DIFFERENTIABLE NEURAL LOGIC NETWORKS AND THEIR APPLICATION ONTO INDUCTIVE LOGIC PROGRAMMING

A Dissertation
Presented to
The Academic Faculty

By

Ali Payani

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology

May 2020

Copyright © Ali Payani 2020

DIFFERENTIABLE NEURAL LOGIC NETWORKS AND THEIR APPLICATION ONTO INDUCTIVE LOGIC PROGRAMMING

Approved by:

Professor Faramarz Fekri, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Matthieu Bloch
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Mark Davenport
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Ghassan AlRegib
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Siva Theja Maguluri
School of Industrial and Systems Engineering
Georgia Institute of Technology

Date Approved: April 20, 2020

to my family

ACKNOWLEDGEMENTS

I like to dedicate this dissertation to my family, specially my mom whom I love dearly, my late father whom I miss everyday and my dearest and kindest sister.

I would like to express my gratitude to my advisor, Dr. Faramarz Fekri for his kindness, continuous support and for the valuable mentorship throughout these years. I am grateful to him for believing in my abilities and for giving me the chance to pursue my goals. I would also like to specially thank my thesis committee members: Dr. Matthieu Bloch, Dr. Mark Davenport, Dr. Ghassan AlRegib and Dr. Siva Maguluri for their valuable feedbacks and their support.

Many thanks to my close friend and collaborator Afshin Abdi for his support in my early days of PhD and to Yashas Malur Saidutta for offering his kind support and help in many situations including my PhD proposal presentation. I would also like to thank my other friends and collaborators in Georgia Tech: Dr. Entao Liu, Dr. Jinwen Tian, Dr. Ahmad Beirami, Dr. Mohsen Sardari, Dr. Nima Torabkhani and Dr. Masoud Gheisari for making the life at campus fun and entertaining. Finally, I would like to thank my friends in Atlanta and specially Maryam Najiarani, for making me feel at home and supporting me during all these challenging years.

TABLE OF CONTENTS

Acknowledgments iv

List of Tables ix

List of Figures xi

Chapter 1: Introduction and Literature Survey 1

1.1 Differentiable Neural Logic Networks 3

1.2 Inductive Logic Programming 5

1.2.1 Introduction to ILP 6

1.2.2 Previous Works 8

1.3 Learning from Uncertain Data via dNL-ILP 11

1.4 Relational Reinforcement Learning 13

1.5 Decoding LDPC codes over Binary Erasure Channels 15

1.6 Independent Component Analysis 15

Chapter 2: Differentiable Neural Logic Networks 17

2.1 Introduction 17

2.2 Neural Conjunction and Disjunction Layers 18

2.3 Neural XOR Layer 21

2.4 dNL vs MLP 22

2.5 Binary Arithmetic 24

2.6 Grammar Verification 27

Chapter 3: Inductive Logic Programming via dNL 29

3.1 Introduction 29

3.2 Logic Programming 30

3.3 Formulating the dNL-ILP 34

3.3.1 Forward Chaining 35

3.3.2 Training 40

3.3.3 Pruning 40

3.3.4 Lack of Uniqueness 41

3.3.5 Predicate Rules ($\mathscr{F}_p^i$) 41

3.4 ILP as a Satisfiability Problem 42

3.5 Interpretability 43

3.6 Implementation Details 43

3.7 Experiments 45

3.7.1 Benchmark ILP Tasks 46

3.7.2 Learning Decimal Multiplication 48

3.7.3 Sorting 51

Chapter 4: Learning from uncertain data via dNL-ILP 53

4.1 Introduction 53

4.2 Classification for Relational Data 53

4.2.1 Implementation Details: . . . 57
4.3 Handling Continuous Data . . . 59
4.4 Dream4 Challenge Experiment (Handling Uncertain Data) . . . 61
4.5 Comparing dNL-ILP to the Past Works . . . 62

Chapter 5: Relational Reinforcement Learning via dNL-ILP . . . 65

5.1 Introduction . . . 65
5.2 Relational Reinforcement Learning via dNL-ILP . . . 67
5.2.1 State Representation . . . 68
5.2.2 Action Representation . . . 72
5.3 Experiments . . . 73
5.3.1 BoxWorld Experiment . . . 73
5.3.2 GridWorld Experiment . . . 77
5.3.3 Relational Reasoning . . . 80
5.3.4 Asterix Experiment . . . 81

Chapter 6: Decoding LDPC codes over Binary Erasure Channels via dNL . . . 84

6.1 Introduction . . . 84
6.2 Message Passing Decoding . . . 85
6.3 Experiments . . . 89
6.3.1 Performance . . . 89
6.3.2 Generalizations . . . 90
6.3.3 dNL vs MLP . . . 92
6.4 ML decoding of LDPC over BEC via dNL . . . 93

Chapter 7: Independent Component Analysis using Variational Autoencoder Framework . . . 97

7.1 Introduction . . . 97

7.2 Proposed Method . . . 98

7.2.1 Encoder . . . 100

7.2.2 Decoder . . . 101

7.2.3 Learning Algorithm . . . 102

7.3 Experiments . . . 104

7.3.1 VAE-nICA vs VAE . . . 106

7.3.2 Performance . . . 106

7.3.3 Robustness to Noise . . . 106

Chapter 8: Conclusion . . . 109

8.1 Summary of Achievements . . . 109

8.2 Future Research Directions . . . 112

8.2.1 Incorporating Human Preference and Safe AI . . . 112

Appendix A: Proof of Theorem 1 . . . 116

Appendix B: BoxWorld Experiment Details . . . 117

Appendix C: GridWorld Experiment Details . . . 120

Appendix D: Relational Reasoning Experiment Details . . . 123

Appendix E: Asterix Experiment Details . . . 126

References 136

LIST OF TABLES

1.1 Program definition of learning daughter relation [11] 7

2.1 Size of required training samples to reach certain levels of accuracy 27

3.1 Some of the notations used in this chapter 38

3.2 dNL-ILP vs dILP and Metagol in benchmark tasks 50

4.1 Dataset Features 55

4.2 Some of the learned rules for classification tasks 55

4.3 AUPR score for the 5 relational classification tasks 56

4.4 Classification accuracy 61

4.5 DREAM4 challenge scores 62

5.1 Number of training episodes required for convergence 79

6.1 Evaluating the generalization performance of the model 92

6.2 Design parameters for functions $\mathscr{F}$ and $\mathscr{G}$ 93

7.1 Maximum correlation results for the 3 different types of mixings; Linear, PNL and MLP. 107

7.2 Maximum correlation results for the linear mixing problem in the presence of white Gaussian noise with three different σ_e. 107

B.1 ILP definition of the BoxWorld 119

C.1 ILP definition of the GridWorld 122

D.1 Flat index for a grid of 4 by 4 used in relational learning task 123
D.2 ILP definition of the relational reasoning task 125

E.1 ILP definition of the Asterix experiment 127

LIST OF FIGURES

1.1 Interpretability vs performance trade-off 3

1.2 Learning connectedness in directed graphs 9

1.3 An example of relational database 12

2.1 Truth table of $F_c(\cdot)$ and $F_d(\cdot)$ functions 19

2.2 Comparing MLP vs dNL for learning Boolean functions 25

2.3 Feed-Forward and recurrent models used in our experiments 26

2.4 Accuracy results for Palindrome grammar check 28

3.1 Internal representation of $\mathscr{F}_{lt}^i$ via a dNL-DNF function with of hidden layer of size 4 36

3.2 The diagram for one step forward chaining for predicate `lt` where $\mathscr{F}_{lt}$ is implemented using a dNL-DNF network. 37

3.3 Background graph used in learning circular predicate 49

4.1 Internal structure of MovieLens dataset 54

4.2 An example of dNL-ILP program definition in python 60

5.1 States representation in the form of predicates in BoxWorld game, before and after an action 68

5.2 Learning explicit relational information from images in our proposed RRL; Images are processed to obtain explicit representation and dNL-ILP engine learns and expresses the desired policy (actions) 69

5.3 Transforming low-level state representation to high-level form via auxiliary predicates 69

5.4 Extracting relational information from visual scene [41] 71

5.5 Comparing deep A2C and the proposed model on BoxWorld task 76

5.6 Effect of background knowledge on learning BoxWorld 76

5.7 GridWorld environment [77] 78

5.8 Effect of background knowledge on learning GridWorld 79

5.9 Asterix game environment example 82

5.10 Score during training in Asterix experiment 83

6.1 Logical functions used in forward/backward operations 87

6.2 Performance comparison between the neural logic decoder and BP 91

6.3 Performance comparison between the dNL and MLP for iteratively decoding LDPC codes over BEC channel 94

6.4 Performance comparison between the dNL-XOR decoder and BP 96

7.1 Graphical Models used in VAE vs Noisy ICA 102

7.2 Block diagram of the noisy ICA model 104

7.3 Six synthetic sources used in our experiments 105

7.4 Comparing the performance of standard VAE vs VAE-nICA 107

7.5 Comparing the performance of VAE-nICA vs Anica in the presence of noise 108

8.1 Grid World environments for safe AI 114

C.1 An example GridWorld scene 121

SUMMARY

Despite the impressive performance of Deep Neural Networks (DNNs), they usually lack the explanatory power of disciplines such as logic programming. Even though they can learn to solve very difficult problems, the learning is usually implicit and it is very difficult, if not impossible, to interpret and decipher the underlying explanations that is implicitly stored in the weights of the neural network models. On the other hand, standard logic programming is usually limited in scope and application compared to the DNNs. The objective of this dissertation is to bridge the gap between these two disciplines by presenting a novel paradigm for learning algorithmic and discrete tasks via neural networks. This novel approach, uses the differentiable neural network to design interpretable and explanatory models that can learn and represent Boolean functions efficiently. We will investigate the application of these differentiable Neural Logic (dNL) networks in disciplines such as Inductive Logic Programming, Relational Reinforcement Learning, as well as in discrete algorithmic tasks such as decoding LDPC codes over Binary erasure Channels.

Inductive Logic Programming (ILP) is an important branch of machine learning that uses formal logic to define and solve problems. Compared to the DNNs, ILP provides high degree of interpretability and can learn from small number of examples and has superior generalization ability. However, standard ILP solvers are not differentiable and cannot benefit from the impressive power of DNNs. In this dissertation, we reformulate the ILP as a differentiable neural network by exploiting the explanatory power of dNL networks. This novel neural based ILP solver (dNL-ILP) is capable of learning auxiliary and invented predicates as well as learning complex recursive predicates. We show that dNL-ILP outperforms the current state of the art ILP solvers in a variety of benchmark algorithmic tasks as well as larger scale relational classification tasks. In particular, the application of the dNL-ILP in classification of relational datasets such as MovieLens, IMDB, UW-CSE, CORA and Mutagenesis is investigated.

Relational Reinforcement Learning (RRL) focuses on the application of logic programming in reinforcement learning. In particular, RRL describes the environment in terms of objects and relations and uses predicate language to describe the state of the environment as well the agent's policies. Despite their great potential and the promise of learning generalized and human readable policies, the traditional RRL systems are not applicable to a variety of the problems that the modern RL framework addresses. This is mainly because the typical RRL systems require explicit representation of the state and cannot directly learn from complex visual scenes. In this dissertation, we show how the proposed differentiable ILP solver can effectively combine the standard deep learning techniques such as convolutional networks with ILP and can learn the intermediate explicit representations that the ILP systems requires. Through various experiments, we show how the proposed deep relational policy learning framework can combine human expertise to learn efficient policies directly from images and outperform both the traditional RRL systems and the current deep RL by incorporating background knowledge effectively.

Despite the impressive performance of DNNs in a variety of tasks, there has been limited success in using DNNs for learning discrete algorithmic problems. Via various experiments, we show that even in discrete algorithmic tasks that do not require the explanatory power of dNL networks, they are more efficient and can learn faster and generalize better than the standard DNNs. In particular, we investigate their application in designing an iterative decoding approach for LDPC codes over binary erasure channels and compare their performance to the state of the art.

CHAPTER 1

INTRODUCTION AND LITERATURE SURVEY

Deep Neural Networks (DNNs) based on Convolution Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) have improved the state of the art in various areas such as natural language processing [1], image and video processing [2], and Speech recognition [3] just to name a few. In recent years, successful reports such as Neural Turing Machine [4] have inspired the machine learning community to apply neural networks for learning algorithmic tasks. The development of several principles such as memory, memory controllers, and recurrent sequence to sequence networks in [4] and many other follow up papers [5, 6] led to the increased capacity of the regular RNNs for learning tasks. Further, [7] proposed to employ several layers of GRU units and process the data as a whole vector; unlike the typical sequence to sequence models. This new design produced excellent performance for binary addition and multiplication tasks. In most typical networks (e.g., MLPs, RNNs), the model is designed using successive applications of a basic atomic layer which computes the weighted sum of the inputs followed by a non-linear function. More precisely, if vector $\mathbf{x} \in \mathbf{R}^n$ is the n dimensional vector, the ith element of output, i.e., y_i, can be expressed as $y_i = \eta(\sum_{j=1}^{n} x_j w_j^{(i)} + bias^{(i)})$, where $\eta(.)$ is a scalar nonlinear mapping such as `sigmoid`, `tanh` or `relu` function and $\mathbf{w}^{(i)}$ is the weight vector for the ith neuron. While in theory, using sufficient number of weights and (hidden) layers, we could approximate any function with this design, in practice this is not always possible. Additionally, MLP based models come with some limitations. These models, in general, do not construct any explicit and symbolic representation of the algorithm they learned and the algorithm is implicitly stored in thousands or even millions of weights, which is typically impossible to be deciphered and verified by human agents. Further, MLP networks are usually suitable for cases where there are many training examples, and usually do not generalize well where there are only limited

training examples.

One of the alternative machine learning approaches that addresses the shortcomings of DNNs for learning discrete algorithmic tasks is Inductive Logic Programming (ILP). Logic programming uses the language of formal logic to express the relation between objects in some problem domain. These relations are stated by using Boolean functions called `predicates`. For example, consider a family tree example, where *tom* is the parent of `john` and `jane`. In logic programming, the facts about this system can be expressed using the predicate language. For example, using predicates such as `female(X)`, `male(X)`, `father(X,Y)`, we may express our knowledge regarding this family in terms of some Boolean clauses such as: `father(tom, john), father(tom, jane), female(jane), male(john).` Inductive logic programming, explores inductive learning in first order logic. An inductive system tries to learn a general description of a concept from a set of instances. The main goal of ILP is then the development of theory and hypothesis for inductive reasoning in first-order (predicate) logic. In other words, ILP uses the logic programming to uniformly express the examples, background knowledge and the target hypothesis for a machine learning task. In ILP, explicit rules and symbolic logical representations can be learned using only a few training examples and these models are usually able to generalize well. Further, the explicit symbolic representation that is obtained via ILP can be understood and verified by human, and can also be used to write programs in any conventional programming language.

Over the last two decades, machine learning systems based on ILP has been widely used in various domain including the knowledge discovery from relation databases as well as robotics. However, in recent years and with the impressive progress of deep learning techniques, the ILP discipline has been largely overlooked. While the application of ILP and DNN based systems do not always overlap, we can crudely compare these two discipline from the perspective of the trade-of between the interpretability and the performance as shown in Fig. 1.1. Generally speaking, DNNs are able to tackle much harder learning tasks

involving difficult reasoning and learning from multidimensional data. Comparatively, ILP systems are usually limited in scope and complexity. On the other hand, the learning in DNNs is usually implicit and hard (if not impossible) to interpret, while ILP models offer a high degree of interpretability.

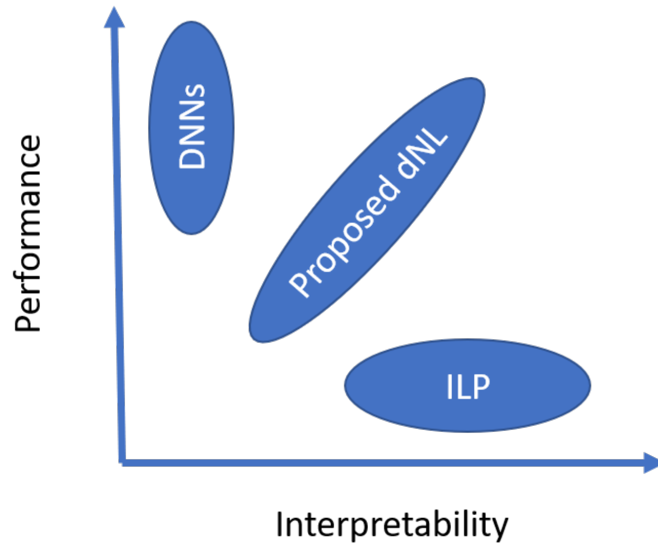


Figure 1.1: Interpretability vs performance trade-off

In this dissertation, we seek new methods and techniques that could bridge the gap between these two disciplines. Our aim is to design a differentiable approach to learning Boolean logic and to use these differentiable Neural Logic (dNL) networks in designing a differentiable neural based ILP solver. We will then study the application of such an approach in learning tasks involving relational databases. We further extend this framework to design a relational reinforcement learning framework. The outline of the contribution of this dissertation are presented below.

1.1 Differentiable Neural Logic Networks

The ability to learn Boolean logic in an explicit and interpretable manner is at the heart of our framework. Our aim is to design a differentiable neural networks dedicated to learn and represent Boolean functions in an explicit and efficient way. The general idea

of representing and learning Boolean functions using neural networks is not new. There is significant body of research from the early days of machine learning using neural networks that is focused on the theoretical aspects of this problem. Some special Boolean functions such as parity-N and XOR has been the subject of special interest, as benchmark tasks for theoretical analysis. Minsky and Papert [8, 9], for example, showed the impossibility of representing all functional dependence and proved this for XOR logic function while other works demonstrate the possibly of representing XOR by adding hidden layers. From the practical standpoint, as was suggested by many works (for example [10]), any Boolean function can be learned by a multi-layer neural network equipped with proper activation functions. However, as we will demonstrate in the following chapters, this approach is not always applicable for two main reasons:

- Since these networks are not specifically designed for learning Boolean functions, the rate of convergence is not always acceptable. Moreover, in various problems they do not fully converge. Specially, in cases with small batch sizes, the weights of the network tend to fluctuate and do not fully converge as training continues. As we will show later, this behavior can result in degrading performance for some applications including the learning of discrete algorithmic tasks.
- Even if the MLP based approaches fully converge and could perfectly learn a Boolean function, their internal representation is not usually interpretable by human. The actual learned Boolean function is implicitly stored in the weights and biases of additive neurons in a complex manner. As such, they are not suitable for applications where the interpretability is desirable.

