]. Our novel approach provides a new perspective on existing feed-forward neural networks to build an end-to-end architecture that can not only handle low-level input such as images but also converge to symbolic relations and rules.

Although early connectionist approaches, such as the perceptron [174], were trained to learn AND and OR gates, the current state-of-the-art deep neural networks have departed from logic based reasoning [58]. Yet, the ingredients to revitalise neural networks that incorporate Boolean connectives without altering their architecture can be traced back to early pioneering works. In Section 2.2.1, we already established that conjunction and disjunction are linearly separable and that a single linear layer from Definition 6 can learn any linearly separable function. This can be easily verified if the training data is generated from the corresponding Boolean connective such as $y = x_1 \wedge x_2$ (Table 2.1) and the layer is required to correctly predict y given some propositional atoms x_1 and x_2. On the other hand, it is not clear as to how this behaviour can be learnt as part of a larger neural network that, for example, does image classification. We would like the overall network to still be able to not only process images using convolutional networks but also eventually exhibit some Boolean algebra that may constitute a logic program with predicates, variables and rules.

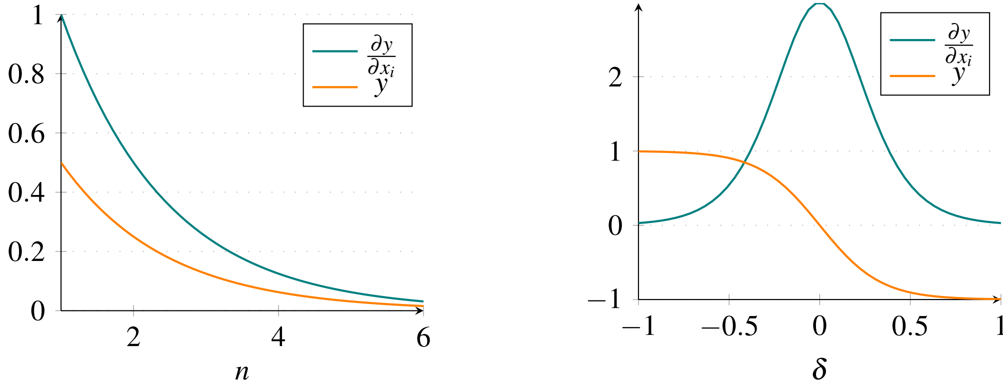
We are interested in end-to-end trainable architectures that propose a holistic neural network to perform logical computation using discrete operations on top of continuous input. This motivation aligns with and stems from the neuro-symbolic account of the human brain as one single neural system capable of symbolic thought over non-symbolic stimuli. Hence, we focus on the question of whether an end-to-end neural network can learn objects, relations and rule-based reasoning going from pixels in an image all the way to logic-based rules. Inspired by the perceptron algorithm [175], we define an inductive bias to an otherwise regular feed-forward neural network in order to enable end-to-end neuro-symbolic learning. The contributions presented in this chapter are based on “pix2rule: End-to-end Neuro-symbolic Rule Learning” [34] and include:

1. A novel way of constraining the bias of a linear layer, called the semi-symbolic layer, in order to obtain the operational semantics of logical $\wedge$ and $\vee$ Boolean connectives. The semi-symbolic layers encourage the desired logical behaviour which might not otherwise emerge via plain linear layers.

2. Combining semi-symbolic layers to construct differentiable multi-layer neural networks that represent the disjunctive normal form (DNF) of real-valued propositional atoms and to use the resulting DNF network not only as a standalone model but also as part of a larger image classification network.

3. A method to prune and threshold the weights of semi-symbolic layers after training to extract symbolic rules.

We evaluate our approach in a controlled environment with two synthetic datasets alongside the analysis of the learnt objects, relations and rules. Our implementation using TensorFlow [1] of the models, experiments and data processing is publicly available online from the corresponding published work cited in Section 1.5.



(a) The output and partial derivative of the product t-norm with increasing number of inputs n where $y = \prod_i x_i$ and $x_i = 0.5$. As n increases, the magnitude of the output and the derivative diminishes.

(b) Visualisation of the semi-symbolic layer with 2 inputs $x_1 = 1, x_2 = -1$ and weights $w_1 = w_2 = 3$. By only adjusting δ, we can obtain either conjunctive $\delta = 1$ or disjunctive semantics $\delta = -1$.

Figure 6.1: Comparison of the product t-norm with vanishing gradients and the semi-symbolic layer which acts similar to a linear layer at $\delta \approx 0$

6.1 Semi-symbolic Layer

In order to learn symbolic rules over real-valued logic, our neuro-symbolic architecture makes use of a differentiable feed-forward layer that behaves like conjunction or disjunction. Given continuous inputs $x_1, x_2, \ldots, x_n \in [\bot, \top]$ where $\bot, \top \in \mathbb{R}$ denote some real-valued constants, we would like to model a layer that can act like conjunction $y = \bigwedge_i x_i$ or disjunction $y = \bigvee_i x_i$.

While t-norms from Definition 1 can model real-valued Boolean connectives, they are not viable for end-to-end neuro-symbolic training due to their tendency towards vanishing gradients [108]. Consider a neural network architecture in which the propositional fuzzy atoms $x_1, x_2, \ldots, x_n$ that we would like to compute the conjunction of are obtained from a randomly initialised upstream convolutional layer with the sigmoid (σ) activation. The sigmoid activation is commonly used to *squash* the values to the standard $[\bot = 0, \top = 1]$ range for fuzzy logic. At initialisation we expect $x_1, x_2, \ldots, x_n \approx 0.5 = \sigma(0)$ since we assume the overall input to the convolutional layer is normalised and the weights are randomly initialised, i.e. prior to any training the convolutional layer outputs unknown $U = 0.5$ values for every propositional atom. Fig. 6.1a demonstrates the vanishing gradient effect for the product t-norm where $y = \prod_i x_i$. As the number of propositional atoms n increases, the output value y and the partial derivative with respect to an input $\frac{\partial y}{\partial x_i}$ vanishes since only values strictly between 0 and 1 are multiplied. The gradients of other t-norms that employ minimum or maximum operators also suffer from gradient cut offs which render the training of any upstream components difficult. Despite this property of t-norms, they are used in existing neuro-symbolic architectures and may work for smaller input sizes [52, 130, 133, 154, 186].

The phenomenon of vanishing gradients occurs due to the 1 out of n failure or success characteristics of conjunction and disjunction respectively rather than the selection of a particular t-norm. If the output of a real-valued conjunction is false ($\approx \bot$), then one cannot reliably determine which input was the culprit as the number of inputs that are false increases regardless of how the output is computed. Hence, modelling directly conjunction or disjunction at the beginning of a randomly initialised neuro-symbolic network may struggle to learn the propositional atoms, particularly as

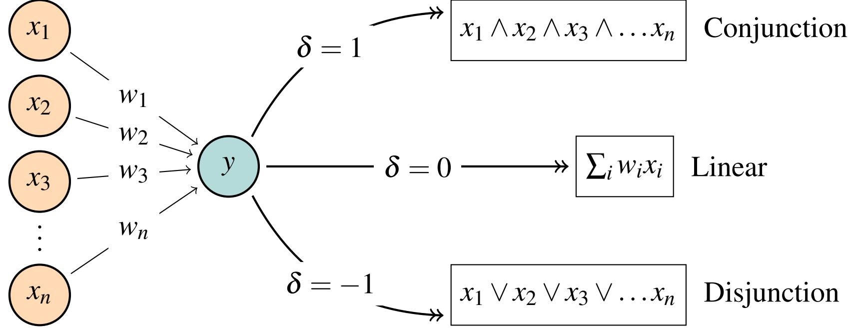


Figure 6.2: Graphical overview of the semi-symbolic layer

part of a larger network with upstream modules. As a result, we are interested in an operation that does starve gradients immediately but *eventually* converges to the desired semantics.

Building on the linearly separable nature of the Boolean connectives $\wedge$ and $\vee$ shown in Fig. 2.4, we extend the linear layer from Definition 6 to propose the semi-symbolic layer (SL):

$$y = f\left(\sum_i w_i x_i + \beta\right) \tag{6.1}$$

$$\beta = \delta\left(\max_i |w_i| - \sum_i |w_i|\right) \tag{6.2}$$

where w_i are the learnable layer weights, f the non-linear activation function and $\delta \in [-1, 1]$ the semantic gate selector. In this formulation, we set $\bot = -1$, $\top = 1$ and f to be the hyperbolic tangent function (tanh). While Eq. (6.1) is the standard feed-forward layer, by adjusting the bias β we can obtain either conjunctive ($\delta = 1$) or disjunctive ($\delta = -1$) semantics. A graphical overview of the semi-symbolic layer is shown in Fig. 6.2. Intuitively, in the conjunctive case, we are looking for a threshold that is at least as small as the sum of the weights but not any bigger than the maximum weight such that if at least one input is false, the output will be false too. For a more detailed derivation and an example implementation, please refer to Appendix B.1.

The weights represent how much an atom is part of the output conjunction or disjunction. When there are no input connections, i.e. $\forall_i w_i = 0$, the output becomes zero and when there is only one input the bias becomes $\beta = 0$ yielding a pass-through, identity gate. A natural interpretation of the sign of the weights is whether the input or its negation is contributing to the output. Thus, logical negation can be regarded as the multiplicative inverse of the input $\neg x_i = -x_i$ which organically works with the weights of the layer. Finally, the gradient profile of the semi-symbolic layer with respect to the inputs $x_1, x_2, \ldots, x_n$ is similar to that of a linear layer as only the bias is modified:

$$\frac{\partial y}{\partial x_i} = \left(1 - \tanh^2\left(\sum_i w_i x_i + \beta\right)\right) w_i \tag{6.3}$$

and is identical when $\delta = 0$. Hence, by gradually adjusting δ from 0 to either 1 or -1, we can

now maintain a feed-forward linear layer that does not initially starve gradients to upstream layers but eventually approximates the desired semantics. Fig. 6.1b shows an example case for two inputs $x_1 = 1, x_2 = -1$ and weights $w_1 = w_2 = 3$. As δ changes, the output of the layer y swings between $\wedge$ and $\vee$ smoothly with the partial derivative of either input reflecting the asymptotes of the hyperbolic tangent function (refer back to Fig. 2.6b for the derivatives of activation functions).

For symbolic inputs $x_i \in \{\bot, \top\}$, $\delta \in \{1, -1\}$ and sufficiently large weights (sufficient so as to saturate tanh), we achieve the proper desired logical behaviour for $\wedge$ and $\vee$ matching Table 2.1. The output of the semi-symbolic layer in this case aligns with the many-valued Łukasiewicz logic [129]. For example, consider the conjunction ($\delta = 1$) of two inputs x_1, x_2 with equal weights $\forall_i w_i = w$. The semi-symbolic layer computes $y = \tanh(wx_1 + wx_2 - w)$ of which the inner term $w(x_1 + x_2 - 1)$ resembles the strong Łukasiewicz conjunction t-norm defined as $y = \max(0, x_1 + x_2 - 1)$ for $x_1, x_2 \in [0, 1]$. Subsequently, when both the inputs are unknown $\forall_i x_i = 0$, the output of the semi-symbolic layer will be negative due to the bias from Eq. (6.2) implying that the conjunction of unknown inputs is false similar to the Łukasiewicz t-norm. In practice, when the input is a vector of learnt features from an upstream layer, our model has the chance to resolve unknown inputs and gradually attain the aforementioned semantics.

Prune & Threshold The forward pass of the semi-symbolic layer in Eq. (6.1) does not place any constraints on the weights. Hence, during training, particularly when $\delta \approx 0$, they can attain values of various magnitudes. In order to obtain symbolic formulas, we prune and then threshold the weights after training. Similar to decision tree pruning methods [50], each weight is set to zero $w_i = 0$ one at a time and pruned if the performance has not dropped by a fixed value ε. This process effectively scans to determine which atoms should be part of the conjunction or disjunction. Then a threshold value is picked by sweeping over potential values $[\min |w_i|, \max |w_i|]$ with a similar performance check to pruning. Each weight is then set to $w_i' = 6\,\text{sign}(w_i)$ which gives sufficient saturation for the hyperbolic tangent function $\tanh(6) \approx 0.999$. Finally, we repeat the pruning step to remove any wrongly amplified weights. The resulting layer weights directly correspond to conjunction or disjunction of propositional input atoms. We opt for this simple post-training regime rather than regularising the weights, for example by using l-norms [67], to limit the number of modifications required to train the semi-symbolic layer.

6.2 Datasets

In order to evaluate the semi-symbolic layer using both symbolic and continuous inputs, we utilise two synthetic datasets: subgraph set isomorphism and image classification. While the former is designed to test the scalability of the semi-symbolic layer, the latter provides the opportunity to process images into objects, relations and rules. For both datasets, the models are required to predict a true or false label. Samples from each dataset are shown in Fig. 6.3 with further examples in Appendix B.2. At this stage, we opt to use synthetic datasets with known and controlled parameters to avoid any inherent biases that may arise in real-world datasets.

Subgraph Set Isomorphism Since modern deep learning architectures can produce large latent spaces such as the 64 dimensional vector space used to represent logical constructs in Chapter 4,

(a) Sample target rules for the easy difficulty of the graph dataset. For each predicate, the last argument denotes the relation id.

(b) Example images from the Relations Game dataset with the top and bottom rows showing correct and incorrect cases respectively

Figure 6.3: Samples from datasets used to evaluate the semi-symbolic layer

we need to understand whether our approach can robustly learn symbolic rules in a scalable fashion. By scalable, we mean in the number of inputs to a logical operation such as conjunction as highlighted as a potential issue in Fig. 6.1a. The subgraph set isomorphism task from Section 3.3 offers an unbiased combinatorial search space for rule learning where the length of the body of the rules to be learnt can be adjusted. Formally, given a graph $\mathscr{G}$ and a set of graphs $\mathbb{H} = \{\mathscr{H}_1, \mathscr{H}_2, \dots, \mathscr{H}_n\}$, the models are required to predict whether $t \leftrightarrow \exists \mathscr{L} \exists i (\mathscr{L} \subseteq \mathscr{G} \land \mathscr{L} \simeq \mathscr{H}_i)$ holds where $\subseteq$ and $\simeq$ mean subgraph of and isomorphic to respectively. The objective is then to learn a suitable $\mathbb{H}$ for a fixed n which is cast as an instance of Inductive Logic Programming. Fig. 6.3a shows a sample from the easy version of the dataset and the correspondence between the graph and the learnable logic programming representations.

Relations Game The Relations Game dataset consists of an input image with different shapes and colours exhibiting compound relations between them. It has been introduced to evaluate object-based relational learning in deep neural networks [191]. Despite their ability to correctly predict the labels, existing methods fail to provide a coherent object, relation and symbolic rule learning in an end-to-end fashion. Hence, we use this dataset to demonstrate a novel holistic neuro-symbolic architecture that is capable of learning object representations, relations between them and logical rules by leveraging semi-symbolic layers described in Section 6.1. Along with the four tasks shown in Fig. 6.3b, we create an *All* multi-task setting and provide the task id as additional input. While the training set contains only pentominoes, shapes with 5 uniformly coloured pixels, the test sets expand to hexominoes (shapes with 6 pixels) as well as striped shapes with unseen colours, shown in Fig. B.1, to evaluate out-of-distribution generalisation.

The Relations Game dataset contains 250k images per set and to ascertain if neuro-symbolic approaches could be more data efficient, we take 100, 1k and 5k for training and 1k examples for validation and test splits. The original resolution of the images are 36x36x3. At this higher resolution, each *block* of the shapes occupies 3x3 pixels of the same colour and since this is redundant, we convert each block to a single pixel reducing the size of images without any loss of information to 12x12x3. This step also reduces the amount computing power required to train the models and allows the experiments as well as the dataset to be more readily distributable on commodity hardware. Finally, we also apply standard data augmentation during training with random horizontal or vertical flips and 90 degree rotations with an added input noise drawn from $\mathscr{N}(0, 0.01)$ in order to counteract over-fitting [67].

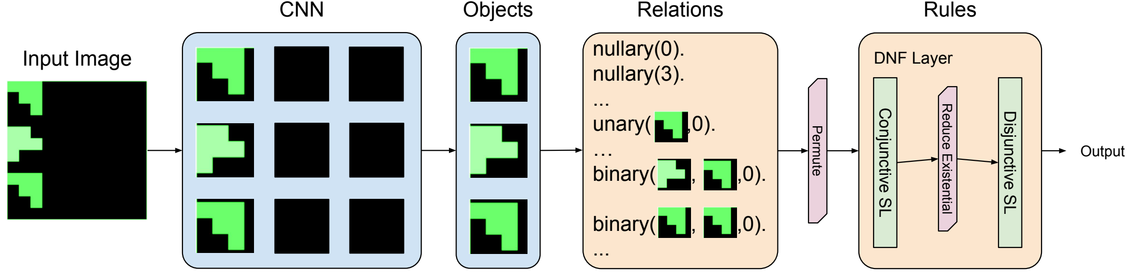


Figure 6.4: Graphical overview of the full neuro-symbolic model used for the Relations Game dataset. The image is processed as a 3x3 grid using a CNN to obtain representations for each patch. Then a subset of those patches are recognised as relevant objects using an MLP. All relations between objects are computed using another MLP with shared weights. Every permutation is constructed to handle variable binding before passed on to the DNF layer to learn rules and predict the final output. Blue squares indicate distributional vector representations while the orange squares highlight real-valued logic.

6.3 DNF Layer & Model

The semi-symbolic layer only offers means to model conjunction or disjunction along with negation of propositional atoms. In order to learn logical rules directly from pixels, we require a larger neural network of which semi-symbolic layers are part of. In this section, we describe the novel disjunctive normal form (DNF) layer and the overall end-to-end differentiable neuro-symbolic architecture that uses it, the DNF model. Fig. 6.4 shows a graphical overview going from an image as input to rules. Table B.1 summarises the symbols used in this section.

Convolutional Layer To process an input image from the Relations Game dataset, $X \in \mathbb{R}^{12\times12\times3}$, we utilise a single convolutional layer with a kernel size of 4x4, stride 4 with 32 filters and ReLU activation. This step produces an intermediate output tensor of shape 3x3x32, i.e. each patch in Fig. 6.4 is represented with a real-valued vector of size 32. We choose this configuration to align the kernel to the grid cells of the image in which the objects are contained such that there is a direct correspondence between the representations and the objects. We leave further state-of-the-art scene parsing and object detection techniques [125, 26, 15] as future work to keep our model size small and focus on neuro-symbolic rule learning rather than object detection. For each patch, we append its coordinates using a linear scale from 0 to 1 and flatten the grid to obtain our set of objects $\boldsymbol{O} \in \mathbb{R}^{9\times36}$:

$$\boldsymbol{O} = \text{flatten}(\text{CNN}_{\text{input}}(X)) \tag{6.4}$$

Object Selection Since semi-symbolic layers work atop propositional atoms, any object based first-order logical representation must be grounded, i.e. every object with every predicate (relation) must be computed. This raises a practical issue regarding computational resources such as time and memory. In order to reduce the number of objects such that our model learns to ignore blank image patches (6 out of 9 in Fig. 6.4), we implement a score based object selection step. Formally, given a set of objects $\boldsymbol{O} \in \mathbb{R}^{n\times d}$ where n is the number of objects and d the embedding size ($n = 9, d = 36$ for Relations Game images), the layer learns to select m relevant objects. The

number of objects to select m is fixed and determined by the tasks shown in Fig. 6.3b: 2 for Same, 3 for Between and 4 for the remaining tasks. The relevance score $\boldsymbol{s}^0 \in \mathbb{R}^n$ of each object is learnt using a single linear layer and a novel *iterative score inversion* is applied:

$$\boldsymbol{s}^0 = \boldsymbol{W}\boldsymbol{O} + \boldsymbol{b} \tag{6.5}$$

$$\boldsymbol{a}^t = \text{Gumbel-Softmax}(\boldsymbol{s}^t) \tag{6.6}$$

$$\boldsymbol{s}^{t+1} = \boldsymbol{a}^t(\boldsymbol{s}^t - c) + (1 - \boldsymbol{a}^t)\boldsymbol{s}^t \tag{6.7}$$

where $\boldsymbol{a}^t \in \mathbb{R}^n$ is a sample from the Gumbel-Softmax [97], also known as the Concrete [128], distribution. We use $c = 100$ in Eq. (6.7) to create a sufficiently large score inversion such that the probability of re-selecting an object becomes really low. By iterating $t = m$ times, we obtain m many objects $\boldsymbol{O}^* \in \mathbb{R}^{m \times d}$ which we collect using the attention maps produced at each time step:

$$\boldsymbol{O}^*_{t,:} = \boldsymbol{O}^T \boldsymbol{a}^t \tag{6.8}$$

Since the object selection layer is fully differentiable, we allow the model to learn which objects to attend to during training. Similar to previous works that use Gumbel-Softmax as a differentiable categorical distribution [7], we anneal the temperature of the distribution starting from 0.5 down to 0.01 with an exponential rate of 0.9 every step after 20 epochs. This allows the model to gradually learn and then sharpen the attention maps from Eq. (6.6), i.e. $\max_i a^t_i \approx 1$. At a temperature of 0.01, the selection becomes a one-hot vector yielding a clear correspondence between the selected objects and the downstream modules that use them. This is in contrast with soft attention [12, 30] which allows continuous mixtures of image patches as a *single* selected object similar to how multiple rules are selected by the Iterative Memory Attention model presented in Chapter 4 and discussed in Section 4.4.

Object Relations So far the representations of objects in $\boldsymbol{O}^*$ are real-valued vectors. However, to transition into a logic programming representation, we need to introduce predicates and compute their truth values. We focus on two predicates to capture object properties and relations between them: *unary(object, property id)* and *binary(object, object, relation id)* and compute them using a single linear layer with tanh activation:

$$\text{unary}(X, i) = \tanh\left(W_i \boldsymbol{x} + b_i\right) \tag{6.9}$$

$$\text{binary}(X, Y, j) = \tanh\left(W_j[\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{x} - \boldsymbol{y}] + b_j\right) \tag{6.10}$$

where W and b are different learnable parameters for each equation and i, j are unique property and relation ids. The notation transitions on the left hand side from logic variables X, Y to vectors $\boldsymbol{x}, \boldsymbol{y}$ corresponding to the continuous, real-valued object representations. This boundary is in some sense the border between neuro and symbolic worlds where the use of tanh yields a balanced

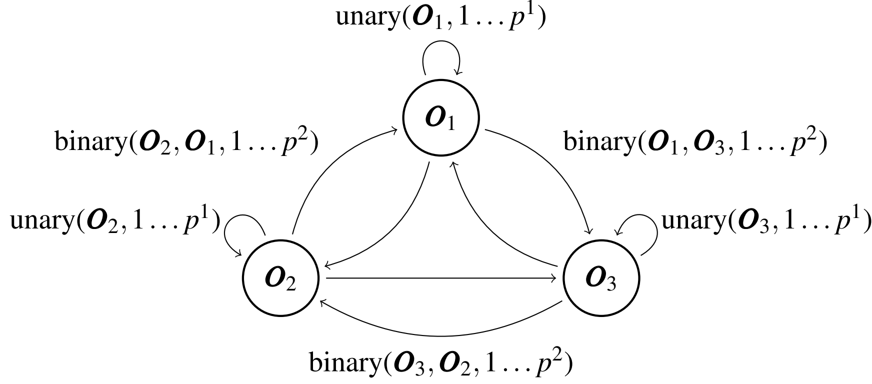


Figure 6.5: An example graphical visualisation of the unary/2 and binary/3 predicates over a set of objects. The unary and binary predicates correspond to self and internode edges and are grounded for every $1 \dots p^1$ and $1 \dots p^2$ many unique relations respectively.

ternary truth value required by the semi-symbolic layers. We take every object and predicate combination to produce the full set of grounded facts $P^1 \in [-1,1]^{m \times p^1}$ and $P^2 \in [-1,1]^{m \times (m-1) \times p^2}$ where m is the number of objects, p^1 and p^2 are the number of unary and binary predicates ids. Since the predicates are learnable, we omit binary relations grounded with the same objects binary(X,X,_) and assume it would be captured instead by unary(X,_). We set the number of learnable predicates to $p^1 = 8$ and $p^2 = 16$ for the Relations Game task and more hyper-parameter details are available in Appendix B.3. We opt for this single linear layer formulation to keep the model computationally less intensive and again focus on downstream differentiable rule learning. We leave more complex and expressive relational layers that compute interactions between dense entity vectors such as Neural Tensor Networks [200] as future work.

The Herbrand base obtained from the combined set of facts using P^1 and P^2 is:

$$P^* = \left[\text{flatten}\left(P^1\right), \text{flatten}\left(P^2\right)\right] \tag{6.11}$$

$P^* \in \mathbb{R}^{mp^1 + m(m-1)p^2}$ can be viewed as a graph where the objects correspond to the vertices and predicates to the edges. Fig. 6.5 shows a visual interpretation of the grounded facts using the unary/2 and binary/3 predicates. This graphical perspective of logic is common in knowledge base completion tasks [38, 186] and brings the logic programming representation of the image processing task in line with that of the subgraph set isomorphism task described in Section 6.2. We can extend this set of ground facts with predicates of different arities such as *nullary(global property id)* (denoted as a vector $P^0 \in [-1,1]^{p^0}$) which graphically corresponds to global graph properties as shown in Fig. 6.3a and described further in Section 3.3. To keep the formalisation simpler, we only focus on the unary and binary predicates in the remainder of this section.

DNF Layer While a single semi-symbolic layer cannot fully represent a logic program, stacking them yields enough expressive power to represent logical rules and therefore logic programs. Recall from Section 2.1.1 that one only needs a functionally complete set of connectives such as $\{\neg, \wedge\}$ to express all possible Boolean connectives. In this case, we have the functionally complete set $\{\neg, \wedge, \vee\}$ which are the connectives needed for any logical formula in conjunctive or

Table 6.1: Example permutation of $m = 2$ constants (objects) $\{a, b\}$ and $|\mathbb{V}| = 2$ variables with $p^1 = 1$ unary and $p^2 = 1$ binary predicates. P^* denotes the Herbrand base, $V^*_:$ the ungrounded atoms and V^*_k the kth permutation.

P^*	$V^*_:$	$V^*_0(X = a, Y = b)$	$V^*_1(X = b, Y = a)$
unary(a, 0)	unary(X, 0)	unary(a, 0)	unary(b, 0)
unary(b, 0)	unary(Y, 0)	unary(b, 0)	unary(a, 0)
binary(a, b, 0)	binary(X, Y, 0)	binary(a, b, 0)	binary(b, a, 0)
binary(b, a, 0)	binary(Y, X, 0)	binary(b, a, 0)	binary(a, b, 0)

disjunctive normal form. We opt to use the disjunctive normal form due to the logical equivalence $p \leftarrow q \equiv \neg q \vee p$ and stack two semi-symbolic layers:

$$\mathrm{DNF}(P^*) = \mathrm{SL}_{\delta \to -1} \circ \mathrm{SL}_{\delta \to 1}(P^*) \tag{6.12}$$

in conjunctive and disjunctive order respectively to obtain the DNF layer in the propositional case. This formulation, however, may not strictly be in DNF since the disjunctive SL may also contain negations of its disjuncts. For example, it may represent $\neg(\neg p \wedge q)$ which can be further simplified to $p \vee \neg q$. One can restrict the weights of the disjunctive SL to be positive but at this stage we refrain from adding constraints to remain as close as possible to the linear layer in Eq. (6.1).

In order to learn first-order rules, we start from a set of variables $\mathbb{V}$ with size $|\mathbb{V}| = l$ as input and ground the rule represented by the DNF layer with every possible constant (object) present in P^*. There are in total $k = \frac{m!}{(m-l)!}$ many permutations for l and m many variables and objects respectively with $l \leq m$. We denote them using $V^* \in [-1, 1]^{k \times (lp^1 + l(l-1)p^2)}$ such that $V^* = \mathrm{permute}(P^*)$. An example with $m = 2$ objects and $l = 2$ variables is shown in Table 6.1 where V^*_k denotes the kth permutation. This permutation step effectively curates all possible object to variable bindings to render the DNF layer invariant to the ordering of objects in P^*. This invariance matches the set like semantics of the input facts P^*, i.e. unary and binary predicates grounded using the unsorted collection of selected objects $\boldsymbol{O}^*$ from Eq. (6.4).

For an arbitrary fixed ordering of the variable set $\mathbb{V}$, the application of the DNF layer over the permuted objects V^* is:

$$\mathrm{DNF}(V^*) = \mathrm{SL}_{\delta \to -1}\left(\max_{\mathbb{V}_{j:l}} \mathrm{SL}_{\delta \to 1}(V^*)\right) \tag{6.13}$$

1. The conjunctive SL takes the permuted ground facts V^*_k as input and computes h many conjunctions, i.e. $\mathrm{SL}_{\delta \to 1}(V^*) \in \mathbb{R}^{k \times h}$ which in return become potential body conditions of the rules. The weight matrix in this case would be $\boldsymbol{W} \in \mathbb{R}^{|V^*_k| \times h}$ and $W_{0,0}$ would correspond to whether unary$(\mathbb{V}_0, 0)$ is in ($W_{0,0} > 0$), not in ($W_{0,0} \approx 0$) or its negation ($W_{0,0} < 0$) is in the first possible rule body.

2. The number of variables in the head $\mathbb{V}_{i:j}$ and in the body $\mathbb{V}_{j:l}$, i.e. those that are existential, are given as hyper-parameters. We can construct rules with no variables in the head (all existential) or learn rules of a certain arity. Since all permutations are computed in V^*, the

ordering of variables does not matter. For example, a rule of the form $p(X) \leftarrow q(X,Y)$ is equivalent to $p(Y) \leftarrow q(Y,X)$. We set the arity of the rules to be learnt based on the task; for example for binary label prediction we learn nullary predicates.

3. To reduce the existential variables $\mathbb{V}_{j:l}$ (from variable j to l) that may occur in the body of the rules, we use the max operator over the corresponding permutations in V^*. As $k = \frac{m!}{(m-l)!}$ yields a tensor of shape $m \times (m-1) \times \ldots \times (m-l+1)$, the existential reduction is performed over the last j to l dimensions of the expanded permutation dimensions. For example, for $m=3$ and $l=2$ with $j=1$, we would get a permutation tensor V^* of shape $\big((3 \times (3-1)) \times (2p^1 + 2(2-1)p^2)\big)$ where k is now expanded and take the maximum over the second (3-1) dimension.

4. Of the h potential body conditions, the disjunctive SL learns which of them form the final rules. Similar to the conjunctive case, each body condition is part, not part or its negation is part of the rule. For a learnable weight matrix $\boldsymbol{W} \in \mathbb{R}^{h \times r}$, $W_{0,0}$ would denote whether first output rule is defined using the first possible body condition.

The total number of learnable parameters of the semi-symbolic layers in the DNF layer does not depend on the number of permutations which may be quite large. The factorial nature of the permutations is a limitation of our approach but the evaluation of the DNF layer over these k many permutations can be computed in parallel benefiting from the increasing parallelism available in modern computing hardware such as the Tensor Processing Units [99]. As encouraged by the object selection step, the actual number of objects that is necessary to reason with for solving a task, we speculate, will be limited in practice.