As an alternative to typical additive neurons, we propose a set of differentiable and multiplicative neurons which are designed with the specific function of learning Boolean functions efficiently and in an explicit manner which can be read and verified by human. Even though these models, referred as differentiable Neural Logic (dNL) networks hereafter,

are designed for Boolean functions, their domain of applicability is not limited to the learning of Boolean logic. Many algorithmic learning tasks that were addressed in the recent literature, e.g., addition, copying, reverse sequence, grammar checking, can indeed be expressed by a Boolean functions of the binary inputs with arbitrary complexity. As an example, consider a very simplistic case, e.g., the binary addition of the two binary digits x and y which can be expressed using Boolean algebra as:

$$\mathbf{r} = x + y \quad \Rightarrow \quad \begin{cases} r_0 = x \bigwedge y \\ r_1 = (x \bigwedge \bar{y}) \bigvee (\bar{x} \bigwedge y) \end{cases} , \tag{1.1}$$

where r_1 and r_0 are two Boolean elements of the vector $\mathbf{r}$. Here, $\bar{x}$ is the compliment of the Boolean variable x, and $\bigwedge$ and $\bigvee$ represent conjunction (AND) and disjunction (OR) operators respectfully. Consequently, a network which is designed with dNL network is capable of efficiently learning various discrete algorithmic tasks as we will show in this work.

In chapter 2, we present an alternative approach to the traditional MLP design for learning Boolean functions (dNL) that aim to address some of the shortcoming of the MLP for learning discrete-algorithmic tasks. We demonstrate the effectiveness of the proposed general purpose dNL layers by various empirical experiments in sections 2.4, 2.5 and 2.6.

1.2 Inductive Logic Programming

Inductive Logic Programming (ILP) is one of the most successful machine learning approaches that addresses the shortcomings of MLPs for learning discrete algorithmic tasks. In the following we give a short introduction on ILP before reviewing the literature on ILP and discussing our contributions.

1.2.1 Introduction to ILP

ILP is a machine learning discipline that uses logic programming both for internal representation of the system as well as representing and learning the target hypothesis. As such, before discussing various ILP systems we need to introduce the basic notations used in logic programming. A logic program consists of a set of `clauses`. We may think of a clause as an `if-then` rule. These if-then rules consist of a conclusion which is called `head` and a condition which is called the `body` of the rule. For example, the rule $q \leftarrow p$ reads as: if p is True, then q is True, where q is the head and p is the body of the rule. Logic programming is also referred to as predicate language. In logic programs, the relation between objects are expressed using Boolean functions called `predicate`. For example, the fatherhood relation between two persons may be expressed by the predicate `father(X,Y)`. For instance, the term `father(Tom,John)` is evaluated to True if `Tom` is the father of `John`. Similar to other programming paradigms, in logic programming we can talk about general rules using variables (i.e., `X, Y`) or about specifics (i.e., instances such as `tom`, `john`). The atomic Boolean expressions in logic programming are created by applying a predicate to some arguments. For example, both expressions `father(X,Y)` and `father(Tom,John)` are called `atoms`. An atom that contains no variables (e.g., `father(Tom,John)`) is called `ground` atom. Most ILP systems use a restrictive form of rules which are called `Horn` clauses. In this format, an if-then rule is a clause of this form:

$$H \leftarrow B_1,\ B2,\ \ldots,\ B_m \tag{1.2}$$

where the head of rule, i.e., H contains one atom and the body of rule is made of conjunction of atoms B_1 to B_m. As an example, consider the following rule that defines the daughter relation using the `parent` and `female` relations:

$$\text{daughter(X,Y)} \leftarrow \text{parent(Y,X)},\ \text{female(X)} \tag{1.3}$$

It is read as ”if Y is the parent of Y and X is female, then X is the daughter of Y”. Here, the body consists of two atoms `parent(Y,X)` and `male(X)` and the conclusion, i.e., the head of rule consists of the atom `daughter(X,Y)`. The goal of the ILP is to use the language of the logic programming to describe the known facts about the system and then, to find generalized hypothesis that could explain those facts. In order to illustrate this, consider the program consists of the background facts and some training examples for learning the `daughter` relation as shown in table 1.1. Here, $\oplus$ and $\ominus$ represent the positive and negative training examples, respectively. In ILP, we are interested in techniques

Table 1.1: Program definition of learning daughter relation [11]

Training Examples		Background knowledge	
daughter(mary, ann).	$\oplus$	parent(ann, mary).	female(ann).
daughter(eve, tom).	$\oplus$	parent(ann, tom).	female(mary).
daughter(tom, ann).	$\ominus$	parent(tom ,eve).	female(eve).
daughter(eve, ann).	$\ominus$	parent(tom, ian).	

that given a program in Table 1.1, can generate a valid hypothesis such as the one in (1.3). Generally speaking, there are two approaches for solving ILP problems as discussed. In the `buttom_up` approaches, we start from the positive examples and by starting from the most specifics examples we try to find generalization. For example, consider the positive example of `daughter(mary,ann)`. We can find facts that are relevant to this example by looking at the partial matching of arguments. For example, we can pick two relevant facts of `parent(ann, mary)` and `female(mary)` and generate a hypothesis such as:

$$\text{daughter(mary,ann)} \leftarrow \text{parent(ann, mary)}, \text{female(mary)} \tag{1.4}$$

We can then generalize the above hypothesis by replacing the specifics with variables to arrive at the desired hypothesis in (1.3). In most practical problems we usually cannot immediately find the general hypothesis as easy as this one. In practice, the button-up approaches iteratively generate and test the candidate hypothesises to find the desired target

hypothesis. To reduce the possible clauses to consider, they reduce the space of possible clauses by introducing declarative biases. For example, such bias declaration may state that the target hypothesis is obeying the transitive form of $P(X, Y) \leftarrow Q(X, Z),\ R(Z, Y)$.

The other technique for solving ILP is the `top-down` approach. In this method we start from an empty set of clauses, and construct a clause explaining some of the positive examples, add this clause to the hypothesis, and remove the positive examples explained. These steps are repeated until all positive examples have been explained. Variations of this top-down approach to ILP is used by most neural ILP solvers as well as other ILP systems dealing with classification tasks. Most traditional approaches to ILP search the space of possible clauses iteratively to find a hypothesis that satisfy examples. On the other hand, the neural approaches to ILP usually needs to enumerate all the possible clauses based on some templates and score them using some techniques to find the hypothesis. In these systems we needs to find the consequences of applying those hypothesis and compare that with the provided examples. These consequences are found by `forward-chaining` algorithm. In this iterative approach, we start by background facts and iteratively generate new facts by applying the rules to the current set of facts, until no further new consequences is found.

1.2.2 Previous Works

Since the early 2000's, a variety of algorithms for solving ILP have been proposed. Most notably, the introduction of Metagol [12] based on meta interpretive learning [13] was one of the recent breakthroughs in solving ILP problems. As shown by a series of publications (e.g. [14, 15]), the impressive performance of Metagol is due to its ability to learn recursive predicates (which is vital for learning many algorithmic tasks) as well as its ability to invent/learn auxiliary predicates. To see the importance of recursion in learning algorithms we can consider a simple program for learning the concept of connectedness in directed graphs. Consider a program that describes a connected graph using the relation (i.e.,

predicate) `edge`. For example, graph in Fig. 1.2 can be described as:

$\mathcal{B}$ = {edge(a,b), edge(b,d), edge(d,a), edge(b,c), edge(b,e), edge(e,f), edge(e,h), edge(f,g)}

In a directed graph, two nodes are considered connected if there is a direct path from one to

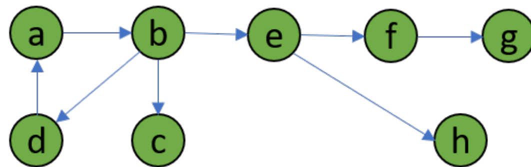


Figure 1.2: Learning connectedness in directed graphs

another. For example, in graph of Fig. 1.2, while `a` is connected to `e` (i.e., connected(a,e) is True), `e` is not connected to `a` since there is no path from `e` to the node `a`. It is easy to verify that a simple recursive definition for the target predicate `connect` via the following two rules can satisfy all the examples:

connected(X,Y) ← egde(X,Y)

connected(X,Y) ← egde(X,Z), connected(Z,Y)

On the other hand, it is not possible to find such a hypothesis without allowing recursive (or mutually recursive) predicates. Without recursion, we can find hypothesis for one or two or any other particular level of connected nodes but it is impossible to solve for the most general form:

connected(X,Y) ← egde(X,Y)

connected(X,Y) ← egde(X,Z), edge(Z,Y)

connected(X,Y) ← egde(X,Z), edge(Z,T), edge(T,Y)

...

In recent years, there has been some attempts to use deep learning methods in solving the ILP problems [16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27]. The connectionist model [20] uses a recurrent neural network to implement a differentiable forward chaining approach. However, the connectionist model only considers acyclic logic programs. Moreover, once the program is learned, it is hidden in the weights of the network and is not explicitly available. Logic Tensor Network [26] is another recently proposed ILP solver that uses the extension of Boolean values to real values in the range of $[0, 1]$ to learn an approximation to the ILP's satisfiability problem. However, their approach is limited in scope and does not support predicate invention or self/mutual recursion among predicates. Another important contribution is the CLIP++ algorithm proposed in [22]. This is a powerful approach for learning first-order logic programs efficiently which uses the button clause $\perp_e$ introduced by Muggleton [28]. For each positive example, the bottom clause $\perp_e$ is the most specific clause whose head unifies with e. Since the clauses form a lattice, there is always a most specific clause for a particular language bias. [22] uses the set of bottom clauses generated from the examples as a set of features: each clause is applied to every tuple of individuals, generating a list of features for each tuple. Even though CLIP++ is fast and scales well to the larger problems, similar to the previously mentioned methods it does not support recursion and hence its application for learning the algorithmic tasks is limited.

While these past neural ILP solvers are shown to learn some specific tasks, until the introduction of differentiable ILP (dILP) by [27], they were not capable of predicate invention and learning the recursive predicates (in an explicit manner). The main limitation of dILP is that it cannot scale well to the more complex problems. In fact, the explosion of parameters in their model, led to a very restrictive approach for choosing rule templates. As such, dILP limits the model to learn clauses with at most two terms (atoms) and to the predicates with at most two arguments, which is a very restrictive approach. In fact, as stated in [27], the need for using program templates to generate a limited set of viable candidate clauses in forming the predicates is the key weakness in all existing (past) ILP systems (neural or non-neural),

severely limiting the solution space of a problem. Metagol [12], mitigates this problem by introducing meta-rule templates to define a family of possible clauses. While this approach allows the system to learn more complex solutions compared to dILP, the design of these rules for any particular problem is itself a complicated task and requires expert knowledge and possibly many trials. Moreover, although many of the neural ILP solvers provide a way to learn the inductive rules, these learned knowledge is not necessarily explicit and human readable (e.g., [29]) or require arbitrary cutoff values for choosing the candidate rules (e.g., dILP).

In Chapter 3 we demonstrate that the explicit representational power of the proposed dNL allows us to transform the inductive logic programming into a differentiable problem and solve it using gradient optimizers more efficiently than the existing neural ILP solvers. We evaluate the performance of our proposed ILP solver in some benchmark ILP tasks and compare the performance of dNL-ILP to the state of the art ILP systems in learning algorithmic tasks such as learning arithmetic and sorting.

1.3 Learning from Uncertain Data via dNL-ILP

In Chapter 4 we study the application of dNL-ILP in learning from uncertain data. In particular, we will study the application of the dNL-ILP in tasks involving the reasoning and extracting knowledge from relational databases.

ILP is concerned with the development of techniques and tools for relational data mining. In relational databases, data is typically resides in multiple tables. Because of the relational nature of ILP, it can directly find patters from multi table databases and other structured data. In contrast, most other machine learning approaches can only deal with data that resides in a single table. As such, they require reprocessing using various types of `join` and `aggregation` operations before they can be applied. However as shown by [30, 31, 11], integrating data trough joining and aggregating tables can lead to the loss of meaning or information. We consider an example similar to the one presented in [11]

for instance. Suppose we are given a database with two tables as shown in Fig. 1.3, and the task is to characterize customers that spend a lot. Please note that each customer can have multiple purchases. Integrating the two relations (each table defines one relation) using join will create a new relation where each row corresponds to a purchase and not to a customer. Alternatively, two tables can be integrated into aggregate customer1(customer_id, age, number_of_purchases, total_value). In this case, however, some information is lost. For example, the following hypothetical hypothesis that an ILP system may learn cannot be directly achieved using either of those integration techniques:

$$\begin{aligned} \text{big_spender(CID)} \leftarrow & \text{customer(CID, PID, date, value, payment_mode)} \wedge \\ & \text{age} > 30 \wedge \\ & \text{payment_mode=credit_card} \wedge \text{ value} > 100, \end{aligned}$$

where CID and PID are short for customer_id and purchase_id, respectively.

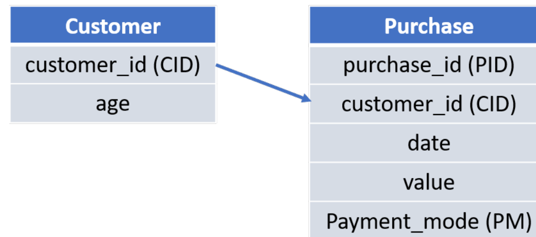


Figure 1.3: An example of relational database

Standard ILP systems such as Metagol are usually not able to directly tackle the problems involving ambiguous and missing data. In classification problems involving relational datasets, no single hypothesis is usually able to classify all the examples correctly. As such, for tackling these tasks, a probabilistic extension of ILP (e.g. probabilistic ILP (PILP) [19] and Markov Logic Networks (MLN) [32]) are usually employed. These models are then usually used to create an ensemble learning algorithm (e.g., via boosting techniques). There

are two downsides to this approach. First, by using ensemble techniques and employing many logical search trees, the interpretability decreases significantly. Moreover, the same probabilistic ILP solvers cannot be employed for other algorithmic tasks involving recursion and predicate invention which limits the applicability of such ILP solvers. In Chapter 4, we use real-world relational datasets including Mutagenesis [33], UW-CSE [32] as well as IMDB, MovieLens (1M) and Cora datasets and we demonstrate the unified design of the dNL-ILP solver makes it possible to seamlessly extend its application to ambiguous and uncertain tasks. Moreover, via experiments we show how the differentiable design of dNL-ILP makes it possible to combine it with standard neural network models to handle continuous data, either by learning boundary decisions or by adopting a probabilistic approach to model the continuous signals.

1.4 Relational Reinforcement Learning

Relational Reinforcement Learning (RRL) is formed in the early 2000s as the intersection of Reinforcement Learning (RL) and Inductive Logic Programming (ILP) and by works such as [34, 35, 36] among others. The main idea behind RRL is to upgrade the typical propositional representation used in reinforcement learning to the first-order relational form. As such, the environment can be described more naturally in terms of objects and relations. As an example, consider an agent playing the simple BoxWorld game. In this game, the three boxes $\{$`a, b, c`$\}$ can be on top of each other or on the floor. Assume the final goal of the agent be to stack all of the boxes on top of each other. The state of this environment can be completely explained in a relational form via groundings of the predicate `On(X,Y)`, implying that the box labeled by `X` is on top of box `Y`. For example, if $S_t = \{on(a, b), on(b, floor), on(c, floor)\}$, the agent can reach the final goal by performing the action `move(c,a)`, i.e., placing box `c` over box `a`. Here, S_t represents the relational state of the system in time t. The first practical implementation of this idea was proposed by [35] and later was improved in [36] which was based on a modification to the famous

Q-Learning algorithm [37] via the standard relational tree-learning algorithm TILDE [38]. The traditional RRL model learns through exploration of the state-space in a way similar to normal Q-learning algorithms. It starts with running a normal episode but uses the encountered states, chosen actions and the received rewards to generate a set of examples that can then be used to build a Q-function generalization. As shown in [36], the RRL system allows for a very natural and human readable decision making and policy evaluation. More importantly, the use of variables in ILP system, makes it possible to learn generally formed policies and strategies. Since these policies and actions are not directly associated with any particular instance and entity, this approach leads to a generalization capability beyond what is possible in most typical RL systems.

In recent years and with the advent of the new deep learning techniques, significant progress has been made to the classical Q-learning RL framework and via algorithms such as deep q-learning and its variants [39, 40], more complex problems are tackled. The classical RRL framework can not be easily employed to tackle the more complex scenes available in recent RL problems. While it is still possible to use the representational power of RRL in the deep RL framework, the explicit relational representation of states, actions and policies and specially the use of variables are lacking in these systems. Since the proposed dNL-ILP solver is a differentiable neural network, it can be combined with the standard deep learning technique. In chapter 5, we formulate a deep relational policy learning algorithm by employing the differentiable dNL-ILP. The various features of this novel RRL system will be studied. To evaluate the performance of this new approach to reinforcement learning, we will use it in RL environments such as BoxWorld, GridWorld and Asterix games. We will also show how this general framework can be used to learn and extract relational information in other benchmark tasks including the visual reasoning tasks for the Sort-of-CLEVR [41] dataset.

1.5 Decoding LDPC codes over Binary Erasure Channels

Unlike the standard DNNs, the dNL networks are specifically designed for learning Boolean functions and are naturally better suited for learning discrete algorithmic tasks. In Chapter 6, we study the generalization performance of the dNL by considering their application in learning a practical algorithmic task. Over the last few years, the application of deep recurrent neural networks in decoding various communication codes, such as LDPC over Gaussian channels, have been extensively studied. For example in [42], the authors successfully employed RNNs to design a very efficient decoder for convolutional and turbo codes over additive white Gaussian noise (AWGN) channels and in [43] the authors proposed a framework for soft decoding of linear codes of arbitrary length. However, as shown in chapter 6, typical neural network layers often fail to learn the exact logical flow in discrete (Binary) arithmetic. This is specially true for decoding LDPC codes over BEC. In Chapter 6, we will demonstrate how the proposed dNL framework can be successfully used to design an LDPC decoder over BEC channels [44]. More specifically, we compare their performance and their generalization ability to the standard neural networks.

1.6 Independent Component Analysis

One of the biggest challenges in unsupervised machine learning is finding nonlinear features from unlabeled data. In the general nonlinear case, there has been some successful applications of the deep neural networks in recent years. Multi-layer deep belief networks, Restricted Boltzmann Machine (RBM) and most recently, variational Autoencoders are some of the most promising new developments. Independent component analysis (ICA) is one of the most basic and successful classical tools in unsupervised learning which has been studied extensively during the early years of 2000s. For the simplest case of linear ICA without the presence of noise, this problem has been solved and the conditions for the identifiability of independent components has been well established [45]. However, for the

more general form of the problem where the independent features (sources) are mixed using nonlinear functions and specifically in the presence of noise, there has been little success. In recent years, the problem of learning ICA from the nonlinear mixtures (and similar related problems) has been addressed in works such as [46, 47]. In particular, in [47] the Adversarial Nonlinear Independent Component Analysis (Anica) algorithm was proposed which is an unsupervised learning method based on Generative Adversarial Networks (GAN) [48].

The problem of linear ICA in the presence of noise has been studied in [49, 50, 51, 52, 53] among others. Some of the most successful approaches (e.g. [52, 53]) use the Mean-Field variational Bayesian framework to solve the linear ICA in a Bayesian setting. However, these approaches are usually very slow and have very limited applicability even for the linear case. In Chapter 7, we formulate the general non-linear ICA with the additive Gaussian noise via employing a variational autoencoder framework [54]. The properties of the new proposed ICA framework will be analyzed and its robustness to the noise compared to the previous methods will be examined.