While, in principle, different mixtures of semi-symbolic layers can extend the space of possible logic programs to a larger language including constraints and predicates with arity greater than two, we focus on learning first-order (recursive) normal logic programs without function symbols and excluding constraints. We also work with the nullary/1, unary/2 and binary/3 predicates of which the last arguments are the property or relations ids since any logic program with predicates of arity up to 2 can be converted to a program using this signature. In cases where the desired rules require specific constants as literal arguments, the instantiations could be restricted at the permutation step to adjust the hypothesis space accordingly.

Image Reconstruction To investigate whether reconstructing the input image as an auxiliary self-supervised learning signal would aid neuro-symbolic rule learning, we optionally create a separate image output pathway from the selected objects $\boldsymbol{O}^*$. For the Relations Game dataset, we spatially broadcast the selected objects back into a 3×3 grid tensor of shape $m \times 3 \times 3 \times 36$. Following the recent state-of-the-art scene parsing Slot Attention model [125], we use two deconvolution layers, also known as transposed convolution, with 32 filters, kernel size 5, ReLU activation and a stride of 2 to expand the tensor to $m \times 12 \times 12 \times 36$. As the final layer, we apply another deconvolution layer with 4 filters, kernel size 5 and a stride of 1 yielding an output reconstruction tensor $C \in \mathbb{R}^{m \times 12 \times 12 \times 4}$. The first three colour channels are combined with the last masking channel to obtain the final input reconstruction $\hat{X}$:

$$\hat{X} = \sum_{i}^{m} \text{softmax}\left(C_{i,:,:,4}\right) C_{i,:,:,1:3} \tag{6.14}$$

and is trained using the mean squared error (Eq. (2.11)). Reconstructing the image might encourage not only the object representations of the input CNN to include all the relevant information but also the object selection to select all the object patches. However, in practice this may not necessary lead to better rule learning on the DNF path as we demonstrate in the next section.

6.4 Experiments

We evaluate whether semi-symbolic layers and consequently the DNF layer can learn symbolic rules in a differentiable manner. We also test their scalability and performance in the presence of input noise. We train all deep models using Adam [106] with a fixed learning rate of 0.001 and the negative log-likelihood loss, on an Intel Core i7 CPU and report the median results of a 5-fold cross-validation regime to avoid outliers. The training consists of a fixed number of batch updates, and evaluation metrics such as accuracy are reported every 200 batch updates. Information about all the hyper-parameters of the experiments are given in Appendix B.3, and further results with supplementary figures and tables can be found in Appendix B.5.

First, we focus on rule learning over symbolic inputs with the subgraph set isomorphism task and compare against state-of-the-art symbolic rule learners. As input, the models receive a set of ground facts P^* describing a single graph and output a prediction whether it is subgraph isomorphic to any graph in the hidden set $\mathbb{H}$ (i.e. t holds or not) which is fixed in advance per fold. The objective is to learn the rules (the fixed set of graphs) from which the samples are drawn. Since the inputs are ground facts, we use a single DNF layer as a standalone model with $r = 1$ unique predicate to learn. In this case, the predicate we desire is t, the target condition that renders input graphs isomorphic or not. The size of the conjunctive SL h is the maximum number of possible rules to learn $|\mathbb{H}|$ set by the level of difficulty shown in Table 3.7. We use a batch size of 128 and train for 10k batch updates.

Table 6.2: Median test results for the subgraph set isomorphism dataset without input noise with median absolute deviation. Although ILASP and FastLAS solve the easy set, they timeout at higher levels of difficulty.

	Test Accuracy			Training Time		
Difficulty	Easy	Medium	Hard	Easy	Medium	Hard
DNF	1.0±0.0	1.00±0.0	1.00±0.00	127.13±4.17	136.90±10.00	129.56± 5.77
DNF+t	1.0±0.0	0.99±0.0	0.99±0.01	125.02±6.53	135.67± 8.56	143.67±22.60
FastLASv3	1.0±0.0	-	-	29.85±0.69	-	-
ILASP-2i	1.0±0.0	-	-	3336.00±994.45	-	-

Can the DNF layer learn symbolic rules in a scalable, differentiable manner? The results with continuous weights (DNF) and with pruning, thresholding (DNF+t) are compared against two state-of-the-art symbolic learners: ILASP [114] (2i as the recommended version for non-noisy tasks) and the recent more scalable FastLAS [115]. We also considered state-of-the-art rule mining

system AMIE [111] but it does not support normal clauses which are required by this problem. Table 6.2 shows that the DNF layer scales better than symbolic learners achieving ≥ 0.99 accuracy against increasing levels of difficulty, and maintains a steady training time because it is trained for a fixed number of iterations. Since DNF+t is thresholded, we have an exact correspondence to the underlying Boolean algebra and thus the logic program. We can convert the thresholded weights into ASP rules and test them using clingo [61] to verify that our approach has indeed learnt correct symbolic rules in a differentiable manner. We were only able to run the symbolic learners for the easy difficulty because they do not report progress or allow checkpoints making them infeasible for distributed shared computing infrastructure. FastLAS on an isolated machine did not terminate after 16 hours on the medium difficulty of which the specific instance is shown in Table 3.8. The full set of results which includes every run of every model can be found in Appendix B.5.

Table 6.3: Median test accuracies for the best out of 5 runs with input noise. The median absolute deviation is less than 0.09 for all entries.

Difficulty	Easy			Medium			Hard		
Noise	0.00	0.15	0.30	0.00	0.15	0.30	0.00	0.15	0.30
DNF	1.00	0.89	0.82	1.00	0.98	0.89	1.00	0.98	0.86
DNF+t	1.00	1.00	0.84	0.99	0.99	0.99	0.99	0.99	0.74

How does the DNF layer cope with input noise? Latent representations of upstream neural networks such as the embeddings of image patches $\boldsymbol{O}$ from Eq. (6.4) are likely to be noisy and fluctuate as they are trained. To ascertain whether the DNF layer is robust against input noise before it is combined with the larger image processing upstream pipeline in Fig. 6.4, we perturb the subgraph set isomorphism dataset by randomly flipping the truth values of the edges of the given input graph $E(\mathcal{G})$ (which correspond to nullary/1, unary/2 and binary/3 predicates) in the training set with a fixed probability as shown in Table 6.3. This added noise forces the DNF layer to approximate the rules since it becomes less likely that the subgraphs will be exactly isomorphic. Firstly, we observe that the DNF layer performs above ≈ 0.9 up to 0.3 where the median accuracy drops below 0.9 suggesting it can cope with some input noise. Secondly, the accuracy for the same input noise at higher difficulties compared to the easy difficulty increases from 0.89 to 0.98 and from 0.82 to 0.86 most likely because more rules together are more robust to noise. Finally, the pruning and thresholding steps seem to improve the performance for lower levels of noise, likely because incorrect weights are removed against a smaller but noise free validation set.

We now focus on the Relations Game image classification task and utilise the full neuro-symbolic architecture shown in Fig. 6.4 and described in Section 6.3 with 3 variations:

- *-i* suffix includes the optional image reconstruction output from Eq. (6.14) as an auxiliary signal to train the object representation and selection components in tandem with the rule learning output. This auxiliary mean squared error is added to the label classification loss.

- *-h* suffix has one extra hidden DNF layer in order to learn new predicates that are part of the induced logic program. In total, we utilise 14: 2 nullary, 4 unary and 8 binary predicates to be learnt along side the desired output label predicate t. The hidden DNF layer is placed before the output DNF layer with its own permutation step. The invented predicates are

Table 6.4: Median test accuracy for the Relations Game tasks with full results in Appendix B.5

Set	Task	All			Between			Occurs			Same			XOccurs		
	Model	100	1000	5000	100	1000	5000	100	1000	5000	100	1000	5000	100	1000	5000
Hex.	DNF	0.94	0.97	0.98	0.90	0.99	0.99	0.56	0.99	0.99	0.94	1.00	1.00	0.49	0.80	0.93
	DNF-h	0.98	0.99	0.99	0.95	1.00	0.99	0.62	0.99	0.99	0.97	1.00	1.00	0.50	0.98	0.96
	DNF-h+t	0.51	0.55	0.92	0.91	0.97	0.98	0.50	0.79	0.96	0.53	1.00	0.98	0.48	0.51	0.51
	DNF-hi	0.98	0.99	1.00	0.97	0.99	1.00	0.69	0.99	0.99	0.97	1.00	1.00	0.51	0.99	0.99
	DNF-r	0.94	0.98	0.98	0.84	1.00	0.99	0.63	0.99	0.99	0.96	1.00	1.00	0.51	0.94	0.51
	PrediNet	0.85	0.95	0.96	0.66	0.99	0.99	0.57	0.95	0.97	0.99	1.00	1.00	0.50	0.58	0.95
Pent.	DNF	0.89	0.96	0.95	0.85	0.99	0.99	0.57	0.96	0.98	0.95	1.00	1.00	0.50	0.74	0.86
	DNF-h	0.95	0.97	0.98	0.92	0.99	0.99	0.62	0.95	0.97	0.94	1.00	1.00	0.51	0.94	0.90
	DNF-h+t	0.50	0.53	0.88	0.90	0.98	0.97	0.49	0.81	0.93	0.50	0.99	0.97	0.50	0.51	0.49
	DNF-hi	0.96	0.99	0.99	0.95	0.99	1.00	0.69	0.98	0.98	0.96	1.00	1.00	0.50	0.96	0.99
	DNF-r	0.93	0.96	0.97	0.81	0.99	0.99	0.65	0.96	0.96	0.95	1.00	1.00	0.52	0.88	0.49
	PrediNet	0.85	0.96	0.95	0.65	0.99	0.98	0.60	0.95	0.97	0.99	1.00	1.00	0.50	0.58	0.95
Stripe	DNF	0.91	0.97	0.95	0.81	0.98	0.99	0.57	0.97	0.99	0.93	0.99	1.00	0.49	0.88	0.94
	DNF-h	0.93	0.98	0.99	0.89	0.99	0.99	0.57	0.97	0.99	0.96	1.00	1.00	0.51	0.98	0.97
	DNF-h+t	0.52	0.53	0.93	0.92	0.97	0.95	0.49	0.84	0.86	0.49	0.99	0.97	0.48	0.50	0.51
	DNF-hi	0.95	0.99	0.99	0.94	0.99	0.99	0.63	0.96	0.99	0.97	1.00	1.00	0.49	0.96	0.97
	DNF-r	0.92	0.97	0.98	0.81	0.99	0.99	0.64	0.95	0.98	0.98	1.00	1.00	0.52	0.92	0.51
	PrediNet	0.84	0.93	0.92	0.64	0.99	0.99	0.54	0.94	0.92	0.99	0.99	1.00	0.51	0.61	0.93

evaluated in parallel using matrix operations similar to a feed-forward network layer with many outputs.

-r suffix iterates the DNF layer of the model twice inducing recursive rules with up to 7 extra predicates: 1 nullary, 2 unary and 4 binary. For the first iteration, the input facts P^* from Eq. (6.11) are padded with zeros representing a truth value of unknown for the extra predicates. Then the DNF layer is iterated twice with the padded values replaced by the computed values at the second iteration. Similar to the hidden suffix, the extra predicates along with the output label predicate t are computed in parallel using matrix operations.

The overall connection patterns of the DNF layers yield the space of possible logic programs similar to the mode biases used in symbolic solvers ILASP and FastLAS. Each DNF layer corresponds to the application and learning of a set of rules. Hence, stacking two DNF layers (as in DNF-h) gives a stratified logic program with two strata while iterating a single DNF layer produces recursive logic programs. One can alter the architecture in order to adjust and adapt the search space to the particular domain demonstrating the modularity and flexibility of our approach.

Is the DNF model more data efficient? Data efficiency in the number of training examples is a desired property of symbolic rule learners in contrast to the data hungry nature of deep artificial neural networks [212]. To investigate whether a neuro-symbolic architecture is more data efficient, we compare our approach against PrediNet [191], an explicitly relational neural network and the current state-of-the-art in the Relations Game dataset. PrediNet is shown to outperform MLP baseline models and Relation Networks [179] both in terms of predictive performance and relational representation learning. Columns with different data sizes in Table 6.4 suggest that the DNF models consistently match or outperform PrediNet, especially DNF-h with training size 100 for the between task (0.95 vs 0.66). This gap eventually narrows as the training size is increased. Despite failing to recognise and reason with four objects in the Occurs and XOccurs tasks with 100 training examples, all the models seem to tackle them better when mixed with other tasks, i.e. the All task. This might suggest that a mixture of tasks is beneficial to learning such that tasks with 2, 3 and 4 objects are presented together. The failure cases with < 0.7 accuracies observed on Occurs and XOccurs tasks using 100 training data points highlight how our approach is still prone to over-fitting due to its neural network based formulation which may be further mitigated using regularisation. Appendix B.4 shows all the training curves for all the models.

Table 6.5: Aggregate median test accuracy for all DNF models with and without image reconstruction loss along with median absolute deviation. Only for XOccurs task do we see an improvement.

Image Reconsturction		Hexos False	Hexos True	Pentos False	Pentos True	Stripes False	Stripes True
All	100	0.94±0.06	0.96±0.05	0.93±0.06	0.94±0.06	0.92±0.07	0.94±0.08
	1000	0.98±0.02	0.99±0.01	0.96±0.03	0.98±0.01	0.98±0.02	0.97±0.01
	5000	0.99±0.04	0.99±0.01	0.97±0.04	0.98±0.01	0.98±0.04	0.98±0.01
Between	100	0.91±0.05	0.93±0.07	0.87±0.05	0.89±0.07	0.86±0.08	0.84±0.07
	1000	0.99±0.01	1.00±0.00	0.99±0.01	0.99±0.00	0.99±0.01	0.99±0.01
	5000	0.99±0.00	1.00±0.00	0.99±0.00	0.99±0.00	0.99±0.01	0.99±0.01
Occurs	100	0.62±0.06	0.63±0.09	0.61±0.06	0.63±0.08	0.57±0.06	0.59±0.08
	1000	0.99±0.01	0.99±0.00	0.96±0.02	0.98±0.01	0.97±0.03	0.97±0.02
	5000	0.99±0.01	0.99±0.00	0.97±0.02	0.98±0.01	0.99±0.01	0.99±0.02
Same	100	0.96±0.02	0.96±0.02	0.94±0.02	0.94±0.02	0.96±0.03	0.96±0.02
	1000	1.00±0.00	1.00±0.00	1.00±0.00	1.00±0.00	1.00±0.00	1.00±0.00
	5000	1.00±0.00	1.00±0.00	1.00±0.00	1.00±0.00	1.00±0.00	1.00±0.00
XOccurs	100	0.50±0.01	0.51±0.02	0.51±0.01	0.51±0.02	0.50±0.01	0.50±0.02
	1000	0.96±0.10	0.98±0.00	0.88±0.10	0.97±0.01	0.94±0.07	0.98±0.01
	5000	0.89±0.17	0.98±0.03	0.82±0.16	0.97±0.03	0.94±0.18	0.96±0.03

Can the DNF model generalise to unseen shapes and colours? After training on pentominoes, the models are tested on unseen hexominoes and striped shapes; Fig. B.1 contains samples from each set. The results for Hex. and Stripe in Table 6.4 are similar to that of pentominoes for all the models. This lack of change might be because of the tasks' requirement to determine whether the shapes match or not as opposed to predicting certain properties such as how many blocks (pixels) a shape consists of. This differential matching might encourage a simple subtraction based representation that could generalise to unseen shapes and colours without adequately representing them. Indeed, PrediNet is pre-engineered to use subtraction as the main relational operator while our model would need to learn that solely from examples using the given features in Eq. (6.10).

Can we extract symbolic rules in an image classification task? The main goal is to create a coherent neuro-symbolic neural network where objects, their relations and the symbolic rules that govern them are learnt in an end-to-end fashion. We take the final weights of the best overall DNF model variant DNF-h (refer to Table B.5 for a comparison) and, as described in Section 6.1, threshold its weights DNF-h+t to get symbolic rules. Comparing rows DNF-h with DNF-h+t in Table 6.4, only if DNF-h achieves ≥ 0.99 accuracy does DNF-h+t consistently yield better than random performance. This highlights the difficulty of learning both the meaning of the predicates and the rules in tandem as well as the fragility of thresholding continuous weights. The DNF models would have two possible avenues for each predicate: adjust the rules that use it or alter the meaning of the predicate. However, in spite of the overall trend, there are some successful outliers ignored by the median which can be found in Appendix B.5 along with the full set of results of all configurations in Table B.6. Since the thresholded weights correspond to exact logic programs, we extract and analyse them further in Section 6.5.

Does image reconstruction improve DNF model performance? Finally, we probe the hypothesis that adding an auxiliary signal to train the object representations would aid downstream rule learning. The main driving force behind this hypothesis is that by reconstructing the image, the object representations $\boldsymbol{O}^*$ from Eq. (6.8) would accommodate their shape, colour and position to

Table 6.6: Example continuous representations of objects that are selected $\boldsymbol{O}^*$ from Eq. (6.8) with an embedding dimension of 32, enumerated from the top left to the bottom right of each row. The last 4 entries are the location coordinates appended based on the object's 3x3 grid location.

Obj 1	0.	3.356	0.728	0.	0.871	1.170	1.726	1.528	1.174	0.117	1.586	0.478
	2.527	0.286	0.724	1.723	0.580	0.	3.770	0.943	0.517	3.114	3.456	0.
	0.710	0.975	2.030	1.570	1.251	1.971	1.830	2.688	0.5	0.5	0.5	0.5
Obj 2	2.624	0.304	0.244	0.	0.	2.959	0.589	3.234	2.745	2.195	0.	2.967
	0.510	2.248	2.414	1.953	2.047	1.409	1.645	2.542	0.	2.825	1.280	0.
	2.709	2.844	0.293	0.349	3.549	0.	0.	0.	1.	1.	0.	0.
Obj 3	2.624	0.304	0.244	0.	0.	2.959	0.589	3.234	2.745	2.195	0.	2.967
	0.510	2.248	2.414	1.953	2.047	1.409	1.645	2.542	0.	2.825	1.280	0.
	2.709	2.844	0.293	0.349	3.549	0.	0.	0.	0.	0.	1.	1.

ease the burden of learning them through the DNF layers. Contrary to this intuition, we do not observe a consistent improvement above 5% despite the extra computation required to reconstruct the images when comparing DNF-hi and DNF-h rows in Table 6.4. We further substantiate this claim with Table 6.5 in which only the XOccurs task alone with at least 1k training data points (last 2 rows) provides an improvement of the median accuracy. Although the representations of objects would need to accommodate both the reasoning and the reconstruction, we believe the simplicity of the shapes might be why the auxiliary loss does not yield any advantage for the rule learning.

6.5 Analysis

In this section, we focus on *how* the downstream symbolic rule learning behaves as an end-to-end differentiable component. Specifically, we probe the symbolic rules learnt by the semi-symbolic layer described in Section 6.1 as well as the rest of the components that make up the overall architecture. Appendix B.5 contains supplementary figures and examples.

What are the object representations? For the Relations Game dataset, we follow the journey of a single successful run (≥ 0.99 test accuracy) of the DNF model on the between task with 1k training examples and image reconstruction (DNF-i). We pick this particular run because the model achieves a very high test accuracy of 0.99 and 0.97 across pentominoes, hexominoes and stripes prior to and after thresholding respectively. Table 6.6 shows the object representations learnt by the input CNN (the first step in Fig. 6.4) with an embedding size of 36 where the last 4 entries correspond to the grid coordinates of the objects. These correspond to the selected objects of the first image shown in Fig. 6.6 where the second and third objects have the same shape and colour. These matching objects have an identical representation as expected with only the coordinates varying, top left vs bottom right. The zeros in the object representations are due to the ReLU activation (see Eq. (2.5)) which cuts off any negative values. These *continuous* real-valued vectors are common in deep neural networks and are one of the reasons why models using them act like black boxes [2]; e.g. it is unclear as to how the colour and shape information is encoded.

What are the selected objects? Unlike soft attention mechanisms such as the key-value based dot-product attention used in PrediNet, our object selection layer approximates a categorical distribution by annealing the temperature of the underlying Gumbel-Softmax distribution. Hence, this ensures the relations and rules downstream have a clear correspondence to the objects that

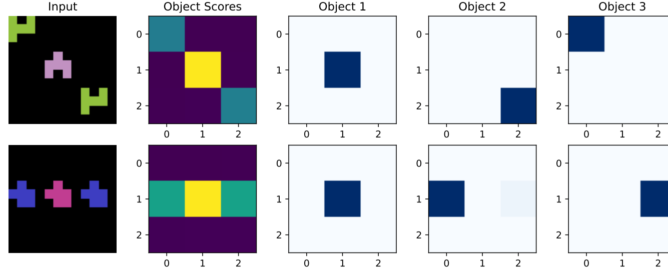


Figure 6.6: Attention maps learnt during the object selection phase. Gumbel-Softmax with a low temperature value (≈ 0.01) approximates hard attention.

Table 6.7: Example pruning steps of the conjunctive SL weights for the standalone DNF layer used in the subgraph set isomorphism easy task with no input noise. These are the weights of the rules shown in Listing 6.1. The weights are rounded two decimal spaces for clarity.

Stage	Weights	Test Acc
Pre-prune	`[ 0.01 -0.07  2.16  0.00 -0.06 -0.06  0.00 -0.00 -2.27  2.23]`	1.0
	`[-1.61  0.02 -1.84 -1.85  0.03  0.06 -1.85 -1.76  1.85  1.85]`	
	`[ 0.98 -0.98 -0.62 -0.75 -0.82  0.83  0.80  0.04 -0.97 -0.67]`	
Prune	`[ 0.00 -0.00  2.16  0.00 -0.00 -0.00  0.00 -0.00 -2.27  2.23]`	1.0
	`[-1.61  0.00 -1.84 -1.85  0.00  0.00 -1.85 -1.76  1.85  1.85]`	
	`[ 0.98 -0.98 -0.62 -0.75 -0.82  0.83  0.80  0.00 -0.97 -0.67]`	
Threshold	`[ 0.00 -0.00  6.00  0.00 -0.00 -0.00  0.00 -0.00 -6.00  6.00]`	1.0
	`[-6.00  0.00 -6.00 -6.00  0.00  0.00 -6.00 -6.00  6.00  6.00]`	
	`[ 6.00 -6.00 -6.00 -6.00 -6.00  6.00  6.00  0.00 -6.00 -6.00]`	
Threshold + Prune	`[ 0.00 -0.00  6.00  0.00 -0.00 -0.00  0.00 -0.00 -6.00  6.00]`	1.0
	`[-6.00  0.00 -6.00 -6.00  0.00  0.00 -6.00 -6.00  6.00  6.00]`	
	`[ 6.00 -6.00 -6.00 -6.00 -6.00  6.00  6.00  0.00 -6.00 -6.00]`	

reside in the image patches. For the between task where $m = 3$ objects are selected, Fig. 6.6 plots the initial relevance scores $\boldsymbol{s}^0$ along with the attention map at each iteration $\boldsymbol{a}^t$ from Eq. (6.6). In both images, the model has learnt to correctly identify by assigning higher scores to and iteratively selecting the image patches with objects. Due to the random sampling, the order of the selected objects may vary with each run despite freezing the weights of the model. The continuous real-valued vector representations of the first row of objects are the ones shown in Table 6.6.

In a successful run, what are the rules learnt? Once the weights are pruned and thresholded, our approach has a one-to-one correspondence with the desired logical operations. Table 6.7 shows the weights of the conjunctive SL (w_i from Eq. (6.1)) of the standalone DNF layer used for the subgraph set isomorphism easy difficulty task without input noise. Recall from Table 3.7 that the easy configuration has 2 nullary, 2 unary, 2 binary with at most $|\mathbb{V}| = 2$ variables and a maximum of 3 rules. This results in an input size of $|V^*| = 2+2\times2+2\times2 = 10$ facts from Eq. (6.11) which correspond to the number of weights in each row following Eq. (6.13). The rows represent the individual rules that define the target predicate t, in this case there are a maximum of 3 definitions. The pre-prune stage contains the final trained values of the weights. Following the procedure described in Section 6.1, the pruning step removes weights without any drop in the test accuracy, most likely due to the noise-free nature of the task. Finally, the threshold stage amplifies the weights to obtain the equivalence between the symbolic Boolean connective $\wedge$ and the computation carried out by the conjunctive SL. The final pruning phase yields the same set of weights meaning that all the

Table 6.8: Example pruning steps of the disjunctive SL weights of the full DNF model with image reconstruction trained on 1k examples. This is the last set of weights that predict the output. We report the test pentominoes accuracy after each step. The accuracy might vary due to stochastic object selection despite having fixed weights, e.g. in the last two rows.

Stage	Weights	Test Pent. Acc
Pre-prune	[1.56 -1.72 7.45 -2.22 -2.27 2.76 -1.46 1.75]	0.996
Prune	[1.56 -1.72 7.45 -0.00 0.00 0.00 -1.46 1.75]	0.991
Threshold	[0.00 -0.00 6.00 0.00 0.00 0.00 -0.00 0.00]	0.942
Threshold + Prune	[0.00 -0.00 6.00 0.00 0.00 0.00 -0.00 0.00]	0.938

weights were correctly thresholded. While it may be more obvious that a weight of magnitude 2.23 (first row, last column) would be more important and therefore contribute more to the final rule definition, the weight with a value of -0.67 (third row, last column) is also required to attain a test accuracy of 1. In other words, there does not seem to be a correlation between the magnitude of the weights and how important they are to the final predictive performance.

```
1 t :- unary(X,0), not binary(Y,X,0), binary(Y,X,1),
2      X != Y, obj(X), obj(Y).
3 t :- not nullary(0), not unary(X,0), not unary(X,1),
4      not binary(X,Y,0), not binary(X,Y,1), binary(Y,X,0),
5      binary(Y,X,1), X != Y, obj(X), obj(Y).
6 t :- nullary(0), not nullary(1), not unary(X,0), not unary(X,1),
7      not unary(Y,0), unary(Y,1), binary(X,Y,0), not binary(Y,X,0),
8      not binary(Y,X,1), X != Y, obj(X), obj(Y).
```

Listing 6.1: Sample rules learnt on the easy set for subgraph set isomorphism dataset. These are thresholded results, i.e. DNF+t model. Random seed is 3. These rules can be passed onto clingo to solve the unseen examples. The obj/1 predicate is added to make the program safe in ASP.

The translation of the final weights from Table 6.7 into a logic program is shown in Listing 6.1. For every remaining weight, either it is included $w_i = 6$ or its negation $w_i = -6$ is included in the rule. The variable inequalities are added for two reasons: (i) graph isomorphism is a bijective mapping such that no two variables will correspond to the same constant and (ii) the permutation step of the DNF layer described in Section 6.3 omits duplicate assignments. While the former is a constraint of the task, the latter is a design choice since any relation of the form `binary(X,X,_)` could be represented using a unary predicate `unary(X,_)`, particularly if the predicates are learnt as in the Relations Game dataset. Finally, the `obj` predicate is added to ensure a safe representation in ASP which is not required for the DNF layer. Despite being learnt using an end-to-end differentiable approach with continuous weights, the rules from Listing 6.1 can be passed along with the test data points to clingo (Section 2.1.5) and clingo attains a test accuracy of 1 substantiating the correspondence of the DNF layer to the Boolean algebra and verifying the symbolic rules.

We perform the same analysis for the full DNF model trained on the between task of the Relations Game dataset. Since the number of latent features of the image classification task is too large to tabulate (8 unary and 16 binary predicates), Table 6.8 shows the pruning steps for the disjunctive

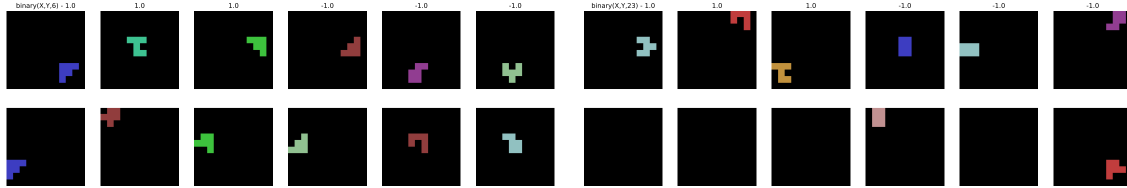
SL weights, which correspond to $h = 8$ possible unique rule definitions. Unlike the subgraph set isomorphism task weights shown in Table 6.7, the test accuracy of the model drops at each step with the biggest drop of 0.05 recorded at the thresholding step. This reduction is expected in an image classification task where the predicates are also learnt since it is an inherently more difficult problem. When the *switch* from continuous to symbolic representation occurs at the thresholding step, we speculate, predictive information from the cut out rules are lost which leads to a drop in accuracy. Coupled with the weights of the conjunctive SL, this situation presents itself as a search problem as to how one can obtain symbolic rules guided by the weights. It may be misleading again to assume that the magnitude of the weights correlate with importance since both -2.22 and 2.76 are pruned with very little degradation in performance.

```
1 between :- unary(X,4), not unary(Y,6), binary(Y,Z,6),
2            not binary(Y,Z,9), not binary(Z,Y,3), binary(Z,Y,5),
3            not binary(Z,Y,9), binary(Z,Y,14), X != Y, X != Z,
4            Y != Z, obj(X), obj(Y), obj(Z).
```

Listing 6.2: Rule learnt by the full DNF model on the between task of the Relations Game dataset. Since the target predicate t corresponds to correctly predicting the between relation in the input image, we alias it for clarity.

The rule that corresponds to the single surviving weight, last row third column in Table 6.8, is shown in Listing 6.2 which achieves a test accuracy of 0.97 for pentominoes and hexominoes and 0.87 for stripes. This is higher than the accuracy of 0.938 recorded after the disjunctive SL was thresholded since when the conjunctive SL weights are also thresholded the performance improves, i.e. first the disjunctive weights are pruned then the conjunctive ones. We opt to perform pruning in that order since the disjunctive SL is further downstream in the overall network and closer to the final prediction so we prune the layers backwards from the final output.

As published in the corresponding paper for this chapter (see Section 1.5), the logic program from Listing 6.2 is the first result we are aware of that presents differentiable rule learning with learnt predicates on continuous representations of images in an end-to-end fashion. To further validate this rule, we threshold the input facts $P^* > 0$ from Eq. (6.11) and pass them along with the rule to clingo. Over a sample test batch of 64 images, clingo solves 91% of them verifying the correctness of the learnt rules by the DNF layer. The extra drop in accuracy is most likely caused by the threshold on the input facts. This phenomenon highlights the struggle of going from continuous real-valued latent representations and weights to a purely discrete symbolic realm as alluded to in Section 1.2. We hypothesise further neural network training techniques such as regularisation might improve pruning and thresholding. For example, adding a sparsity constraint to the loss function similar to Eq. (5.20) might already act as a differentiable form of pruning whereby unused weights are pushed towards zero. The unwanted side-affect in that case would be the diminishing magnitude of all the weights so as to satisfy the constraint as previously analysed in Section 5.5. Future work into differentiable methods for pruning using regularisation and thresholding with weight amplification might yield more robust symbolic rule extraction.



(a) Object arguments (top and bottom rows) that make `binary(X,Y,6)` true or false exhibit no common pattern of the principal concepts of the dataset. The drop in accuracy after removing it is 0.18.

(b) Example truth values for binary(X,Y,23) predicate from the logic program in Listing 6.3. Removing 'not binary(V3,V2,23)' from the logic program results in a drop in accuracy of 0.39.

Figure 6.7: Plots of the learnt predicates that contribute the most to the final prediction

```
% Learnt predicates from the hidden DNF layer
    unary(X,10) :- not c3unary(X,10), obj(X).
  c3unary(X,10) :- not nullary(3), unary(X,3), binary(X,Y,6), not binary(Y,X,1),
                   binary(Y,X,3), binary(Y,X,10), binary(Y,X,14),
                   obj(Y), Y != X, obj(X).
    unary(X,11) :- not c1unary(X,11), obj(X).
  c1unary(X,11) :- not nullary(3), not binary(X,Y,1), not binary(X,Y,9),
                   not binary(X,Y,11), binary(Y,X,10), not binary(Y,X,11),
                   binary(Y,X,13), binary(Y,X,14),
                   obj(Y), Y != X, obj(X).
  binary(X,Y,23) :- not binary(Y,X,13), obj(Y), Y != X, obj(X).
  binary(X,Y,23) :- not c3binary(X,Y,23), obj(X), obj(Y), X != Y.
c3binary(X,Y,23) :- not nullary(0), not binary(X,Y,1), binary(X,Y,3),
                   binary(X,Y,15), not binary(Y,X,1), binary(Y,X,10),
                   not binary(Y,X,11), binary(Y,X,13), binary(Y,X,14),
                   obj(Y), Y != X, obj(X).
% Output DNF layer
                t :- not c4t.
              c4t :- unary(Z,10), unary(Z,11), obj(Z).
                t :- binary(X,Y,23), not binary(V,Z,23), obj(Z),
                     Z != Y, Z != V, Z != X, obj(Y), Y != V,
                     Y != X, obj(V), V != X, obj(X).
```

Listing 6.3: Learnt logic program of a successful run with DNF-hi on all tasks with a training size of 1k. The logic program can be passed to clingo, along with a thresholded interpretation from the neural network to predict the answer. Atoms nullary(0), ..., nullary(3) corresponds to the task ids for Same, Between, Occurs and XOccurs respectively. Since this is a multi-task setting, we observe that the model is utilising the nullary predicates in the logic program.

To highlight the capability of learning larger logic programs with invented predicates, Listing 6.3 shows the set of rules learnt by the DNF model with the extra hidden DNF layer (DNF-hi) with image reconstruction trained on all the Relations Game tasks mixed together (All task as described in Section 6.2). The nullary/1 predicate in this case indicates which task it is, i.e. nullary(0) is true when it is the Same task and so on. This particular logic program achieves a test of accuracy of 0.97 on pentominoes and hexominoes with 0.96 for stripes. Given the amalgamated complexity of the All task, we see a larger logic program with the two strata corresponding to the two DNF layers stacked onto each other. The nullary/1 predicate is used through the first strata most likely to differentiate between the tasks, yet only nullary(0) (Same) and nullary(3) (XOccurs) are used. This behaviour might arise from the fact that the first three tasks Same, Between and Occurs share the common step of recognising whether two objects have the same shape and colour whereas XOccurs requires pairs of comparisons, refer to Figs. 6.3b and B.1 for examples.

What do the learnt predicates mean? Despite obtaining a logic program for classifying a compound relation in an image, we have no idea as to what these predicates mean. To investigate

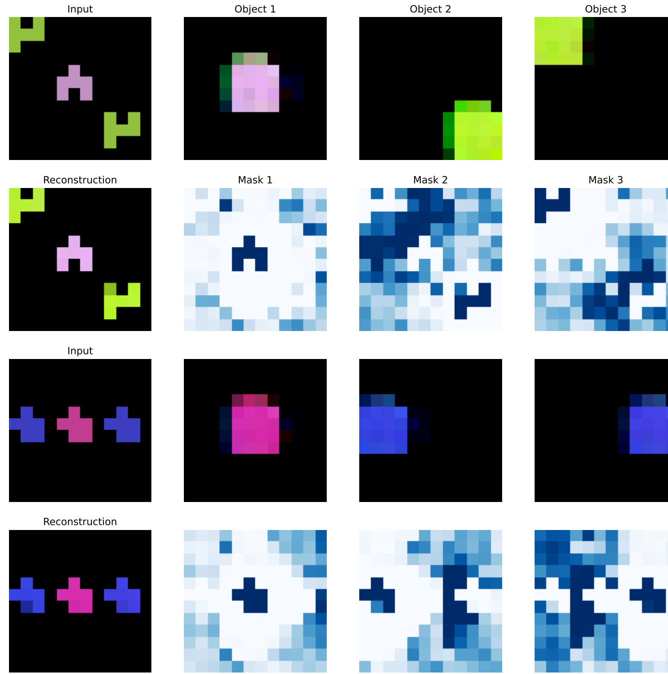


Figure 6.8: Example image reconstructions of the analysed DNF model with image reconstruction. The model seems to paint a colour and use the mask like a cookie-cutter to extract out the shape.

further, we iteratively remove one atom from the body of the rules and check the drop in accuracy over the same sample test batch so as to find the predicate that contributes the most to the final prediction. For the logic program in Listing 6.2, removing binary(X,Y,6) yields the highest drop of 18%. Hence, we plot input image patches that make binary(X,Y,6) true and false in Fig. 6.7a but fail to notice any alignment between the primary concepts of the dataset such as shape, colour or position. Although at first this may seem odd, since there is no regularisation for any sort of disentanglement, we would not expect or assume any correlation between the concepts and the learnt relations beyond any coincidence. This finding contrasts the promise of interpretability using neuro-symbolic methods under the umbrella of explainable AI [2] which seems to be fundamentally limited by our ability to decode the learnt predicates. A similar picture is painted if we perform the same analysis on the larger logic program from Listing 6.3 in which binary(X,Y,23) emerges as the most important predicate with a drop in accuracy of 0.39. Yet, it is difficult to conclude whether Fig. 6.7b shows a common pattern for the meaning of binary(X,Y,23). On the other hand, this phenomenon does not arise when the input predicates are known and fixed such as in the subgraph set isomorphism dataset where the learnt rules have interpretable meanings in the form of graphs as shown in Fig. 6.3a.

How are the images reconstructed? For the DNF-i model trained on the between task of the Relations Game dataset, we also plot its image reconstructions for the same sample images shown in Fig. 6.6. Fig. 6.8 shows the reconstructed outputs $\hat{X}$ from Eq. (6.14) in which the upper row is the RGB reconstruction of the ith object $C_{i,:,:,1:3}$ and the lower row is the corresponding mask $C_{i,:,:,4}$. The reconstruction paints a colour in the correct position and uses the mask as a cookie-cutter to produce the correct shape similar to the Slot Attention [125] network it's based on. The mean absolute error of the reconstructions in the test sets are 0.012, 0.030 and 0.133 for pentominoes, hexominoes and stripes respectively. Figs. B.15 and B.16 show a similar reconstruction

behaviour for the larger DNF-hi model of which the learnt rules are shown in Listing 6.3. These results suggest that the image reconstruction output is within an acceptable performance range both quantitatively with an absolute error bound of ≤ 0.2 and qualitatively despite yielding little consistent improvement in the final label prediction as highlighted in Table 6.5.

6.6 Related Work

Neuro-symbolic architectures attempting to integrate logic with neural networks have a prolific history as outlined in Section 1.3. In this section, we focus on related work that achieves a direct correspondence between the computations carried out by a neural network and some Boolean algebra as done by the novel semi-symbolic layer from Section 6.1. Our approach is related to these works from the following perspective: constrain or make neural networks exhibit behaviours of logical reasoning. That is, to build an architecture with the right biases in order to learn and leverage structured logical reasoning in the form of objects, relations and rules.

While our approach falls into the category of adding architectural biases to neural networks in order to elicit logic inspired behaviour, the application of DNF layers after pruning and thresholding offers a one-to-one translation to a symbolic logic program. This ability is novel in contrast to similar works in this category of neuro-symbolic architectures. For example, Relation Networks [179] create a pairwise object representation bottleneck and demonstrate improvements in visual question answering tasks but have no correlation to any form of logic programming. This architectural inductive bias can be traced to many models all the way to and including PrediNet [191]. We use PrediNet as a baseline in Section 6.4 because it is not only the state-of-the-art model in the Relations Game dataset but also one step closer to logic programming by computing predicates as output before falling short of learning symbolic rules. More recently, Neural Production Systems [69] leverage the Gumbel-Softmax [97] distribution to sparsely apply rule templates to latent representations. These encoded *rules*, however, are continuous matrix transformations of latent vectors rather than logical rules as part of a principled framework.

Closer to a logic programming based neural network architecture, Neural Logic Machines [47] use principles of forward-chaining (Section 2.1.4) with nullary, unary and binary predicates implemented as MLPs with input permutations to tackle algorithmic tasks. While the permutation and forward-chaining nature of Neural Logic Machines are similar to the DNF layer described in Section 6.3, the learnt rules or reasoning steps cannot be symbolically extracted due to the use of MLPs which learn some, potentially unknown and non-linear, Boolean operations.

If the logic program is given, one can also construct neural networks, often using t-norms, to implement the matching logical computation. For example, Lifted Relational Neural Networks [202] use rule sets as templates for constructing neural networks, i.e. the connection paths. Logic Tensor Networks [186] use dense embeddings of constants while also constructing a deductive neural network. TensorLog [38] similarly builds factor graphs which in return yield the neural network architecture. These approaches have similarities to the more recent Logical Neural Networks [169] which assemble neural networks from logical formulae and constrain the weights along with a non-standard activation function to achieve conjunction and disjunction semantics. Sen et al. [184]

demonstrate how Logical Neural Networks are used to induce rules for grid-world and knowledge base completion tasks albeit they require rule templates such as $r(X,Z) \leftarrow \underline{p}(X,Y) \wedge \underline{q}(Y,Z)$ where $\underline{p}$ and $\underline{q}$ are adjustable (learnable) predicates. One can also utilise a given logic program as a form of regularisation to transfer structured information into the weights of the networks using the loss function [87]. Our more flexible approach is not bound to a fixed logic program or template and does not require any constraints on the weights whilst also handling negation organically.

We can always decompose the full neuro-symbolic network shown in Fig. 6.4 into a two-stage pipeline whereby the relations could be learnt and then the symbolic rules as opposed to an end-to-end approach. This perspective relates to pipeline or two-stage neuro-symbolic approaches such as neural scene parsing which decomposes or parses input images completely into symbolic entities or concepts and then performs symbolic reasoning. Neuro-Symbolic VQA [226] and following works decompose images into objects and their properties before performing visual question answering. This is similar in nature to the Neural State Machine [89] which converts an image into objects and their relations using a separate pre-trained object detector network before carrying out the reasoning aspect, and Neural Logic Inductive Learning [222] which deploys Transformer [216] based neural networks to induce rules on processed knowledge graphs of images. The equivalent for our approach would be to train the DNF model up until the ground facts P^* from Eq. (6.11) and then use any other symbolic or neuro-symbolic method to predict the labels. For datasets where the decomposition is successful, they outperform any end-to-end approach as the combined difficulty of learning visual features with symbolic rules is decoupled. The Neuro-symbolic Concept Learner [131] extends scene decomposition by learning concepts given in a parsed question such as red in a joint fashion but leverages reinforcement learning in which the environment itself is a symbolic program executor. A similar line of work utilises unsupervised deep representation learning using auto-encoders to convert images into propositional atoms and then utilise symbolic solvers for planning tasks [7, 8].

Finally, there are a group of methods that leverage gradient descent to train upstream neural networks similar to the input CNN layer in Eq. (6.4) of the DNF model. δ-ILP [52] and its derivations [195] implement t-norms to create differentiable logic programs to learn symbolic rules in an Inductive Logic Programming (Section 2.1.6) setting. However, they do not scale to large hypothesis spaces or input sizes due to their template based rule search and are limited when it comes to end-to-end learning due to their use of t-norms. By modelling rule membership of atoms as learnable weights, Neural Logic Networks [154, 153] outperform δ-ILP but again employ t-norms. As shown in Fig. 6.1a, usage of t-norms such as the product t-norm expresses vanishing gradients when dealing with large number of inputs. The influence of gradient descent can also be found in full symbolic reasoning systems like DeepProblog [130] and NeurASP [224] which attempt to propagate gradients through logic programs in order to train input neural networks such as CNNs that recognise hand-written digits and perform addition of the recognised digits. These resemble the continuous formulations of logic such as differentiable stable and supported semantics that utilise matrix multiplications and non-linear activations to realise logic operations [9]. The core ideological difference between these approaches and our DNF layer is that while *they attempt to make logic differentiable, we attempt to make neural networks logical.*

6.7 Discussion

The core novelty presented in this chapter stems from the constrained bias value of the semi-symbolic layer in Eq. (6.2) which when adjusted accordingly creates an approximation of the logical $\wedge$ and $\vee$ Boolean connectives. The DNF layer offers a configuration of semi-symbolic layers to realise a single round of rule application as part of a logic program. While the overall effective computation is novel due to its differentiable nature, it is akin to the T_P operator used in forward-chaining from Section 2.1.4 Definition 4. From that perspective, the DNF layer captures only one of many ways of performing logical inference and there is room for further exploration how semi-symbolic layers as Boolean connective approximators could realise other forms of entailment such as the stable model semantics described in Section 2.1.5.

While the analyses provided in Section 6.5 is that of individual components such as object selection, the modular differentiable components are part of a single larger neuro-symbolic model. We highlight this view since although the behaviour of those layers in isolation may make sense, for example checking if the images are reconstructed in Fig. 6.8, their training and performance is ultimately tied to the overall network and how every component interacts with each other to create one neuro-symbolic system. The DNF model shown in Fig. 6.4 offers a seamless realisation of symbolic deduction and rule induction over a neural substrate. Referring back to the motivation outlined in Section 1.1, this perspective aligns with our overarching goal of creating holistic neural network architectures capable of symbolic inference in a coherent manner similar to the human brain. The integration of symbolic relations and rules with distributed representations of objects in our architecture reflects on how the human brain might be using symbolic and sub-symbolic representations harmoniously [103]. While our novel approach is an approximation of symbolic inference in a neural network, particularly when it comes to the failure modes during thresholding of the weights as shown in Table 6.4, it is still far from what the human brain regarded as a neuro-symbolic computing engine is capable of.

One of the main limitations of our approach presented in this chapter is the transition from continuous real-valued vectors to discrete symbolic entities and the subsequent drop in performance. This challenge of reconciling neural and symbolic representations is a recurring theme of neuro-symbolic models as emphasised in Section 1.2. In this chapter, it manifests in two cases: the hyperbolic tangent function applied to the relations in Eqs. (6.9) and (6.10) and when the weights of the semi-symbolic layers are thresholded after training as described in Section 6.1. The former allows the model to transition from unbounded scores to a truth value that forms the basis of the logical framework. In other words, the activation function allows us to deem something as true or false to kickstart the logical calculus. The latter converts continuous approximations of rules into discrete symbolic programs which we can extract. However, the concern is, particularly for thresholding weights, the consistent degradation of performance across all the model configurations when the training procedure results in a sub-optimal state as observed in the full results Table B.6. That is to say, unless the original model performs at a test accuracy of ≈ 0.99, thresholding the weights w_i from Eq. (6.1) and the relations P^* from Eq. (6.11) to evaluate using a symbolic solver such as clingo leads to an accuracy range of $\approx [0.5, 0.6]$. This limitation is equally highlighted in cases when the original model (DNF) has an accuracy of ≈ 0.7 but its thresholded

counterpart (DNF-t) yields an accuracy of 0.5. Hence, this phenomenon beckons further incorporation of neural network training techniques such as regularisation to improve the otherwise crude thresholding procedure; although, one would desire an emergent logical behaviour from the semi-symbolic layers without any modifications to the training regime.

Finally, we draw attention to the lack of human interpretable representations despite obtaining symbolic rules. While the logic program in Listing 6.2 is clear and verifiable, the meaning of the predicates shown in Fig. 6.7a along with the object representations they operate on shown in Table 6.6 are ambiguous and unidentifiable. This property is not idiosyncratic to our approach and is common to methods on the more neural end of the spectrum shown in Fig. 1.2. When the features are learnt by neural networks as opposed to pre-engineered by a human expert, they may lack extrinsic meaning. This property is referred to as the black-box nature of neural networks and is often highlighted as a limitation in cases when the networks are deployed in decision making tasks [2]. Learning sparse disentangled representations is an active area of research with variational neural networks such as Variational Auto-Encoders [105] and object-centric models that use them [73, 26]. Even if such methods are used, one must *decode* what, if any, of the disentangled features correspond to. On the other hand, we raise the question whether we should expect these latent features to be human interpretable in the context of end-to-end trainable neuro-symbolic architectures. After all, if a model is to learn every component from scratch without human intervention, any viable solution with respect to the loss function may be acceptable since the optimisation carried out by gradient descent solely depends on the loss function as in Eq. (2.7). In other words, it may be unlikely that any recognisable features or behaviour will emerge purely from training on a single predictive loss function such as classification as done in this chapter.

This brings the chapters containing the novel contributions of this thesis to an end. In the next and final concluding chapter, we summarise and discuss the current state, future work and potential alternative avenues of the neuro-symbolic ideas presented thus far.

Chapter 7

Conclusion

We presented three new logic inspired end-to-end differentiable neural networks in Chapters 4 to 6 along with reusable datasets in Chapter 3. Chapter 4 provided further understanding of how deep recurrent neural networks might encompass symbolic deductive reasoning in the domain of logic programming. Chapter 5 presented a new network architecture capable of identifying varying symbols and in return lifting examples into invariants through the use of soft unification. Chapter 6 proposed a unified neuro-symbolic framework for learning objects, their relations and symbolic logic-based rules in an end-to-end fashion using semi-symbolic layers. Across all the chapters, the quantitative results discussed in the experiments section (Sections 4.3, 5.4 and 6.4) demonstrated better if not competitive performance across metrics such as accuracy, data efficiency and scalability against comparable baselines; the analysis sections of each chapter (Sections 4.4, 5.5 and 6.5) provided a qualitative evaluation of our neuro-symbolic models.

The perceived successes of each model in this thesis are obtained from the novel architectural inductive biases baked into the neural networks. These biases are different for each problem domain and form the basis for the type of behaviour one would expect from the networks once trained. For example, the iterative memory updates following the steps of backward-chaining for goal-oriented deductive reasoning in Section 4.1, the design of the soft unification function g that allows unification networks to recognise and utilise varying symbols in Section 5.1 and the semi-symbolic layer with an adjustable bias value to mimic Boolean connectives in Section 6.1, all seem to act as the key ingredient. As mentioned in Chapter 1, we are interested in such architectural biases as means to elicit some form symbolic inference following the inspiration that the human brain may be one complete and coherent neuro-symbolic system over a single neuronal substrate with a predisposition for neural and symbolic reasoning [51]. However, it remains a mystery as to what, if any, the ultimate architectural bias we require in order to attain the human level seamless integration of neural and symbolic reasoning as part of a single end-to-end trainable artificial neural network capable of learning and performing logic-based inference.

This trend of finding the right architectural bias has already been driving a paradigm shift in the field of natural language processing. Transformer based deep neural language models such as BERT [45], XLNet [223] and GPT-3 [25] have been outperforming existing state-of-the-art ap-

proaches on almost every benchmark. These models are huge, compared to the average 40k learnable weights of the models presented in this thesis (e.g. the larger DNF-hi model from Section 6.4 has 65,204), the number of learnable parameters for the aforementioned models range from ≈ 300 million to ≈ 173 billion. Beyond benchmark tasks such as sentiment analysis, these language models have also been shown to generate coherent pieces of text [25], induce computer programs for solving algorithmic problems [10] and act as knowledge bases by extracting information post-training [158]. This sudden improvement may be attributed to the architectural change of deep language models going from recurrent neural networks as described in Section 2.2.4 to state-of-the-art transformer models [216] along with unsupervised autoregressive learning on vast amounts of data like an offline copy of all the natural language text available on the internet [25]. This trend of exploiting huge amounts of unlabelled data through some self-supervision also extends to the image domain with works such as the Barlow Twins [229] and Masked Auto-encoders [77]. Neuro-symbolic systems might be facing a bottleneck to find a single, flexible, scalable and efficient architecture along with a self-supervising regime on a wealth of data because most of the research surrounding end-to-end systems, as it is in this thesis, involves a search for and the presentation of novel architectural biases within a supervised learning setting. In spite of the huge success of language models, there is modern criticism for lack of understanding on how they work, why they fail on certain occasions and their impact on the society [20] as well as their computational resource requirements [212].

Although the neuro-symbolic account of the human mind is a driving source for end-to-end neural networks capable of learning logic-based inference, we conclude this thesis by highlighting the unknowns of the anatomical nature of the human brain and the questions they raise. Similar to how convolutional neural networks align with the perceptive design of a biological eye as described in Section 2.2.3, further interdisciplinary research combining not only the cognitive aspects of the human mind but also its physiological properties surrounding symbolic reasoning with deep learning might be necessary for advancing the field. In neuroscience, the problem of segregating sensory inputs such as colour and sound into discrete objects or symbols along with how they are combined into a single coherent experience is explored under the umbrella of the binding problem [167]. There are already similarities with the work from Chapter 6 whereby an image is processed into objects and then recombined into a single prediction albeit done from a practical, logic programming perspective as opposed to a neurological one. Greff, Steenkiste, and Schmidhuber [72] draw inspiration from the binding problem to offer a unifying framework on how neural networks may segregate inputs such as images into meaningful entities and then recombine them to make predictions. However, even with interdisciplinary integration of ideas, a compromise between practical abstraction and biological reality seems inevitable given the admirable complexity of the human brain and its many unknowns.

7.1 Future Work

In the first instance, one may consider amalgamating the novel modular components presented in Chapters 4 to 6 due to their end-to-end differentiable nature. For example, one might be able use semi-symbolic layers from Section 6.1 inside an iterative neural reasoning network described

in Section 4.1 in order to build a neuro-symbolic architecture capable of logical deduction in which the steps of backward-chaining are logical rules themselves. In other words, how the model determines the next sub-goal might be obtained as symbolic rules such as

$$goal(X,Y) \leftarrow match(predicate(X), predicate(Y)) \wedge match(args(X), args(Y))$$

in which X,Y may represent atoms and $predicate(X), args(X)$ are function symbols to obtain the predicate symbol and the arguments of the atom respectively. A logic program of this nature would have to integrate more of first-order logic described in Section 2.1.2 such as the aforementioned function symbols. This expansion into more expressive logics such as the inclusion of default reasoning [165] as well as different entailment semantics such as stable models from Section 2.1.5 may improve the overall expressiveness of the neuro-symbolic networks presented in this thesis, in particular the DNF layer from Chapter 6 in which only forward-chaining over stratified logic programs are considered.

Another prominent piece of future work would be to apply neuro-symbolic approaches of this thesis to a more decision critical domain such as reinforcement learning. Unlike a direct mapping between inputs and outputs in a supervised learning setting as in Section 2.2, reinforcement learning captures the process of learning and performing in an environment whereby an agent receives observations and acts accordingly [100]. There is an existing and growing neuro-symbolic reinforcement learning scene where the proposed approaches can be categorised based on how neural or symbolic they are similar to Fig. 1.2. On the neural end of the spectrum we have Deep Q-Learning [142] and its variants [80] which use convolutional neural networks to approximate the value function to successfully learn how to play Atari games while the other end comprises of techniques that utilise state-of-the-art symbolic rule learners such as using ILASP to learn sub-goals [57]. As with many neuro-symbolic methods covered in Chapter 1, the practical goal is to maintain the benefits of both worlds such as data efficiency with robust learning. Incorporating a relational architectural bias through an attention mechanism similar to that of PrediNet [191], Zambaldi et al. [227] demonstrate better generalisation of their neuro-symbolic agent against baselines in a box-world and the challenging StarCraft 2 environments. Similarly, Garnelo, Arulkumaran, and Shanahan [59] propose an end-to-end deep neuro-symbolic reinforcement learning paradigm which improves data efficiency and high-level abstraction compared against Deep Q-Learning. On the more logic-based side, Logical Neural Networks have been integrated into a reinforcement learning setting in text-based games [104]. Inspired by the dual-systems of the human brain, Botvinick et al. [22] provide a modern narrative on how deep reinforcement learning agents may include both fast and slow forms of learning through episodic memory architectures along with learning-to-learn meta-level training algorithms. All these works leverage similar neuro-symbolic principles to those explored in this thesis and encourage future work in integrating aspects of Chapters 4 to 6 such as soft unification and semi-symbolic layers into neural network based reinforcement learning agents.

Finally, the novel methods presented in this thesis can be further expanded or optimised following literature surrounding few-shot deep learning. Few-shot learning includes methods and algorithms that are geared towards learning or predicting from very few examples [218] in contrast to the

relatively large number of data points used across Sections 4.2, 5.2 and 6.2. For example, Matching Networks [217] offer a differentiable narrative of the non-parametric machine learning algorithm k-nearest neighbours in which the distance metric is based on the embeddings obtained from a convolutional neural network. After an initial training phase, previously unseen examples from new classes (a classification that has not been made during training) are correctly predicted with very few examples of that new class (their neighbours). Similarly, taking the idea of differentiable metric learning further, Prototypical Networks [199] allow generalisation to unseen classes by using clusters of examples of which the mean (the centroid) is the prototype for that class. One may consider integrating ideas such as nearest-neighbours or clustering into a neuro-symbolic architecture through the use of invariants as described in Chapter 5 or symbolic rules as learnt in an end-to-end fashion in Chapter 6. For example, the learnt invariant may act as the prototype for a class of data points or logic-based rules that make two examples similar, i.e. neighbours, may be learnt. A mixture of these methods to create a neuro-symbolic few-shot learning paradigm may achieve further gains in data efficiency and out-of-distribution generalisation.

7.2 Alternative Avenues

Due to the popularity and unprecedented success of deep learning in recent years [67], neuro-symbolic methods using artificial neural networks have been focused on gradient descent as the key ingredient for *learning* (refer back to Section 2.2.2 on how gradient descent is used to optimise the weights of a neural network). In Chapters 4 to 6, we also propose and compare our methods to end-to-end differentiable neural networks trained using gradient descent. In this section, however, we highlight alternative avenues one may consider in the pursuit of neuro-symbolic learning of logic-based inference outside the boundaries of gradients and gradient descent.

Genetic algorithms are a family of optimisation methods inspired by the natural selection process [176]. The optimal values for the learnable weights (parameters) of a network are searched through biologically motivated operators such as mutation, crossover and selection of best the individual from a population. Informally, a high-level implementation might involve:

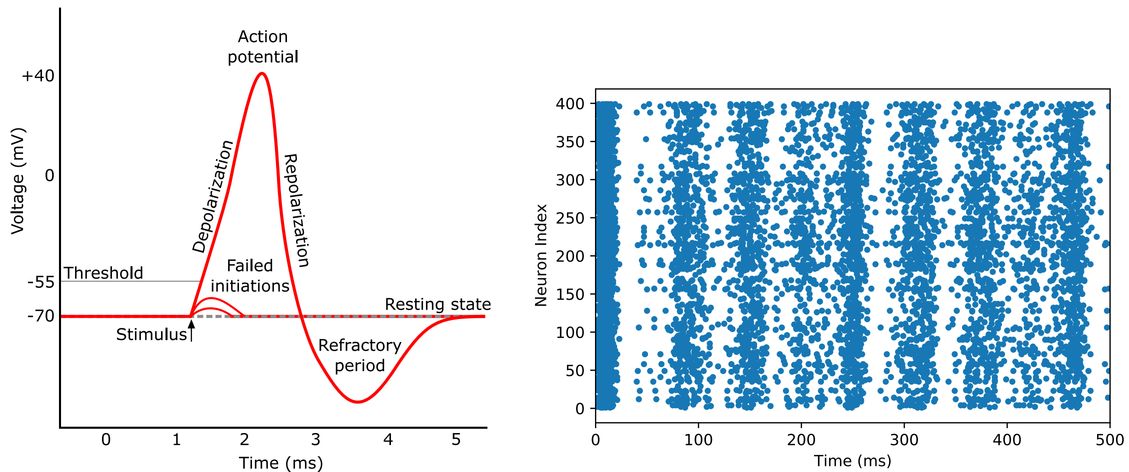
1. Generating a population of n randomly initialised networks
2. Evaluating each individual's performance such as accuracy against a training set and selecting the best two individuals from the population
3. Crossing over (swapping) the weights between the selected parents at random and mutating the weights by adding noise
4. Repeating the previous step n many times to obtain a new population
5. Repeating steps 2 to 4 until a performance criteria is reached

The unique selling point of genetic algorithms and the overall evolutionary computing domain are their gradient-free nature. Consequently, the entire model does not need to be end-to-end differentiable which opens up room for novel architectures, search procedures and performance improvements. For example, Salimans et al. [178] utilise Evolution Strategies which just mutate

the weights of a population of networks towards higher rewards in a reinforcement learning environment and obtain competitive results in terms of not only performance but also data efficiency, scalability, robustness and better parallelisation over a cluster of machines. These are all desirable enhancements one would seek from neuro-symbolic architectures.

An overarching theme of this thesis is architectural bias and the search for the right one in order to create coherent and holistic end-to-end neuro-symbolic learning of logic-based inference. The restrictions on automating this architectural search using gradient descent stems from its non-differentiable nature. We cannot readily search through different neural network architectures in a differentiable manner along with optimising the weights. However, using genetic algorithms and its variants one can consider the architecture, i.e. the connection patterns of the network, part of the learning process. Indeed, Stanley and Miikkulainen [207] introduce the NeuroEvolution of Augmenting Topologies (NEAT) approach for not only evolving the connection patterns of a neural network but also its connection weights as part of a single process. The field of Neuroevolution offers the means to search for neural network architectures, its connection weights and even potentially the learning algorithm itself [208]. Further examples include Compositional Pattern Producing Networks (CPPN) [206] which can generate complex patterns using a compact coordinate based encoding and Hyper-Neat [60] that attempts to automatically evolve the aforementioned patterns such as a convolution by combining NEAT and CPPNs. Therefore, neuro-symbolic systems might benefit from an intrinsically coupled architectural search using Neuroevolution, for example by evolving the connections of the DNF layer from Section 6.3 as opposed to training all possible connections (weights) and pruning afterwards.