CHAPTER 2

DIFFERENTIABLE NEURAL LOGIC NETWORKS

2.1 Introduction

Although a simple single perception layer equipped with a threshold (e.g., sign) function as the nonlinearity can mimic the behavior of the OR and AND operators, these models are very difficult to train in general. Consider the conjunction operator as an example. Let assume we have an input vector $\mathbf{x} \in \{0,1\}^n$ and a fully connected layer, consisted of n weights and a bias term, is used to learn the logical function $s = x_1 \bigwedge x_2$. It is clear that by setting $w_1 = w_2 = 1,\ w_{i>2} = 0$, and using a suitable bias term (i.e., $bias = -1$) followed by the $sigmoid(cz)$ activation function, where $c > 1$, the output resembles the desired Boolean function $s = x_1 \bigwedge x_2$. Consider now the case for $s = x_1 \bigwedge x_2 \bigwedge x_3$. In this case, the bias should be set as -2. A good example of these types of differentiable Boolean functions can be found in [10]. We found that however, in general these models are very difficult to train in more difficult problems specially since they are dependent on a bias term. Further, their flexible weights make it difficult to extract the underlying explicit Boolean representation in a simple and intuitive way. This is specially true when combining multiple layers in order to learn more complex Boolean functions in a general Disjunctive Normal Form (DNF) as an example.

In this chapter, we introduce a new design for the logical operators by using membership weights without the use of an adjustable bias terms[55]. First, we design a general purpose conjunction and disjunction layers. Then, we introduce the exclusive OR layer which enhances the capacity of the network significantly. We would then demonstrate the properties and characteristics of the proposed model in three areas :

- *Learning Boolean functions efficiently*: In Section 2.4, we demonstrate how the

proposed method compares to the MLP design in learning Boolean functions using two simplistic and synthetic examples.

- *Generalization*: In Chapter 6, we show how well the proposed dNL based models can generalize in discrete iterative algorithms compared to the traditional MLP by comparing their performance in learning a message passing decoder for Low Density Parity Check Codes (LDPC) over erasure channels.

- *Explicit symbolic representation*: Unlike MLP, the learned Boolean functions in dNL can be expressed explicitly. In Chapter Inductive Logic Programming via dNL, we propose a new paradigm for solving ILP problems efficiently by exploiting the explicit representational power of dNL networks. The proposed method is able to use features such as predicate invention, recursion and functions while using a remarkably small number of training weights.

2.2 Neural Conjunction and Disjunction Layers

Let $\{x_1, \ldots, x_n\}$ be the input vector consists of n Boolean variables and consider the case of conjunction function. To each variable x_i we associate a Boolean flag m_i such that it control the inclusion or exclusion of the term x_i in the resulting conjunction function. To this end, we define the following Boolean function:

$$f_{conj} = (\overline{m_1} \vee x_1) \wedge \cdots \wedge (\overline{m_n} \vee x_n) \tag{2.1}$$

It is easy to see that when m_i is True, the corresponding term simplifies to x_i and when m_i is False, x_i term is excluded from the expression. We can similarly design a parameterized disjunction function:

$$f_{disj} = (m_1 \wedge x_1) \vee \cdots \vee (m_n \wedge x_n) \tag{2.2}$$

x_i	m_i	F_c
0	0	1
0	1	0
1	0	1
1	1	1

(a)

x_i	m_i	F_d
0	0	0
0	1	0
1	0	0
1	1	1

(b)

Figure 2.1: Truth table of $F_c(\cdot)$ and $F_d(\cdot)$ functions

As an example, if m_1 and m_2 are True and for $i > 2$, $m_i = False$, the corresponding functions reduce to $f_{conj} = x_1 \wedge x_2$ and $f_{disj} = x_1 \vee x_2$, respectively.

In order to design a differentiable network, we use the continuous relaxation of Boolean, i.e., we assume fuzzy Boolean values are real values in the range $[0, 1]$ and we use 1 (True) and 0 (False) representations for the two states of a binary variable. We also define the fuzzy unary and dual Boolean functions of two Boolean variables x and y as:

$$\bar{x} = 1 - x \quad , \quad \bar{y} = 1 - y \tag{2.3a}$$

$$x \wedge y = xy \tag{2.3b}$$

$$x \vee y = \overline{\bar{x} \wedge \bar{y}} = 1 - (1 - x)(1 - y) \tag{2.3c}$$

This algebraic representation of the Boolean logic allows us to manipulate the logical expressions via Algebra. Let $\mathbf{x}^n \in \{0, 1\}^n$ be the input vector in a typical logical neuron. To implement the fuzzy conjunction function, we would like to select a subset in $\mathbf{x}^n$ and apply the fuzzy conjunction (i.e. multiplication) to the selected elements. One way to accomplish this is to use a `softmax` function and select the elements that belong to the conjunction function similar to the concept of pointer networks [56]. This requires knowing the number of items in the subset (i.e. the number of terms in the conjunction function) in advance. Moreover, in our experiment we found that the convergence of model using this approach is very slow for larger input vectors. Alternatively, inspired by the equations (2.2) and (2.1), we associate a trainable Boolean membership weight m_i to each input elements x_i from

vector $\mathbf{x}^n$. Further, we define a Boolean function $F_c(x_i, m_i)$ with the truth table as in Fig. 2.1a which is able to include (exclude) each element in (out of) the conjunction function. This design ensures the incorporation of each element x_i in the conjunction function only when the corresponding membership weight is 1. Consequently, the neural conjunction function can be defined as:

$$O_{conj}(\mathbf{x}) = \prod_{i=1}^{n} F_c(x_i, m_i)$$
$$\text{where, } F_c(x_i, m_i) = \overline{x_i \overline{m_i}} = 1 - m_i(1 - x_i)\,, \tag{2.4}$$

where O_{conj} is the output of conjunction neuron. To ensure the trainable membership weights remain in the range $[0, 1]$ we use a `sigmoid` function, i.e. $m_i = sigmoid(c\,w_i)$ where $c \geq 1$ is a constant. Similar to perceptron layers, we can stack m neural conjunction neurons to create a conjunction layer of size m. This layer has the same complexity as a typical perceptron layer without incorporating any bias term. More importantly, this way of implementing the conjunction layer makes it possible to interpret the learned Boolean function directly from the values of the membership weights.

The disjunctive neuron can be defined similarly by introducing membership weights but using the function F_d with truth table as depicted in Fig. 2.1b. This function ensures an output 0 from each element when the membership is zero which correspond to excluding the x_i element from the neuron outcome. Therefore, the neural disjunction function can be expressed as:

$$O_{disj}(\mathbf{x}) = \overline{\prod_{i=1}^{n} \overline{F_d(x_i, m_i)}} = 1 - \prod_{i=1}^{n}(1 - F_d(x_i, m_i))\,,$$
$$\text{where, } F_d(x_i, m_i) = x_i m_i \tag{2.5}$$

By cascading a conjunction layer with a disjunctive layer, we can create a multi-layer structure which is able to learn and represent Boolean functions using the Disjunctive

Normal Form (DNF). Similarly we can construct the Conjunctive Normal Form (CNF) by cascading the two layers in the reverse order. The total number of possible logical functions over a Boolean input vector $\mathbf{x} \in \{0,1\}^n$ is very large (i.e. 2^{2^n}). Further, in some cases, a simple clause in one of those standard forms can lead to an exponential number of clauses when expressed in the other form For example, it is easy to verify that converting $(x_1 \vee x_2) \wedge (x_3 \vee x_4) \wedge \cdots \wedge (x_{n-1} \vee x_n)$ to DNF leads to $2^{\frac{n}{2}}$ number of clauses. As such, using only one single form of Boolean network for learning all possible Boolean functions is not always the best approach. The general purpose design of the proposed conjunction and disjunction layers allows us to define the appropriate Boolean function suitable for the problem.

2.3 Neural XOR Layer

Exclusive OR (XOR) is another important Boolean function which has been the subject of many researches over the years, especially in the context of parity-N learning problem. It is easy to verify that expressing XOR of an n-dimensional input vector in DNF form requires 2^{n-1} clauses. Although, it is known that it cannot be implemented using a single perceptron layer [8, 57], it can be implemented, for example, using multi-layer perceptron or multiplicative networks combined with small threshold values and sign function activation [58]. However, none of these approaches allow for explicit representation of the learned XOR functions directly. Here we propose a new algorithm for learning XOR function (or equivalently the parity-N problem). To form the logical XOR neuron, we first define k functions of the form:

$$
\begin{aligned}
f_1(\mathbf{x}) &= x_1 + x_2 + \cdots + x_k \boxed{-x_{k+1} - \cdots - x_n} \\
f_2(\mathbf{x}) &= x_1 + x_2 + \dots \boxed{-x_k - \cdots - x_{n-1}} + x_n \\
\vdots \quad & \quad \vdots \quad \quad \vdots \quad \quad \vdots \quad \quad \vdots \\
f_k(\mathbf{x}) &= x_1 \boxed{-x_2 - \cdots - x_k - x_{k+1}} + \cdots + x_n
\end{aligned}
\tag{2.6}
$$

where $k = \frac{n}{2}$ (assuming n is even). Then, we define the XOR function as in Theorem 1.

Theorem 1. *Given the set of k functions as defined in (2.6) we have:*

$$XOR(\mathbf{x}) = g_1(\mathbf{x}) \bigwedge g_2(\mathbf{x}) \bigwedge \cdots \bigwedge g_k(\mathbf{x}) \,, \tag{2.7a}$$

$$\textit{where, } g_i(\mathbf{x}) = \begin{cases} 0 & \textit{if } \; f_i(\mathbf{x}) = 0 \\ 1 & else \end{cases} \tag{2.7b}$$

Proof. See Appendix A. □

Inspired by the Theorem.1, we design our XOR neuron as:

$$O_{XOR}(\mathbf{x}) = \prod_{i=1}^{k} hs\Big(\left| \mathbf{x} \times (\mathbf{M}_i \odot \mathbf{w})^T \right| \Big) \tag{2.8}$$

Here, $hs(\cdot)$ is the `hard-sigmoid` function, and $\times$ and $\odot$ denote matrix and element-wise multiplication correspondingly. Further, vector $M_i \in \{-1, 1\}^n$ is the set of coefficients used in $f_i(\mathbf{x})$ and $\mathbf{w}$ is the vector of membership weights. The resulting XOR logical neuron uses only one weight variable per input element for learning. However, its complexity is k times higher than the conjunction and disjunction neurons for an input vector of length $n = 2k$.

2.4 dNL vs MLP

To evaluate the proposed dNL model in learning Boolean functions we compare their performance to the standard MLP networks for the task of learning Boolean functions using two synthetic experiments. It is worth noting that in these experiments we are not concerned with the interpretability. Needless to say, while dNL network generates an easily interpretable Boolean functions, the MLP counterpart does not provide explicit representation of the Boolean formulas.

Learning DNF form

For this experiment, we randomly generate some Boolean functions over a 10 bits input vectors and a randomly generated batches of 50 samples as training data. We train two models; one designed via our proposed DNF network (with 200 disjunction functions) and another designed by two layers MLP network with hidden layer of size 1000 and `relu` activation and use `sigmoid` activation function for the output layer. We use ADAM optimizer [59] with learning rate of 0.001 for both models and count the number of errors in 1000 randomly generated test samples. When we used a Bernoulli distribution with parameter $p = 0.5$ (i.e. fair coin toss) for generating the bits of each training samples, both models quickly converge and the number of test error drops to zero for both. However, in many realistic discrete problems, the 0's and 1's are not usually equiprobable. As such, next we use Bernoulli with parameter $p = 0.75$. Fig. 2.2a depicts the comparative performance of the two models. The proposed DNF model converges fast and remains at 0 error. On the contrary, the MLP model continues to generate errors. In our experiments, while the number of errors decreases as training continues for an hour, the MLP model never fully converges to the true logical function and occasionally generates some errors. While for some tasks, this may be a negligible error, in some logical applications such as the ILP learning task, this behavior prevents the model from learning.

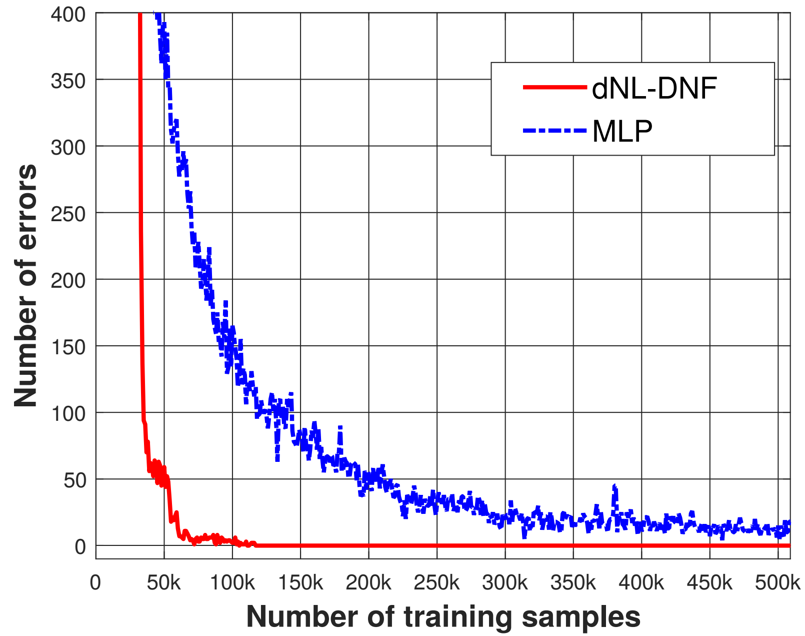
Learning XOR function

Next, we compare the two models for a much more complex task of learning the XOR logic. We use a multi layer MLP with `relu` as activation functions in the hidden layers and `sigmoid` function in the output layer as usual. As for dNL, we use a single XOR neuron as described in (2.8). For the small size inputs both models quickly converge. However, for larger size input vectors ($n > 30$) the MLP model fails to converge at all. Fig. 2.2b shows the average bit error over the number of training samples. We tried various setups for MLP, including different number of hidden layers (from 1 to 3) and different sizes of the hidden

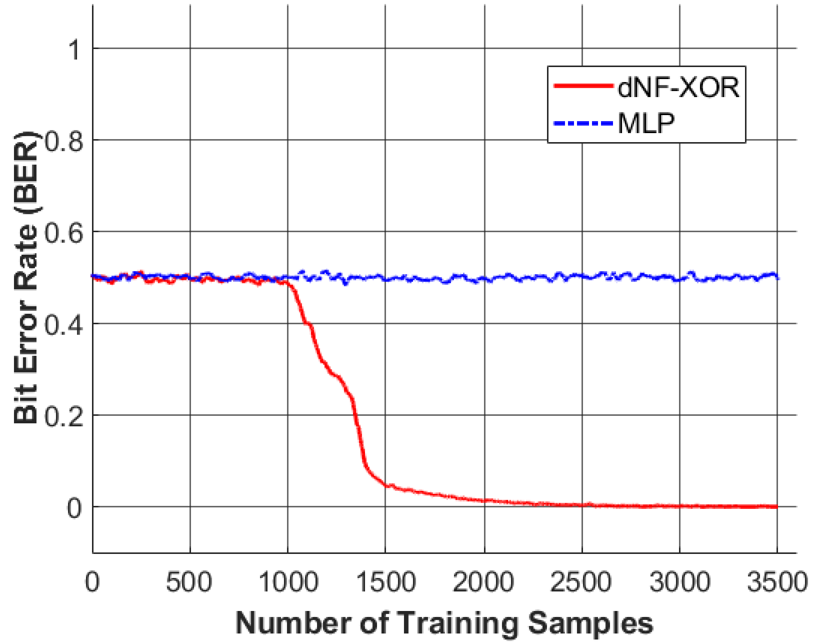
layers. For both models we evaluated the performance for 100 randomly seeded runs and we report the best results for each model. The error rate for MLP was around $.5$, which indicates it failed to learn the XOR function. On the contrary, the XOR logic layer was able to converge and learn the objective in most of the runs. This is significant considering the fact that the number of parameters in our proposed XOR layer is equal to the input length, i.e., one membership per input variable. In conclusion, although a typical MLP structure is very powerful and can encode many complex functions using the adjustable weights, it lacks the rigidness and specificity of our proposed Logical Layers. As such, often it is difficult to combine MLP layers effectively and simply adding more layers will not increase the network performance in many problems. On the contrary, the logical layers, by design, are forced to learn describable logical relationship between their inputs. As such, combining them in various arrangements enables the network to learn very complex logical flows.

2.5 Binary Arithmetic

Binary arithmetic tasks are among the widely used experiments in the topic of algorithm and pattern learning via DNNs. While binary operations are quite easy to learn for small size binary numbers, learning becomes very difficult as the size increases. This problem has been tackled in two different ways. In the feed-forward approach, the models are designed such that the complete input stream of symbols is processed sequentially through multiple layers and the output of the last layer generates the stream of symbols corresponding to the desired result of the arithmetic operation all at once (e.g., [7]). Alternatively, this problem can be looked at as a case of the sequence-to-sequence learning problem [4] where we first process the input symbols one by one using a recurrent cell (e.g., an LSTM cell) and then another recurrent cell generates the output sequence of symbols sequentially while accessing all the outputs of the first recurrent layer generated at each time stamp. Various attention mechanisms and memory access models have also been proposed to improve the performance of these recurrent networks [60]. To evaluate our proposed logical layers in

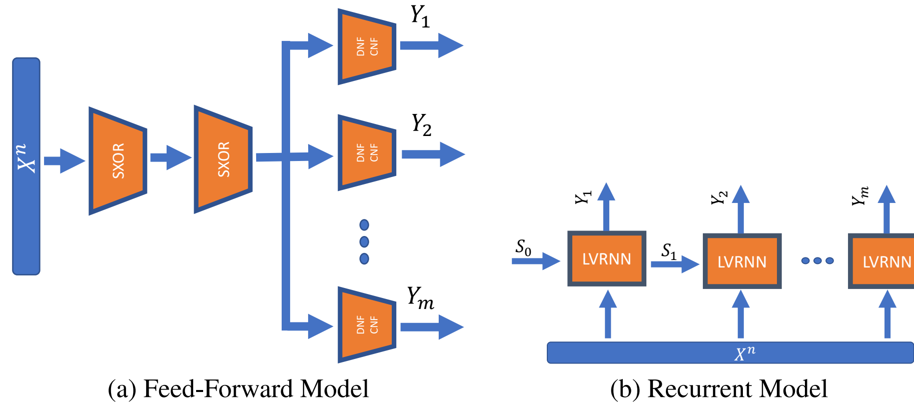


(a) DNF Task



(b) Xor 50 Task

Figure 2.2: Comparing MLP vs dNL for learning Boolean functions



(a) Feed-Forward Model (b) Recurrent Model

Figure 2.3: Feed-Forward and recurrent models used in our experiments

such a standard task, we first consider the addition of two long binary numbers. We represent each digit in a one-hot vector with $depth = 4$ to denote $\{0, 1, +, space\}$ symbols, where $space$ is used for padding the sequences to have identical lengths. Among the existing models in the literature, Neural GPU model [7] proved to be the most successful approach in solving such problems using a feed-forward network. In [7], the authors proposed a model consisted of a multi-layer feed-forward network where each layer is designed using a recurrent convolutional GRU cell [61]. This model (combined with curriculum learning) is reported to attain 100% accuracy for the addition of up to 2000 long binary numbers. In our experiments, we were also able to achieve 100% accuracy in binary addition of any lengths we tested (we did not test above 200 long sequences though).