Even if neuroevolution offers a viable method for evolving neuro-symbolic systems, one might not encounter the desired behaviour surrounding logic-based inference. In other words, beyond *what* a system does, we are equally interested in *how* it does it. Throughout Sections 4.4, 5.5 and 6.5, we analyse how our novel models behave beyond their quantitative performance such as whether they select the right objects in Fig. 6.6 and the style of induced symbolic rules in Listing 6.3. Hence, there is another perspective of the term neuro-symbolic which is more concerned with the *behaviour* of a system as opposed to its architecture. For example, one may consider a multi-layer perceptron to have neuro-symbolic properties based on its actions as a reinforcement learning agent, interacting with objects and solving puzzles. All the entries to the Animal-AI Olympics [40] irrespective of their neuro-symbolic inductive bias, attempt to solve common-sense reasoning tasks involving objects, relations and rules which arguably require some form of neuro-symbolic reasoning. Such common-sense reasoning still persists as one of the biggest challenges in Artificial Intelligence [192]. This interest in behaviour relates to quality diversity in which the goal is to discover a range of high performing models with different behavioural characteristics [41]. The key driving ingredient is *novelty* quantified in behaviour space such as the location of a robot at the end of a random walk [42]. Stretching this idea of behavioural evolution to its limits, AutoML-Zero [163] attempts to evolve machine learning algorithms automatically from scratch over three phases involving setup, prediction and learning with a fixed memory size and a set of pre-defined operations. After approximately 5 days, the system behaves like a 2 layer neural network with a stochastic gradient descent optimisation procedure as described in Section 2.2.2.



(a) Visualisation of the membrane voltage of a biological neuron during an action potential. After the membrane voltage crosses the threshold of -55mV, it quickly depolarises to generate an action potential. Credit: Wikimedia Commons

(b) Example spiking neural network using 400 mixed inhibitory and excitatory Izhikevich neurons with random connections. Each point corresponds to an action potential (a spike). The overall behaviour is oscillatory with an approximate frequency of $\approx$ 20Hz.

Figure 7.1: Unlike regular Artificial Neural Networks, Spiking Neural Networks model the membrane potential of each neuron.

This result hints at the potentially unbounded complexity neuroevolution can generate through the use of simple building blocks. Unlike the learning paradigm of deep learning as practised in this thesis, we would perhaps like neuro-symbolic properties to be emergent behaviours of ever evolving systems akin to the history of evolution of the human brain.

Finally, we draw attention to the core difference between artificial and biological neurons. While neurons in artificial neural networks are modelled as linear transformations of the sum of their inputs with real-valued outputs, biological neurons operate by generating discrete action potentials carried along their axons. These rapid rises and falls in membrane voltage as shown in Fig. 7.1a are also referred to as *spikes* [64]. Using computational approximations of biological neurons, one can construct Spiking Neural Networks (SNN) in a similar fashion to Section 2.2.1 where the output calculation of a neuron from Definition 6 is replaced with an integrate-and-fire model of the membrane voltage. We leave the reader to explore various computational models of spiking neurons such as the popular Izhikevich neuron [91] as it is outside the scope of this thesis. In summary, the membrane voltage is modelled such that when a spike arrives the voltage increases by a small amount to the point of reaching a threshold at which the neuron itself generates a spike. Fig. 7.1b plots the spikes of a neural network with 400 mixed excitatory and inhibitory Izhikevich neurons. In this particular case, we observe an oscillatory behaviour in which neurons fluctuate between regions of high and low activity. So why are Spiking Neural Networks not as popular despite offering a biologically more plausible medium for neuronal computation? The answer lies in the discrete nature of the spikes which by default do not lend themselves for the ubiquitous and powerful gradient based optimisation techniques that have been driving the success of large deep artificial neural networks. In other words, training spiking neurons is not directly possible with gradient descent and one has to resort to other methods such as genetic algorithms or Spike Timing

Dependent Plasticity [159]. For example, using activity-dependent plasticity, Müller et al. [146] show that clusters of spiking neurons can be trained to store and retrieve values creating a form of associative memory. There are further success cases on deploying spiking neural networks in more complex domains such as robotics [18]. Taking the human mind as a source of inspiration, it is natural to consider expanding neuro-symbolic methods to a spiking neural substrate for insights into biological similarity and plausibility. However, the complex structure of the spiking neurons in the human brain is an active area of research with topics including the connective core [190], global workspaces [188] and the dynamics of networks with small-world properties [189] all representing challenges for biologically plausible neuro-symbolic spiking networks. Even research into avian brains which are relatively much smaller uncover intricate connectivity patterns with similar aforementioned properties [193]. Beyond their biological roots, spiking neural networks are also valuable for low-power high-efficiency computing paradigm using analog as opposed to digital signals called neuromorphic computing [181]. This avenue might offer a solution to the large computing resources currently required by deep learning and allow neuro-symbolic research to experiment with bigger architectures.

7.3 Broader Impact

We end our discussion of end-to-end neuro-symbolic learning of logic-based inference by briefly reviewing how, if any, the research presented in this thesis and released to the public domain through publications and online source code repositories in Section 1.5 may impact the wider world. In Chapter 4, we introduced and analysed end-to-end differentiable recurrent memory based neural networks capable of learning an approximation of logical deduction in the realm of logic programming. We expect Neural Reasoning Networks such as the Iterative Memory Attention model described in Section 4.1 to act as a baseline for further neuro-symbolic research instead of any direct commercial applications due to the drop in performance surrounding extended testing conditions shown in Section 4.4. Furthermore, while we were able to observe visually which rules the models select in Fig. 4.3 and the clustering of predicate representations in Fig. 4.5, none of the neural networks offer *how* exactly they perform such deductive reasoning which should inhibit deployment of such systems in their current state.

In Chapter 5, we proposed Unification Networks to learn invariant patterns that may be present in the input data. As it is with any machine learning model aimed at extracting patterns solely from data, learning invariants through soft unification is prone to being influenced by spurious correlations and biases. There is no guarantee that even a clear, high accuracy invariant might correspond to a valid inference or casual relationship as previously mentioned in Section 5.5 with some mismatching invariants for the relatively simple synthetic datasets shown in Fig. 5.11. Consequently, if our approach succeeds in solving the task with an invariant, it does not necessarily mean that there is a pattern or in the failure case, a lack of patterns in the data. There has been recent work on tackling a different notion of invariance formed of features that are consistent (hence invariant) across different training dataset environments, to learn more robust predictors [6, 78] that is outside the scope of this thesis. Unification Networks is instead targeted at research and researchers involved with combining cognitive aspects such as variable learning and assignment with neural

networks under the umbrella of neuro-symbolic systems. A differentiable formulation of variables could accelerate the research of combining logic based symbolic systems with neural networks but a public facing commercial application of soft unification based invariant learning in its current form may be misleading. Furthermore, how the invariants are used by the downstream predictor networks are unknown as highlighted in Section 5.5 which may further undermine real-world applications with respect to interpretability.

In Chapter 6, we introduced the semi-symbolic and the DNF layers using a novel inductive bias in the feed-forward computation of a linear layer. Unlike symbolic solvers such as ILASP [114], the DNF layer applied on symbolic inputs, i.e. to an instance of inductive logic programming problem (Section 2.1.6), do not provide any theoretical guarantees on finding any solution. In other words, our differentiable rule learning approach is presented on an empirical basis. It is not the case that if a set of rules is learnt, it is correct or if none are found then there are no solutions to the problem. However, the target use case for semi-symbolic layers is learning symbolic rules from unstructured inputs such as images as opposed to competing in the symbolic rule learning domain. As a result, while our novel approach may be more widely applicable, it is also more fragile inheriting the disadvantages of its neural network foundation such as over-fitting. Our target audience again is researchers aiming to advance neuro-symbolic systems and one would expect further improvements before semi-symbolic or DNF layers are used in industrial settings.

In conclusion, the ideas and results presented in this thesis can be viewed as stepping stones towards intelligent machines of the neuro-symbolic kind that can perform high-level reasoning and low-level perception. Yet, the ultimate goal of Artificial General Intelligence [66] as envisioned by the public still invites many years of research.

Bibliography

[1] Martin Abadi et al. “TensorFlow: A system for large-scale machine learning”. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016, pp. 265–283. URL: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf.

[2] Amina Adadi and Mohammed Berrada. “Peeking Inside the Black-Box: A Survey on Explainable Artificial Intelligence (XAI)”. In: *IEEE Access* 6 (2018), pp. 52138–52160. DOI: 10.1109/access.2018.2870052.

[3] Julius Adebayo et al. “Sanity Checks for Saliency Maps”. In: *Advances in Neural Information Processing Systems* (Oct. 8, 2018), pp. 9505–9515. arXiv: 1810.03292 [cs.CV].

[4] Krzysztof R. Apt and Roland N. Bol. “Logic programming and negation: A survey”. In: *The Journal of Logic Programming* 19-20 (May 1994), pp. 9–71. DOI: 10.1016/0743-1066(94)90024-8.

[5] Krzysztof R. Apt and M. H. van Emden. “Contributions to the Theory of Logic Programming”. In: *Journal of the ACM* 29.3 (July 1982), pp. 841–862. DOI: 10.1145/322326.322339.

[6] Martin Arjovsky et al. “Invariant Risk Minimization”. In: (July 5, 2019). arXiv: 1907.02893 [stat.ML].

[7] Masataro Asai. “Unsupervised Grounding of Plannable First-Order Logic Representation from Images”. In: *ICAPS* (Feb. 2019). arXiv: 1902.08093 [cs.AI].

[8] Masataro Asai and Alex Fukunaga. “Classical Planning in Deep Latent Space: Bridging the Subsymbolic Symbolic Boundary”. In: (Apr. 29, 2017). arXiv: 1705.00154 [cs.AI].

[9] Yaniv Aspis et al. “Stable and Supported Semantics in Continuous Vector Spaces”. In: *Proceedings of the Seventeenth International Conference on Principles of Knowledge Representation and Reasoning*. International Joint Conferences on Artificial Intelligence Organization, 2020. DOI: 10.24963/kr.2020/7.

[10] Jacob Austin et al. “Program Synthesis with Large Language Models”. In: (Aug. 16, 2021). arXiv: 2108.07732 [cs.PL].

[11] Sebastian Bach et al. “On Pixel-Wise Explanations for Non-Linear Classifier Decisions by Layer-Wise Relevance Propagation”. In: *PLOS ONE* 10.7 (July 2015). Ed. by Oscar Deniz Suarez, e0130140. DOI: 10.1371/journal.pone.0130140.

[12] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. “Neural Machine Translation by Jointly Learning to Align and Translate”. In: *ICLR* (Sept. 1, 2014). arXiv: 1409.0473 [cs.CL].

[13] David G. T. Barrett et al. “Measuring abstract reasoning in neural networks”. In: *ICML* (July 11, 2018). arXiv: 1807.04225 [cs.LG].

[14] Peter W. Battaglia et al. “Relational inductive biases, deep learning, and graph networks”. In: (June 4, 2018). arXiv: 1806.01261 [cs.LG].

[15] Daniel M. Bear et al. “Learning Physical Graph Representations from Visual Scenes”. In: *NeurIPS* (June 2020). arXiv: 2006.12373 [cs.CV].

[16] Yoshua Bengio et al. "Curriculum learning". In: *Proceedings of the 26th Annual International Conference on Machine Learning - ICML '09*. ACM. ACM Press, 2009, pp. 41–48. DOI: 10.1145/1553374.1553380.

[17] Tarek R. Besold et al. "Neural-Symbolic Learning and Reasoning: A Survey and Interpretation". In: (Nov. 10, 2017). arXiv: 1711.03902 [cs.AI].

[18] Zhenshan Bing et al. "A Survey of Robotics Control Based on Learning-Inspired Spiking Neural Networks". In: *Frontiers in Neurorobotics* 12 (July 2018). DOI: 10.3389/fnbot.2018.00035.

[19] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer-Verlag New York Inc., Apr. 6, 2011. 738 pp. ISBN: 0387310738. URL: https://www.ebook.de/de/product/5324937/christopher_m_bishop_pattern_recognition_and_machine_learning.html.

[20] Rishi Bommasani et al. "On the Opportunities and Risks of Foundation Models". In: (Aug. 16, 2021). arXiv: 2108.07258 [cs.LG].

[21] Matko Bošnjak et al. "Programming with a Differentiable Forth Interpreter". In: *ICML* (May 21, 2016). arXiv: 1605.06640 [cs.NE].

[22] Matthew Botvinick et al. "Reinforcement Learning, Fast and Slow". In: *Trends in Cognitive Sciences* 23.5 (May 2019), pp. 408–422. DOI: 10.1016/j.tics.2019.02.006.

[23] Samuel R. Bowman et al. "A large annotated corpus for learning natural language inference". In: *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Aug. 21, 2015. DOI: 10.18653/v1/d15-1075. arXiv: 1508.05326 [cs.CL].

[24] Krysia B. Broda, Artur S. D'Avila Garcez, and Dov M. Gabbay. *Neural-Symbolic Learning Systems*. Springer London, Aug. 6, 2002. 288 pp. ISBN: 1852335122. DOI: 10.1007/978-1-4471-0211-3.

[25] Tom B. Brown et al. "Language Models are Few-Shot Learners". In: (May 28, 2020). arXiv: 2005.14165 [cs.CL].

[26] Christopher P. Burgess et al. "MONet: Unsupervised Scene Decomposition and Representation". In: (Jan. 2019). arXiv: 1901.11390 [cs.CV].

[27] Jose Camacho-Collados and Mohammad Taher Pilehvar. "From Word To Sense Embeddings: A Survey on Vector Representations of Meaning". In: *Journal of Artificial Intelligence Research* 63 (2018), pp. 743–788. DOI: 10.1613/jair.1.11259. arXiv: 1805.04032 [cs.CL].

[28] Oana-Maria Camburu et al. "e-SNLI: Natural Language Inference with Natural Language Explanations". In: *NeurIPS* (Dec. 4, 2018). arXiv: 1812.01193 [cs.CL].

[29] Aditya Chattopadhay et al. "Grad-CAM++: Generalized Gradient-Based Visual Explanations for Deep Convolutional Networks". In: *2018 IEEE Winter Conference on Applications of Computer Vision (WACV)*. IEEE. IEEE, Mar. 2018, pp. 839–847. DOI: 10.1109/wacv.2018.00097.

[30] Sneha Chaudhari et al. "An Attentive Survey of Attention Models". In: *IJCAI* (Apr. 5, 2019). arXiv: 1904.02874 [cs.LG].

[31] Kyunghyun Cho et al. "On the Properties of Neural Machine Translation: Encoder–Decoder Approaches". In: *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*. Association for Computational Linguistics, 2014. DOI: 10.3115/v1/w14-4012.

[32] Nuri Cingillioglu and Alessandra Russo. "DeepLogic: Towards End-to-End Differentiable Logical Reasoning". In: *AAAI-MAKE* (Mar. 20, 2019). arXiv: 1805.07433 [cs.NE].

[33] Nuri Cingillioglu and Alessandra Russo. "Learning Invariants through Soft Unification". In: *NeurIPS* (Oct. 24, 2020). arXiv: 1909.07328 [cs.LG].

[34] Nuri Cingillioglu and Alessandra Russo. "pix2rule: End-to-end Neuro-symbolic Rule Learning". In: *IJCLR-NeSy* (June 14, 2021). arXiv: 2106.07487 [cs.LG].

[35] Keith L. Clark. "Negation as Failure". In: *Logic and Data Bases*. Springer US, 1978, pp. 293–322. DOI: 10.1007/978-1-4684-3384-5_11.

[36] Peter Clark, Oyvind Tafjord, and Kyle Richardson. "Transformers as Soft Reasoners over Language". In: *IJCAI* (Feb. 14, 2020). arXiv: 2002.05867 [cs.CL].

[37] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog*. Springer Berlin Heidelberg, 2003. DOI: 10.1007/978-3-642-55481-0.

[38] William W. Cohen. "TensorLog: A Differentiable Deductive Database". In: *arXiv:1605.06523* (May 20, 2016). arXiv: http://arxiv.org/abs/1605.06523v2 [cs.AI].

[39] Stephen A. Cook. "The complexity of theorem-proving procedures". In: *Proceedings of the third annual ACM symposium on Theory of computing - STOC '71*. ACM Press, 1971. DOI: 10.1145/800157.805047.

[40] Matthew Crosby, Benjamin Beyret, and Marta Halina. "The Animal-AI Olympics". In: *Nature Machine Intelligence* 1.5 (May 2019), pp. 257–257. DOI: 10.1038/s42256-019-0050-3.

[41] Antoine Cully and Yiannis Demiris. "Quality and Diversity Optimization: A Unifying Modular Framework". In: *IEEE Transactions on Evolutionary Computation* 22.2 (Apr. 2018), pp. 245–259. DOI: 10.1109/tevc.2017.2704781. arXiv: 1708.09251 [cs.NE].

[42] Antoine Cully et al. "Robots that can adapt like animals". In: *Nature* 521.7553 (May 2015), pp. 503–507. DOI: 10.1038/nature14422.

[43] Daniel Cunnington et al. "NSL: Hybrid Interpretable Learning From Noisy Raw Data". In: (Dec. 2020). arXiv: 2012.05023 [cs.LG].

[44] Wang-Zhou Dai and Stephen H. Muggleton. "Abductive Knowledge Induction From Raw Data". In: *IJCAI* (Oct. 7, 2020). arXiv: 2010.03514 [cs.AI].

[45] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North*. Association for Computational Linguistics, 2019. DOI: 10.18653/v1/n19-1423.

[46] Liya Ding. "Neural Prolog-the concepts, construction and mechanism". In: *1995 IEEE International Conference on Systems, Man and Cybernetics. Intelligent Systems for the 21st Century*. IEEE, 1995. DOI: 10.1109/icsmc.1995.538347.

[47] Honghua Dong et al. "Neural Logic Machines". In: *ICLR* (Apr. 2019). arXiv: 1904.11694 [cs.AI].

[48] Sašo Džeroski. "Inductive Logic Programming and Knowledge Discovery in Databases". In: *Advances in Knowledge Discovery and Data Mining*. USA: American Association for Artificial Intelligence, 1996, pp. 117–152. ISBN: 0262560976.

[49] Herbert Enderton. *A mathematical introduction to logic*. San Diego, Calif: Academic Press, 2001. ISBN: 9780122384523.

[50] F. Esposito et al. "A comparative analysis of methods for pruning decision trees". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 19.5 (1997), pp. 476–493. DOI: 10.1109/34.589207.

[51] Jonathan St.B.T. Evans. "In two minds: dual-process accounts of reasoning". In: *Trends in Cognitive Sciences* 7.10 (Oct. 2003), pp. 454–459. DOI: 10.1016/j.tics.2003.08.012.

[52] Richard Evans and Edward Grefenstette. "Learning Explanatory Rules from Noisy Data". In: *Journal of Artificial Intelligence Research* 61 (2018), pp. 1–64. DOI: 10.1613/jair.5714. arXiv: 1711.04574 [cs.NE].

[53] Richard Evans et al. "Can Neural Networks Understand Logical Entailment?" In: (Feb. 23, 2018). arXiv: 1802.08535 [cs.NE].

[54] Shane Frederick. "Cognitive Reflection and Decision Making". In: *Journal of Economic Perspectives* 19.4 (Nov. 2005), pp. 25–42. DOI: 10.1257/089533005775196732.

[55] Gottlob Frege. "Sense and Reference". In: *The Philosophical Review* 57.3 (May 1948), p. 209. DOI: 10.2307/2181485.

[56] Joe Frost. *The Developmental Benefits Of Playgrounds*. Olney MD: Association for Childhood Education International, 2004. ISBN: 0871731649.

[57] Daniel Furelos-Blanco et al. "Induction and Exploitation of Subgoal Automata for Reinforcement Learning". In: *Journal of Artificial Intelligence Research* (Sept. 8, 2020). DOI: 10.1613/jair.1.12372. arXiv: 2009.03855 [cs.AI].

[58] Artur d'Avila Garcez and Luis C. Lamb. "Neurosymbolic AI: The 3rd Wave". In: (Dec. 10, 2020). arXiv: 2012.05876 [cs.AI].

[59] Marta Garnelo, Kai Arulkumaran, and Murray Shanahan. "Towards Deep Symbolic Reinforcement Learning". In: (Sept. 18, 2016). arXiv: 1609.05518 [cs.AI].

[60] Jason Gauci and Kenneth Stanley. "Generating large-scale neural networks through discovering geometric regularities". In: *Proceedings of the 9th annual conference on Genetic and evolutionary computation - GECCO '07*. ACM Press, 2007. DOI: 10.1145/1276958.1277158.

[61] Martin Gebser et al. "Theory Solving Made Easy with Clingo 5". en. In: (2016). DOI: 10.4230/OASICS.ICLP.2016.2.

[62] Michael Gelfond and Vladimir Lifschitz. "The stable model semantics for logic programming." In: *ICLP/SLP*. Vol. 88. 1988, pp. 1070–1080.

[63] Gerhard Gentzen. "Untersuchungen uber das logische Schlieben." In: *Mathematische Zeitschrift* 39.1 (Dec. 1935), pp. 176–210. DOI: 10.1007/bf01201353.

[64] Wulfram Gerstner and Werner M. Kistler. *Spiking Neuron Models*. Cambridge, U.K. New York: Cambridge University Press, Aug. 2002. ISBN: 9780511815706. DOI: 10.1017/cbo9780511815706.

[65] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, 2010, pp. 249–256. URL: https://proceedings.mlr.press/v9/glorot10a.html.

[66] Ben Goertzel. *Artificial General Intelligence*. Berlin: Springer, 2007. ISBN: 9783540686774.

[67] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, Jan. 3, 2017. 800 pp. ISBN: 0262035618. DOI: 10.1016/b978-0-12-804291-5.00010-6.

[68] Ian J. Goodfellow et al. "Generative Adversarial Networks". In: *NeurIPS* (June 10, 2014). arXiv: 1406.2661 [stat.ML].

[69] Anirudh Goyal et al. "Neural Production Systems". In: (Mar. 2021). arXiv: 2103.01937 [cs.AI].

[70] Alex Graves, Greg Wayne, and Ivo Danihelka. "Neural Turing Machines". In: *arXiv:1410.5401* (Oct. 20, 2014). arXiv: http://arxiv.org/abs/1410.5401v2 [cs.NE].

[71] Alex Graves et al. "Hybrid computing using a neural network with dynamic external memory". In: *Nature* 538.7626 (Oct. 2016), pp. 471–476. DOI: 10.1038/nature20101.

[72] Klaus Greff, Sjoerd van Steenkiste, and Jürgen Schmidhuber. "On the Binding Problem in Artificial Neural Networks". In: (Dec. 9, 2020). arXiv: 2012.05208 [cs.NE].

[73] Klaus Greff et al. "Multi-Object Representation Learning with Iterative Variational Inference". In: *ICML 2019 (PMLR 97:2424-2433)* (Mar. 1, 2019). arXiv: 1903.00450 [cs.LG].

[74] Abhishek Gupta et al. "Deep learning for object detection and scene perception in self-driving cars: Survey, challenges, and open issues". In: *Array* 10 (July 2021), p. 100057. DOI: 10.1016/j.array.2021.100057.

[75] M.M. Gupta and J. Qi. "Theory of T-norms and fuzzy inference methods". In: *Fuzzy Sets and Systems* 40.3 (Apr. 1991), pp. 431–450. DOI: 10.1016/0165-0114(91)90171-l.

[76] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, June 2016. DOI: 10.1109/cvpr.2016.90.

[77] Kaiming He et al. “Masked Autoencoders Are Scalable Vision Learners”. In: (Nov. 11, 2021). arXiv: 2111.06377 [cs.CV].

[78] Christina Heinze-Deml, Jonas Peters, and Nicolai Meinshausen. “Invariant Causal Prediction for Nonlinear Models”. In: *Journal of Causal Inference* 6.2 (June 26, 2017). arXiv: 1706.08576 [stat.ME].

[79] Mikael Henaff et al. “Tracking the World State with Recurrent Entity Networks”. In: *ICLR 2017* (2017). arXiv: 1612.03969 [cs.CL].

[80] Matteo Hessel et al. “Rainbow: Combining Improvements in Deep Reinforcement Learning”. In: *AAAI* (Oct. 6, 2017). arXiv: 1710.02298 [cs.AI].

[81] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.

[82] Wilfrid Hodges. *Logic*. Harmondsworth New York: Penguin, 1977. ISBN: 9780140136364.

[83] Ian Hodkinson and Alessandra Russo. *Logic*. Imperial College London. 2019. URL: https://www.doc.ic.ac.uk/~imh/index.html (visited on 09/07/2021).

[84] Paul W. Holland. “Statistics and Causal Inference”. In: *Journal of the American Statistical Association* 81.396 (Dec. 1986), pp. 945–960. DOI: 10.1080/01621459.1986.10478354.

[85] David Holmes. “Reconstructing the retina”. In: *Nature* 561.7721 (Sept. 2018), S2–S3. DOI: 10.1038/d41586-018-06111-y.

[86] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. “Multilayer feedforward networks are universal approximators”. In: *Neural Networks* 2.5 (Jan. 1989), pp. 359–366. DOI: 10.1016/0893-6080(89)90020-8.

[87] Zhiting Hu et al. “Harnessing Deep Neural Networks with Logic Rules”. In: *ACL* (Mar. 21, 2016). DOI: 10.18653/v1/p16-1228. arXiv: 1603.06318 [cs.LG].

[88] Drew A. Hudson and Christopher D. Manning. “Compositional Attention Networks for Machine Reasoning”. In: *ICLR* (Mar. 8, 2018). arXiv: http://arxiv.org/abs/1803.03067v2 [cs.AI].

[89] Drew A. Hudson and Christopher D. Manning. “Learning by Abstraction: The Neural State Machine”. In: *NeurIPS* (July 9, 2019). arXiv: 1907.03950 [cs.AI].

[90] Marco A. P. Idiart et al. “How the Brain Represents Language and Answers Questions? Using an AI System to Understand the Underlying Neurobiological Mechanisms”. In: *Frontiers in Computational Neuroscience* 13 (Mar. 2019). DOI: 10.3389/fncom.2019.00012.

[91] E.M. Izhikevich. “Simple model of spiking neurons”. In: *IEEE Transactions on Neural Networks* 14.6 (Nov. 2003), pp. 1569–1572. DOI: 10.1109/tnn.2003.820440.

[92] Allan Jabri, Andrew Owens, and Alexei A. Efros. “Space-Time Correspondence as a Contrastive Random Walk”. In: *NeurIPS* (June 25, 2020). arXiv: 2006.14613 [cs.CV].

[93] Peter Jackson. *Introduction to expert systems*. Harlow, England Reading, Massachusetts: Addison-Wesley, 1999. ISBN: 9780201876864.

[94] Herbert Jaeger. “Deep neural reasoning”. In: *Nature* 538.7626 (Oct. 2016), pp. 467–468. DOI: 10.1038/nature19477.

[95] Andrew Jaegle et al. “Perceiver IO: A General Architecture for Structured Inputs and Outputs”. In: (July 30, 2021). arXiv: 2107.14795 [cs.LG].

[96] Andrew Jaegle et al. “Perceiver: General Perception with Iterative Attention”. In: *ICML* (Mar. 4, 2021). arXiv: 2103.03206 [cs.CV].

[97] Eric Jang, Shixiang Gu, and Ben Poole. “Categorical Reparameterization with Gumbel-Softmax”. In: *ICLR* (Nov. 3, 2016). arXiv: 1611.01144 [stat.ML].

[98] Armand Joulin and Tomas Mikolov. “Inferring Algorithmic Patterns with Stack-Augmented Recurrent Nets”. In: *NIPS* (Mar. 3, 2015), pp. 190–198. arXiv: http://arxiv.org/abs/1503.01007v4 [cs.NE].

[99] Norman P. Jouppi et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, June 2017. DOI: 10.1145/3079856.3080246.

[100] L. P. Kaelbling, M. L. Littman, and A. W. Moore. "Reinforcement Learning: A Survey". In: *Journal of Artificial Intelligence Research* 4 (May 1996), pp. 237–285. DOI: 10.1613/jair.301.

[101] Daniel Kahneman. *Thinking, Fast and Slow*. FARRAR STRAUSS GIROUX, Oct. 1, 2011. 499 pp. ISBN: 0374275637. URL: https://www.ebook.de/de/product/14769249/daniel_kahneman_thinking_fast_and_slow.html.

[102] Lukasz Kaiser et al. "Model-Based Reinforcement Learning for Atari". In: *ICLR* (Mar. 1, 2019). arXiv: 1903.00374 [cs.LG].

[103] Troy D. Kelley. "Symbolic and Sub-Symbolic Representations in Computational Models of Human Cognition". In: *Theory & Psychology* 13.6 (2003), pp. 847–860. DOI: 10.1177/0959354303136005.

[104] Daiki Kimura et al. "Neuro-Symbolic Reinforcement Learning with First-Order Logic". In: *EMNLP* (Oct. 21, 2021). arXiv: 2110.10963 [cs.AI].

[105] Diederik P Kingma and Max Welling. "Auto-Encoding Variational Bayes". In: *CoRR* (Dec. 20, 2013). arXiv: 1312.6114 [stat.ML].

[106] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *ICLR* (Dec. 22, 2014). arXiv: 1412.6980 [cs.LG].

[107] Ekaterina Komendantskaya. "Unification neural networks: unification by error-correction learning". In: *Logic Journal of IGPL* 19.6 (May 2010), pp. 821–847. DOI: 10.1093/jigpal/jzq012.

[108] Emile van Krieken, Erman Acar, and Frank van Harmelen. "Analyzing Differentiable Fuzzy Implications". In: *KR* (June 4, 2020). arXiv: 2006.03472 [cs.AI].

[109] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet classification with deep convolutional neural networks". In: *Communications of the ACM* 60.6 (May 2017), pp. 84–90. DOI: 10.1145/3065386.

[110] Ankit Kumar et al. "Ask Me Anything: Dynamic Memory Networks for Natural Language Processing". In: *ICML*. 2016, pp. 1378–1387. arXiv: 1506.07285 [cs.CL].

[111] Jonathan Lajus, Luis Galárraga, and Fabian Suchanek. "Fast and Exact Rule Mining with AMIE 3". In: *The Semantic Web*. Springer International Publishing, 2020, pp. 36–52. DOI: 10.1007/978-3-030-49461-2_3.

[112] Brenden M. Lake et al. "Building Machines That Learn and Think Like People". In: *Behavioral and Brain Sciences* 40 (Nov. 2016). DOI: 10.1017/s0140525x16001837. arXiv: 1604.00289 [cs.AI].

[113] David Landy, Colin Allen, and Carlos Zednik. "A perceptual account of symbolic reasoning". In: *Frontiers in Psychology* 5 (Apr. 2014). DOI: 10.3389/fpsyg.2014.00275.

[114] Mark Law, Alessandra Russo, and Krysia Broda. "The ILASP system for Inductive Learning of Answer Set Programs". In: (May 2020). arXiv: 2005.00904 [cs.AI].

[115] Mark Law et al. "FastLAS: Scalable Inductive Logic Programming Incorporating Domain-Specific Optimisation Criteria". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34.03 (Apr. 2020), pp. 2877–2885. DOI: 10.1609/aaai.v34i03.5678.

[116] Y. LeCun et al. "Backpropagation Applied to Handwritten Zip Code Recognition". In: *Neural Computation* 1.4 (Dec. 1989), pp. 541–551. DOI: 10.1162/neco.1989.1.4.541.

[117] Richard L. Lewis. "Cognitive modeling, symbolic". In: *The MIT encyclopedia of the cognitive sciences* (1999), pp. 525–527. URL: http://www-personal.umich.edu/~rickl/pubs/lewis-1999-mitecs.pdf.

[118] Yujia Li et al. "Graph Matching Networks for Learning the Similarity of Graph Structured Objects". In: (Apr. 2019). arXiv: 1904.12787 [cs.LG].

[119] Chen Liang et al. "Neural Symbolic Machines: Learning Semantic Parsers on Freebase with Weak Supervision". In: *ACL* (Oct. 31, 2016). arXiv: http://arxiv.org/abs/1611.00020v4 [cs.CL].

[120] Vladimir Lifschitz. *What is Answer Set Programming?* AAAI'08. Menlo Park, Calif: AAAI Press, 2008, pp. 1594–1597. ISBN: 9781577353683.

[121] Grace W. Lindsay. "Convolutional Neural Networks as a Model of the Visual System: Past, Present, and Future". In: *Journal of Cognitive Neuroscience* (Feb. 2020), pp. 1–15. DOI: 10.1162/jocn_a_01544.

[122] Zachary C. Lipton. "The Mythos of Model Interpretability". In: *Communications of the ACM* 61.10 (2018), pp. 36–43. DOI: 10.1145/3233231. arXiv: 1606.03490 [cs.LG].

[123] Dianbo Liu et al. "Discrete-Valued Neural Communication". In: *NeurIPS* (July 6, 2021). arXiv: 2107.02367 [cs.LG].

[124] Fei Liu and Julien Perez. "Gated End-to-End Memory Networks". In: *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*. Association for Computational Linguistics, 2017, pp. 1–10. DOI: 10.18653/v1/e17-1001. arXiv: 1610.04211 [cs.CL].

[125] Francesco Locatello et al. "Object-Centric Learning with Slot Attention". In: (June 2020). arXiv: 2006.15055 [cs.LG].

[126] Jelena Luketina et al. "A Survey of Reinforcement Learning Informed by Natural Language". In: *IJCAI* (June 10, 2019). arXiv: 1906.03926 [cs.LG].

[127] Thang Luong, Hieu Pham, and Christopher D. Manning. "Effective Approaches to Attention-based Neural Machine Translation". In: *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Aug. 17, 2015. DOI: 10.18653/v1/d15-1166. arXiv: 1508.04025 [cs.CL].

[128] Chris J. Maddison, Andriy Mnih, and Yee Whye Teh. "The Concrete Distribution: A Continuous Relaxation of Discrete Random Variables". In: (Nov. 2016). arXiv: 1611.00712 [cs.LG].

[129] Grzegorz Malinowski. *Many-valued logics*. Oxford England New York: Clarendon Press Oxford University Press, 1993. ISBN: 9780198537878.

[130] Robin Manhaeve et al. "DeepProbLog: Neural Probabilistic Logic Programming". In: (May 2018). arXiv: 1805.10872 [cs.AI].

[131] Jiayuan Mao et al. "The Neuro-Symbolic Concept Learner: Interpreting Scenes, Words, and Sentences From Natural Supervision". In: *ICLR* (Apr. 26, 2019). arXiv: 1904.12584 [cs.CV].

[132] Gary Marcus. "Deep Learning: A Critical Appraisal". In: (Jan. 2, 2018). arXiv: 1801.00631 [cs.AI].

[133] Hongyuan Mei et al. "Neural Datalog Through Time: Informed Temporal Modeling via Logical Specification". In: *Proceedings of the 37th International Conference on Machine Learning*. Ed. by Hal Daumé III and Aarti Singh. Vol. 119. Proceedings of Machine Learning Research. PMLR, June 30, 2020, pp. 6808–6819. arXiv: 2006.16723 [cs.LG]. URL: http://proceedings.mlr.press/v119/mei20a.html.

[134] Tomas Mikolov et al. "Distributed Representations of Words and Phrases and their Compositionality". In: *NIPS*. Oct. 16, 2013, pp. 3111–3119. arXiv: http://arxiv.org/abs/1310.4546v1 [cs.CL].

[135] Tim Miller. "Explanation in Artificial Intelligence: Insights from the Social Sciences". In: *Artificial Intelligence* 267 (2019), pp. 1–38. DOI: 10.1016/j.artint.2018.07.007. arXiv: 1706.07269 [cs.AI].

[136] Pasquale Minervini et al. "Differentiable Reasoning on Large Knowledge Bases and Natural Language". In: *AAAI* (Dec. 17, 2019). arXiv: 1912.10824 [cs.LG].

[137] Pasquale Minervini et al. "Learning Reasoning Strategies in End-to-End Differentiable Proving". In: *Proceedings of the 37th International Conference on Machine Learning* (July 13, 2020). arXiv: 2007.06477 [cs.AI].

[138] Pasquale Minervini et al. "Towards Neural Theorem Proving at Scale". In: *FAIM Workshop on Neural Abstract Machines and Program Induction* (July 21, 2018). arXiv: 1807.08204 [cs.AI].

[139] Marvin Minsky. *Perceptrons; an introduction to computational geometry*. Cambridge, Mass: MIT Press, 1969. ISBN: 0262130432.

[140] Arindam Mitra et al. "Declarative Question Answering over Knowledge Bases containing Natural Language Text with Answer Set Programming". In: *AAAI* (May 1, 2019). arXiv: 1905.00198 [cs.AI].

[141] Sarthak Mittal et al. "Learning to Combine Top-Down and Bottom-Up Signals in Recurrent Neural Networks with Attention over Modules". In: *ICML* (June 30, 2020). arXiv: 2006.16981 [cs.LG].

[142] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. DOI: 10.1038/nature14236.

[143] J. D. Monk. *Mathematical Logic*. Springer New York, Sept. 7, 1976. 548 pp. ISBN: 0387901701. URL: https://www.ebook.de/de/product/4248362/j_d_monk_mathematical_logic.html.

[144] Adam Morton and Jerry A. Fodor. *The Language of Thought.* Vol. 75. 3. Philosophy Documentation Center, Mar. 1978, p. 161. DOI: 10.2307/2025426.

[145] Stephen Muggleton and Luc de Raedt. "Inductive Logic Programming: Theory and methods". In: *The Journal of Logic Programming* 19-20 (May 1994), pp. 629–679. DOI: 10.1016/0743-1066(94)90035-3.

[146] Michael G. Müller et al. "A model for structured information representation in neural networks". In: (Nov. 11, 2016). arXiv: 1611.03698 [q-bio.NC].

[147] Allen Newell and Herbert A. Simon. "Computer science as empirical inquiry: symbols and search". In: *Communications of the ACM* 19.3 (Mar. 1976), pp. 113–126. DOI: 10.1145/360018.360022.

[148] Maximilian Nickel, Lorenzo Rosasco, and Tomaso Poggio. "Holographic Embeddings of Knowledge Graphs". In: *AAAI* (Oct. 16, 2015), pp. 1955–1961. arXiv: http://arxiv.org/abs/1510.04935v2 [cs.AI].

[149] Maxwell Nye et al. "Improving Coherence and Consistency in Neural Sequence Models with Dual-System, Neuro-Symbolic Reasoning". In: (July 6, 2021). arXiv: 2107.02794 [cs.AI].

[150] Aaron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu. "Neural Discrete Representation Learning". In: *NeurIPS* (Nov. 2, 2017). arXiv: 1711.00937 [cs.LG].

[151] Rasmus Berg Palm, Ulrich Paquet, and Ole Winther. "Recurrent Relational Networks". In: *NeurIPS* (Nov. 21, 2017). arXiv: 1711.08028 [cs.AI].

[152] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[153] Ali Payani and Faramarz Fekri. "Inductive Logic Programming via Differentiable Deep Neural Logic Networks". In: (June 2019). arXiv: 1906.03523 [cs.AI].

[154] Ali Payani and Faramarz Fekri. "Learning Algorithms via Neural Logic Networks". In: (Apr. 2019). arXiv: 1904.01554 [cs.LG].

[155] Judea Pearl. "Probabilities of causation: three counterfactual interpretations and their identification". In: *Synthese* 121.1/2 (1999), pp. 93–149. DOI: 10.1023/a:1005233831499.

[156] Judea Pearl. "The Seven Tools of Causal Inference with Reflections on Machine Learning". In: *Communications of the ACM* 62.3 (Feb. 2019), pp. 54–60. DOI: 10.1145/3241036.

[157] Jeffrey Pennington, Richard Socher, and Christopher Manning. "Glove: Global Vectors for Word Representation". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2014, pp. 1532–1543. DOI: 10.3115/v1/d14-1162.

[158] Fabio Petroni et al. "Language Models as Knowledge Bases?" In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Association for Computational Linguistics, 2019. DOI: 10.18653/v1/d19-1250.

[159] Michael Pfeiffer and Thomas Pfeil. "Deep Learning With Spiking Neurons: Opportunities and Challenges". In: *Frontiers in Neuroscience* 12 (Oct. 2018). DOI: 10.3389/fnins.2018.00774.

[160] Jean Piaget. *The Psychology of Intelligence*. London New York: Routledge, 2001. 216 pp. ISBN: 0415254019.

[161] Steven Pinker and Alan Prince. "On language and connectionism: Analysis of a parallel distributed processing model of language acquisition". In: *Cognition* 28.1-2 (Mar. 1988), pp. 73–193. DOI: 10.1016/0010-0277(88)90032-7.

[162] Dag Prawitz. *Natural Deduction: A Proof-Theoretical Study*. DOVER PUBN INC, Feb. 1, 2006. 113 pp. ISBN: 0486446557. URL: https://www.ebook.de/de/product/3560412/dag_prawitz_natural_deduction_a_proof_theoretical_study.html.

[163] Esteban Real et al. "AutoML-Zero: Evolving Machine Learning Algorithms From Scratch". In: *ICML* (Mar. 6, 2020). arXiv: 2003.03384 [cs.LG].

[164] Scott Reed and Nando de Freitas. "Neural Programmer-Interpreters". In: *ICLR* (Nov. 19, 2015). arXiv: http://arxiv.org/abs/1511.06279v4 [cs.LG].

[165] R. Reiter. "A logic for default reasoning". In: *Artificial Intelligence* 13.1-2 (Apr. 1980), pp. 81–132. DOI: 10.1016/0004-3702(80)90014-4.

[166] Michael Rescorla. "The Language of Thought Hypothesis". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Summer 2019. Metaphysics Research Lab, Stanford University, June 21, 2019. URL: https://plato.stanford.edu/archives/sum2019/entries/language-thought/.

[167] Antti Revonsuo and James Newman. "Binding and Consciousness". In: *Consciousness and Cognition* 8.2 (June 1999), pp. 123–127. DOI: 10.1006/ccog.1999.0393.

[168] John C. Reynolds. "Transformational systems and algebraic structure of atomic formulas". In: *Machine intelligence* 5 (1970), pp. 135–151.

[169] Ryan Riegel et al. "Logical Neural Networks". In: (June 2020). arXiv: 2006.13155 [cs.AI].

[170] Tim Rocktäschel and Sebastian Riedel. "End-to-End Differentiable Proving". In: *NIPS* (May 31, 2017), pp. 3791–3803. arXiv: 1705.11040 [cs.NE].

[171] Tim Rocktäschel et al. "Low-Dimensional Embeddings of Logic". In: *Proceedings of the ACL 2014 Workshop on Semantic Parsing*. Association for Computational Linguistics, 2014, pp. 45–49. DOI: 10.3115/v1/w14-2409.

[172] Tim Rocktäschel et al. "Reasoning about Entailment with Neural Attention". In: *ICLR* (Sept. 22, 2015). arXiv: 1509.06664 [cs.CL].

[173] Neal J. Roese. "Counterfactual thinking." In: *Psychological Bulletin* 121.1 (1997), pp. 133–148. DOI: 10.1037/0033-2909.121.1.133.

[174] F. Rosenblatt. *The Perceptron, a Perceiving and Recognizing Automaton Project Para*. Report: Cornell Aeronautical Laboratory. Cornell Aeronautical Laboratory, 1957.

[175] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach, Global Edition*. Addison Wesley, Nov. 28, 2018. ISBN: 1292153962. URL: https://www.ebook.de/de/product/25939961/stuart_russell_peter_norvig_artificial_intelligence_a_modern_approach_global_edition.html.

[176] S. N. Sivanandam S. N. Deepa. *Introduction to Genetic Algorithms*. Springer-Verlag GmbH, Oct. 24, 2007. 442 pp. ISBN: 3540731903. URL: https://www.ebook.de/de/product/11428707/s_n_deepa_s_n_sivanandam_introduction_to_genetic_algorithms.html.

[177] Mohammed Saeed et al. "RuleBert: Teaching Soft Rules to Pre-trained Language Models". In: *EMNLP-2021* (Sept. 24, 2021). arXiv: 2109.13006 [cs.AI].

[178] Tim Salimans et al. "Evolution Strategies as a Scalable Alternative to Reinforcement Learning". In: (Mar. 10, 2017). arXiv: 1703.03864 [stat.ML].

[179] Adam Santoro et al. "A simple neural network module for relational reasoning". In: *NeurIPS*. June 5, 2017, pp. 4974–4983. arXiv: 1706.01427 [cs.CL].

[180] Julian Schrittwieser et al. "Mastering Atari, Go, chess and shogi by planning with a learned model". In: *Nature* 588.7839 (Dec. 2020), pp. 604–609. DOI: 10.1038/s41586-020-03051-4.

[181] Catherine D. Schuman et al. "A Survey of Neuromorphic Computing and Neural Networks in Hardware". In: (May 19, 2017). arXiv: 1705.06963 [cs.NE].

[182] John R. Searle. "Minds, brains, and programs". In: *Readings in Cognitive Science*. Elsevier, 1988, pp. 20–31. DOI: 10.1016/b978-1-4832-1446-7.50007-8.

[183] Ramprasaath R. Selvaraju et al. "Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization". In: *2017 IEEE International Conference on Computer Vision (ICCV)*. IEEE, Oct. 2017, pp. 618–626. DOI: 10.1109/iccv.2017.74.

[184] Prithviraj Sen et al. "Neuro-Symbolic Inductive Logic Programming with Logical Neural Networks". In: (Dec. 6, 2021). arXiv: 2112.03324 [cs.AI].

[185] Minjoon Seo et al. "Query-Reduction Networks for Question Answering". In: *ICLR* (2017). arXiv: 1606.04582 [cs.CL].

[186] Luciano Serafini and Artur d'Avila Garcez. "Logic Tensor Networks: Deep Learning and Logical Reasoning from Data and Knowledge". In: *Proceedings of the 11th International Workshop on Neural-Symbolic Learning and Reasoning* (June 14, 2016). arXiv: http://arxiv.org/abs/1606.04422v2 [cs.AI].

[187] Marek Sergot. *Knowledge Representation*. Imperial College London. 2019. URL: https://www.doc.ic.ac.uk/~mjs/teaching/491.html (visited on 09/07/2021).

[188] Murray Shanahan. "A spiking neuron model of cortical broadcast and competition". In: *Consciousness and Cognition* 17.1 (Mar. 2008), pp. 288–303. DOI: 10.1016/j.concog.2006.12.005.

[189] Murray Shanahan. "Dynamical complexity in small-world networks of spiking neurons". In: *Physical Review E* 78.4 (Oct. 2008), p. 041924. DOI: 10.1103/physreve.78.041924.

[190] Murray Shanahan. "The brain's connective core and its role in animal cognition". In: *Philosophical Transactions of the Royal Society B: Biological Sciences* 367.1603 (Oct. 2012), pp. 2704–2714. DOI: 10.1098/rstb.2012.0128.

[191] Murray Shanahan et al. "An Explicitly Relational Neural Network Architecture". In: *ICML* (May 24, 2019). arXiv: 1905.10307 [cs.LG].

[192] Murray Shanahan et al. "Artificial Intelligence and the Common Sense of Animals". In: *Trends in Cognitive Sciences* 24.11 (Nov. 2020), pp. 862–872. DOI: 10.1016/j.tics.2020.09.002.

[193] Murray Shanahan et al. "Large-scale network organization in the avian forebrain: a connectivity matrix and theoretical analysis". In: *Frontiers in Computational Neuroscience* 7 (2013). DOI: 10.3389/fncom.2013.00089.

[194] Ehud Y. Shapiro. *Inductive inference of theories from facts*. Yale University, Department of Computer Science, 1981.

[195] Hikaru Shindo, Masaaki Nishino, and Akihiro Yamamoto. "Differentiable Inductive Logic Programming for Structured Examples". In: *AAAI 21* (Mar. 2021). arXiv: 2103.01719 [cs.AI].

[196] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529.7587 (Jan. 2016), pp. 484–489. DOI: 10.1038/nature16961.

[197] David Silver et al. "Mastering the game of Go without human knowledge". In: *Nature* 550.7676 (Oct. 2017), pp. 354–359. DOI: 10.1038/nature24270.

[198] Koustuv Sinha et al. "Compositional Language Understanding with Text-based Relational Reasoning". In: *Relational Representation Learning Workshop, NeurIPS* (Nov. 7, 2018). arXiv: 1811.02959 [cs.CL].

[199] Jake Snell, Kevin Swersky, and Richard S. Zemel. "Prototypical Networks for Few-shot Learning". In: *NeurIPS* (Mar. 15, 2017). arXiv: 1703.05175 [cs.LG].

[200] Richard Socher et al. "Reasoning with Neural Tensor Networks for Knowledge Base Completion". In: *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1*. NIPS'13. Lake Tahoe, Nevada: Curran Associates Inc., 2013, pp. 926–934. URL: http://papers.nips.cc/paper/5028-reasoning-with-neural-tensor-networks-for-knowledge-base-completion.pdf.

[201] Richard Socher et al. "Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank". In: *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*. Seattle, Washington, USA: Association for Computational Linguistics, Oct. 2013, pp. 1631–1642. URL: https://www.aclweb.org/anthology/D13-1170.

[202] Gustav Sourek et al. "Lifted Relational Neural Networks". In: (Aug. 20, 2015). arXiv: 1508.05128 [cs.AI].

[203] Robyn Speer, Joshua Chin, and Catherine Havasi. "ConceptNet 5.5: An Open Multilingual Graph of General Knowledge". In: *AAAI 31*. 2017, pp. 4444–4451. arXiv: 1612.03975 [cs.CL].

[204] Courtney J. Spoerer, Patrick McClure, and Nikolaus Kriegeskorte. "Recurrent Convolutional Neural Networks: A Better Model of Biological Object Recognition". In: *Frontiers in Psychology* 8 (Sept. 2017). DOI: 10.3389/fpsyg.2017.01551.

[205] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *J. Mach. Learn. Res.* 15.1 (Jan. 2014), pp. 1929–1958. ISSN: 1532-4435.

[206] Kenneth O. Stanley. "Compositional pattern producing networks: A novel abstraction of development". In: *Genetic Programming and Evolvable Machines* 8.2 (May 2007), pp. 131–162. DOI: 10.1007/s10710-007-9028-8.

[207] Kenneth O. Stanley and Risto Miikkulainen. "Evolving Neural Networks through Augmenting Topologies". In: *Evolutionary Computation* 10.2 (June 2002), pp. 99–127. DOI: 10.1162/106365602320169811.

[208] Kenneth O. Stanley et al. "Designing neural networks through neuroevolution". In: *Nature Machine Intelligence* 1.1 (Jan. 2019), pp. 24–35. DOI: 10.1038/s42256-018-0006-z.

[209] Sainbayar Sukhbaatar et al. "End-To-End Memory Networks". In: *NIPS*. Mar. 31, 2015, pp. 2440–2448. arXiv: 1503.08895 [cs.NE].

[210] Oyvind Tafjord, Bhavana Dalvi Mishra, and Peter Clark. "ProofWriter: Generating Implications, Proofs, and Abductive Statements over Natural Language". In: *ACL* (Dec. 24, 2020). arXiv: 2012.13048 [cs.CL].

[211] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. "Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks". In: (Feb. 28, 2015). arXiv: 1503.00075 [cs.CL].

[212] Neil C. Thompson et al. "The Computational Limits of Deep Learning". In: (July 10, 2020). arXiv: 2007.05558 [cs.LG].

[213] Seiya Tokui et al. "Chainer: a Next-Generation Open Source Framework for Deep Learning". In: *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*. 2015. URL: http://learningsys.org/papers/LearningSys_2015_paper_33.pdf.

[214] A. M. Turing. “COMPUTING MACHINERY AND INTELLIGENCE”. In: *Mind* LIX.236 (Oct. 1950), pp. 433–460. DOI: 10.1093/mind/lix.236.433.

[215] J. Marshall Unger and Terrence W. Deacon. *The Symbolic Species: The Co-Evolution of Language and the Brain*. Vol. 82. Wiley, 1998, p. 437. DOI: 10.2307/329984.

[216] Ashish Vaswani et al. “Attention Is All You Need”. In: *NeurIPS* (June 12, 2017). arXiv: 1706.03762 [cs.CL].

[217] Oriol Vinyals et al. “Matching Networks for One Shot Learning”. In: *NeurIPS* (June 13, 2016). arXiv: 1606.04080 [cs.LG].

[218] Yaqing Wang et al. “Generalizing from a Few Examples”. In: *ACM Computing Surveys* 53.3 (July 30, 2020), pp. 1–34. DOI: 10.1145/3386252. arXiv: 2007.15484 [cs.CV].

[219] Jason Weston, Sumit Chopra, and Antoine Bordes. “Memory Networks”. In: *ICLR* (2015). arXiv: 1410.3916 [cs.AI].

[220] Jason Weston et al. “Towards AI-Complete Question Answering: A Set of Prerequisite Toy Tasks”. In: *ICLR* (2016). arXiv: 1502.05698 [cs.AI].

[221] Caiming Xiong, Stephen Merity, and Richard Socher. “Dynamic Memory Networks for Visual and Textual Question Answering”. In: *ICML* (Mar. 4, 2016), pp. 2397–2406. arXiv: 1603.01417 [cs.NE].

[222] Yuan Yang and Le Song. “Learn to Explain Efficiently via Neural Logic Inductive Learning”. In: *ICLR* (Oct. 6, 2019). arXiv: 1910.02481 [cs.AI].

[223] Zhilin Yang et al. “XLNet: Generalized Autoregressive Pretraining for Language Understanding”. In: *NeurIPS* (June 19, 2019). arXiv: 1906.08237 [cs.CL].

[224] Zhun Yang, Adam Ishay, and Joohyung Lee. “NeurASP: Embracing Neural Networks into Answer Set Programming”. In: *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*. International Joint Conferences on Artificial Intelligence Organization, 2020. DOI: 10.24963/ijcai.2020/243.

[225] Kexin Yi et al. “CLEVRER: CoLlision Events for Video REpresentation and Reasoning”. In: *ICLR* (Oct. 3, 2019). arXiv: 1910.01442 [cs.CV].

[226] Kexin Yi et al. “Neural-Symbolic VQA: Disentangling Reasoning from Vision and Language Understanding”. In: *NeurIPS* (Oct. 4, 2018). arXiv: 1810.02338 [cs.AI].

[227] Vinicius Zambaldi et al. “Relational Deep Reinforcement Learning”. In: *ICLR* (June 5, 2018). arXiv: 1806.01830 [cs.LG].

[228] Wojciech Zaremba et al. “Learning Simple Algorithms from Examples”. In: *ICML*. Nov. 23, 2015, pp. 421–429. arXiv: http://arxiv.org/abs/1511.07275v2 [cs.AI].

[229] Jure Zbontar et al. “Barlow Twins: Self-Supervised Learning via Redundancy Reduction”. In: *ICML* (Mar. 4, 2021). arXiv: 2103.03230 [cs.CV].

[230] Jie Zhou et al. “Graph neural networks: A review of methods and applications”. In: *AI Open* 1 (2020), pp. 57–81. DOI: 10.1016/j.aiopen.2021.01.001.

Appendix A

Soft Unification

A.1 Training Curves

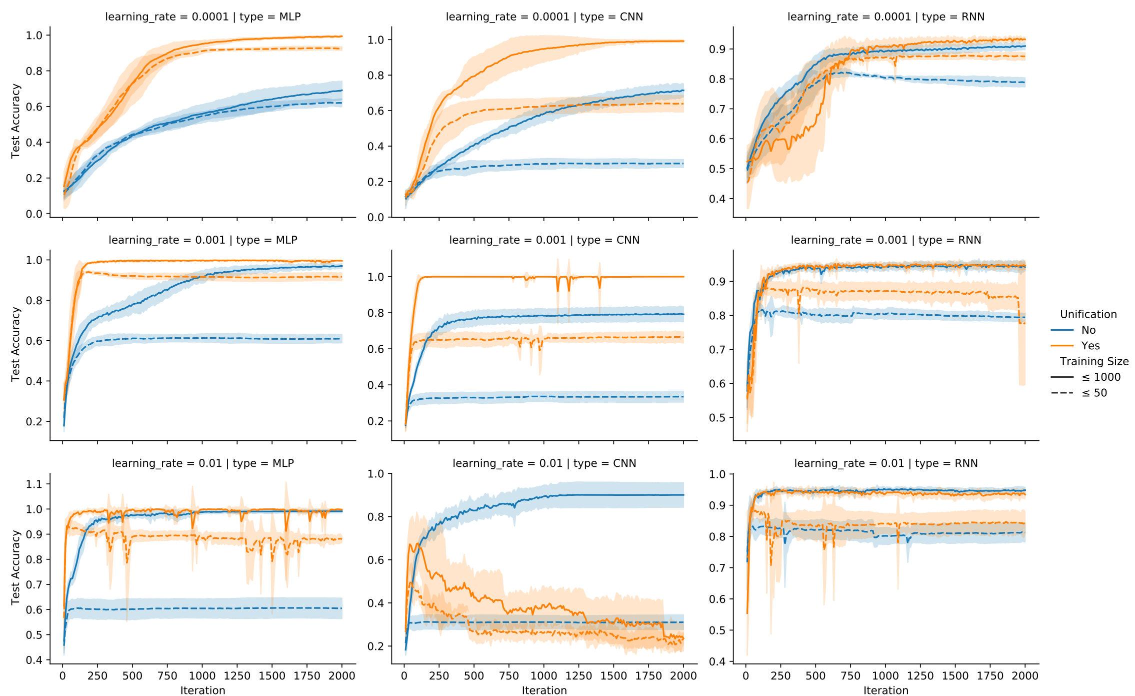


Figure A.1: Results of Unification MLP, CNN and RNN with different learning rates, complements Figs. 5.4 and 5.5. We observe that the improvement is maintained except at 0.01 for which our approach degrades in training performance. With the more real-world movie sentiment dataset, we can see that our approach maintains a noisier (fluctuating) training curve.

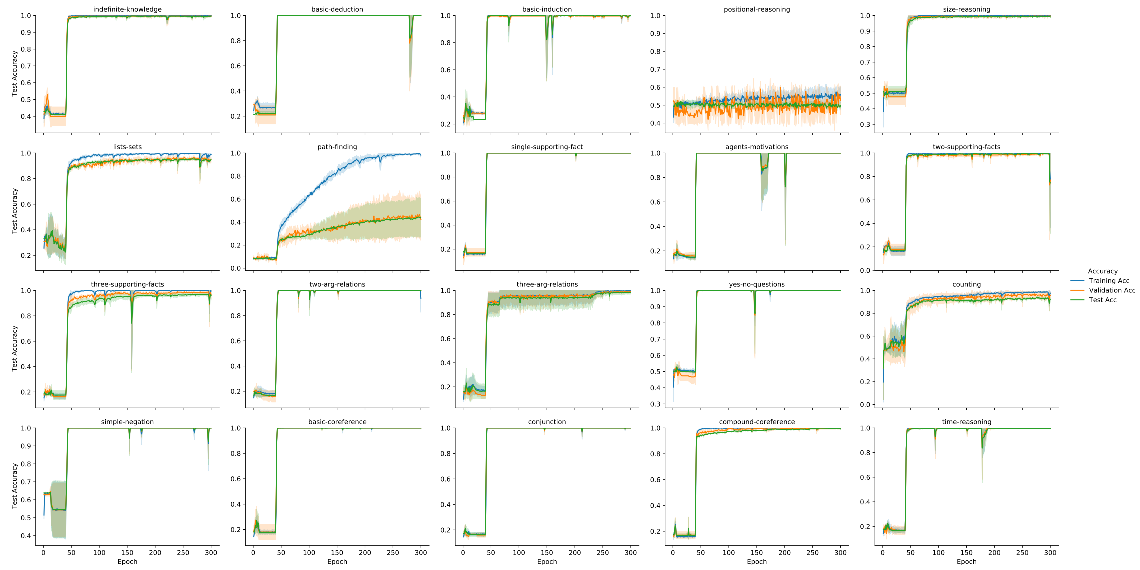


Figure A.2: Training curves for UMN on the bAbI dataset with strong supervision, 1 invariant and 1k training size. The initial 40 epochs pre-train the memory network; then unification is enabled which causes the sudden jump in the training curves. After unification is enabled, our approach learns to solve the tasks through soft unification within 5 epochs for most tasks. Tasks involving positional reasoning and path finding still prove difficult emphasising the dependency of our approach on the upstream memory network f which struggles to solve or over-fits.

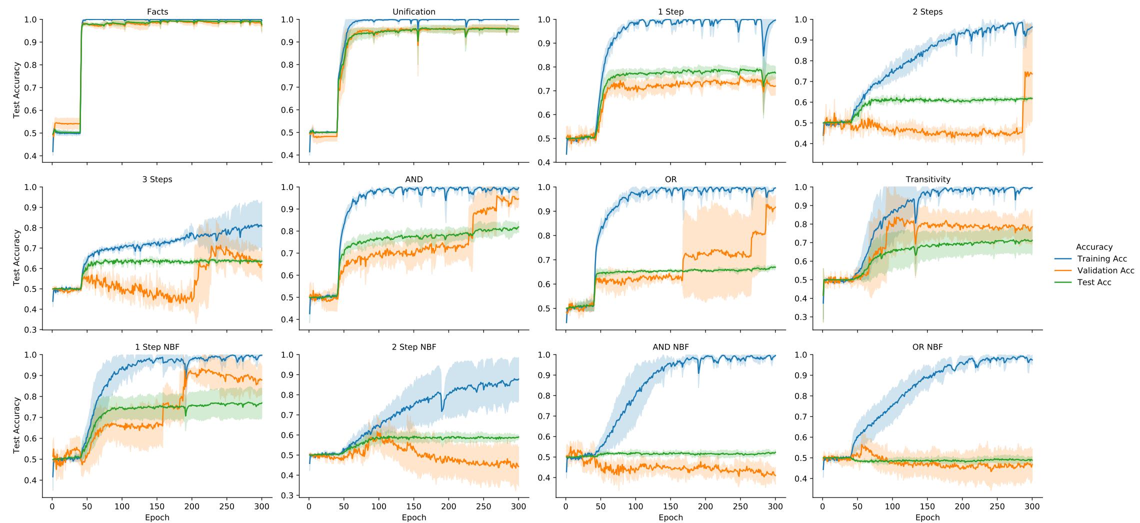


Figure A.3: Training curves for UMN on the logic dataset with strong supervision, 1 invariant and 2k training size. The initial 40 epochs pre-train the memory network and then unification is enabled. We observe that besides the very basic tasks that involve just Facts or Unification (these are single iteration tasks), our approach tends to over-fit.