We trained the Neural GPU using its default set of parameters but for our model we used two layers of XOR logic with length 100 each followed by a set of DNF-CNF layers each with a hidden layer of size 50 in our feed-forward model (as shown in Fig. 2.3a). Table 6.1 compares the speed of the convergence for the two models by displaying the number of the training samples required to reach a certain level of accuracy for $n = 10$ and $n = 20$ digits inputs. As Table 6.1 suggests while both models are capable of reaching 100% accuracy, our proposed model converges much faster. Our experiments show that our model reaches 99% accuracy for various lengths in less than 200k training samples. However, it usually requires much more training samples to reach the 100% accuracy level.

Table 2.1: Size of required training samples to reach certain levels of accuracy

Model \ Accuracy	75%	90%	100%
Neural GPU (n=10)	77k	400k	490k
Logic CNF-DNF (n=10)	4k	25k	260k
Neural GPU (n=20)	400k	448k	830k
Logic CNF-DNF (n=20)	8k	45k	610k

2.6 Grammar Verification

In many of the works related to the algorithm learning tasks, the models are claimed to reach 100% accuracy whenever they attain perfect accuracy on the validation set during the training. However, we confirmed via many experiments that even though such models reduced the error rate significantly on the average, they rarely learn the underlying logical flow of the problem fully. In other words, even after achieving the perfect accuracy, these models usually continue to fluctuate. In contrast, the proposed logical layer model is usually able to learn the underlying logical flow. Here, we compare the efficiency of the Long Short Term Memory (LSTM)[62] recurrent cell with our proposed LVRNN recurrent cell for the problem of understanding Palindrome grammar. The language of Palindrome grammar contains all strings in the form WcW' where W is an arbitrary string of symbols from some alphabet, W' is the reverse of the string W, and c is a divider symbol. For example, in the binary alphabet system, "$11001c10011$" is a legal Palindrome string but "$11000c11000$" is not. Because the number of illegal phrases is much larger than the legal phrases, we ensure that around 5% of the training samples for each training batch are legal phrases. We use the model depicted in Fig. 2.3b and use only the last output of the recurrent network to create a discriminator network with the help of a dense layer. We train the network for the phrases of length 41 binary symbols using an ADAM optimizer [63] with the learning rate of 0.001. Fig. 2.4 compares the performance of the two models, in terms of the number of mismatches per 5000 samples processed during training of each epoch, versus the epoch number. We observed that although the MLP based network approaches to the perfect accuracy and can

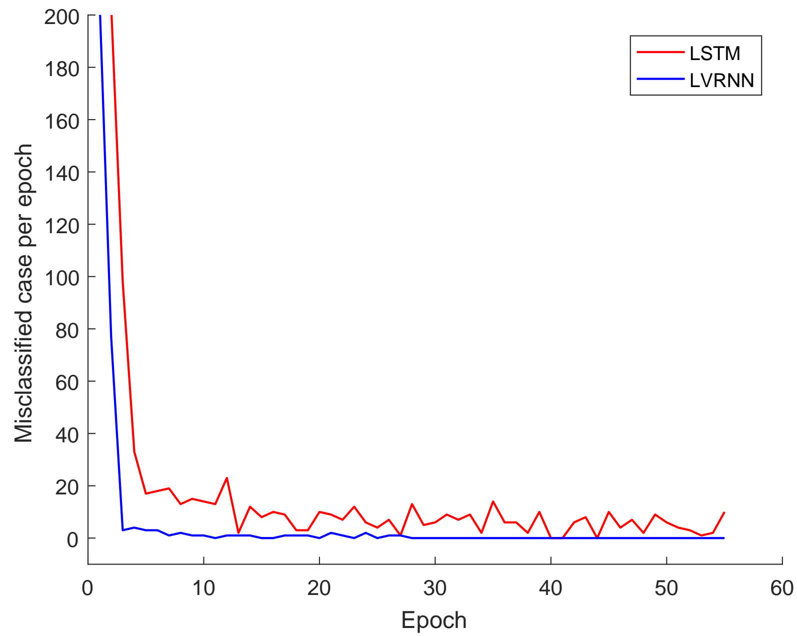


Figure 2.4: Accuracy results for Palindrome grammar check

produce perfect results in a few epochs, it fails to fully learn the underlying pattern and continues to fluctuate and generate a few errors. On the contrary, our proposed logical layers cell (LVRNN) converges faster and reaches the perfect accuracy level without any further fluctuations afterwards.

CHAPTER 3
INDUCTIVE LOGIC PROGRAMMING VIA DNL

3.1 Introduction

In the previous chapter the performance of dNL networks in learning binary arithmetic tasks was evaluated using a variety of algorithmic tasks involving discrete and binary data. In those problems, instead of focusing on interpretability of the learned relations, we were mainly concerned with the learning performance. One of the main characteristics of dNL networks is that while they can efficiently learn discrete and binary tasks, they also provide an intuitive and easy way of interpreting the underlying Boolean function. This becomes clear by observing the design of any logical neuron from dNL. For instance, consider the conjunction function defined as $f_{conj} = (\overline{m_1} \vee x_1) \wedge \cdots \wedge (\overline{m_n} \vee x_n)$. Since in the fuzzy implementation of this function, we are associating a (fuzzy Boolean variable m_i to each element of the input vector, i.e., x_i, we can directly create a valid fuzzy Boolean formula from the weights of the dNL networks. As an example, it is easy to see that if $m_1 = 1$ and $m_3 = 1$ and other membership weights of the conjunctive neuron are equal to 0, the resulting Boolean function can be interpreted as the Boolean function of $f_{conj} = x_1 \wedge x_3$. This is also the case for multi-layer design. Since the representation in each layer is explicit, if we cascade two conjunction and disjunction layers to form a dNL-DNF structure, the resulting multi-layer network still represents the Boolean logic in an explicit manner which is easily interpretable and human-readable.

Inductive Logic Programming (ILP) is an important branch of machine learning that uses formal logic to define and solve problems. Since in this framework the ability to express and learn symbolic Boolean functions is usually required, standard DNN models are not easy to use. Indeed, most of the neural implementations of ILP provide no explicit representation of

the learned logic. In this chapter we demonstrate how to exploit the explicit representational power of dNL to design a neural based differentiable ILP solver which unlike most neural based approaches provides interpretable hypothesis.

3.2 Logic Programming

Logic programming is a programming paradigm in which we use formal logic (and usually first-order-logic) to describe relations between facts and rules of a program domain. In this framework, a definite clause is a rule of the form:

$$H \leftarrow B_1,\ B_2,\ \dots,\ B_m \tag{3.1}$$

where H is called `head` of the clause and $B_1,\ B_2,\ \dots,\ B_m$ is called `body` of the clause. This rule expresses that if all the atoms in the `body` are true, the `head` is necessarily true. We assume each of the terms H and B are made of `atoms`. Each `atom` is created by applying an n-ary Boolean function called `predicate` to some constants or variables. A `predicate` states the relation between some variables or constants in the logic program. Throughout this chapter we will use small letters for constants and capital letters $(A, B, C, \dots)$ for variables. In ILP, a problem can be defined as a tuple ($\mathcal{B}$, $\mathcal{P}$, $\mathcal{N}$) where $\mathcal{B}$ is the set of background assumptions and $\mathcal{P}$ and $\mathcal{N}$ are the set of positive and negative examples, respectively. Given this setting, the goal of the ILP is to construct a logic program (usually expressed as a set of definite clauses, $\mathcal{R}$) such that it explains all the examples. More precisely,

$$\mathcal{B}, \mathcal{R} \models e, \forall e \in \mathcal{P} \quad , \quad \mathcal{B}, \mathcal{R} \not\models e, \forall e \in \mathcal{N}, \tag{3.2}$$

where, $\models$ and $\not\models$ are the logical entailment operator and its negation, respectively. Let's consider the logic program for learning the `lessThan` predicate over natural numbers and assume that our constants contains the set $\mathcal{C} = \{0, 1, 2, 3, 4\}$ and the ordering of the natural numbers is defined using the predicate `inc` (which defines increments of 1). The set of

background atoms which describe the known facts about this problem is:

$$\mathcal{B} = \{\text{inc}(0,1),\ \text{inc}(1,2),\ \text{inc}(2,3),\ \text{inc}(3,4)\} \tag{3.3}$$

Further, we can easily provide the positive and negative examples as:

$$\begin{aligned} \mathcal{P} &= \{lt(a,b)\,|\,a,b \in \mathcal{C},\ a < b\} \\ \mathcal{N} &= \{lt(a,b)\,|\,a,b \in \mathcal{C},\ a \geq b\} \end{aligned} \tag{3.4}$$

It is easy to verify that the program with rules defined in the following entails all the positive examples and rejects all the negative ones:

$$\begin{aligned} &\text{lessThan}(A,B) \leftarrow \text{inc}(A,B) \\ &\text{lessThan}(A,B) \leftarrow \text{inc}(A,C),\ \text{lessThan}(C,B) \end{aligned} \tag{3.5}$$

We can view the ILP as a supervised learning problem that generates a valid set of rules such as (3.5), given the input facts in (3.3) and the training labels provided as in (3.4). The `consequences` of applying the rules of the program are obtained via `forward chaining`. Following the notation used in [27], define $cn_R(X)$ as immediate consequence of applying rules R on the set of ground atoms in X, i.e.:

$$cn_R(X) = X \bigcup \{\gamma\,|\,\gamma \leftarrow \gamma_1, \ldots, \gamma_m \in Ground(R), \gamma_i \in X\}, \tag{3.6}$$

where $Ground(R)$ are the ground rules of R. If we start from the set of background facts, we can iteratively obtain all the consequences of the rules. As an example, for the logic

program `lessThan` we can obtain the consequences of the two defined rules in (3.5) as:

$$
\begin{aligned}
Cn_R^{(0)} &= \mathcal{B} = \{\text{inc}(0,1),\ \text{inc}(1,2),\ \text{inc}(2,3),\ \text{inc}(3,4)\} \\
Cn_R^{(1)} &= Cn_R^{(0)} \bigcup \{lt(0,1),\ lt(1,2),\ lt(2,3),\ lt(3,4)\} \\
Cn_R^{(2)} &= Cn_R^{(1)} \bigcup \{lt(0,2),\ lt(1,3),\ lt(2,4)\} \\
Cn_R^{(3)} &= Cn_R^{(2)} \bigcup \{lt(0,3),\ lt(1,4)\} \\
Cn_R^{(4)} &= Cn_R^{(3)} \bigcup \{lt(0,4)\},
\end{aligned}
\tag{3.7}
$$

where, we use lt as shorthand for lessThan and the $Cn_R^{(t)}$ denotes the consequences at time stamp t. Notice that applying the rules after the time stamp $t = 4$ does not yield any new consequences.

In general, there are two main approaches used by various ILP solver systems to generate a valid hypothesis:

- In the *bottom-up* family of approaches (e.g., Progol), the solver starts by examining the provided examples and extracts specific clauses from those and tries to generalize from those specific clauses.

- In the *top-down* approaches (e.g., most neural implementations such as dILP [27] as well as Metagol), the possible clauses are generated via a template and the generated clauses are tested against positive and negative examples.

Since the space of possible clauses are vast, in most of these systems, very restrictive template rules are employed to reduce the size of the search space. For example, dILP [27] allows for rules with at most two atoms in the body and only two rules per each predicate. Metagol [12] employs a more flexible approach by allowing the programmer to define the rule templates via some meta-rules. However, in practice, this approach does not resolve the issue completely. Even though it allows for more flexibility, defining those templates is itself a complicated task which requires expert knowledge and possible trials and it can still

lead to exponentially large space of possible solutions.

Alternatively, we propose a novel approach which allows for learning an arbitrary complex Boolean functions as rules. Here, we describe the intuition behind our proposed approach. In most ILP systems the rules are restricted to Horn clauses. A Horn clause is a clause with at most one positive literal (atom). For example, (3.1) defines a generic horn clause in the implication form (if-then form). In our system, instead of restricting the rules to the Horn clauses, we can consider a more general form of clause where the body could be defined as a DNF (disjunctive normal form) Boolean function. In this view, multiple Horn clause rules involving each predicate can be viewed as disjunctive terms in a single DNF rule. For example, the two rules in (3.5) can be combined into a single rule described as:

$$\text{lessThan}(A, B) \leftarrow \text{inc}(A, B) \vee (\text{lessThan}(A, C) \wedge \text{inc}(C, B)) \tag{3.8}$$

where $\vee$ and $\wedge$ represent Boolean disjunction and conjunction functions, respectively. It is easy to see that the consequences of these two forms (i.e., (3.5) and (3.8)) are equivalent. We recall that the differentiable dNL-DNF functions introduced in Chapter 2 are able to learn and represent any arbitrary DNF functions. We associate a dNL-DNF function to each DNF rule in our logic programs. We can view any dNL-DNF function, as a differentiable symbolic Boolean function, parameterized by ϕ. Here, ϕ represents the set of all membership weights for the conjunctive and disjunctive neurons used in designing a dNL-DNF function. We notice that each instance of ϕ corresponds to a distinct Boolean function. At each point in the learning, given the current parameters ϕ, we can apply the forward chaining and obtain all the consequences of the rules iteratively. By comparing the obtained consequences against the provided examples (i.e., $\mathcal{P}$ and $\mathcal{N}$), we can adjust ϕ. Moreover, since this parameterization is differentiable, this approach leads us to reformulating the ILP as a differentiable problem which can be learned by a gradient based optimization technique.

3.3 Formulating the dNL-ILP

We assume that for each predicate we can define N_p rules. We do not restrict the rules to Horn clauses. We assume the body of rules can be represented by a (fuzzy) Boolean function from the dNL, i.e., $\mathscr{F}_p^i$ and the head of each rule contains only one atom. If we allow for $num_var^i(p)$ variables in the body of the ith rule for the predicate p, the set of possible (symbolic) atoms in the body of that rule is given by:

$$\mathbb{I}_p^i = \bigcup_{p^* \in \mathbb{P}} \mathbb{T}(p^*, V_p^i)\,, \text{ where} \tag{3.9}$$

$$\mathbb{T}(p, V) = \{p(arg) \,|\, arg \in Perm(V,\, arity(p)\,)\,\} \tag{3.10}$$

where V_p^i is the set of variables ($|V_p^i| = num_var^i(p)$) for the ith rule and $\mathbb{P}$ is the set of all the predicates in the program. Further, the function $Perm(S, n)$ generates the set of all the permutations of tuples of length n from the elements of a set S and the function $arity(p)$ returns the number of arguments in predicate p.

As an example, assume that in the `lessThan` program, we allow for one rule for learning the predicate `lessThan`, and we set $num_var^1(lt) = 3$, i.e., $V_{lt}^1 = \{A, B, C\}$. Consequently, the set of possible atoms that can be used in the definition of the predicate `lessThan` can be enumerated as:

$$\mathbb{I}_{lt}^1 = \{\text{inc}(A, A), \text{inc}(A, B), \text{inc}(A, C), \ldots, \text{inc}(C, C)\}$$
$$\bigcup \{lt(A, A), lt(A, B), \ldots,\, lt(C, C)\}$$

Please note the neural dNL functions are parameterized by membership weights but they are not symbolic functions themselves. For each substitution θ (of constants into variables), the actual fuzzy Boolean vector that is passed to a dNL network during the program runtime, is the Boolean vector ($\mathbb{I}_p^i|_\theta$) generated by the substitution θ of constants into the variables in $\mathbb{I}_p^i$.

In other words, $\mathbb{I}_p^i|_\theta$ is the vector corresponding to the groundings of the atoms in set $\mathbb{I}_p^i$.

3.3.1 Forward Chaining

We are now able to formulate the ILP problem as an end-to-end differentiable neural network. Let G be the set of all ground atoms and G_p be the subset of G associated with predicate p. We associate a (fuzzy) value vector to each predicate p at time-stamp t as $X_p^{(t)}$ which holds the (fuzzy) Boolean values corresponding to the groundings of p. We set the initial values of the valuation vectors via the background atoms. i.e.,

$$\forall p,\, \forall e \in G_p, \quad X_p^{(0)}[e] = \begin{cases} 1 & \text{if } \; e \in \mathcal{B} \\ 0 & else \end{cases} \tag{3.11}$$

For the example in consideration (i.e., `lessThan`), we will have two valuation vectors corresponding to the two predicates in the program; the vector $X_{\text{inc}}^{(t)}$ includes the Boolean values for the atoms in $\{\text{inc}(0,0), \text{inc}(0,1), \ldots, \text{inc}(4,4)\}$ and similarly, the vector $X_{\text{lt}}^{(t)}$ includes the Boolean values for the atoms in $\{\text{lt}(0,0), \text{lt}(0,1), \ldots, \text{lt}(4,4)\}$.

For every ground atom $e \in G_p$, let $\Theta_p^i(e)$ be the set of all the substitutions which would result in the atom e. For example, in the `lessThan` program, for the ground atom $lt(0,2)$ we have $\Theta_{lt}^2(\, lt(0,2)\,) = \{\{A \mapsto 0, B \mapsto 2, C \mapsto 0\}, \ldots, \{A \mapsto 0, B \mapsto 2, C \mapsto 4\}\}$, where, $X \mapsto x$ denotes the substitution of constant x into variable X. For every predicate p, we can now define the one step forward inference at time stamp t as:

$$\begin{aligned} &\forall e \in G_p, \\ &X_p^{(t+1)}[e] = X_p^{(t)}[e] \bigvee \left(\bigvee_{i=1}^{N_p} \bigvee_{\theta \in \Theta_p^i(e)} \mathscr{F}_p^i|_\theta \right) \end{aligned} \tag{3.12}$$

To define forward chaining, we must distinguish among predicates, as explained in the following. In general there are two types of predicates. `extensional` predicates

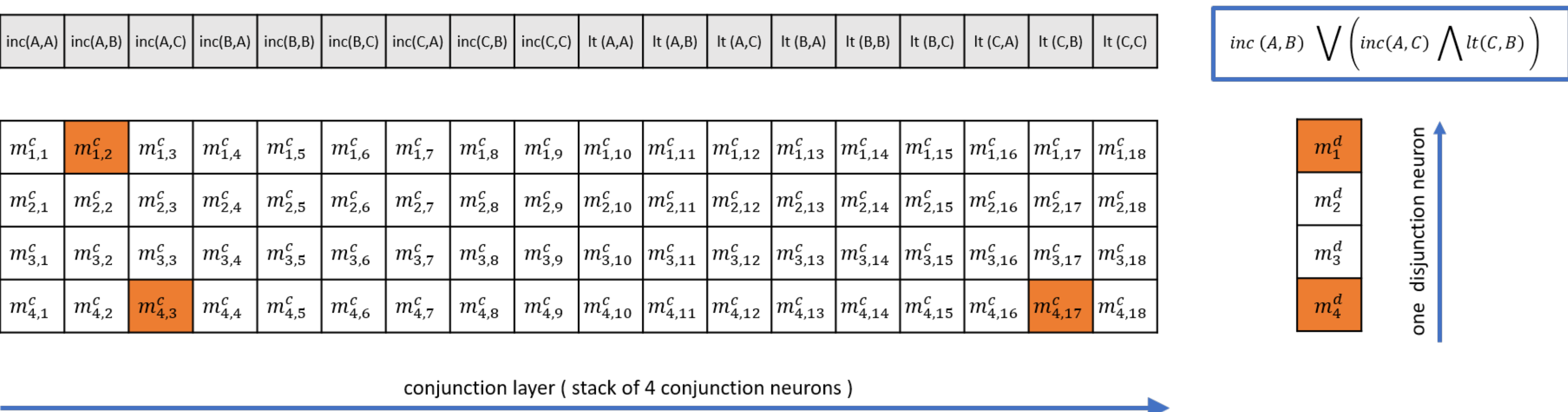


Figure 3.1: Internal representation of $\mathscr{F}^i_{lt}$ via a dNL-DNF function with of hidden layer of size 4

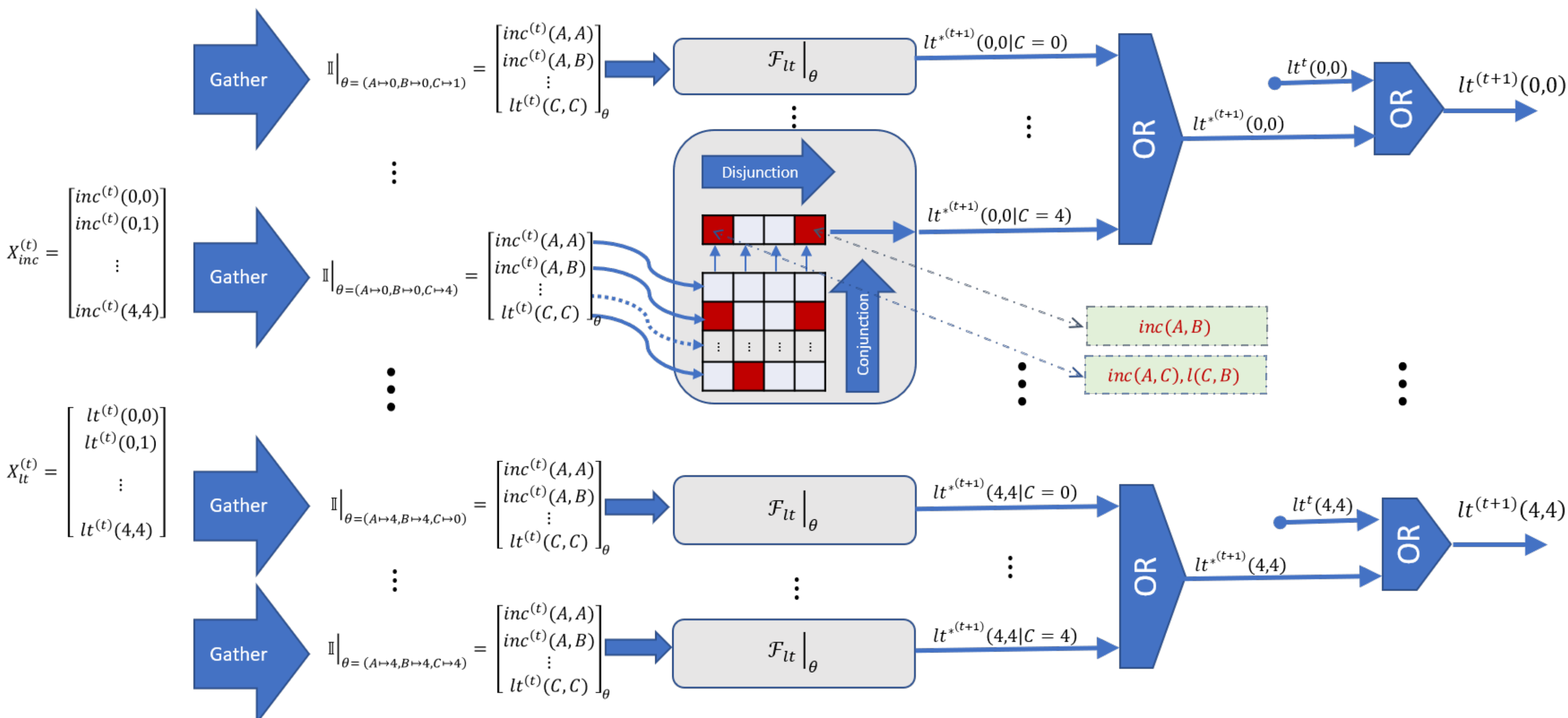


Figure 3.2: The diagram for one step forward chaining for predicate `lt` where $\mathscr{F}_{lt}$ is implemented using a dNL-DNF network.

Table 3.1: Some of the notations used in this chapter

Notation	Explanation
p/n	a predicate p of arity n
$\mathcal{C}$	the set of constants in the program
$\mathcal{B}$	the set of background atoms
$\mathcal{P}$	the set of positive examples
$\mathcal{N}$	the set of negative examples
G_p	the set of ground atoms for predicate p
G	the set of all ground atoms
$\mathbb{P}$	the set of all predicates in the program
$X_p^{(t)}$	the fuzzy values of all the ground atoms for predicate p at time t
$A \mapsto a$	substitution of constant a into variable A
$\mathcal{R}$	the set of all the rules in the program
$perm(S, n)$	the set of all the permutations of tuples of length n from the set S
$\mathbb{T}(p, V)$	the set of atoms involving predicate p and using the variables in the set V
$Cn_R^{(t)}$	the consequences of applying rules in R at timestamp t
$\mathbb{I}_p^i$	the set of all atoms that can be used in the body of ith rule for predicate p

are those predicates that are solely defined by the background facts. On the other hand, `intensional` predicates use other predicates and also variables in their definition. In the `lessThan` example, predicate `inc` in extensional and the target predicate `lessThan` is an intensional predicate. Please note that the forward chaining is only applied for the intensional predicates. For the extensional predicates, the groundings of the predicate is not changing during the forward chaining and is given by the background knowledge. Further, For brevity, we did not introduce the indexing notations in (3.12). By $X_p^{(t)}[e]$, we actually mean the entry in X_p^t corresponding to the grounding e. Algorithm 1 shows the outline of the t_{max} steps of forward chaining in the proposed dNL-ILP via the application of the equation (3.12).

To understand the equation (3.12) and the Algorithm 1, we use our `lessThan` example once again. Assume that we use a dNL-DNF function with hidden layer of size 4 (i.e., applying a disjunction neuron to the output of 4 conjunction neurons). Fig. 3.1 shows the internal representation of the symbolic predicate function $\mathscr{F}_{lt}^1 = \text{inc}(A, B) \vee (\text{lessThan}(A, C) \wedge \text{inc}(C, B))$ in the corresponding dNL-DNF functions. Here, only the membership weights

corresponding to the atoms used in $\mathscr{F}_{lt}^1$ (i.e., conjunction membership weights $m^c_{1,2}$, $m^c_{4,3}$ and $m^c_{4,17}$ as well as two disjunction membership weights $m^d_{1,1}$ and $m^d_{1,4}$) are equal to 1 and the rest of the membership weights are zero. Fig. 3.2 shows one step forward chaining via (6.4) for the predicate lt. Steps 4 through 8 in the Algorithm 1 show how the values in X_{lt} corresponding to each grounding e are updated at each time stamp t. For example, consider the case of $e = lt(1, 2)$ at the first time stamp $t = 1$. Initially, according to (3.11), all entries in X_{lt} will be zero. To find all the consequences, we need to apply the rules (i.e., predicate functions) for each possible substitutions (all possible groundings) and aggregate all the resulting consequences. Steps 4 to 7 in Algorithm 1 simply aggregate (via disjunction) those consequences that have the same grounding. For instance, when $e = lt(1, 2)$, this includes all the substitutions $\theta = \{A \mapsto 1,\ B \mapsto 2,\ C \mapsto 0\}$ through $\theta = \{A \mapsto 1,\ B \mapsto 2,\ C \mapsto 4\}$ (see Fig. 3.2). It is easy to verify that $\mathscr{F}_{lt}^1|_\theta$ returns 1 (True) for each of these substitution. As such, the new calculated consequence containing the grounding $e = lt(1, 2)$ will be evaluated as $1 \vee 1 \vee 1 \vee 1 \vee 1 = 1$. After obtaining the new value, we have to aggregate it with previous consequence (similar to the role of $\bigcup$ in (3.6) and (3.7)) according to step 8 in the algorithm. By following the same process for all the groundings, we can obtain all consequences similar to (3.7).

Algorithm 1: Outline of the forward chaining in the proposed dNL-ILP solver

Data: N_p : number of rules used for defining predicate p
Data: X_p^0 for $p \in \mathbb{P}$
Result: $X_p^{t_{max}}$ for $\in \mathbb{P}$

1 **for** $t \in \{1, \ldots, t_{max}\}$ **do**
2   **for** $p \in \mathbb{P}$ **do**
3   **for** $e \in G_p$ **do**
4             $new_value = 0$
5   **for** $i \in \{1, 2, \ldots, N_p\}$ **do**
6   **for** $\theta \in \Theta_p^i$ **do**
7                     $new_value = new_value \bigvee \mathscr{F}_p^i|_\theta$

            `/* Updating the groundings */`
8             $X_p^t[e] = X_p^{t-1}[e] \bigvee new_value$

3.3.2 Training

We interpret the final values of $X_p^{(t_{max})}[e]$ (after t_{max} steps of forward chaining) as the conditional probability of the value of ground atoms given the model's parameters. We define the loss as the average cross-entropy between the ground truth (provided by the positive and negative examples for the corresponding predicate p) and $X_p^{(t_{max})}$:

$$\mathcal{L} = - \underset{p \in \mathbb{P}}{\mathbb{E}} \underset{(e,\lambda_p) \in \Lambda_p}{\mathbb{E}} \left\{ \lambda \log X_p^{(t_{max})}[e] + (1-\lambda) \log(1 - X_p^{(t_{max})}[e]) \right\}, \tag{3.13}$$

where, $\Lambda_p = \{(\gamma, 1) | \gamma \in \mathcal{P} \bigcap G_p\} \bigcup \{(\gamma, 0) | \gamma \in \mathcal{N} \bigcap G_p\}$. We train the model using ADAM [59] optimizer to minimize the loss function with the learning rate of 0.001 (in some cases we may increase the rate for faster convergence). After the training is completed, a zero cross-entropy loss indicates that the model has been able to satisfy all the examples in the positive and negative sets.

3.3.3 Pruning

There might exist a few atoms with membership weights of '1' in the corresponding dNL network for a predicate which are not necessary for the satisfiability of the solution. For example, consider the predicate function with two rules in (3.5). It is easy to see that if we replace the first rule with (lessThan$(A, B) \leftarrow$ inc(A, B), inc(A, C)), the resulting rules still satisfy the examples. This is because in this example we can always find a substitution that generates the same consequences (e.g., substitutions with $B = C$) for the first rule. Since such a solution also leads to a zero loss function, we cannot remove the redundant atom (inc(A, C)) via the training objective function (3.13). We consider two approaches to deal with these situations. In the first approach, we consider adding a penalty term corresponding to the number of terms in each clause. For each predicate function $\mathscr{F}_p^i$ we add a corresponding penalty term as:

$$
\begin{aligned}
\mathcal{L}_{agg} &= \mathcal{L} + \lambda_{prn}\mathcal{L}_{prn} \\
\mathcal{L}_{prn} &= \sum_{p \in \mathbb{P}} relu(\sum_{m_i \in \mathscr{F}_p^i} m_i - N_{max}),
\end{aligned} \tag{3.14}
$$

where subscript prn is short for pruning and N_{max} is the number of allowed terms in the $\mathscr{F}_p^i$. We usually use a very small λ_{prn} to make sure it only affects the learning at the latter stages of conversion where $\mathcal{P}$ is very small.

Alternatively, we can use a simpler approach to remove those necessary atoms from solutions. In the final stage of algorithm, we remove each remaining atom from a rule if by switching its membership variable from 1 to 0, the loss function does not change.

3.3.4 Lack of Uniqueness

In general, the solutions that the dNL-ILP finds are not unique. In addition to the cases that we discussed in pruning section, there can be various correct answer for a problem. For example, by replacing the body of second rule in (3.5) with inc(C, B), lessThan(A, C), we can obtain another valid solution to the ILP problem. Since the membership weights are randomly initialized, in different runs of the algorithm we may obtain different, yet still valid solutions. Hence, in the rest of this chapter, the provided solutions are only one of the possible obtained solutions during a run of the algorithm.

3.3.5 Predicate Rules ($\mathscr{F}_p^i$)

In the majority of the ILP systems, the rules are restricted to Horn clause, i.e., the body of each rule is made of the conjunction of some atoms. Even though we can still use the same convention, in our model it is sometimes easier and more efficient to combine all the rules involving a predicate via learning the body of rule as a dNL-DNF function. Alternatively, we can also restrict the program to Horn clauses by using a set of dNL-CONJ functions as

the body of predicate rules instead.

Moreover, compared to the most ILP systems, our approach allows for more flexible way of learning predicates. Let $L = |\mathbb{I}_p^i|$ be the number of all distinct atoms that can be used in the body of a rule. The neural ILP solver, dILP [27], limits the search space for the choices in the body of each rule to the $\binom{L}{2}$ combinations of two atoms. This is indeed a very small fraction of 2^L total possibilities that our model considers. While the ability to restrict the space of possible clauses is sometimes desirable and may speed up the learning, in most systems this approach significantly reduces the possible formulas. In our proposed approach, the space of possible clauses can still be restricted to speed up the learning. For example, we can exclude certain predicates form a clause. However, we do not need to explicitly specify the form of possible formula similar to Metagol [12] and dILP. Finally, we can easily allow for including the negation of each atom in the formula by concatenating the vector $\mathbb{I}_p|_\theta$ and its fuzzy negation, i.e., $(1.0 - \mathbb{I}_p|_\theta)$ as the input to the $\mathscr{F}_p^i$ function. This would only double the number of parameters of the model assuming Horn clause rules. In contrast, in most other implementations of ILP, this would increase the number of parameters and the problem complexity at much higher rates.

3.4 ILP as a Satisfiability Problem

In our model, we associate a dNL function $\mathscr{F}_p^i$ to the ith rule of every intensional predicate p. We can view the membership weights in the dNL function corresponding to each $\mathscr{F}_p^i$ as Boolean flags that indicates whether each atom in a rule is `off` or `on`. In this view, the problem of ILP can be seen as finding an assignment to these membership Boolean flags such that the resulting rules applied to the background facts, entail all positive examples and reject all negative examples. Moreover, by allowing these membership weights to be learnable weights, we are formulating a continuous relaxation of the satisfiability problem. This approach is in some ways similar to the approach in dILP [27], but differs in how we define Boolean flags. In dILP, a Boolean flag is assigned to each of the possible combinations

of two atoms from the set $\mathbb{I}_p^i$. In contrast, in our approach the membership weights of the conjunction (or any other logical function from dNL) can be directly interpreted as the flags in the satisfiability interpretation. As such, we can learn any arbitrary Boolean function of the atoms in vector $\mathbb{I}_p^i$ instead of a limited set of possibilities defined by restrictive templates.

3.5 Interpretability

Since our formulation is based on the fuzzy notion of Boolean logic, in general, the membership weights of the dNL networks do not necessarily converge to 0 or 1 and hence the learned formula may be difficult interpret. Fortunately, in problems that a definite solution can be found, as we will show via experiments (e.g. experiments in sections 3.7.1 and 3.7.3), the membership weights converge to 0 and 1 and as such, the solution is a valid Boolean formula. On the other hand, in cases where there is ambiguity in data and no exact Boolean function can describe all the background facts (e.g., the task of classification of relational datasets such as Mutagenesis (section 4.2)), there is a trade-off between the interpretability and the performance of the learning task. To be able to control this trade-off we have added an optional penalty term to the loss function, i.e.

$$\mathcal{L}_{agg} = \mathcal{L} + \lambda_{prn}\mathcal{L}_{prn} + \lambda_{int}\mathcal{L}_{int}, \quad \text{where,}$$
$$\mathcal{L}_{int} = \mathop{\mathbb{E}}_{p\in\mathbb{P}} \mathop{\mathbb{E}}_{m\in\mathscr{F}_p} m(1-m) \tag{3.15}$$

where subscript int stands for interpretability and m represent each of the membership weights used in predicate functions. It is easy to see that $m(1-m)$ is at minimum (0) when membership weights are at 0 or 1 and the constant λ_{int} controls the trade-off.

3.6 Implementation Details

For the dNL-DNF functions associated to each learnable predicate we need to initialize the membership weights in the start of the training. During the experiments, we realized that

in general, the models are not sensitive to the initial weights of membership weights w_i. While the speed of convergence somewhat depends on the initial values, in the presented experiments the network is able to find the optimal settings and converges to the desired output when weights are initialized using random Gaussian functions. In cases where the dimension of the input vector is very large, extra care may be needed. Due to the multiplicative design of the neurons in dNL functions, when many of the membership variables have values between zero and one, the gradient can becomes very small. To avoid this situation, we must ensure that most of the membership variables are almost zero in the beginning of training. As such, we usually initialize weights by a normal distribution with negative mean, (e.g., mean of -1 or -2 depending on the size of input vector).

We have implemented[1] the dNL-ILP solver model using python and Tensorflow [64]. In the previous sections, we have outlined the process in a sequential manner. However, in the actual implementation we first create index matrices using all the background facts before starting the optimization task (Gather operation in Fig. 3.2). Further, at each time-stamp and for each intensional predicate, all instances of applying (executing) the neural function $\mathscr{F}_p^i$ are carried in a batch operation and in parallel. In our approach, usually there is no need for any tuning and parameter specification other than the signature of each predicate rule (arity and the number of additional variables). On the other hand, since we use a propositionalization step (typical to almost all neural ILP solvers), special care may be required when the number of constants in the program is very large.

In general, we have found that the program is not sensitive to the learning rate and we have set the default learning rate to 0.001. In some of the problems we may increase the learning rate to speed up the convergence. For example, for the case of the classification of IMDB dataset, we needed more than 200 epochs for convergence when learning rate was set to 0.001, compared to only 5 epochs that was needed for the learning rate of 0.1. Further, in all the experiments presented in this chapter, we assume the interpretability penalty term

[1] Available at `https://github.com/DemoRep79/dNLILP`

(i.e., λ_{int}) is zero unless stated otherwise. For all the intensional predicates we need to specify the number of variables as well as the type of dNL function that is used to represent $\mathscr{F}_p^1$. Since in most cases we use only one rule with body of the rule defined as a dNL-DNF function, the size of the hidden layer for the DNF must be specified This corresponds to the maximum number of Horn clauses that are allowed for a learnable predicate. Since we may not know this parameter in advance, we usually start by choosing a number based on the complexity of the problem. For example, in case of the `lessThan` example in Fig. 3.1, we started with hidden layer of size 4 but the program needed only two conjunction neurons and the membership weights in the disjunction function corresponding to the two other rules was found to be zero. In the provided experiments we have used t_{max} from 1 to 8 depending on the task. Increasing the maximum number of forward chaining linearly increases the runtime of the program. This is due to the fact that the forward chaining steps cannot be parallelized and should be calculated sequentially. As such we do not increase this number beyond necessity to speed up the programs.