A.2 Further Results

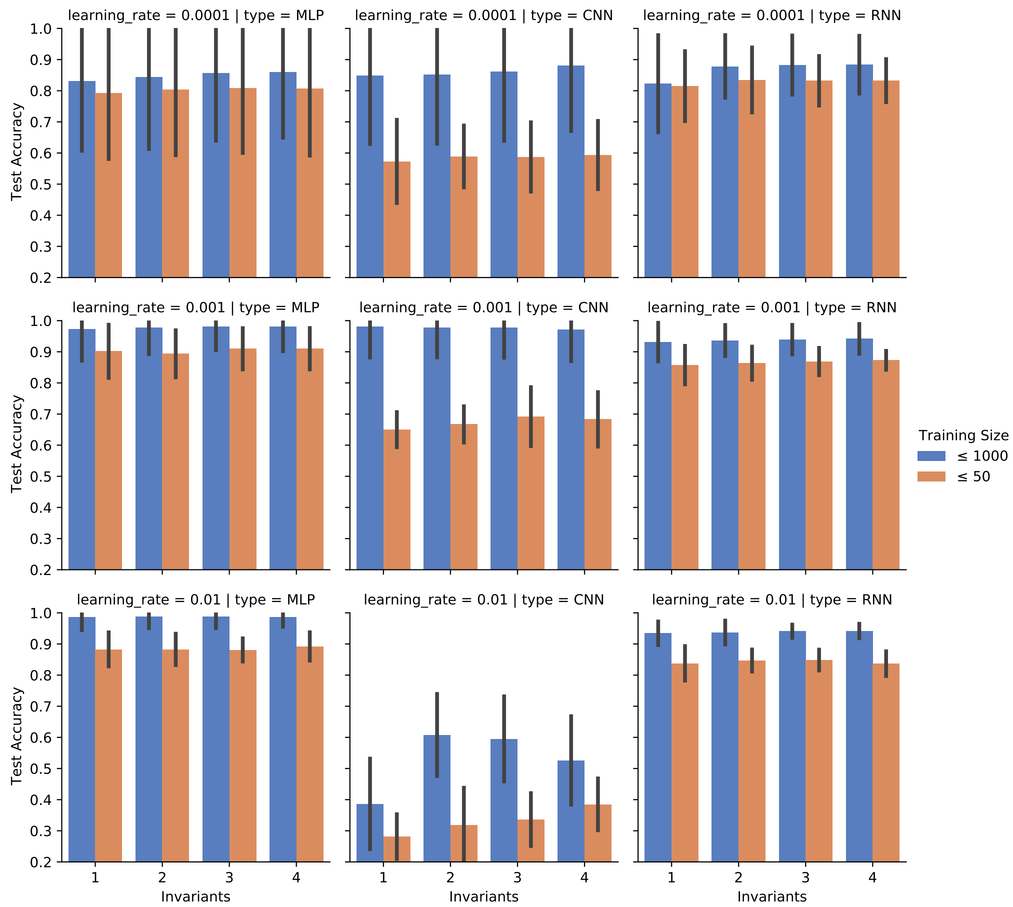


Figure A.4: Results of Unification MLP, CNN and RNN on increasing number of invariants with different learning rates, complements Fig. 5.6. There is no impact on performance when more invariants per task are given. We speculate that the models converge to using 1 invariant because the datasets can be solved with a single invariant due to the regularisation applied on ψ zeroing out unnecessary invariants.

Table A.1: Individual task error rates on bAbI tasks for Unification Memory Networks, complements Table 5.3. Following previous work, these are the best error rates out of 3 runs and are obtained by taking the test accuracy at the epoch for which the validation accuracy is highest. For the variance across runs, you can refer to the sample training curves for each task in Fig. A.2.

Supervision	Weak		Strong		
# Invs	1	3	1	3	3
Training Size	1k	1k	1k	1k	50
1	0.0	0.0	0.0	0.0	1.1
2	62.3	60.4	0.1	0.4	40.4
3	58.8	63.7	1.2	1.3	52.1
4	0.0	0.0	0.0	0.0	36.9
5	1.9	1.6	0.5	1.6	29.7
6	0.0	0.1	0.0	0.0	15.4
7	20.5	22.3	6.4	7.7	22.4
8	7.4	7.7	4.2	2.9	31.9
9	0.3	0.0	0.0	0.0	20.6
10	0.1	0.5	0.2	0.3	26.5
11	0.0	0.0	0.0	0.0	21.1
12	0.0	0.0	0.0	0.0	23.5
13	0.4	4.7	0.0	0.2	5.6
14	15.3	17.4	0.1	0.1	57.3
15	17.8	0.0	0.0	0.0	0.0
16	52.7	53.3	0.0	0.0	45.4
17	39.9	49.3	49.5	48.4	45.8
18	7.2	7.9	0.3	0.8	10.9
19	90.4	90.7	38.9	67.8	86.2
20	0.0	0.0	0.0	0.0	1.8
Mean	18.8	19.0	**5.1**	6.6	28.7
Std	27.7	28.0	13.6	18.0	21.8
# $>$ 5%	10	9	3	3	17

***V*:sandra** went back to the ***V*:bathroom**
is ***V*:sandra** in the ***V*:bathroom**
yes

Figure A.5: bAbI task 6, yes or no questions. The invariant captures the varying components of the story which are the person and the location they have been, similar to task 1 of the bAbI dataset. However, the invariant captures nothing about how this task is actually solved by the upstream network f, i.e. how f uses the interpolated unified representation.

$$\begin{aligned} \boldsymbol{V}\textbf{:m}\ (\ \boldsymbol{V}\textbf{:e}\ ) &\ \vdash \boldsymbol{V}\textbf{:m}\ (\ \boldsymbol{V}\textbf{:e}\ ) \\ \boldsymbol{V}\textbf{:a}\ (\ \boldsymbol{V}\textbf{:w}\ ,\ \boldsymbol{V}\textbf{:e}\ ) &\ \vdash \boldsymbol{V}\textbf{:a}\ (\ \boldsymbol{V}\textbf{:w}\ ,\ \boldsymbol{V}\textbf{:e}\ ) \\ \boldsymbol{V}\textbf{:m}\ (\ \texttt{T}\ ) &\ \vdash \boldsymbol{V}\textbf{:m}\ (\ \texttt{c}\ ) \\ \boldsymbol{V}\textbf{:x}\ (\ \texttt{A}\ ) \leftarrow \textit{not}\ \boldsymbol{V}\textbf{:q}\ (\ \texttt{A}\ ) &\ \vdash \boldsymbol{V}\textbf{:x}\ (\ \boldsymbol{V}\textbf{:z}\ ) \end{aligned}$$

Figure A.6: Invariants learned on tasks 1, 2 and 11 with arity 1 and 2 from the logical reasoning dataset. Last invariant on task 11 lifts the example around the negation by failure, denoted as *not*, capturing its semantics. We present the context on the left and the query on the right, $C \vdash q$ means $f(C,q) = 1$ as described in Sections 3.1 and 5.2.

Table A.2: Comparison of individual task error rates (%) on the bAbI [220] dataset of the best run, complements Table 5.3. We preferred 1k results if a model had experiments published on both 1k and 10k for data efficiency. We present our approach (UMN) with a single invariant for weak and strong supervision, taken from Table A.1.

Support	Weak							Strong		
Size	1k					10k		1k		10k
Model	N2N	GN2N	EntNet	QRN	UMN	DMN+	DNC	MemNN	UMN	DMN
Ref	[209]	[124]	[79]	[185]	Ours	[221]	[71]	[219]	Ours	[110]
1	0.0	0.0	0.7	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	8.3	8.1	56.4	0.5	62.3	0.3	0.4	0.0	0.1	1.8
3	40.3	38.8	69.7	1.2	58.8	1.1	1.8	0.0	1.2	4.8
4	2.8	0.4	1.4	0.7	0.0	0.0	0.0	0.0	0.0	0.0
5	13.1	1.0	4.6	1.2	1.9	0.5	0.8	2.0	0.5	0.7
6	7.6	8.4	30.0	1.2	0.0	0.0	0.0	0.0	0.0	0.0
7	17.3	17.8	22.3	9.4	20.5	2.4	0.6	15.0	6.4	3.1
8	10.0	12.5	19.2	3.7	7.4	0.0	0.3	9.0	4.2	3.5
9	13.2	10.7	31.5	0.0	0.3	0.0	0.2	0.0	0.0	0.0
10	15.1	16.5	15.6	0.0	0.1	0.0	0.2	2.0	0.2	2.5
11	0.9	0.0	8.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1
12	0.2	0.0	0.8	0.0	0.0	0.0	0.0	0.0	0.0	0.0
13	0.4	0.0	9.0	0.3	0.4	0.0	0.1	0.0	0.0	0.2
14	1.7	1.2	62.9	3.8	15.3	0.2	0.4	1.0	0.1	0.0
15	0.0	0.0	57.8	0.0	17.8	0.0	0.0	0.0	0.0	0.0
16	1.3	0.1	53.2	53.4	52.7	45.3	55.1	0.0	0.0	0.6
17	51.0	41.7	46.4	51.8	39.9	4.2	12.0	35.0	49.5	40.6
18	11.1	9.2	8.8	8.8	7.2	2.1	0.8	5.0	0.3	4.7
19	82.8	88.5	90.4	90.7	90.4	0.0	3.9	64.0	38.9	65.5
20	0.0	0.0	2.6	0.3	0.0	0.0	0.0	0.0	0.0	0.0
Mean	13.9	12.7	29.6	11.3	18.8	**2.8**	3.8	6.7	5.1	6.4
# $>$ 5%	11	10	15	5	10	1	2	4	3	2

Table A.3: Individual task error rates (%) of UMN against IMA (Section 4.1) with 2k training size, complements Table 5.4. Following previous work, we present the best task error rates across 3 runs. Each error rate is obtained by taking the test accuracy at the epoch when validation accuracy is highest. For the variance across runs, refer to the sample training curves of each task in Fig. A.3.

Model	UMN					IMA	
Training Size	2k				100	2k	
Supervision	Weak		Strong			Weak	Strong
# Invs	1	3	1	3	3	-	
Facts	1.6	0.9	0.2	0.3	37.0	2.8	3.8
Unification	2.4	3.0	2.1	8.8	45.0	9.7	7.2
1 Step	48.9	45.2	21.1	18.7	49.2	42.6	36.3
2 Steps	49.7	49.1	37.6	36.9	48.4	49.5	36.7
3 Steps	48.9	49.9	35.4	36.9	46.6	48.3	35.9
AND	35.1	35.6	16.4	18.9	48.7	40.3	21.4
OR	37.2	38.1	32.3	32.9	45.9	38.4	25.3
Transitivity	50.0	50.0	25.5	23.2	49.4	48.6	49.3
1 Step NAF	30.8	30.8	19.6	23.2	46.8	39.1	34.2
2 Steps NAF	49.9	50.0	38.8	48.0	49.0	49.5	34.1
AND NAF	48.5	48.6	49.8	49.7	49.4	48.8	45.5
OR NAF	49.2	50.4	49.8	50.0	49.7	48.4	48.8
Mean	37.7	37.6	**27.4**	29.0	47.1	38.8	31.5
# $>$ 5%	10	10	10	11	12	11	11

Table A.4: Individual task error rates (%) of UMN with extra configurations on the logical reasoning dataset as well as the reported results of baselines DMN and IMA taken from Section 4.3. The baseline results are trained on a different instance of the dataset that uses the same data generating script from Section 3.1. When the logic programs are restricted to arity 1, the length of the predicates become fixed, e.g. $p(a)$, which simplifies soft unification by removing any ambiguity. In this case, the identity matrix would produce the correct value assignments for variables in our approach and as a result we see a clear improvement over arity 2 version of the dataset. Note that one cannot generate transitivity tasks with only arity 1 predicates. Generating more logic programs improves the overall performance of our approach which fails to solve only 5 tasks with strong supervision and a single invariant.

Size	2k				20k					100	40k	
Support	Weak		Strong		Weak		Strong				Weak	
Arity	1	2	1	2	1	2	1	2	2	2	2	2
# Invs / Model	1	3	1	3	1	3	1	1	3	3	DMN	IMA
Facts	1.2	0.9	0.0	0.4	0.0	0.0	0.0	0.0	0.0	33.5	0.0	0.0
Unification	0.0	10.3	0.0	10.8	0.0	0.0	0.0	0.0	0.0	41.3	13.0	10.0
1 Step	50.3	49.8	4.4	20.0	1.2	27.8	0.1	1.3	5.7	50.2	26.0	2.0
2 Steps	47.5	50.0	5.7	35.0	37.2	47.8	0.0	29.7	28.7	49.9	33.0	5.0
3 Steps	47.6	49.2	10.4	38.7	39.6	45.6	0.0	26.0	26.1	48.3	23.0	6.0
AND	31.3	37.4	10.7	16.4	29.8	29.0	0.2	0.4	1.2	50.0	20.0	5.0
OR	25.2	38.1	21.0	35.0	20.5	30.2	4.4	20.6	17.4	47.6	13.0	3.0
Transitivity		50.0		26.6		39.6		5.0	6.0	49.2	50.0	50.0
1 Step NAF	46.4	38.7	3.8	28.8	1.1	21.6	0.1	1.1	8.0	47.6	21.0	2.0
2 Steps NAF	48.5	48.9	7.7	39.6	30.4	48.2	0.1	33.4	28.7	50.3	15.0	4.0
AND NAF	51.0	50.1	43.1	48.6	29.4	44.2	0.1	1.3	40.1	49.5	16.0	8.0
OR NAF	51.4	48.4	50.8	47.3	47.6	47.8	21.3	27.6	30.5	47.3	25.0	14.0
Mean	36.4	39.3	14.3	28.9	21.5	31.8	**2.4**	12.2	16.0	47.1	21.2	9.1
Std	18.7	15.9	16.4	14.1	17.1	16.7	6.1	13.2	13.6	4.7	12.3	13.4
# $>$ 5%	9	11	7	11	7	10	1	5	9	12	11	5

Appendix B

Pixels to Rules

B.1 Semi-symbolic Layer

The semi-symbolic layer builds on the semantics of how a regular single-layer perceptron from Definition 6 behaves in order to achieve logical $\wedge$ semantics. Let's start with the formulation for the pre-activation value for a single-layer perceptron:

$$\sum_i w_i x_i + \beta = z \tag{B.1}$$

Now, suppose we are interested in obtaining AND gate semantics with a pre-activation value z if all inputs are true and $-z$ if one of the inputs are false. To simplify the derivation, suppose the magnitudes of the weights are equal $\forall_{ij} w_i = w_j$:

$$\sum_i |w_i| + \beta = z \tag{B.2}$$

$$\sum_i |w_i| - |w_i| + \beta = -z \tag{B.3}$$

where a true input means $x_i = 1.0$ if the weight is positive or $x_i = -1.0$ otherwise. We use the absolute value of the weights since the sum of the weights multiplied by matching inputs would yield the sum of the absolute value of the weights. Solving the above system of equations for β, we obtain:

$$2\beta = |w_i| - 2\sum_i |w_i| \tag{B.4}$$

$$\beta = \frac{|w_i|}{2} - \sum_i |w_i| \tag{B.5}$$

Table B.1: Summary of symbols for the DNF layer and model formulation

Symbol	Shape	Component	Comment
X	$W \times H \times c$	Input CNN	The input to the overall neuro-symbolic architecture in Fig. 6.4.
$\boldsymbol{O}$	$n \times d$	Input CNN	The set of objects with real-valued vector representations, Eq. (6.4).
$\boldsymbol{s}^t, \boldsymbol{a}^t$	n	Selection	The relevance score and attention map of each object iteration t from Eqs. (6.6) and (6.7).
$\boldsymbol{O}^*$	$m \times d$	Selection	Selected objects from Eq. (6.8).
$\text{unary}(X,i)$	scalar	Relations	The ith unary relation of a given object representation X, Eq. (6.9).
$\text{binary}(X,Y,j)$	scalar	Relations	The jth binary relation of a given object representations X and Y.
P^*	$mp^1 + m(m-1)p^2$	Relations	Vector of all ground facts obtained from the selected objects, Eq. (6.11).
$\text{SL}_{\delta \to 1}$	h	DNF	A conjunctive SL with h many outputs.
$\text{SL}_{\delta \to -1}$	r	DNF	A disjunctive SL with r many outputs.
$\text{DNF}(P^*)$	r	DNF	The DNF layer over propositional inputs from Eq. (6.12).
V^*	$k \times (lp^1 + l(l-1)p^2)$	DNF	Object to variable binding permutation tensor with k many permutations and l variables.
$\mathbb{V}_{j:l}$		DNF	Variables j to l from an ordered set of l many variables.
$\text{DNF}(V^*)$	r	DNF	The DNF layer over first-order variable inputs, Eq. (6.13).
C	$m \times W \times H \times c + 1$	Reconstruction	The image reconstruction tensor with the extra masking channel, Eq. (6.14).

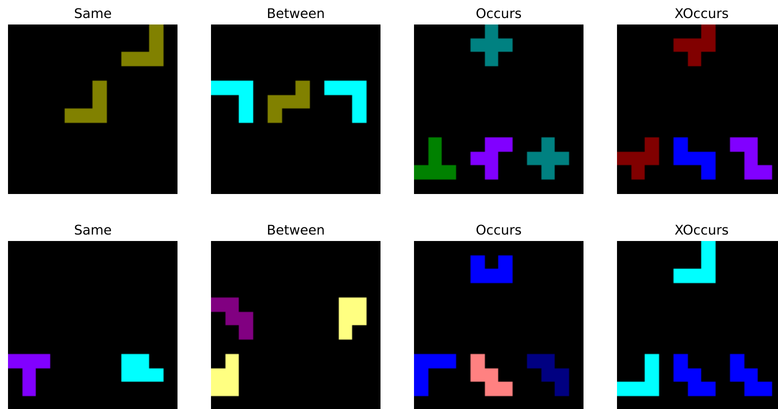
Since the weights would potentially not be equal during training, we replace the first term to obtain the final equation Eq. (6.2) presented in Section 6.1:

$$\beta = \max_i |w_i| - \sum_i |w_i| \tag{B.6}$$

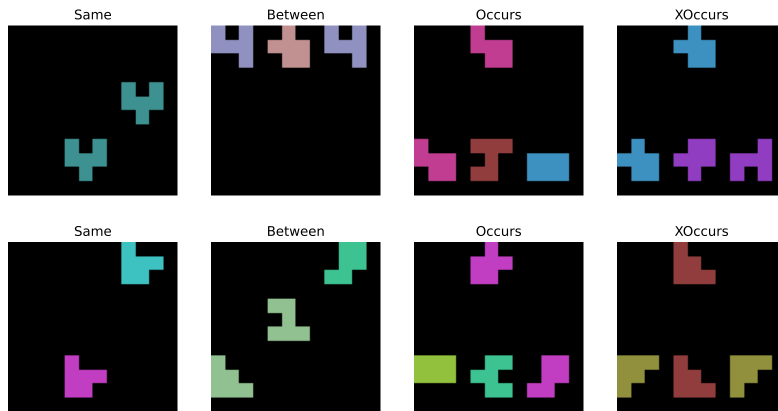
which preserves the desired AND gate semantics. The derivation for the disjunctive case is identical and leads to the bias terms flipped, i.e. sum - max. A fully functional implementation of the semi-symbolic layer in TensorFlow is shown in Listing B.1.

Listing B.1: Implementation of semi-symbolic layer in Tensorflow

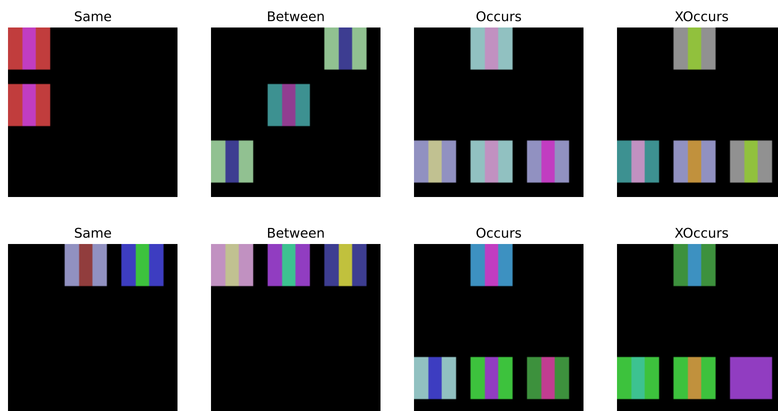
```python
import tensorflow as tf

def semi_symbolic(in_tensor: tf.Tensor, kernel: tf.Tensor, delta: float):
    """Compute semi-symbolic layer outputs of a given input tensor."""
    # in_tensor (..., H), kernel (H,), delta [1,-1]
    abs_kernel = tf.math.abs(kernel) # (H,)
    bias = tf.reduce_max(abs_kernel) - tf.reduce_sum(abs_kernel) # ()
    conjuncts = tf.reduce_sum(in_tensor * kernel, -1) + delta*bias
    return tf.nn.tanh(conjuncts)
```



(a) Example pentominoes where each shape consists of 5 pixels organised in a 3x3 grid.



(b) Example hexominoes where each shape consists of 6 pixels with unseen colours to pentominoes.



(c) Example striped shapes where each object has 9 pixels with a striped colour pattern.

Figure B.1: Further samples from the Relations Game dataset. The top and bottom rows are for true and false cases respectively.

Table B.2: All hyper-parameters used for training the DNF models

Key	Value	Comment
# batch updates Rels. Game	30k	Number of training batch updates used.
# batch updates Graph	10k	
# batch updates per eval	200	How often to evaluate model on test dataset, number of batch updates divided by this quantity gives the number of epochs.
learning rate	0.001	Optimiser learning rate, fixed throughout training.
batch size Rels. Game	64	Batch size used for training.
batch size Graph	128	
# of repeated runs	5	Every deep learning model on both datasets are trained 5 times with the same configuration.
Input noise stddev Rels. Game	0.01	Noise added to input image in the relations game dataset.
rng seed Rels. Game	42	Random number generator seed used in relations game data augmentation.
rng seeds Graph	.	We use the 7 winning numbers of EuroMillions 25 December 2020.

B.2 Dataset Details

B.3 Hyper-parameters

There are three main categories of hyper-parameters used: model, dataset and training. We cover all of them in Tables B.2 and B.3. The magnitude of the semantic gate selector hyper-parameter $|\delta|$ presented in Section 6.1 is gradually adjusted during training according to a fixed exponential schedule. For the image classifier model, we start with 0.01 and increase to 1.0 with an exponential rate of 1.1 while for the subgraph set isomorphism dataset we start higher 0.1 and use the same rate of 1.1. We start the gate at a higher value for the graph isomorphism dataset since the input is already symbolic, i.e. $\forall_i x_i \in \{-1,1\}$. The sign of δ is pre-determined by the layer type, conjunctive or disjunctive within a DNF layer from Section 6.3.

B.4 Training Curves

The training curves for every deep model trained on every dataset and hyperparameter configuration, can be found in Figs. B.2 to B.9. For relations game training curves, the slight dip in accuracy around epoch 70 when trained with only 100 examples corresponds to the point at which the semantic gate $|\delta|$ becomes close to 1. This indicates that if the model is over-fitting, it almost has to relearn the task with the desired semantics since this phenomenon is less pronounced with more training examples and almost absent in the graph dataset. We also observe mode collapses in the recursive configuration of the DNF model (DNF-r) in which the accuracy suddenly drops when the semantic gate $|\delta|$ becomes closer to 1 around epoch 70. We believe this is due to the recursion in the model making a recovery to the desired disjunctive normal form semantics more difficult. The DNF-r either performs well with some runs surviving the saturation $|\delta| = 1.0$ and some collapsing. All the hyper-parameters used in training can be found in Table B.2. The models are trained on a shared pool of computers all having Intel Core i7 CPUs. Note that the run times of the deep models may vary depending on the shared workload on the worker machine. After training, we prune and threshold the weights to obtain DNF+t variants as described in Section 6.1.

Table B.3: All hyper-parameters used for constructing DNF the models

Key	Value	Comment
# selected objects Same	2	Number of objects selected in the Same task in the relations game dataset.
# variables Same	2	Number of variables used in the final DNF layer used to learn rules in the Relations Game dataset.
# selected objects Between	3	
# variables Between	3	
# selected objects Occurs	4	
# variables Occurs	2	
# selected objects XOccurs	4	
# variables XOccurs	4	
# selected objects All	4	
# variables All	4	
# unary relations	8	Number of unary relations computed for the selected objects prior to DNF layer, the relations box in Fig. 6.4.
# binary relations	16	
# relations PrediNet	16	Number of relations computed by the shared weights of each head.
# heads PrediNet	.	The number of heads is adjusted to match the output size of the relations computed prior to the DNF layer for fair comparison. Let s, u, b be the number of selected objects, unary and binary relations used in the DNF models, then the number of heads is $h = su + s(s-1)b$. We adjust this so that the relation representation sizes are equal for a fairer comparison.
PrediNet key size	32	The key size used for computing dot-product attention maps.
PrediNet output hidden size	64	Size of the hidden layer of the output MLP used in PrediNet model.
input CNN hidden size	32	The number of filters used by the input CNN for both DNF and PrediNet models.
input CNN activation	ReLU	
# variables hidden DNF	2	Number of variables that can appear in the rules learnt by the hidden DNF layer of the DNF-h models.
# rule definitions hidden DNF	4	Maximum number of rule definitions per predicate in the hidden layer.
# invented predicates hidden DNF	14	The number of predicates learnt by the hidden DNF layer. Specifically, 2 nullary, 4 unary and 8 binary predicates.
# rule definitions DNF Rels. Game	8	The target label can be defined at most by 8 different rules.
# invented predicates recursive DNF	7	The number of extra predicates learnt by the DNF layer in the recursive configuration DNF-r. Specifically, 1 nullary, 2 unary and 4 binary predicates.
# iterations recursive DNF	2	
# rule definitions recursive DNF	2	Maximum number of rule definitions per predicate including the label in the recursive DNF layer, DNF-r.

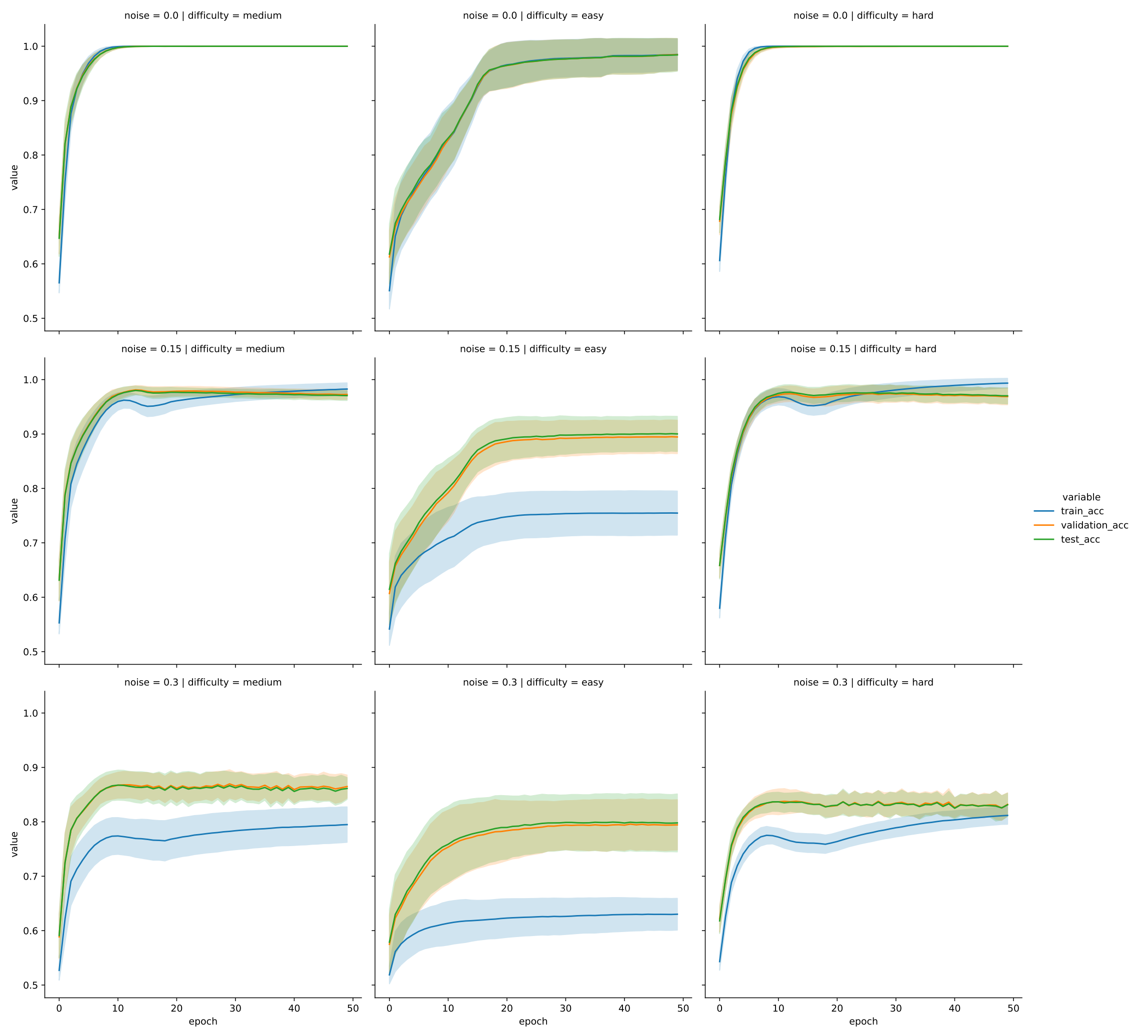


Figure B.2: Training curves for DNF layer on the subgraph set isomorphism task. The model is training for 100k batch updates logging every 200 steps. This gives a total of 50 epochs shown on the x axis.