3.7 Experiments

Different ILP systems can be compared in various ways. ILP systems such as Metagol and dILP are able to learn the logical relationships in an explicit manner and as such are good candidates for learning recursive and algorithmic tasks. Many other ILP systems are mainly focused on larger scale tasks of learning from relational datasets and are usually evaluated by their performance in relational classification tasks. It suffices to say that no single ILP system can outperform all others in every single task. The proposed dNL-ILP system offers explicit representation of the learned logic and is capable of learning recursive predicates of arbitrary complex form without the need to specify the meta-rules similar to Metagol [12] and dILP [27]. Further, unlike the Metagol, because of the fuzzy nature of membership weights, it can also used for the classification tasks similar to approaches based on probabilistic ILP frameworks. Additionally, the differentiable aspect of this solver makes

it possible to combine with deep learning models such as convolutional network to directly learn relational information from images. In the Chapter 5 we exploit this characteristic of the dNL-ILP to formulate a policy learning relational reinforcement learning framework which can directly learn from images. On the other hand, since our proposed model works by propositionalization, it cannot naturally handle partially defined clauses and is not suitable for such applications. In the following, we first examine the performance of dNL-ILP in learning algorithmic tasks and specially tasks involving recursion and compare its performance to the Metagol and dILP. Next, we will consider the classification tasks involving real-life and large scale datasets and compare its performance to the other ILP systems that are mainly based on the probabilistic approach to ILP [19].

3.7.1 Benchmark ILP Tasks

We tested the proposed algorithm on the 20 symbolic tasks described in [27] which are taken from 4 domains: arithmetic, lists, graphs and family tree. The details of these experiments can be found in Appendices $G.1$ to $G.20$ of [27]. Even though some of these tasks require predicate invention and recursion (e.g., the task of learning the `circular` predicate in graph domain), all of these tasks are still considered simple algorithmic tasks as they would only involve predicates with arity of at most 2 and small number of constants. However, as reported in [27], dILP cannot always find the right solutions (e.g. only 49% of time it can solve even/odd problem). Further, Metagol, if not provided with the exact form of meta rules, is not able to solve some of the tasks involving recursion (e.g. Learning the `connectedness` and also `circular` predicates in graph domain). In contrast, as shown in Table 3.2, our proposed algorithm can always find a solution to these problems. The implementation of many of these tasks as well as some additional experiments are provided in the accompanying source code. Here we briefly describe a few tasks and the results of our algorithm.

- **Circular predicate**. In this experiment, the goal is to learn the circular property of a node

in a directed graph. A node is called circular if there exist a connected path from that node to itself. In this and other graph related tasks, the background knowledge consists of the groundings of predicate $edge(A, B)$ which defines a directed graph between nodes. The constants in this example is the set of nodes labeled as $C = \{a, b, c, d, e, f, g, h, i, u, v\}$. In addition to the predicate `edge`, we define two intensional predicates; a learnable auxiliary predicate of arity 2 (`i.e., aux/2`) and the target predicate `circular/1`. We allow for one additional variable in definition of each of these predicates. The set of positive and negative example corresponding to the graph depicted in Fig. 3.3, i.e. $\mathcal{P} = \{b, c, d, e, u, v\}$, $\mathcal{N} = \{a, f, g, h, i\}$. For $\mathscr{F}^1_{aux}$ we allow 3 variables and we use dNL-DNF function with hidden layer of size 4. For $\mathscr{F}^1_{circular}$ we allow for 3 variables and we use dNL-DNF function with hidden layer of size 2. One of the obtained solutions is:

$$\begin{aligned} aux(A, B) &\leftarrow edge(A, B) \\ aux(A, B) &\leftarrow edge(A, C), aux(C, B) \\ circular(A) &\leftarrow aux(A, A) \end{aligned}$$

This simple example uses predicate invention and involves recursive predicates. Even though we did not specify any example for the predicate `aux`, the learned formula for this auxiliary predicate is meaningful and represents the connectedness relation. It is worth mentioning that these solutions are not unique. In a different run, we obtained the below solution for this problem:

$$\begin{aligned} aux(A, B) &\leftarrow edge(A, B) \\ aux(A, B) &\leftarrow edge(A, C), aux(C, B) \\ circular(A) &\leftarrow edge(A, B), aux(B, A) \end{aligned}$$

Even though these two sets of rules appear to be different, It is easy to verify that the two

solutions are logically equivalent.

- **Family Tree: Grandparent** In this subset of English royals family tree, consist of the background relations such as 'son','father',... and including 25 constants, the task is to learn the grandparent relation. One of the obtained solutions is:

$$grandparent(A, B) \leftarrow son(C, A), uncle(C, B)$$
$$grandparent(A, B) \leftarrow son(C, A), father(C, B)$$

  Obviously, different background knowledge would have resulted a different definition for the target predicate. For instance, in another problem formulation, we add an auxiliary predicate of arity 2 with no additional variables. One of the solutions found by dNL-ILP is:

$$aux(A, B) \leftarrow mother(A, B)$$
$$aux(A, B) \leftarrow father(A, B)$$
$$grandparent(A, B) \leftarrow aux(A, C), aux(C, B)$$

  Even though no definition was provided for the auxiliary predicate, it is noteworthy to observe that just by introducing the auxiliary predicates into learning, the model learns higher level concepts and use that to define the target predicate. In the above example, clearly the auxiliary predicate resembles the concept of parenthood and was used as the building block define the target predicate `grandparent`.

3.7.2 Learning Decimal Multiplication

As was discussed in Chapter 2, neural based approaches based on sequence to sequence recurrent networks and GRU networks such as [7] are able to learn arithmetic to some degrees and for example neural GPU can learn binary multiplications of two 20 bits number

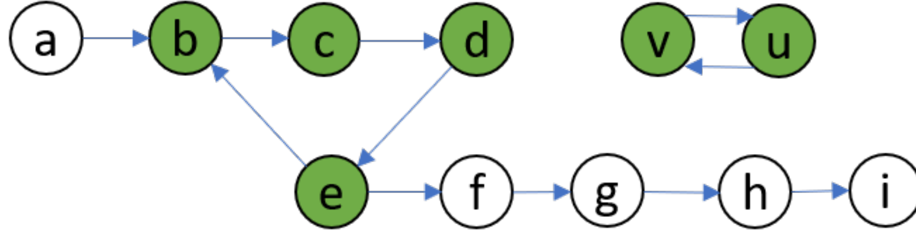


Figure 3.3: Background graph used in learning circular predicate .

after training over millions of samples and in general those methods are not able to generalize well above the length of training samples. In contrast, here we demonstrate how the proposed dNL-ILP can efficiently learn difficult algorithmic tasks such as multiplications of decimal numbers by using only a few examples. Moreover, the learned recursive formula can be applied to any number and can perfectly generalize to all natural numbers since it captures the logical operations required to perform decimal multiplication. We use dNL-ILP solver for learning the predicates $mul/3$ for decimal multiplication using only the positive and negative examples. We use $\mathcal{C} = \{0, 1, 2, 3, 4, 5, 6\}$ as constants and our background knowledge consists of the extensional predicates $\{zero/1, inc/2, add/3\}$, where `zero/1` define the decimal number 0, `inc/2` defines increment of one and `add/3` defines the addition.

It is worth noting that the dNL-ILP solver does not consider constants in learning predicates and the predicate formula can only contain other predicates and variables. Thus, when there is a need for referring to any specific constant in the formula, we should define a corresponding predicate as part of background knowledge. Here for example, since we need to distinguish the special number, zero, we need to define an extensional predicate `zero/1`. In this case, the background fact corresponding to this predicate is the ground atom `zero(0)` which states that the predicate `zero/1` is evaluated as True when its argument is the decimal number, 0.

The target predicate is `mul/3` and we allow for using 5 variables (i.e., $num_var^i(mul) = 5$) in each rule. We use a dNL-DNF network with 4 disjunction terms (4 conjunctive rules)

Table 3.2: dNL-ILP vs dILP and Metagol in benchmark tasks

Domain	Task	dILP	Metagol	dNL-ILP
Arithmetic	Predecessor	**100**	**100**	**100**
Arithmetic	Even	**100**	**100**	**100**
Arithmetic	Even-Odd	49	**100**	**100**
Arithmetic	Less than	**100**	**100**	**100**
Arithmetic	Fizz	10	**100**	**100**
Arithmetic	Buzz	35	**100**	**100**
List	Member	**100**	**100**	**100**
List	Length	93	**100**	**100**
Family Tree	Son	**100**	**100**	**100**
Family Tree	GrandParent	97	**100**	**100**
Family Tree	Husband	**100**	**100**	**100**
Family Tree	Uncle	70	**100**	**100**
Family Tree	Relatedness	**100**	0	**100**
Family Tree	Father	**100**	**100**	**100**
Graph	Undirected Edge	**100**	**100**	**100**
Graph	Adjacent to Red	51	**100**	**100**
Graph	Two Children	95	**100**	**100**
Graph	Graph Colouring	95	**100**	**100**
Graph	Connectedness	**100**	0	**100**
Graph	Cyclic	**100**	0	**100**

for learning $\mathscr{F}_{mul}$. It is worth noting that since we do not know in advance how many rules would be needed, we should pick an arbitrary number and increase in case the ILP program cannot explain all the examples. Further, we set the $t_{max} = 8$. One of the solutions that our model finds is:

$$mul(A, B, C) \leftarrow zero(B),\ zero(C)$$

$$mul(A, B, C) \leftarrow mul(B, A, C)$$

$$mul(A, B, C) \leftarrow mul(A, D, E),\ inc(D, B),\ add(E, A, C)$$

Please note that the two last rules are learned as recursive rules. Furthermore, the first rule has learned that if one of the operands is zero the answer would be zero.

3.7.3 Sorting

In this experiment we consider the task of sorting an ordered list via ILP. Our goal is to find a recursive formula for sort predicate such that `sort(A,B)` is true only if list B is the sorted version of list `A`. The sorting task is more complex than the previous tasks since it requires not only the list semantics, but also many more constants compared to the arithmetic problem. We implement the list semantic by allowing the use of functions in defining predicates. For a data of type `list`, we define two functions H and t which allow for decomposing a list into head and tail elements, i.e $A = [A_H|A_t]$. We use elements of $\{a, b, c, d\}$ and all the ordered lists made from permutations of up to three elements as constants in the program (i.e., $|\mathcal{C}| = 40$). We use extensional predicates such as gt (greater than), eq (equals) and lte (less than or equal) to define ordering between the elements of lists as part of the background knowledge. We use a dNL-DNF with hidden layer of size 4 for $\mathscr{F}^1_{sort}$ and we allow for using 4 variables (and their head and tail functions). One of the solution that our model finds is:

$$sort(A,B) \leftarrow sort(A_H,C),\ lte(C_t,A_t),\ eq(B_H,C),\ eq(A_t,B_t)$$

$$sort(A,B) \leftarrow sort(A_H,C),\ gt(C_t,A_t),\ sort(D,B_H),\ eq(B_t,C_t),\ eq(D_H,C_H),\ eq(A_t,D_t)$$

We may observe that the first two atoms in the body of each learned rules, act like an if/then instruction. If the tail element of sorted part of A_H is less than the tail of A the first recursive rule applies and otherwise the second rule applies. Even though the above examples involve learning tasks that may not seem very difficult on the surface, and deal with relatively small number of constants, they are far from trivial. To the best of our knowledge, learning a recursive predicate for a complex algorithmic task such as `sort` which involves multiple recursive rules with 6 atoms and includes 12 variables (by counting two functions head and tail per variables) is beyond the power of any existing ILP solver. Here for example, the total number of possible atoms to choose from is $|\mathbb{I}^2_{sort}| = 176$. For comparison, consider

template based approaches such as dILP. In such an approach, all the possible combinations with 6 atoms in the body, i.e., $\binom{176}{6} > 3 \times 10^{10}$ should be considered. This shows that the algorithms that require enumerating the possible candidates via templates (such as dILP and Metagol) are not able to directly tackle problems with this level of complexity.

CHAPTER 4

LEARNING FROM UNCERTAIN DATA VIA DNL-ILP

4.1 Introduction

The experiments in previous chapter showcased the performance of the proposed ILP solver model in learning algorithmic tasks such as learning arithmetic and sorting. Since the dNL-ILP is a differentiable model, implemented via neural networks and uses fuzzy membership weights to form Boolean functions, it can also be used in ambiguous problems where data is either noisy or in cases where no single hypothesis may satisfy all the examples. Classification of relational datasets is one of these tasks which has many applications in a wide variety of fields. Standard ILP systems, e.g. Metagol, are not usually suited for handling these kind of tasks. On the other hand, most neural ILP solvers are either incapable of learning complex and recursive predicates or in the case of dILP [27], cannot handle large scale problems. Alternatively, a probabilistic extension of ILP ([19]) and its variants are usually employed in these tasks. In this chapter we study the application of dNL-ILP in scenarios where we are dealing with uncertain data. First, in Section 4.2 we explore the application of dNL-ILP in the classification of benchmark relational databases and we compare its performance to the state of the art algorithms based on probabilistic notion of ILP. Next, we will consider scenarios where by combining differentiable dNL-ILP with standard neural networks we can learn from continuous data; either by learning decision boundaries (Section 4.3) or by learning the distribution of data via fitting probabilistic models (Section 4.4).

4.2 Classification for Relational Data

Standard classification algorithms such as regression techniques as well as methods based on decision trees usually require one feature vector for each data-point and are not able to

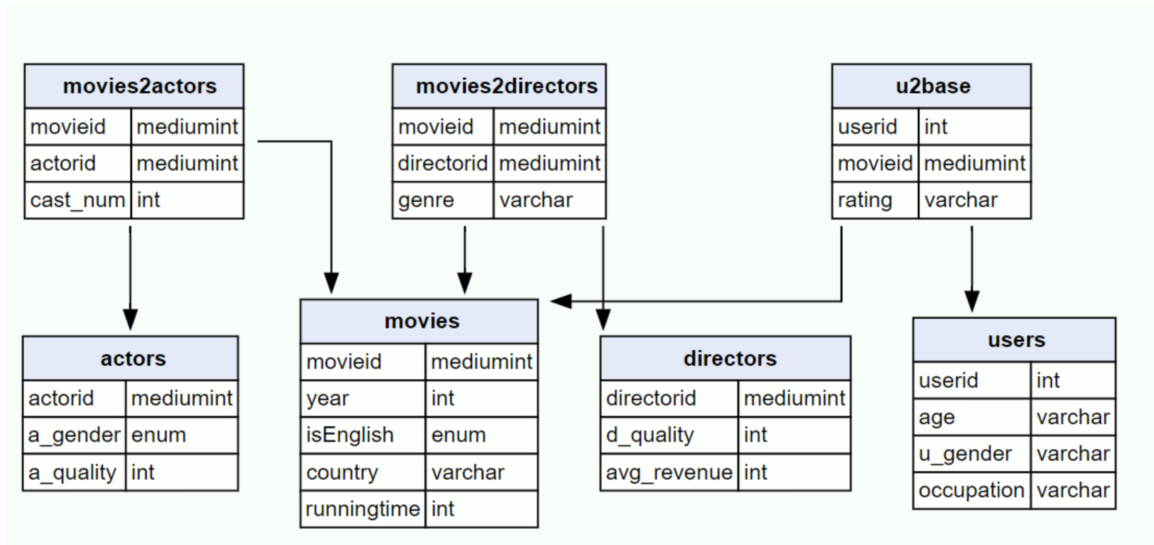


Figure 4.1: Internal structure of MovieLens dataset

exploit the intricate relations within various tables in a relational dataset. Fig. 4.1 depicts the internal table structure for one of the datasets used in our experiments. Here for example, one of the benchmark classification tasks is defined as predicating the sex of a movie reviewer (male or female) based on the reviewer's data. This includes all the information available for each movie the reviewer has reviewed, such as director's and actor's information as shown in Fig. 4.1. Classical approaches usually require joining all the tables to create a feature vector which may lead to the explosion of the records and the need for handling missing values. In practice, techniques based on ILP are usually employed for extracting knowledge and reasoning tasks involving relational datasets.

In this experiment, we evaluate the performance of our proposed ILP solver in some benchmark ILP classification tasks. We use real-world relational datasets Mutagenesis [33], UW-CSE [32] as well as IMDB, MovieLens (1M) and Cora datasets[1]. Table 4.1 summarizes the features of these datasets. In all the cases we used the original full-size dataset instead of subsets that are sometime used in various reports (e.g. Movielens 60K, Mutagenesis1, Mutagenesis2, . . .). For the cases involving unary target predicates (e.g Mutagenesis and Cora) we create a separate background knowledge for each possible groundings of the

[1]Publicly-available at `https://relational.fit.cvut.cz/`

target predicate. For other cases we create only one background knowledge for each fold in cross validation scheme that consists of all the groundings in that portion of training data. As baseline, we are comparing dNL-ILP with the state of the art algorithm based on

Table 4.1: Dataset Features

Dataset	Constants	Target Predicate
Mutagenesis	7045	$active(A)$
UW-CSE	1158	$advisedBy(A,B)$
Cora	3079	$sameBib(A,B)$
IMDB	316	$workingUnder(A,B)$
MovieLens	2627	$female(A)$

Table 4.2: Some of the learned rules for classification tasks

IMDB Task
$workedUnder(A,B) \leftarrow director(B), actor(A), movie(C,A), movie(C,B), \neg actor(B)$
$workedUnder(A,B) \leftarrow isFemale(B), movie(C,A), \neg actor(B), \neg movie(C,A)$

UW-CSE task
$advisedBy(A,B) \leftarrow publication(A,C), publication(B,C),$ $hasanyPosition(B), inanyPhase(A)$
$inanyPhase(A) \leftarrow inPhase(A,B)$
$hasanyPosition(A) \leftarrow hasPosition(A,B)$

Markov Logic Networks such as GSLP [65], Problog [66], MLN-B (Boosted MLN)[67]. Ideally, we would liked to include all the recent published works, however for many cases the source codes are not publicly available. Further, since in most of these datasets, the number of negative examples are significantly greater than the positive examples, we report the Area Under Precision Recall (AUPR) curve as a more reliable measure of the classification performance. We use 5-fold cross validations except for the Mutagenesis dataset which we have used 10-fold and we report the average AUPR over all the folds. Table 4.3 summarizes the classification performance for the 4 relational datasets. As the results show, dNL-ILP outperforms the previous algorithms in most cases. The end-to-end design of our differentiable ILP solver makes it possible to combine some other forms of

Table 4.3: AUPR score for the 5 relational classification tasks

Dataset	GSLP	MLN-B	problog	dNL-ILP
Mutagenesis	0.77	NA	**0.99**	**0.99**
UW-CSE	0.46	0.22	0.28	**0.51**
Cora	0.89	**0.96**	0.90	**0.96**
IMDB	0.79	0.90	0.92	**1.00**
MovieLens	0.70	0.80	**0.82**	0. 78

learnable functions with the dNL networks. For example, while handling continuous data is usually difficult in most ILP solvers, we can directly learn some threshold values to create binary predicates from the continuous data (see Section 4.3). We have used this method in the Mutagenesis task to handle the continuous data in this dataset.