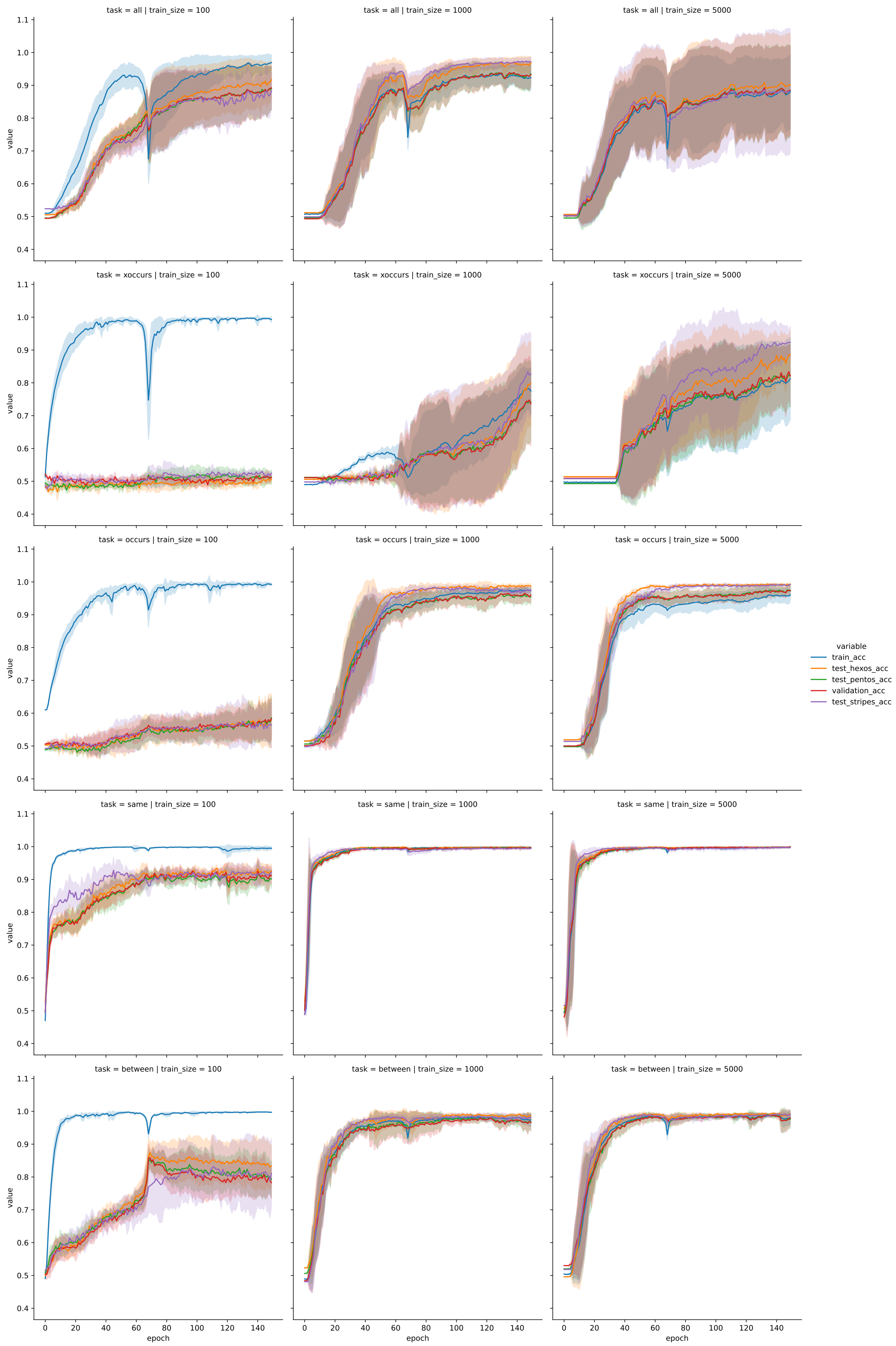


Figure B.3: Training curves for the DNF model on the Relations Game dataset

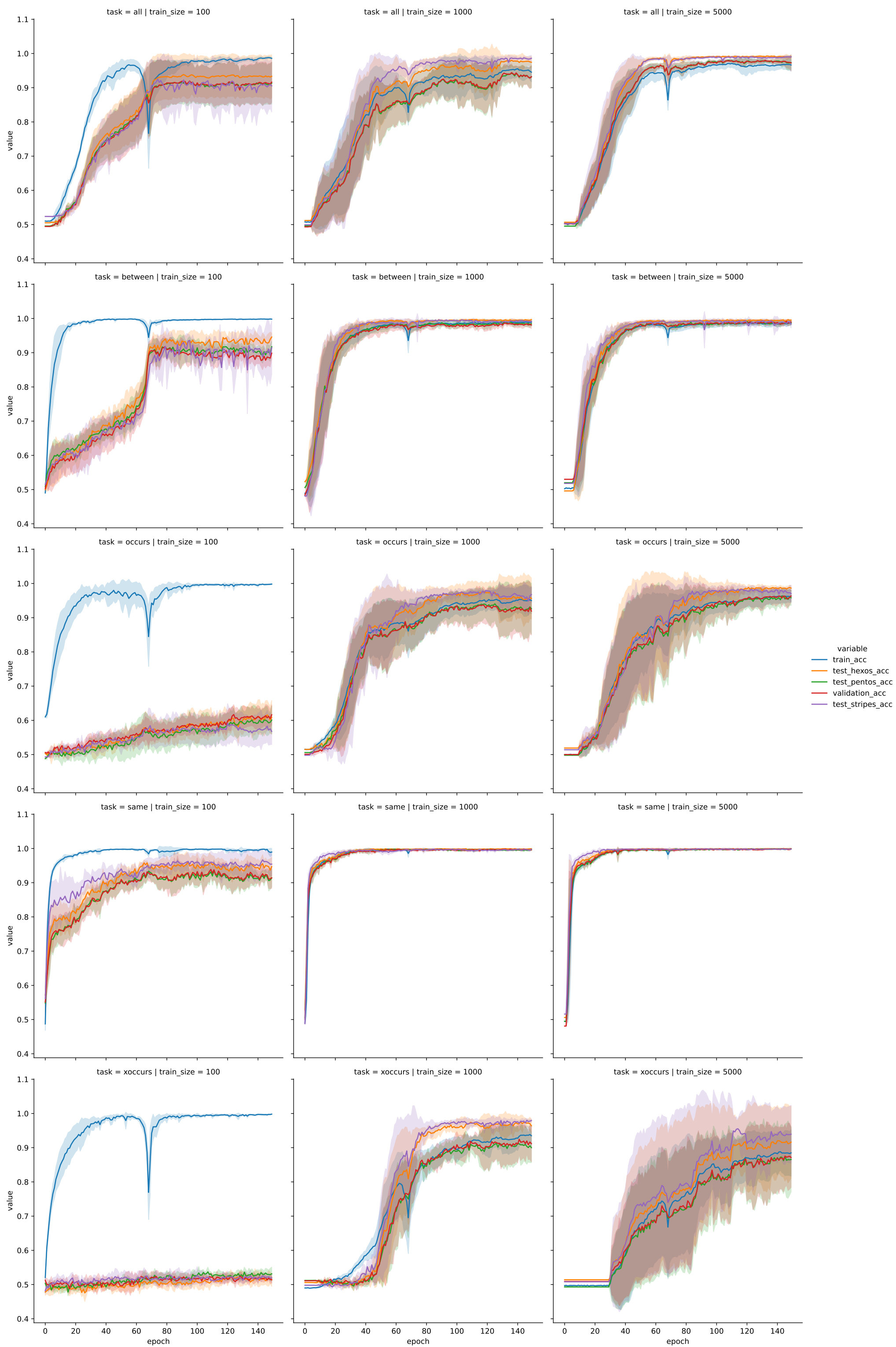


Figure B.4: Training curves for the DNF model with a hidden layer (DNF-h) on the Relations Game

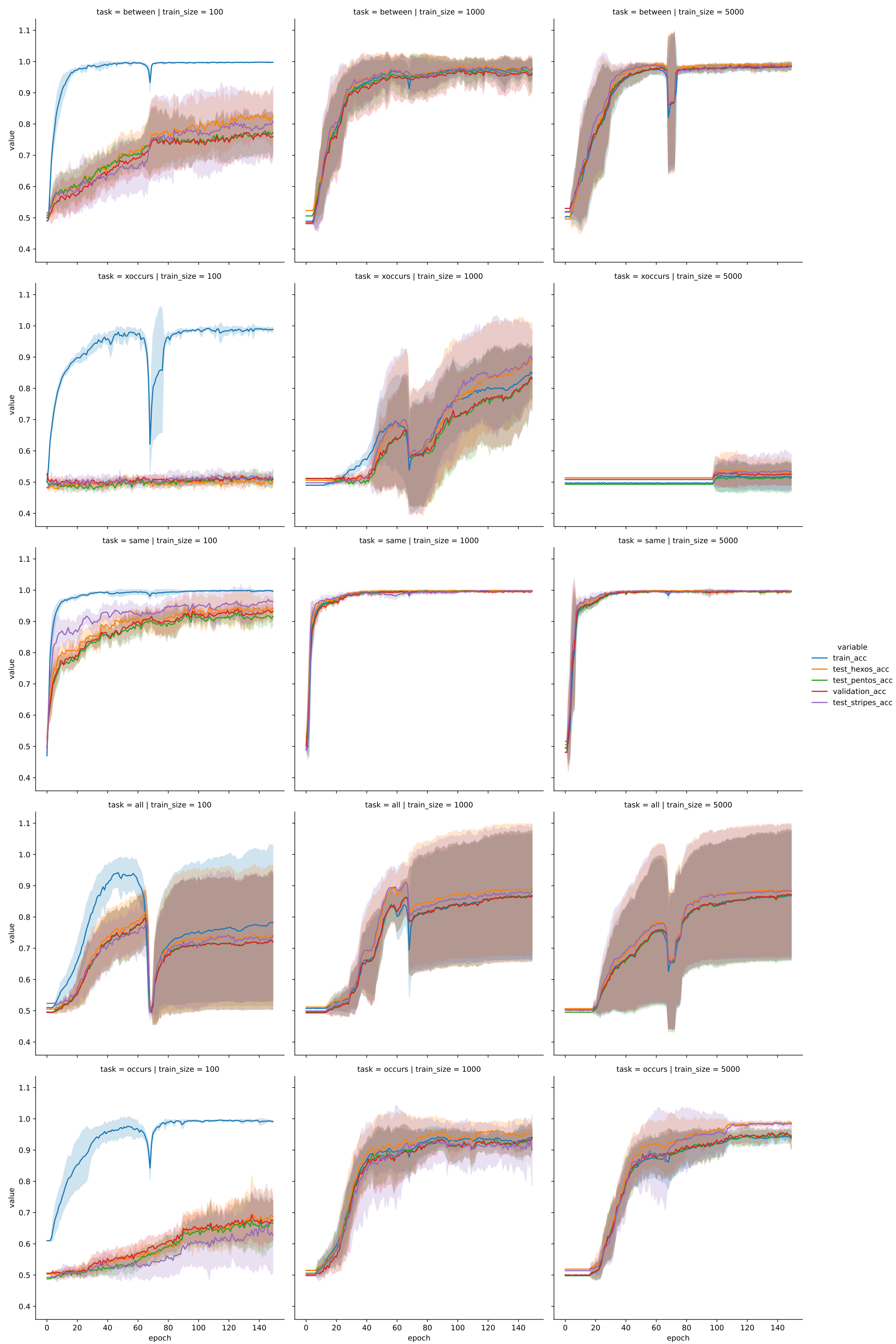


Figure B.5: Training curves for the recursive DNF model (DNF-r) on the Relations Game

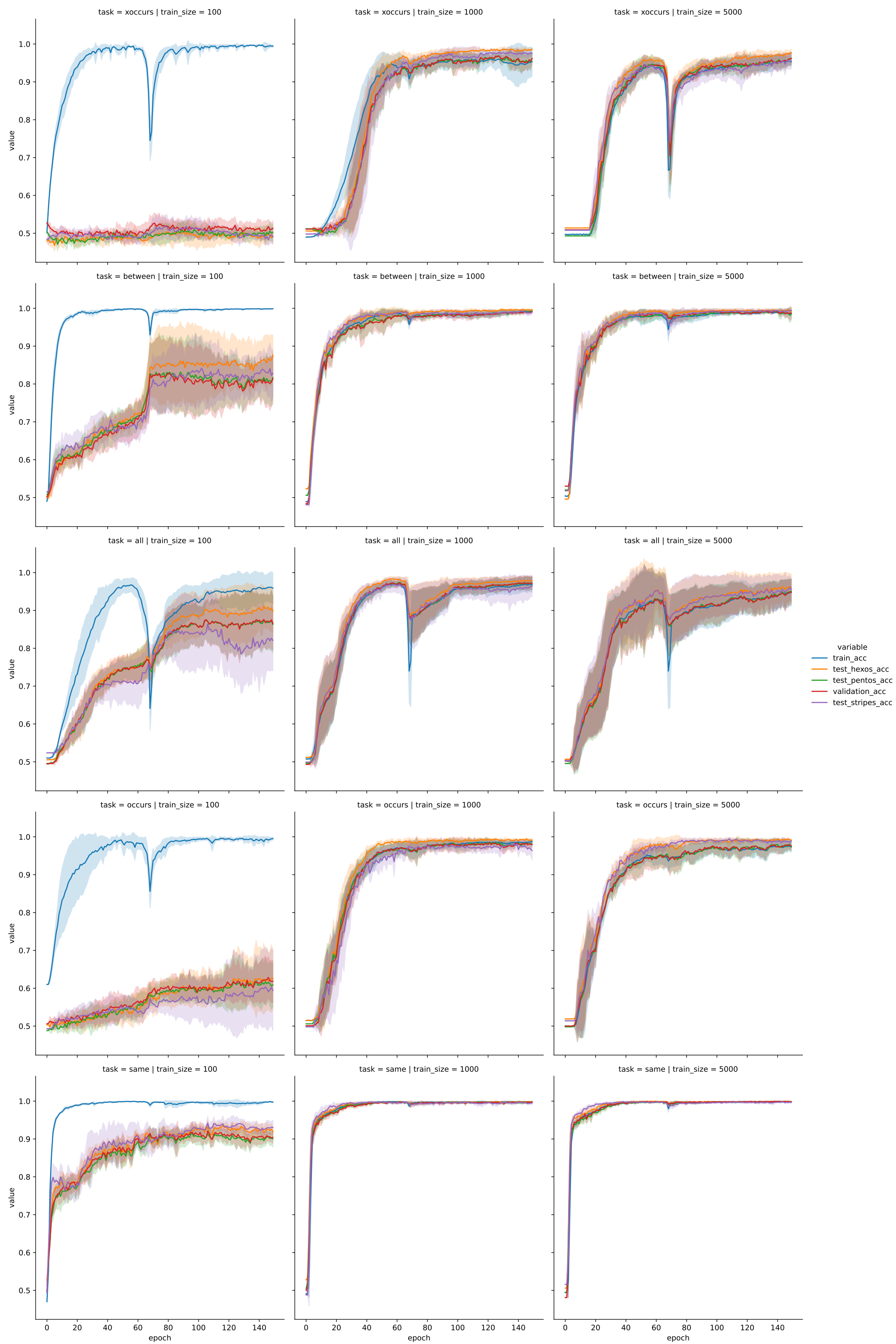


Figure B.6: Training curves for the DNF model with image reconstruction loss (DNF-i)

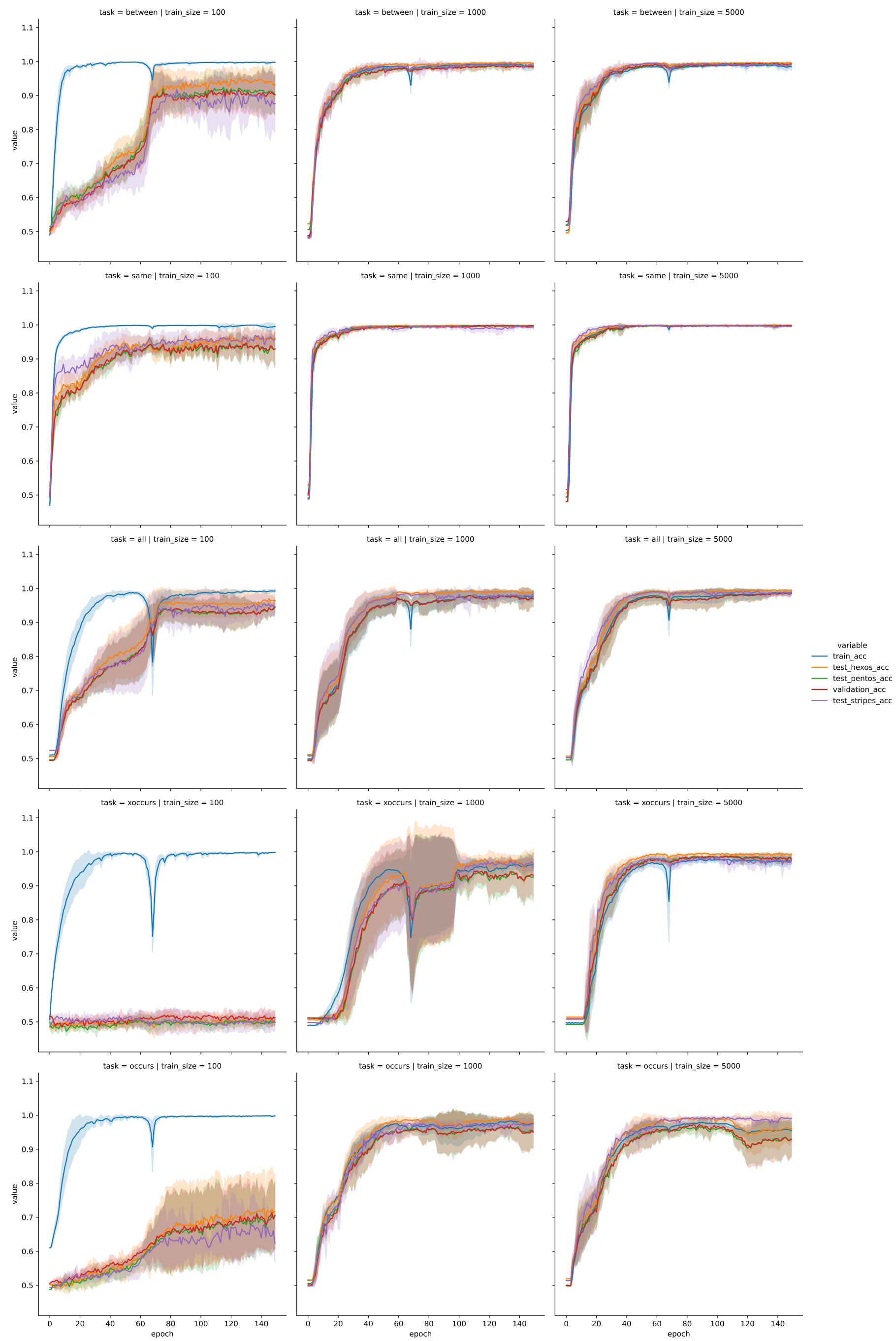


Figure B.7: Training curves for the DNF model with a hidden layer and image reconstruction (DNF-hi)

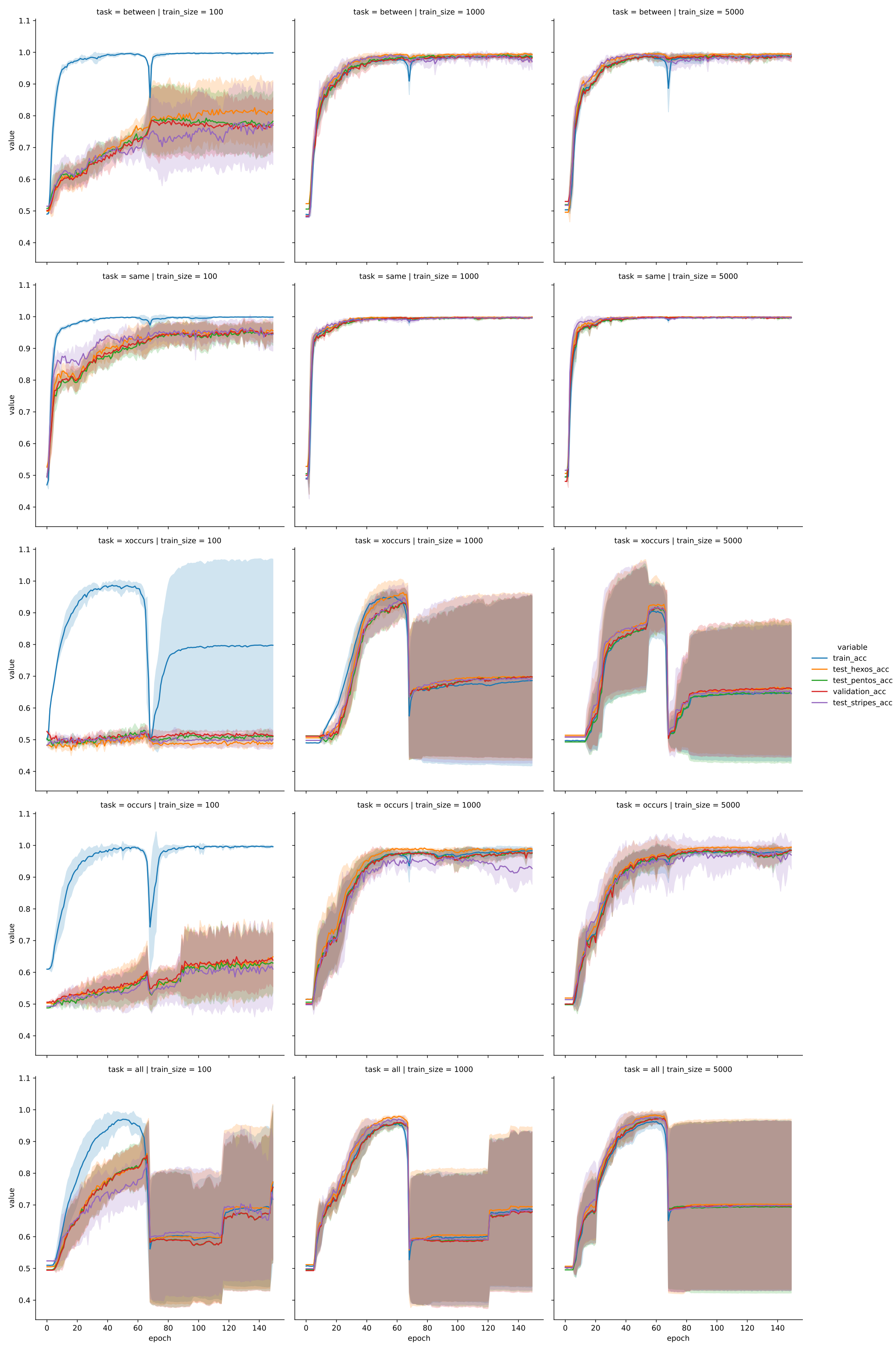


Figure B.8: Training curves for the recursive DNF model with reconstruction loss (DNF-ri)

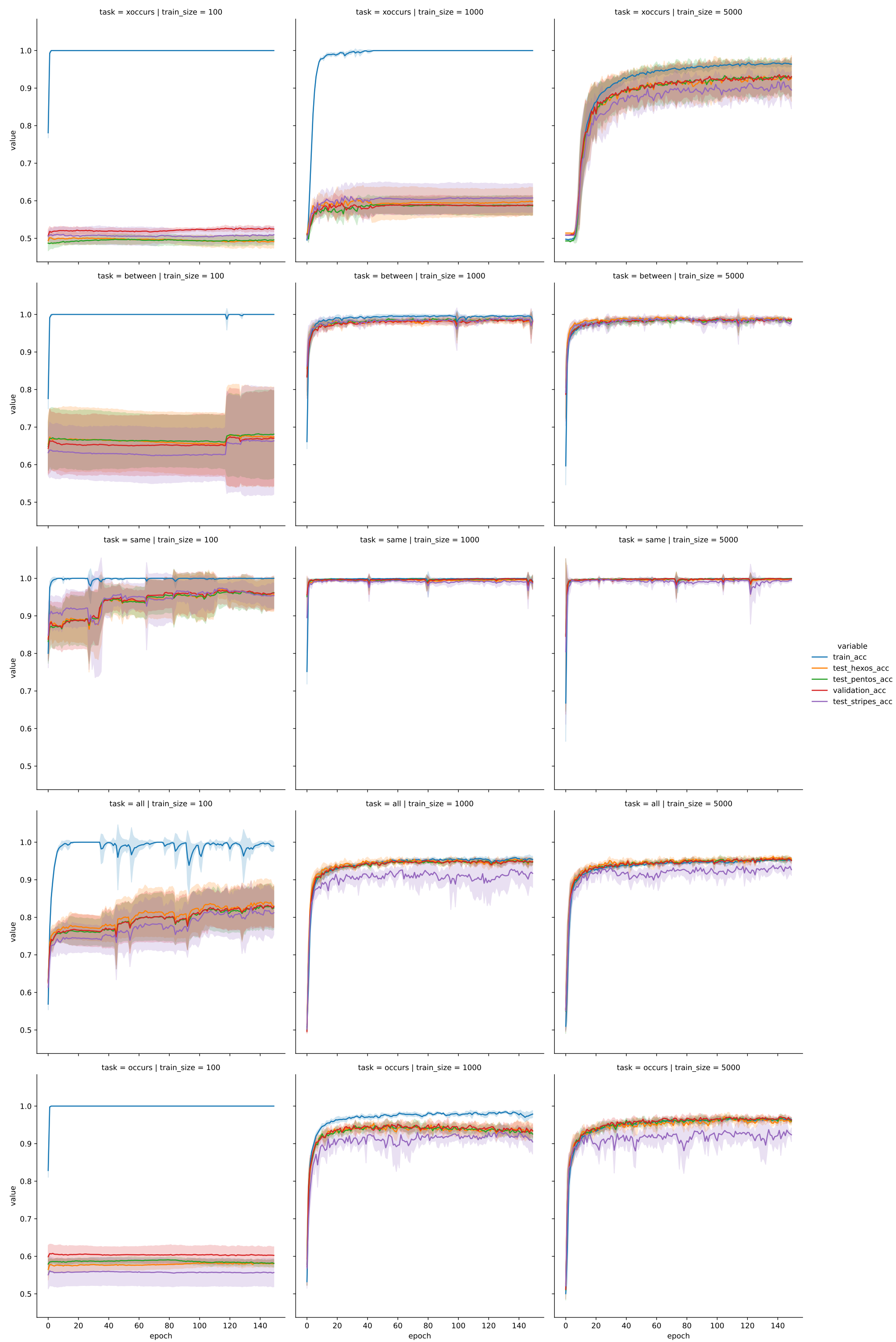


Figure B.9: Training curves for PrediNet on Relations Game

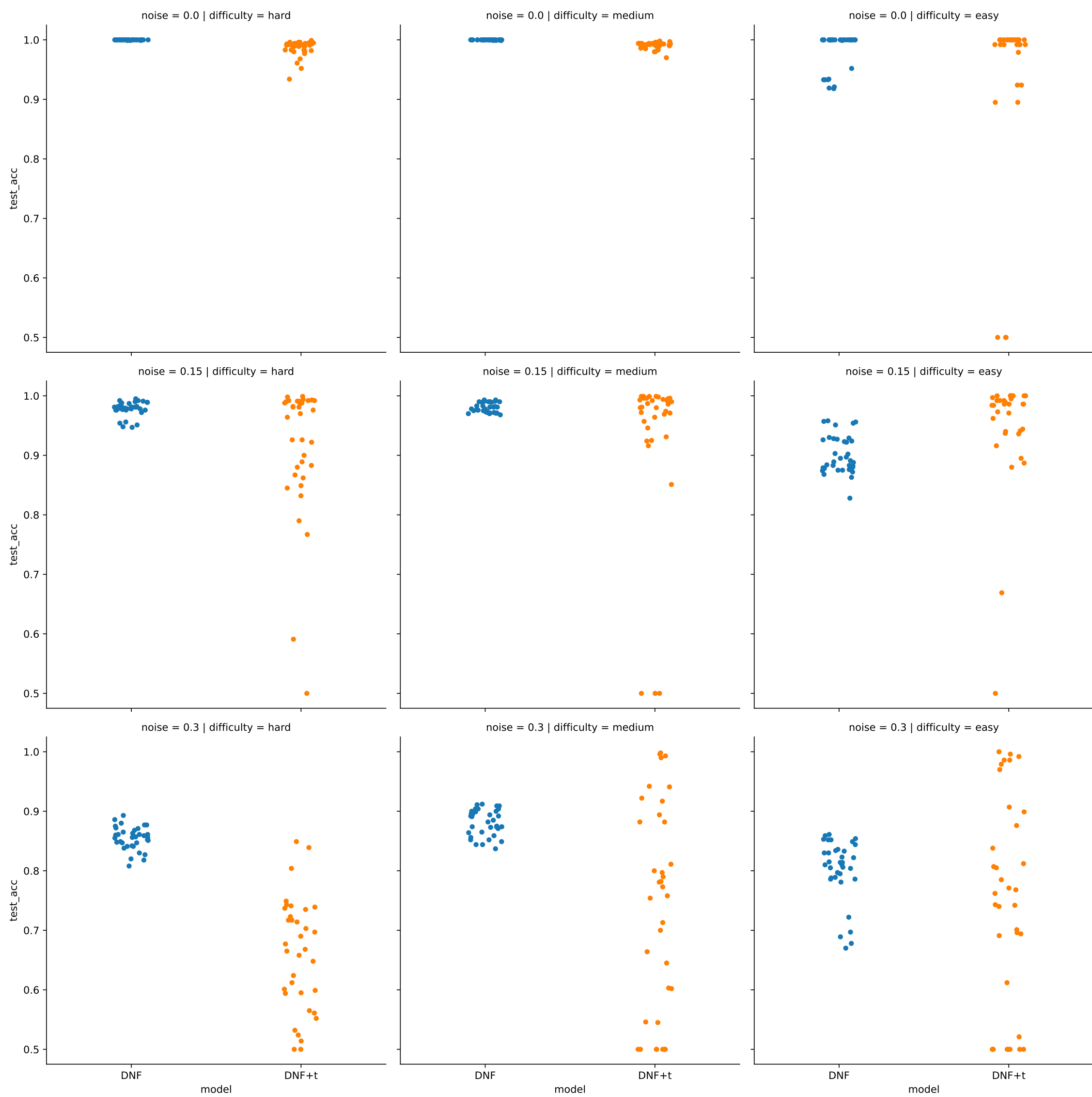


Figure B.10: All the results for the DNF layer on the subgraph set isomorphism task. There are 315 unique runs: 5 runs for each 7 seeds for every difficulty (3) with 3 levels of noise. For each run, the thresholded results are shown as DNF+t.