In previous experiments in Sections 3.7.1 to 3.7.3, when learning is complete (i.e., zero cross entropy loss), all the membership weights are either zero and one and as such we did not incorporate any interpretability loss (i.e. $\lambda_{int} = 0$). However, in complex and ambiguous datasets, this is not the case and we usually never obtain zero cross entropy loss during training. Hence, we need to incorporate the λ_{int} if we want to interpret the learned hypothesis as a valid Boolean function. The result presented in Table 4.3 are obtained $\lambda_{int} = 0$. We can trade some performance and achieve better interpretability by adjusting the λ_{int} parameter. In IMDB task, adding penalty term did not decrease the classification accuracy, while in case of the UW-CSE task, setting the $\lambda_{int} = 0.1$ decreased the AUPR from 0.51 to 0.49 but resulted in a valid Boolean function (i.e., all the membership weights converged to 0 and 1). Table 4.2 displays the learned rules for these two tasks after setting $\lambda_{int} = 0.1$.

Because of the difference in hardware, it is difficult to directly compare the speed of algorithms. In our case, we have evaluated the models using a 3.70GHz CPU, 16GB RAM and GeForce GTX 1080TI graphic card. Using this setup, the problems such as IMDB, Mutagenesis are learned in just a few seconds. For Cora, the model creation takes about one minute and the whole simulation for any fold takes less than 3 minutes.

4.2.1 Implementation Details:

In our implementation, an ILP program consists of these elements:

1. the list of all the variables from each data type
2. the definition of the auxiliary predicates
3. the background facts
4. the signature for learning the target predicate
5. the positive and negative examples

Fig. 4.2 shows a screen shot of a python definition of the graph/connected problem using dNL-ILP.

In the remaining of this section we summarize the details of each experiments and we explain the choice of parameters used in each dataset:

- **Cora**: In this task our background knowledge contains facts involving predicates: HasWordAuthor/2, HasWordTitle/2, HasWordVenue/2, SameAuthor/2, SameTitle/2, Author/2 , Title/2. We also define an auxiliary predicate sameWA as:

  $$\text{sameWA(A,B)} \leftarrow \text{HasWordAuthor(A,C)}, \text{HasWordAuthor(B,C)}$$

  For the target predicate SameBib/2 we allow for two rules, each consists of a DNF with hidden layer of size 1 (i.e., just one conjunction function). For each of these two learnable rules we allow for tree variables. For one of them, the extra variable is of type 'Author' and for the other one it is of data type 'Title'. In learning, we use 5 fold cross validations. Given 5 different sets of background facts, at each training step, we use data from one of the background fact sets randomly and we test the performance on the held-out portion of data.

- **MovieLens**: In this task our background contains facts involving predicates: age/1, occupation/1, rate/1, genre/2, d_quality/2, avg_revenue/2, isEnglish/2 and movie_year/2. For efficiency reasons, we define auxiliary predicates corresponding to each of the possibilities for some of the multi valued data types. For example, if there are three ratings of 1, 2 and 3, we can define three auxiliary predicates such as:

  $$\text{rate1(A)} \leftarrow \text{rate(A,B), is_1(B)}$$
  $$\text{rate2(A)} \leftarrow \text{rate(A,B), is_2(B)}$$
  $$\text{rate3(A)} \leftarrow \text{rate(A,B), is_3(B)}$$

  For the target predicate isFemale/0, we define 10 DNF rules (each with hidden layer of 4), and we allow for one extra variable in defining each of those rules. Please note that if we had not defined auxiliary predicates such as rate1/1, rate2/1, rate3/1, we would needed many more variables in the body of the target predicates which would be more difficult for our models to handle. Further, unlike the previous dataset, we create separate background facts corresponding to each movie rater. In our training, for every training step, each training batch is consisted of randomly selected background facts of 500 users from all the total user in the training set. Again, we are using 5-fold cross validation and report the result on the held-out group of users during each fold.

- **MovieLens**: Here, our background predicates are director/1, actor/1, genre/2, isFemale/1, movie/2 and our target predicate is workedUnder/2. We allow for 4 variables and we use one rule of type DNF with conjunction neurons.

- **UW-CSE**: Here, our background predicates are student/1, professor/1, publication/2, tought_by/2, ta/2, courseLevel/2, has_position/2, projectMember/2. Similar to the Cora dataset case, we create multiple predicates for each distinct value of multi-valued quantities such as position (i.e., position_faculty, position_professor, . . .). We also define these two

auxiliary predicates using transitive relation:

$$\text{sameProj(A,B)} \leftarrow \text{projectMember(A,C)},\ \text{projectMember(B,C)}$$

$$\text{samePublications(A,B)} \leftarrow \text{publication(A,C)},\ \text{publication(B,C)}$$

Please note that by introducing these two auxiliary predicates we may eliminate the need for adding a variable of type ProjectID for leaning the target predicate. This can reduces the complexity of the problem. We allow for 2 variables and we use one rule of type DNF with hidden layer of size 6 to learn the target predicate advisedBy/2.

4.3 Handling Continuous Data

Reasoning using continuous data has been an ongoing challenge for ILP. Most of the current approaches either model continuous data as random variables and use probabilistic ILP framework [19], or use some forms of discretization via iterative approaches. The former approach cannot be applied to the cases where we do not have reasonable assumptions for the probability distributions. Further, the latter approach is usually limited to small scale problems (e.g. [68]) since the search space grows exponentially as the number of continuous variables and the boundary decisions increases. Alternatively, the end-to-end design of dNL-ILP makes it rather easy to handle continuous data. Recall that even though we usually use dNL based functions, the predicate functions $\mathscr{F}'s$ can be defined as any arbitrary Boolean function in our model. Thus, for each continuous variable x we define k lower-boundary predicates $gt_{x_i}(x, l_{x_i})$ as well as k upper-boundary predicates $lt_{x_i}(x, u_{x_i})$ where $i \in \{1, \ldots, k\}$. We let the boundary values l_{x_i}'s and u_{x_i}'s be trainable weights and we define the upper-boundary and lower-boundary predicate functions as:

$$\mathscr{F}_{gt_{x_i}} = \sigma(c\,(x - u_{x_i}))\ ,\ \mathscr{F}_{lt_{x_i}} = \sigma(-c\,(x - l_{x_i})),$$

```python
from  Lib.ILPRLEngine import *
import argparse
from Lib.DNF import DNF

#define constants
Constants = dict({'C': [ 'a','b','c','d' ,'e'] })
predColl = PredCollection (Constants)

#define predicates
predColl.add_pred(dname='edge'  ,arguments=['C','C'])
predColl.add_pred(dname='connected',arguments=['C','C'] , variables=['C'  ] ,
    pFunc = DNF('connected',terms=4 ))
predColl.initialize_predicates()

#add background
bg = Background( predColl )
bg.add_backgroud ('edge', ('a','b') )
bg.add_backgroud ('edge', ('b','c') )
bg.add_backgroud ('edge', ('c','d') )
bg.add_backgroud ('edge', ('e','d') )

#add examples
bg.add_example('connected',('a','b'))
bg.add_example('connected',('a','c'))
bg.add_example('connected',('a','d'))
bg.add_example('connected',('b','c'))
bg.add_example('connected',('b','d'))
bg.add_example('connected',('c','d'))
bg.add_example('connected',('e','d'))
bg.add_all_neg_example('connected')

# callback provides set of background facts for each training step
def callback(it,is_training):
    return [bg,]

# set training parameters
parser = argparse.ArgumentParser()
parser.add_argument('--T',default=5 ,help='Number of forward chain',type=int)
parser.add_argument('--LR_SC',default={(-1e3,1e3):.001}, help='Learning rate')

# training the model
args = parser.parse_args()
model = ILPRLEngine( args=args ,predColl=predColl ,bgs=callback )
model.train_model()
```

Figure 4.2: An example of dNL-ILP program definition in python

where σ is the `sigmoid` function and $c \gg 1$ is a constant. To evaluate this approach we use it in a classification task for two datasets containing continuous data; Wine and Sonar from UCI Machine learning dataset [69] and compare its performance to the ALEPH [70], a state-of-the-art ILP system, as well as the recently proposed FOLD+LIME algorithm [71]. The wine classification task involves 13 continuous features and three classes and the Sonar task is a binary classification task involving 60 features. For each class we define a corresponding intensional predicate via dNL-DNF and learn that predicate from a set of lt_{x_i} and gt_{x_i} predicates corresponding to each continuous feature. We set $k = 6$. Table 4.4 summarizes the results.

Table 4.4: Classification accuracy

Task \ Algorithm	ALEPH+LIME	FOLD+LIME	dNL-ILP
Wine	0.92	0.93	**0.98**
Sonar	0.74	0.78	**0.85**

4.4 Dream4 Challenge Experiment (Handling Uncertain Data)

Inferring the causal relationship among different genes is one of the important problems in biology. In this experiment, we study the application of dNL-ILP for inferring the structure of gene regulatory networks using 10-genes time-series dataset from the DREAM4 challenge tasks [72]. In the 10-gene challenge, the data consists of 5 different biological systems, each composed of 10 genes. For each system a time-series containing 105 noisy readings of the genes expressions (in range $[0, 1]$) is provided. The time-series is obtained via 5 different experiments, each created using a simulated perturbation in a subset of genes and recording the gene expressions over time. To tackle this problem using dNL-ILP framework, we simply assume that each gene can be in one of the two states: `on` (excited or perturbed) or `off`. The key idea here is to model each gene's state (`off` or `On`) using two different approaches and then aim to get a consensus on the gene's state using these two different probes. To accomplish this, for each gene G_i we define the predicate off_i which evaluates

the state of G_i using the corresponding continuous values. We also use predicate inf_off_i which takes the state of all the predicates off_j's ($j \neq i$) to infer the state of G_i. To ensure that at each background data inf_off_i would be close to off_i, we define another auxiliary predicate aux_i with predicate function defined as $\mathscr{F}_{aux_i} = 1 - |\text{inf_off}_i - \text{off}_i|$.

Since the state of genes are uncertain, we use a probabilistic approach and assume that each gene state is conditionally distributed according to a Gaussian mixture (with 4 components), i.e., $x|\text{off} \sim \text{GMM}_{\text{off}}$ and $x|\text{on} \sim \text{GMM}_{\text{on}}$. As such, we design $\mathscr{F}_{\text{inf_off}_i}$ such that it returns the probability of the G_i gene being `off`, and let all the parameters of the mixture models be trainable weights. For the $\mathscr{F}_{\text{Inf_off}_i}$ we use a dNL-CNF network with only one term. Each data point in the time series corresponds to one background knowledge and consists of the continuous value of each gene expression. For each gene, we assign 5 percent of data points with the lowest and highest absolute distance from mean as positive and negative examples for predicate off_i, respectively. We interpret the values of the membership weights in the trained dNL-CNF networks which are used in $\mathscr{F}_{\text{Inf-off}_i}$ as the degree of connection between two genes. Table 4.5 compares the performance of dNL-ILP to the two state of the art algorithms NARROMI [73] and MICRAT [74] for 10-gene classification tasks of DREAM4 dataset.

Table 4.5: DREAM4 challenge scores

Metric \ Method	NARROMI	MICRAT	dNL-ILP
Accuracy	0.82	**0.87**	0.86
F-Score	0.35	0.32	**0.36**
MCC	0.24	0.33	**0.35**

4.5 Comparing dNL-ILP to the Past Works

Addressing all important past contributions in ILP is a tall order and given the limited space we will only focus on a few recent approaches that are in some ways relevant to our work. Among the ILP solvers that are capable of learning recursive predicates (in an explicit and

symbolic manner), the most notable examples are Metagol [12] and dILP [27]. Metagol is a powerful method that is capable of learning very complex tasks via user-provided meta-rules. The main issue with Metagol is that while it allows for some flexibility in terms of providing the meta-rules, it is not always clear how to define those meta formulas. In practice, unless the expert already has some knowledge regarding the form of the possible solution, it would be very difficult to use this method. dILP, on the other hand, is a neural ILP solvers that formulates a differentiable version of ILP. However, because of the way it define templates, dILP is limited to learning simple predicates with arity of at most two and with maximum two atoms in each rule and cannot handle large scale relational datasets [27]. CILP++ [22] is another noticeable neural ILP solver which also uses propositionalization. CLIP++ is a very efficient algorithm and is capable of learning large scale relational datasets. However, since this algorithm uses the bottom clause propositionalization, it is not able to learn recursive predicates. In dealing with uncertain data and specially in the tasks involving classification of the relational datasets, the most notable frameworks are those based on the probabilistic ILP (PILP) [19] and Markov Logic Networks (MLN) [32]. These types of algorithms extend the framework of ILP to handle uncertain data by introducing a probabilistic framework. Our proposed approach is related to PILP in that we also associate a real number to each atom and each rule in the formula. In practice, as we have shown in Section 4.2, our approach outperforms those ensemble techniques based on probabilistic ILP.

We have shown that for simple algorithmic tasks such as those in Section 3.7.1 dNL-ILP is able to learn the rules. However, compared to the efficient and fast implementations such as Metagol, our approach for these kind of problems is still relatively slow and dNL-ILP usually needs a few second to solve most problems. The real advantage of our proposed approach is apparent when dealing with ambiguous data as in the case of relational datasets or when we need to learn a difficult recursive algorithmic task as in Section 3.7.3. More notably, the differentiable implementation of forward chaining makes it possible to combine dNL-ILP with deep learning models such as Convolutional Neural Networks (CNNs) to

learn relational knowledge from complex visual scenes. In the next chapter we explore this important application of dNL-ILP by introducing a relational reinforcement learning framework which can directly learn the relational policy from the images.

CHAPTER 5
RELATIONAL REINFORCEMENT LEARNING VIA DNL-ILP

5.1 Introduction

Relational Reinforcement Learning (RRL) has been investigated in early 2000s by works such as [34, 35, 36] among others. The main idea behind RRL is to describe the environment in terms of objects and relations. One of the first practical implementation of this idea was proposed by [35] and later was improved in [36] based on a modification to Q-Learning algorithm [37] via the standard relational tree-learning algorithm TILDE [38]. As shown in [36], the RRL system allows for very natural and human readable decision making and policy evaluations. More importantly, the use of variables in ILP system, makes it possible to learn generally formed policies and strategies. Since these policies and actions are not directly associated with any particular instance and entity, this approach leads to a generalization capability beyond what is possible in most typical RL systems. Generally speaking RRL framework offers several benefits over the traditional RL:

- It allows for the incorporation of higher level concepts and prior background knowledge.
- The learned policy is usually human interpretable, and hence can be viewed, verified and even tweaked by an expert observer.
- The learned program can generalize better than the classical RL counterpart.
- Since the language for the state representation is chosen by the expert, it is possible to incorporate inductive biases into learning. This can be a significant improvement in complex problems as it might be used to manipulate the agent to choose certain actions without accessing the reward function. One of such areas is designing safer AIs where

certain actions and state can be prohibited by the choice of state representation and the allowed actions.

In recent years and with the advent of the new deep learning techniques, significant progress has been made to the classical Q-learning RL framework. By using algorithms such as deep Q-learning and its variants [39, 40], as well as the deep policy learning algorithms such A2C and A3C [75], more complex problems are now been tackled.

Recently, some new methods for extracting relational information from the output of Convolutional Neural Networks (CNNs) have been proposed [76, 77]. However, despite their attempts in learning relational information, technically, they do not belong to the RRL framework since they do not employ any explicit relational language. While these techniques offer new methods for learning non-local information from the local features obtained by the CNNs, their internal representation is purely implicit (similar to other typical deep neural networks) and they cannot incorporate explicit relational biases.

On the other hand, the classical RRL framework cannot be easily employed to handle complex visual scenes that exist in recent RL problems. Since standard RRL framework is in not usually able to learn from complex visual scenes and cannot be easily combined with differentiable deep neural, none of the inherent benefits of RRL have been materialized in the deep learning frameworks thus far.

In previous chapter, a novel differentiable ILP solver, dNL-ILP, was introduced. The key aspect of the dNL-ILP solver is a differentiable deduction engine that was designed based in a differentiable forward chaining algorithm. In this chapter, we employ this engine to design a new differentiable RRL framework. We show that this differentiable RRL framework can be used similar to the deep RL in an end-to-end learning paradigm, trainable via the typical gradient optimizers. Further, in contrast to the early RRL frameworks, this framework is flexible and can learn from ambiguous and fuzzy information. Finally, it can be combined with deep learning techniques such as CNNs to extract relational information from the visual scenes.

5.2 Relational Reinforcement Learning via dNL-ILP

Early works on RRL [36, 78] mainly relied on access to the explicit representation of states and actions in terms of relational predicate language. In the most successful instances of these approaches, a regression tree algorithm is usually used in combination with a modified version of Q-Learning algorithms. The fundamental limitation of the traditional RRL approaches is that the employed ILP solvers are not differentiable. Therefore, those approaches are typically only applicable the problems for which the explicit relational representation of states and actions is provided.

In this chapter, we establish that differentiable dNL-ILP provides a platform to combine RRL with deep learning methods, constructing a new RRL framework with the best of both worlds. Although the dNL-ILP can also be used to formulate RL algorithms such as deep Q-learning, we focus only on deep policy gradient learning algorithm. This formulation is very desirable because it makes the learned policy to be interpretable by human. One of the other advantages of using policy gradient in our RRL framework is that it enables us to restrict actions according to some rules obtained either from human preferences or from problem requirements. This in turn makes it possible to account for human preferences or to avoid certain pitfalls, e.g., as in safe AI.

In our RRL framework, although we use the generic formulation of the policy gradient with the ability to learn stochastic policy, certain key aspects are different from the traditional deep policy gradient methods, namely state representation, language bias and action representation. In the following, we will explain these concepts in the context of BoxWorld game. In this game, the agent's task is to learn how to stack the boxes on top of each other (in a certain order). For illustration, consider the simplified version of the game as in Fig. 5.1 where there are only three boxes labeled as `a,b`, and `c`. A box can be on top of another or on the `floor`. A box can be moved if it is not covered by another box and can be either placed on the floor or on top of another uncovered box. For this game, the environment state

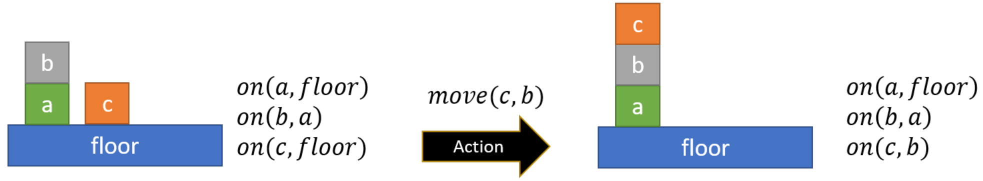


Figure 5.1: States representation in the form of predicates in BoxWorld game, before and after an action

can be fully explained via the predicate `on(X,Y)`. Fig. 5.1 shows the state representation of the scene before and after an action (indicated by the predicate `move(c,b)`). In the following we discuss each distinct elements of the proposed framework using the BoxWorld environment. Fig. 5.2 displays the overall design of our proposed RRL framework.