B.5 Further Results

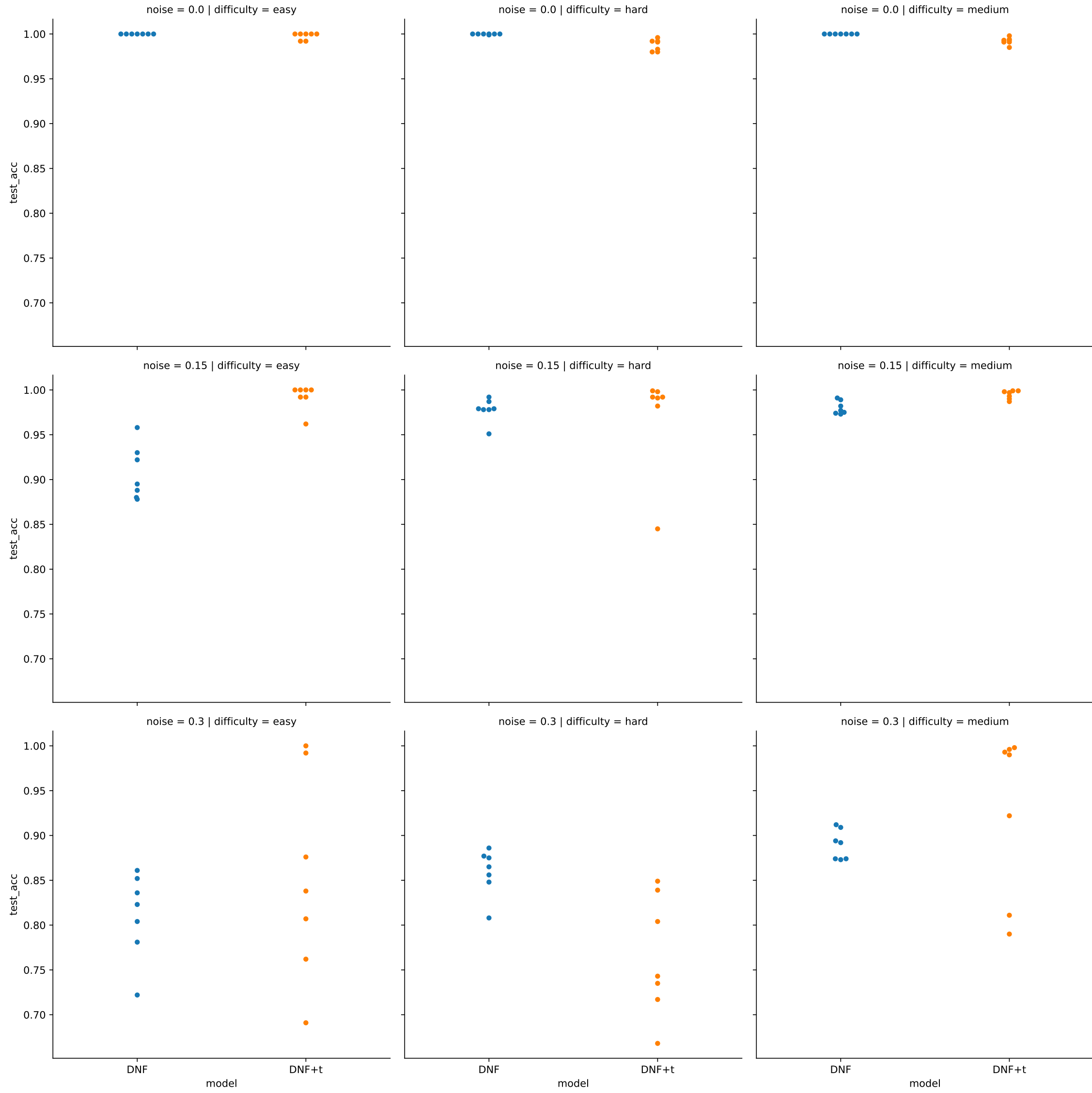


Figure B.11: Best results filtered out of 5 runs from Fig. B.10. There is one point for every random seed used, 7 in total. These are all the data points used in Table B.4.

Table B.4: Full results of the best runs of the DNF layer on the subgraph set isomorphism task

		median				mad			
		test acc		validation acc		test acc		validation acc	
	model	DNF	DNF+t	DNF	DNF+t	DNF	DNF+t	DNF	DNF+t
difficulty	noise								
easy	0.00	1.000	1.000	1.000	1.000	0.000	0.003	0.000	0.002
	0.15	0.895	1.000	0.909	1.000	0.025	0.009	0.022	0.010
	0.30	0.823	0.838	0.809	0.827	0.036	0.089	0.030	0.085
hard	0.00	1.000	0.991	1.000	1.000	0.000	0.006	0.000	0.000
	0.15	0.979	0.992	0.981	0.995	0.008	0.036	0.008	0.034
	0.30	0.865	0.743	0.877	0.769	0.019	0.056	0.012	0.044
medium	0.00	1.000	0.993	1.000	1.000	0.000	0.003	0.000	0.000
	0.15	0.977	0.997	0.983	1.000	0.006	0.004	0.005	0.002
	0.30	0.892	0.990	0.902	0.997	0.014	0.075	0.017	0.074

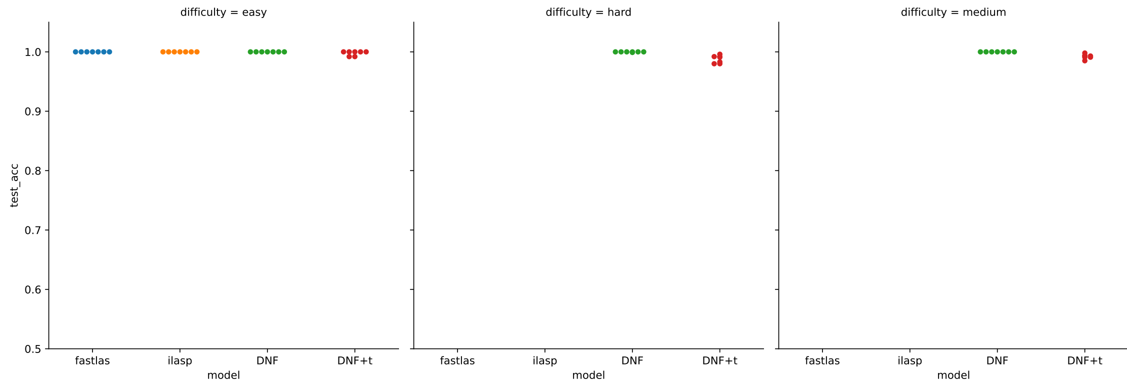


Figure B.12: All the data points used in the aggregation of Table 6.2. There is one point for each random seed used (7).

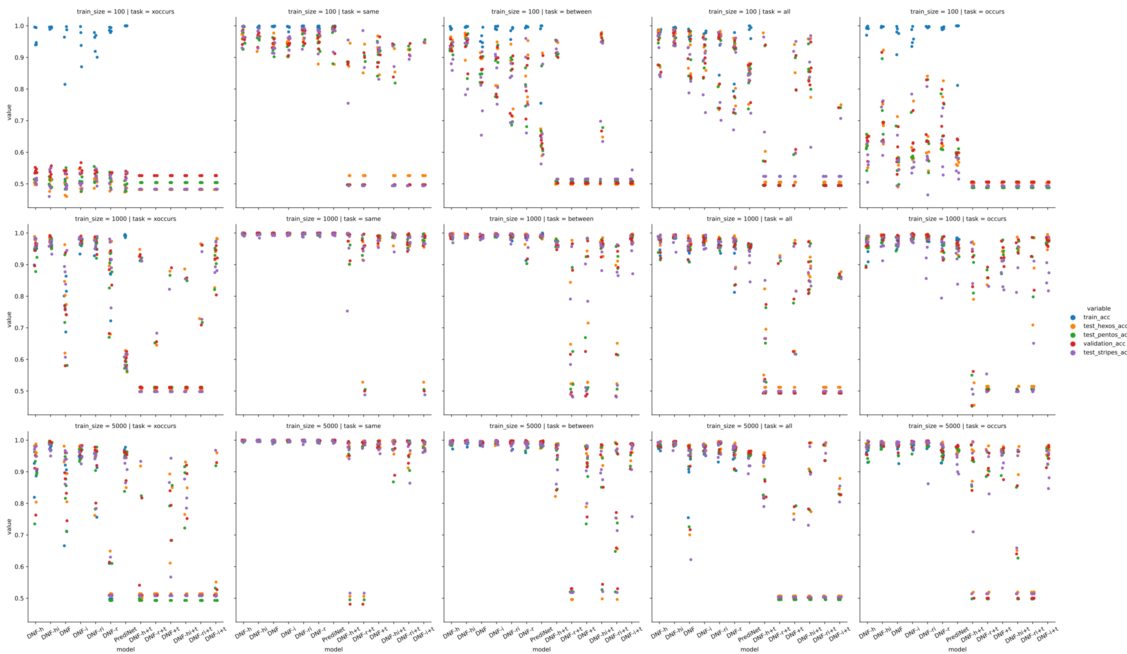


Figure B.13: All of the runs on the Relations Game dataset, there are 75 PrediNet, 225 DNF and 225 DNF with image reconstruction configurations trained in total. For each DNF configuration there is also the thresholded results noted with suffix +t in the model name.

Table B.5: Median of test accuracies of all DNF configurations across all Relations Game setups with mean absolute deviation. Here we observe that the hidden layer version DNF-h has a slight advantage.

Model	Hexominoes	Pentominoes	Stripes
DNF	0.953±0.186	0.932±0.183	0.926±0.182
DNF-h	**0.964**±0.176	**0.943**±0.172	**0.949**±0.177
DNF-r	0.868±0.214	0.863±0.212	0.819±0.212

Table B.6: Median of test accuracies for all models across the Relations Game from Table 6.4

	task	all			between			occurs			same			xoccurs		
	train_size	100	1000	5000	100	1000	5000	100	1000	5000	100	1000	5000	100	1000	5000
	model															
Hex.	DNF	0.94	0.97	0.98	0.90	0.99	0.99	0.56	0.99	0.99	0.94	1.00	1.00	0.49	0.80	0.93
	DNF+t	0.60	0.63	0.51	0.50	0.71	0.96	0.50	0.94	0.97	0.92	0.98	0.99	0.48	0.51	0.61
	DNF-h	0.98	0.99	0.99	0.95	1.00	0.99	0.62	0.99	0.99	0.97	1.00	1.00	0.50	0.98	0.96
	DNF-h+t	0.51	0.55	0.92	0.91	0.97	0.98	0.50	0.79	0.96	0.53	1.00	0.98	0.48	0.51	0.51
	DNF-hi	0.98	0.99	1.00	0.97	0.99	1.00	0.69	0.99	0.99	0.97	1.00	1.00	0.51	0.99	0.99
	DNF-hi+t	0.86	0.90	0.77	0.95	0.97	0.95	0.50	0.95	0.65	0.53	0.99	1.00	0.48	0.51	0.76
	DNF-i	0.93	0.99	0.98	0.93	1.00	1.00	0.58	0.99	0.99	0.94	1.00	1.00	0.49	0.99	0.98
	DNF-i+t	0.51	0.51	0.51	0.50	0.99	0.99	0.50	0.97	0.97	0.53	0.97	0.99	0.48	0.94	0.51
	DNF-r	0.94	0.98	0.98	0.84	1.00	0.99	0.63	0.99	0.99	0.96	1.00	1.00	0.51	0.94	0.51
	DNF-r+t	0.51	0.51	0.51	0.50	0.65	0.50	0.50	0.51	0.90	0.53	0.95	0.99	0.48	0.51	0.51
	DNF-ri	0.96	0.98	0.99	0.88	0.99	1.00	0.60	0.99	0.99	0.98	1.00	1.00	0.52	0.98	0.96
	DNF-ri+t	0.51	0.51	0.51	0.50	0.90	0.75	0.50	0.71	0.52	0.53	0.97	0.99	0.48	0.51	0.51
	PrediNet	0.85	0.95	0.96	0.66	0.99	0.99	0.57	0.95	0.97	0.99	1.00	1.00	0.50	0.58	0.95
Pent.	DNF	0.89	0.96	0.95	0.85	0.99	0.99	0.57	0.96	0.98	0.95	1.00	1.00	0.50	0.74	0.86
	DNF+t	0.59	0.62	0.50	0.51	0.67	0.92	0.49	0.93	0.96	0.91	0.98	0.99	0.50	0.51	0.68
	DNF-h	0.95	0.97	0.98	0.92	0.99	0.99	0.62	0.95	0.97	0.94	1.00	1.00	0.51	0.94	0.90
	DNF-h+t	0.50	0.53	0.88	0.90	0.98	0.97	0.49	0.81	0.93	0.50	0.99	0.97	0.50	0.51	0.49
	DNF-hi	0.96	0.99	0.99	0.95	0.99	1.00	0.69	0.98	0.98	0.96	1.00	1.00	0.50	0.96	0.99
	DNF-hi+t	0.86	0.85	0.78	0.95	0.96	0.94	0.49	0.94	0.63	0.50	0.99	0.99	0.50	0.51	0.72
	DNF-i	0.89	0.99	0.98	0.88	0.99	0.99	0.59	0.98	0.99	0.93	1.00	1.00	0.51	0.97	0.96
	DNF-i+t	0.50	0.50	0.50	0.51	0.99	0.99	0.49	0.97	0.95	0.50	0.98	0.99	0.50	0.93	0.49
	DNF-r	0.93	0.96	0.97	0.81	0.99	0.99	0.65	0.96	0.96	0.95	1.00	1.00	0.52	0.88	0.49
	DNF-r+t	0.50	0.50	0.50	0.51	0.62	0.52	0.49	0.51	0.88	0.50	0.96	0.98	0.50	0.51	0.49
	DNF-ri	0.95	0.96	0.99	0.86	0.99	1.00	0.63	0.98	0.99	0.97	1.00	1.00	0.52	0.97	0.96
	DNF-ri+t	0.50	0.50	0.50	0.51	0.89	0.74	0.49	0.80	0.50	0.50	0.98	0.99	0.50	0.51	0.49
	PrediNet	0.85	0.96	0.95	0.65	0.99	0.98	0.60	0.95	0.97	0.99	1.00	1.00	0.50	0.58	0.95
Stripes	DNF	0.91	0.97	0.95	0.81	0.98	0.99	0.57	0.97	0.99	0.93	0.99	1.00	0.49	0.88	0.94
	DNF+t	0.60	0.62	0.50	0.51	0.78	0.95	0.49	0.88	0.95	0.93	0.98	0.97	0.48	0.50	0.57
	DNF-h	0.93	0.98	0.99	0.89	0.99	0.99	0.57	0.97	0.99	0.96	1.00	1.00	0.51	0.98	0.97
	DNF-h+t	0.52	0.53	0.93	0.92	0.97	0.95	0.49	0.84	0.86	0.49	0.99	0.97	0.48	0.50	0.51
	DNF-hi	0.95	0.99	0.99	0.94	0.99	0.99	0.63	0.96	0.99	0.97	1.00	1.00	0.49	0.96	0.97
	DNF-hi+t	0.89	0.87	0.73	0.70	0.96	0.90	0.49	0.81	0.66	0.49	0.99	0.99	0.48	0.50	0.79
	DNF-i	0.89	0.96	0.97	0.87	0.99	0.99	0.55	0.98	0.99	0.96	1.00	1.00	0.50	0.98	0.96
	DNF-i+t	0.52	0.50	0.50	0.51	0.96	0.97	0.49	0.87	0.93	0.49	0.97	0.97	0.48	0.89	0.51
	DNF-r	0.92	0.97	0.98	0.81	0.99	0.99	0.64	0.95	0.98	0.98	1.00	1.00	0.52	0.92	0.51
	DNF-r+t	0.52	0.50	0.50	0.51	0.58	0.52	0.49	0.50	0.83	0.49	0.94	0.99	0.48	0.50	0.51
	DNF-ri	0.94	0.97	0.99	0.84	0.99	0.99	0.58	0.95	0.99	0.98	1.00	1.00	0.52	0.98	0.95
	DNF-ri+t	0.52	0.50	0.50	0.51	0.87	0.75	0.49	0.50	0.51	0.49	0.97	0.98	0.48	0.50	0.51
	PrediNet	0.84	0.93	0.92	0.64	0.99	0.99	0.54	0.94	0.92	0.99	0.99	1.00	0.51	0.61	0.93

Table B.7: Median absolute deviation across the Relations Game, complements Table B.6

	task	all			between			occurs			same			xoccurs		
	train_size	100	1000	5000	100	1000	5000	100	1000	5000	100	1000	5000	100	1000	5000
	model															
Hex.	DNF	0.05	0.01	0.09	0.03	0.00	0.00	0.06	0.00	0.00	0.01	0.00	0.00	0.02	0.08	0.05
	DNF+t	0.17	0.17	0.08	0.00	0.18	0.06	0.00	0.01	0.00	0.03	0.01	0.01	0.00	0.12	0.16
	DNF-h	0.05	0.01	0.00	0.01	0.00	0.00	0.03	0.01	0.00	0.01	0.00	0.00	0.00	0.01	0.05
	DNF-h+t	0.14	0.11	0.06	0.21	0.01	0.05	0.00	0.20	0.14	0.18	0.03	0.15	0.00	0.21	0.13
	DNF-hi	0.01	0.00	0.00	0.03	0.00	0.00	0.08	0.00	0.00	0.02	0.00	0.00	0.01	0.01	0.00
	DNF-hi+t	0.07	0.04	0.18	0.19	0.01	0.15	0.00	0.21	0.18	0.18	0.01	0.02	0.00	0.12	0.16
	DNF-i	0.03	0.01	0.01	0.06	0.00	0.00	0.06	0.00	0.00	0.01	0.00	0.00	0.02	0.00	0.01
	DNF-i+t	0.08	0.17	0.17	0.00	0.01	0.02	0.00	0.01	0.01	0.13	0.14	0.01	0.00	0.05	0.14
	DNF-r	0.08	0.03	0.01	0.07	0.01	0.00	0.09	0.02	0.01	0.03	0.00	0.00	0.01	0.08	0.04
	DNF-r+t	0.00	0.13	0.00	0.00	0.17	0.00	0.00	0.16	0.21	0.18	0.14	0.16	0.00	0.04	0.00
	DNF-ri	0.10	0.00	0.01	0.09	0.00	0.00	0.08	0.00	0.00	0.01	0.00	0.00	0.01	0.00	0.06
	DNF-ri+t	0.00	0.00	0.23	0.00	0.16	0.17	0.00	0.17	0.00	0.19	0.01	0.03	0.00	0.16	0.00
	PrediNet	0.04	0.00	0.00	0.09	0.00	0.01	0.01	0.01	0.01	0.04	0.00	0.00	0.01	0.02	0.03
Pent.	DNF	0.05	0.02	0.07	0.02	0.00	0.00	0.05	0.02	0.01	0.02	0.00	0.00	0.02	0.08	0.06
	DNF+t	0.15	0.16	0.10	0.00	0.19	0.07	0.00	0.03	0.02	0.03	0.01	0.01	0.00	0.11	0.14
	DNF-h	0.05	0.02	0.00	0.01	0.00	0.00	0.03	0.02	0.02	0.01	0.00	0.00	0.01	0.02	0.06
	DNF-h+t	0.13	0.10	0.05	0.20	0.01	0.04	0.00	0.18	0.14	0.19	0.03	0.15	0.00	0.19	0.11
	DNF-hi	0.01	0.00	0.00	0.04	0.00	0.00	0.07	0.01	0.01	0.01	0.00	0.00	0.01	0.01	0.00
	DNF-hi+t	0.04	0.05	0.19	0.18	0.01	0.13	0.00	0.22	0.18	0.18	0.00	0.04	0.00	0.11	0.17
	DNF-i	0.04	0.01	0.01	0.06	0.00	0.00	0.04	0.00	0.01	0.01	0.00	0.00	0.01	0.01	0.02
	DNF-i+t	0.08	0.18	0.16	0.00	0.01	0.03	0.00	0.01	0.02	0.15	0.15	0.01	0.00	0.04	0.13
	DNF-r	0.09	0.04	0.02	0.07	0.03	0.01	0.07	0.02	0.01	0.02	0.00	0.00	0.01	0.08	0.04
	DNF-r+t	0.00	0.13	0.00	0.00	0.18	0.00	0.00	0.18	0.20	0.19	0.15	0.16	0.00	0.04	0.00
	DNF-ri	0.09	0.01	0.01	0.09	0.00	0.00	0.08	0.00	0.01	0.02	0.00	0.00	0.02	0.02	0.06
	DNF-ri+t	0.00	0.00	0.22	0.00	0.17	0.16	0.00	0.19	0.00	0.21	0.01	0.03	0.00	0.16	0.00
	PrediNet	0.04	0.00	0.00	0.08	0.00	0.01	0.01	0.01	0.00	0.04	0.00	0.00	0.01	0.01	0.04
Stripes	DNF	0.06	0.01	0.11	0.08	0.01	0.01	0.04	0.01	0.00	0.01	0.00	0.00	0.01	0.09	0.03
	DNF+t	0.19	0.17	0.08	0.00	0.18	0.06	0.00	0.04	0.02	0.03	0.01	0.01	0.00	0.10	0.18
	DNF-h	0.06	0.01	0.00	0.03	0.01	0.00	0.03	0.01	0.01	0.02	0.00	0.00	0.01	0.01	0.02
	DNF-h+t	0.14	0.12	0.06	0.20	0.00	0.04	0.00	0.20	0.16	0.17	0.07	0.15	0.00	0.20	0.14
	DNF-hi	0.02	0.02	0.01	0.08	0.00	0.00	0.06	0.02	0.00	0.02	0.00	0.00	0.03	0.01	0.01
	DNF-hi+t	0.10	0.03	0.17	0.16	0.02	0.12	0.00	0.19	0.19	0.20	0.02	0.01	0.00	0.11	0.15
	DNF-i	0.08	0.02	0.01	0.06	0.00	0.00	0.07	0.01	0.01	0.01	0.00	0.00	0.01	0.01	0.02
	DNF-i+t	0.06	0.17	0.16	0.01	0.03	0.07	0.00	0.05	0.04	0.15	0.15	0.01	0.00	0.04	0.15
	DNF-r	0.11	0.03	0.01	0.09	0.01	0.01	0.08	0.05	0.00	0.02	0.00	0.00	0.02	0.06	0.04
	DNF-r+t	0.00	0.14	0.00	0.00	0.17	0.00	0.00	0.10	0.19	0.21	0.15	0.15	0.00	0.06	0.00
	DNF-ri	0.11	0.01	0.02	0.08	0.01	0.01	0.08	0.04	0.04	0.02	0.00	0.00	0.01	0.01	0.05
	DNF-ri+t	0.00	0.00	0.23	0.00	0.19	0.15	0.00	0.13	0.00	0.19	0.01	0.04	0.00	0.16	0.00
	PrediNet	0.04	0.03	0.02	0.08	0.00	0.00	0.03	0.03	0.02	0.02	0.00	0.00	0.02	0.02	0.03

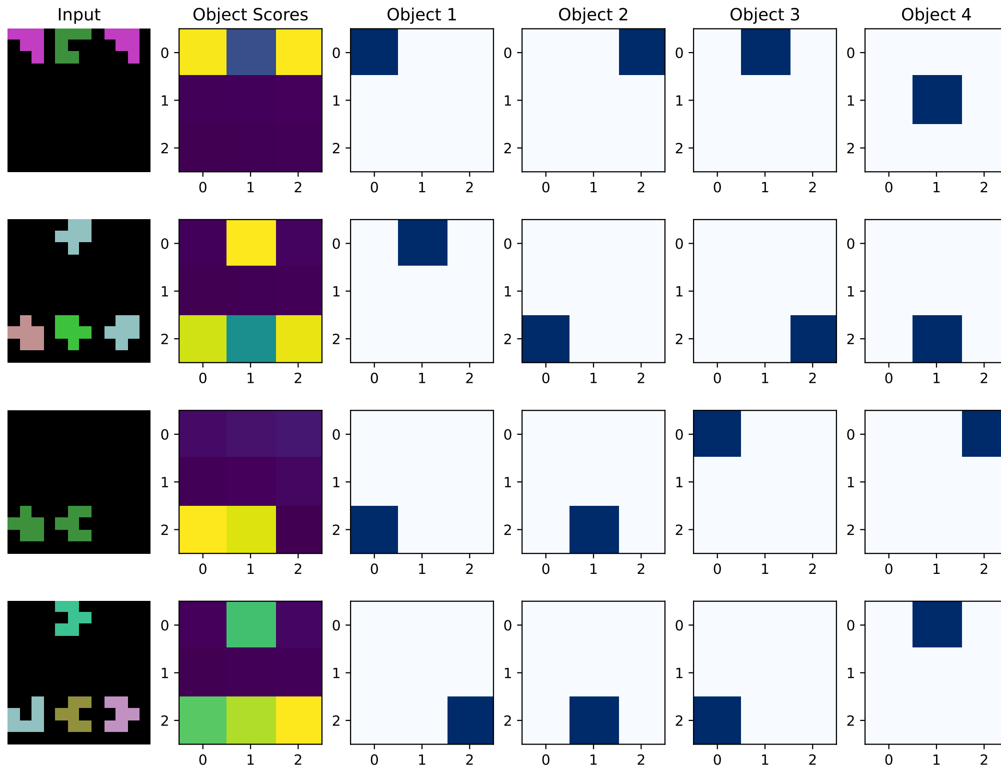


Figure B.14: Attention maps with the DNF model trained on all the tasks with 1k training size per task. When there are extra slots, the model is forced to select an empty black patch as an object.

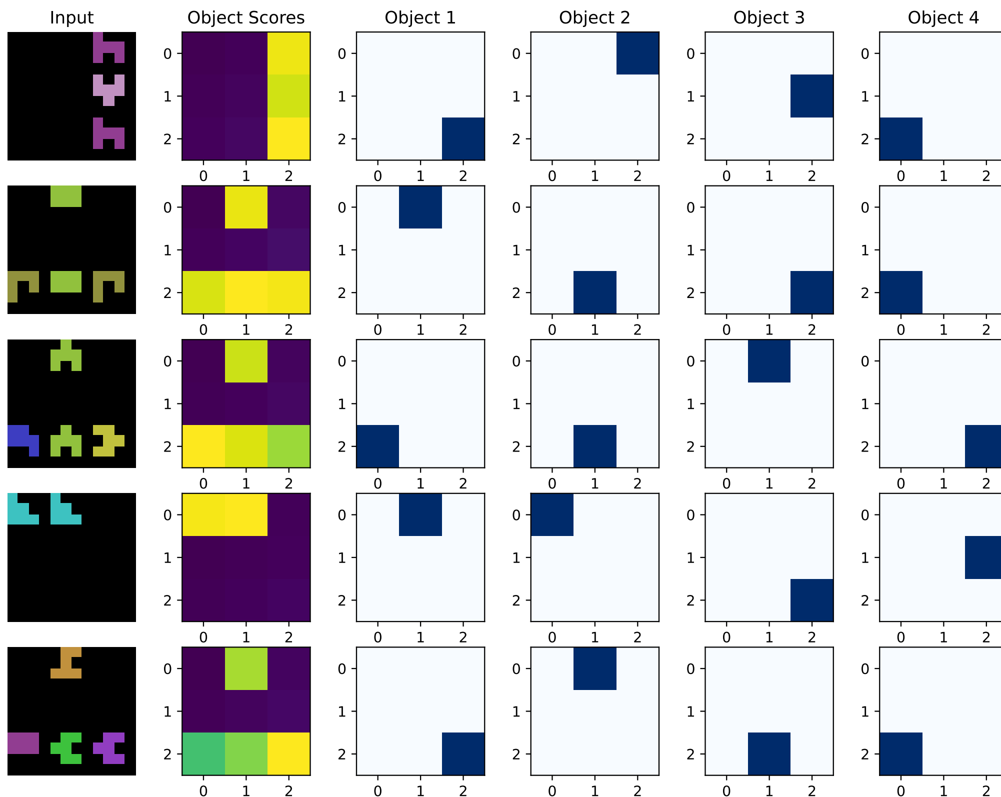


Figure B.15: Further attention maps learnt by DNF-hi on all tasks with 1k training size per task

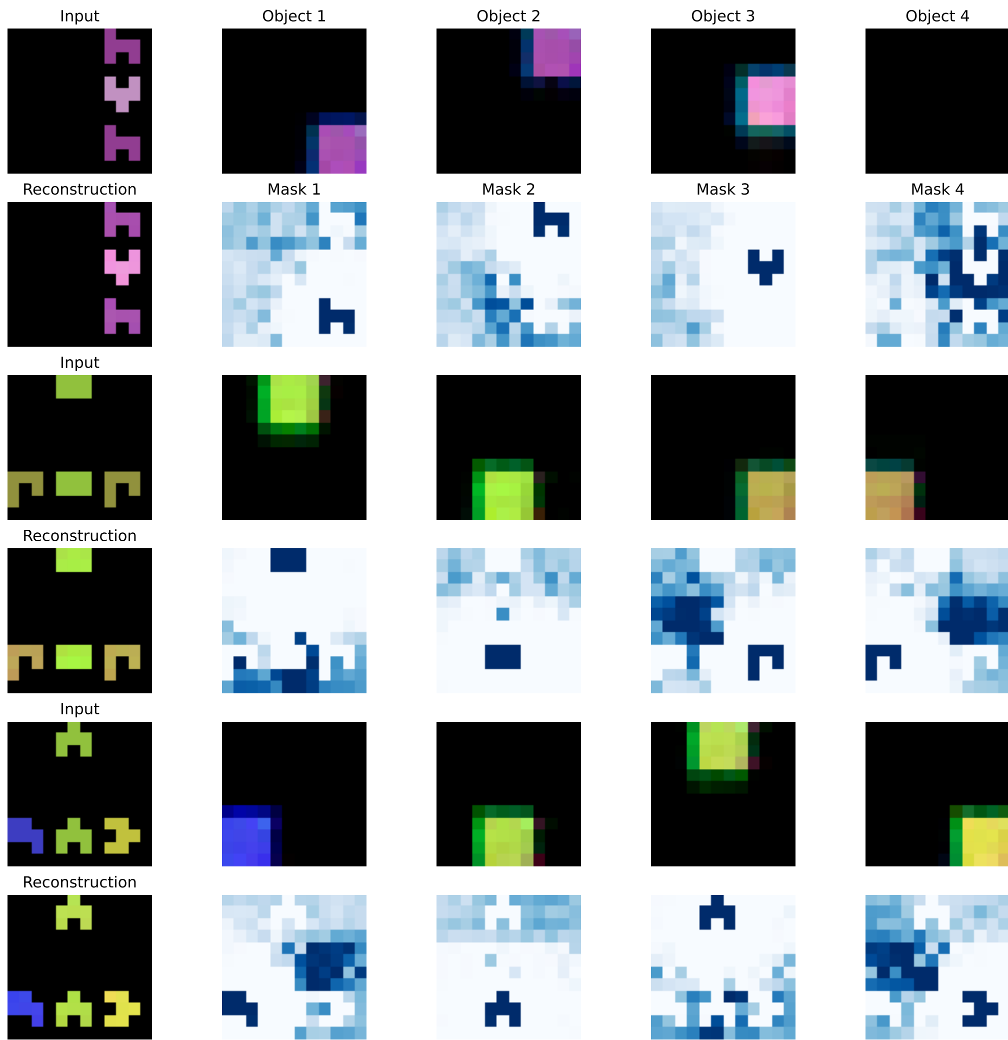


Figure B.16: Further image reconstruction examples obtained from DNF-hi trained on all tasks with 1k examples per task