5.2.1 State Representation

In the previous approaches to the RRL [35, 36, 79], state of the environment is expressed in an explicit relational format in the form of predicate logic. This significantly limits the applicability of RRL in complex environments where such representations are not available. Our goal in this section is to develop a method in which the explicit representation of states can be learned via typical deep learning techniques in a form that will support the policy learning via our differentiable dNL-ILP. As a result, we can utilize the various benefits of the RRL discipline without being restricted only to the environments with explicitly represented states.

For example, consider the BoxWorld environment explained earlier where the predicate `on(X,Y)` is used to represent the state explicitly in the relational form (as shown in Fig. 5.1). Past works in RRL relied on access to explicit relational representation of states, i.e., all the groundings of the state representation predicates. Since this example has 4 constants, i.e. $\mathcal{C}$ ={`a,b,c,floor`}, these groundings would be the binary values ('true' or 'false') for the atoms `on(a,a)`, `on(a,b)`, `on(a,c)`, `on(a,floor)`, `...`, `on(floor,floor)`. In recent years, extracting relational information from visual scenes has been investigated. Fig. 5.4 shows two types of relational representation extracted from

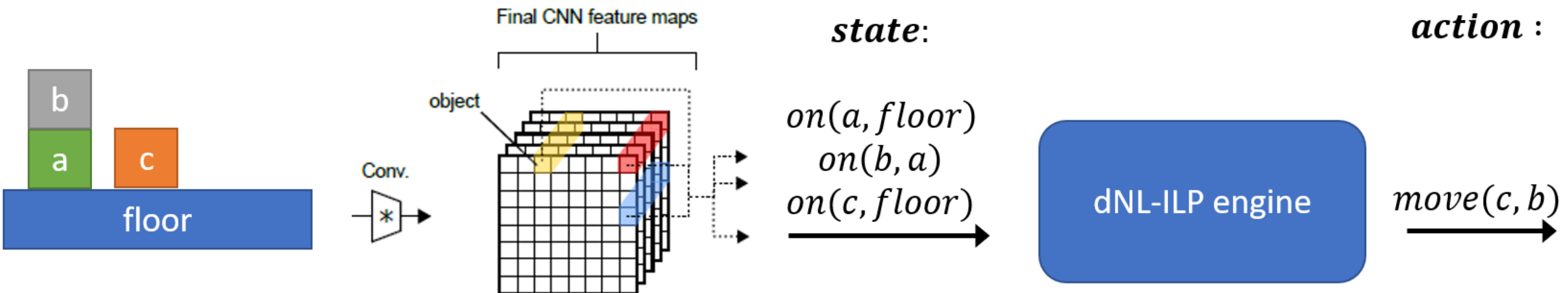


Figure 5.2: Learning explicit relational information from images in our proposed RRL; Images are processed to obtain explicit representation and dNL-ILP engine learns and expresses the desired policy (actions)

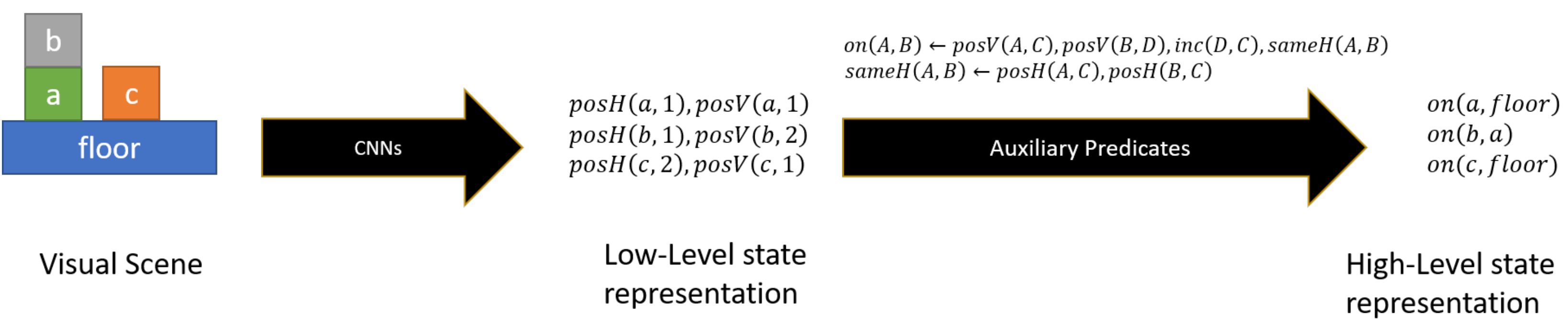


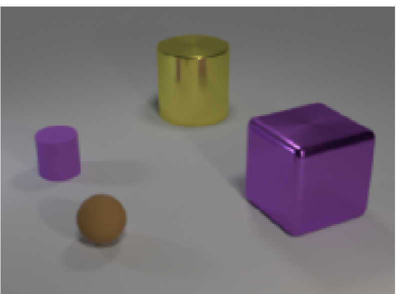
Figure 5.3: Transforming low-level state representation to high-level form via auxiliary predicates

images in [41]. The idea is to first process the images through multiple CNN networks. The last layer of the convolutional network chain is treated as the feature vector and is usually augmented with some non-local information such as the absolute position of each point in the final layer of the CNN network. This feature map is then fed into a relational learning unit which is tasked with extracting non-local features. Various techniques have been then introduced recently for learning these non-local information from the local feature maps, namely, self attention network models [76, 41] as well as graph networks [80, 81]. Unfortunately, none of the resulting presentations from past works is in the form of predicates needed in ILP.

In our approach, we use similar networks discussed earlier to extract non-local information. However given the relational nature of state representation in our RRL model, we consider three strategies in order to facilitate learning the desired relational state from images. Namely:

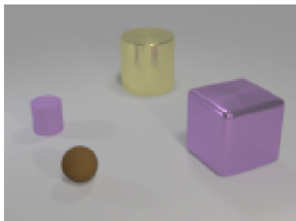
1. **Finding a suitable state representation:** In our BoxWorld example, we used the predicate `on(X,Y)` to represent the state of the environment. However, learning this predicate requires inferring relation among various objects in the scene. As shown by previous works (e.g., [41]), this is a difficult task even in the context of a fully supervised setting (i.e., all the labels are provided) which is not applicable here. Alternatively, we propose to use lower-level relations for state representation and build higher level representation via predicate language. In the game of BoxWorld as an example, we can describe states by the respective position of each box instead of the relative location of boxes via predicate `on`. In particular, we define two predicates `posH(X,Y)` and `posV(X,Y)` such that variable X is associated with the individual box, whereas Y indicate horizontal or vertical coordinates of the box, respectively. Fig. 5.3 shows how this new lower-level representations can be transformed into the

Original Image:



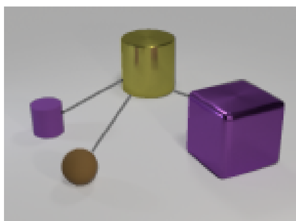
Non-relational question:

What is the size of the brown sphere?

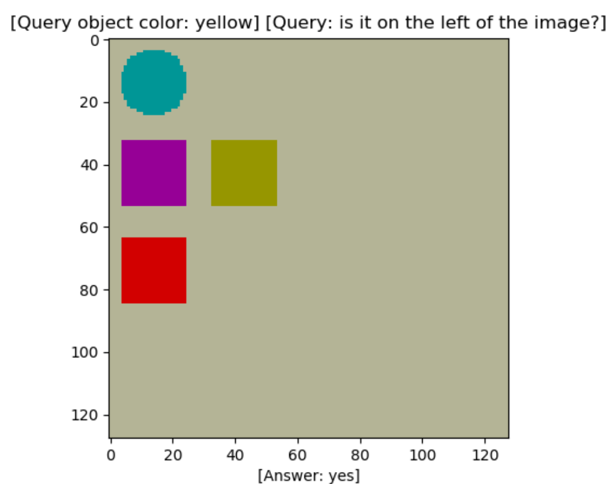


Relational question:

Are there any rubber things that have the same size as the yellow metallic cylinder?



(a) A sample from CLEVER datset



(b) A sample from sort-of-CLEVER datset

Figure 5.4: Extracting relational information from visual scene [41]

higher level description by the appropriate predicate language:

$$\begin{aligned} \text{on(X, Y)} &\leftarrow \text{posH(X, Z), posH(Y, T), inc(T, Z), sameH(X, Y)} \\ \text{sameH(X, Y)} &\leftarrow \text{posH(X, Z), posH(Y, Z)} \end{aligned} \tag{5.1}$$

2. **State constraints:** When applicable, we may incorporate relational constraint in the form of a penalty term in the loss function. For example, in our BoxWorld example we can notice that posY(floor) should be always 0. In general, the choice of relational language makes it possible to pose constraints based on our knowledge regarding the scene. Enforcing these constraints does not necessarily speed up the learning as we will show in the BoxWorld experiment in Section 5.3.1. However, it will ensure that the (learned) state representation and consequently the learned relational policy resemble our desired structure of the problem.

3. **Semi-supervised setting:** While it is not desirable to label every single scene that may happen during learning, in most cases it is possible to provide a few labeled scene to help the model to learn the desired state representation faster. These reference points can then be incorporated to the loss function to encourage the network to learn a representation that matches to those provided labeled scenes. We have used a similar approach in Asterix experiment (see Section E) to significantly increase the speed of learning.

5.2.2 Action Representation

We formulate the policy gradient in a form that allows the learning of the actions via one (or multiple) target predicates. These predicates exploit the background facts, the state representation predicates, as well as auxiliary predicates to incorporate higher level concepts. In a typical Deep Policy Gradient (DPG) learning, the probability distribution of actions is usually learned by applying a multi layer perception with a `softmax` activation function

in the last layer. In our proposed RRL, the action probability distributions can usually be directly associated with groundings of an appropriate predicate. For example, in BoxWorld example in Fig. 5.1, we define a predicate `move(A,B)` and associate the actions of the agent with the groundings of this predicate. In an ideal case, where there is deterministic solution to the RRL problem, the predicate `move(A,B)` may be learned in such a way that, at each state, only the grounding (corresponding to the correct action) would result 1 ('true') and all the other groundings of this predicate become 0. In such a scenario, the agent will follow the learned logic deterministically. Alternatively, we may get more than one grounding with value equal to 1 or we get some fuzzy values in the range of $[0, 1]$. In those cases, we estimate the probability distribution of actions similar to the standard deep policy learning by applying a `softmax` function to the valuation vector of the learned predicate `move` (i.e., the value of `move(X,Y)` for `X,Y`$\in$ $\{$`a,b,c,floor`$\}$).

5.3 Experiments

In this section we explore the features of the proposed RRL framework via several experiments. We have implemented[1] the models using Tensorflow [64].

5.3.1 BoxWorld Experiment

BoxWorld environment has been widely used as a benchmark in past RRL systems [36, 78, 79]. In these systems the state of the environment is usually given as an explicitly relational data via groundings of the predicate `on(X,Y)`. While ILP based systems are usually able to solve variations of this environments, they rely on explicit representation of state and they cannot infer the state from the image. Here, we consider the task of stacking boxes on top of each other. We increase the difficulty of the problem compared to the previous examples [36, 78, 79] by considering the order of boxes and requiring that the stack is formed on top of the blue box (the blue box should be on the floor). To make sure the

[1]The python implementation of the algorithms in this chapter is available at `https://github.com/dnlRRL2020/RRL`

models learn generalization, we randomly place boxes on the floor in the beginning of each episode. We consider up to 5 boxes. Hence, the scene constants in our ILP setup is the set {a,b,c,d,e,floor}. The dimension of the observation images is $64 \times 64 \times 3$ and no explicit relational information is available for the agents. The action space for the problem involving n boxes is $(n+1) \times (n+1)$ corresponding to all possibilities of moving a box (or the floor) on top of another box or the floor. Obviously some of the actions are not permitted, e.g., placing the floor on top of a box or moving a box that is already covered by another box.

Comparing to Baseline: In the first experiment, we compare the performance of the proposed RRL technique to a baseline. For the baseline we consider standard deep A2C (with up to 10 agents) and we use the implementation in `stable-baseline` library [82]. We considered both MLP and CNN policies for the deep RL but we report the results for the CNN policy because of its superior performance. For the proposed RRL system, we use two convolutional layers with the kernel size of 3 and strides of 2 with $tanh$ activation function. We apply two layers of MLP with `softmax` activation functions to learn the groundings of the predicates `posH(X,Y)` and `posV(X,Y)`. Our presumed grid is $(n+1) \times (n+1)$ and we allow for positional constants {0,1,...,n} to represent the locations in the grid in our ILP setting. As constraint we add penalty terms to make sure `posV(floor,0)` is True. We use vanilla gradient policy learning and to generate actions we define a learnable hypothesis predicate `move(X,Y)`. Since we have $n+1$ box constants (including floor), the groundings of this hypothesis correspond to the $(n+1) \times (n+1)$ possible actions. Since the value of these groundings in dNL-ILP will be between 0 and 1, we generate `softmax` logits by multiplying these outputs by a large constant c (e.g., $c = 10$). For the target predicate `move(X,Y)`, we allows for 6 rules in learning (corresponding to dNL-DNF function with 6 disjunctions). The complete list of auxiliary predicates and parameters and weights used in the two models are given in Section B. As indicated in Fig. 5.3 and defined in (5.1), we introduce predicate `on(X,Y)` as a function of the low-level

state representation predicates `posV(X,Y)` and `posH(X,Y)`. We also introduce higher level concepts using these predicates to define the aboveness (i.e., `above(X,Y)`) as well as `isCovered(X,Y)`. Fig. 5.5 compares the average success per episode for the two models for the two cases of $n = 4$ and $n = 5$. The results shows that for the case of $n = 4$, both models are able to learn a successful policy after around 7000 episodes. For the more difficult case of $n = 5$, our proposed approach converges after around 20K episodes whereas it takes more than 130K episodes for the A2C approach to converge, and even then it fluctuates and does not always succeed.

Effect of background knowledge: Contrary to the standard deep RL, in an RRL approach, we can introduce our prior knowledge into the problem via the powerful predicate language. By defining the structure of the problem via ILP, we can explicitly introduce inductive biases [83] which would restrict the possible form of the solution. We can speed up the learning process or shape the possible learnable actions even further by incorporating background knowledge. To examine the impact of the background knowledge on the speed of learning, we consider three cases for the BoxWorld problem involving $n = 4$ boxes. The baseline model (RRL1) is as described before. In RRL2, we add another auxiliary predicate which defines the movable states as:

$$\text{movable(X,Y)} \leftarrow \neg\text{isCovered(X)},\ \neg\text{isCovered(Y)},\ \neg\text{same(A,B)},\ \neg\text{isfloor(X)},\ \neg\text{on(X,Y)}$$

where $\neg$ indicates the negation of a term. In the third model (RRL3), we go one step further, and we force the target predicate move(X,Y) to incorporate the predicate `movable(X,Y)` in each of the conjunction terms. Fig. 5.6 compares the learning performance of these models in terms of average success rate (between [0,1]) vs the number of episodes.

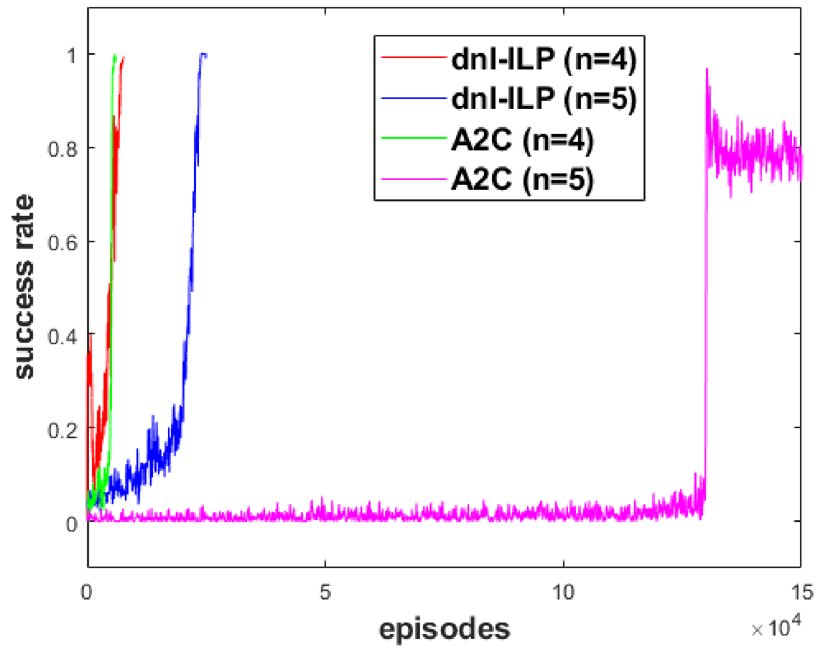


Figure 5.5: Comparing deep A2C and the proposed model on BoxWorld task

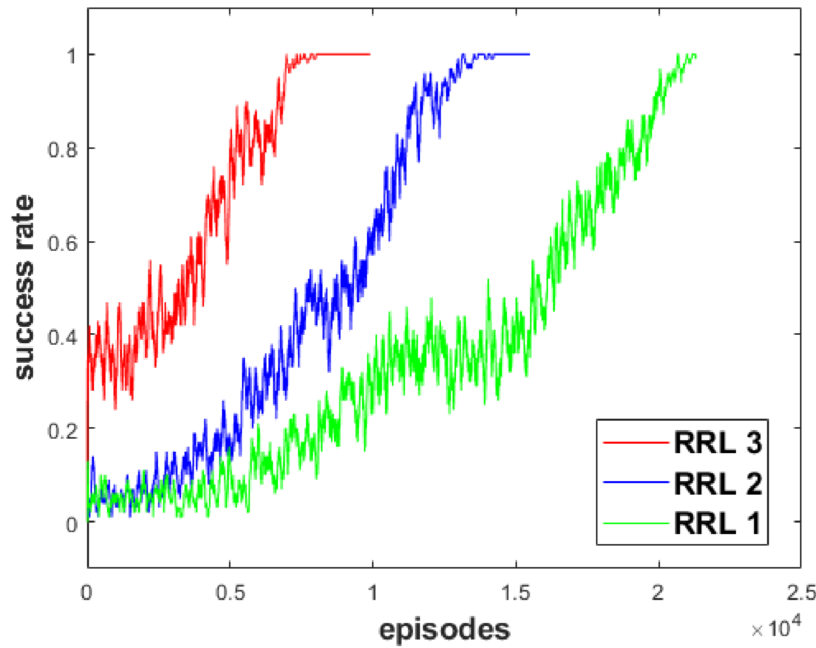


Figure 5.6: Effect of background knowledge on learning BoxWorld

Interpretability: In the previous experiments, we did not consider the interpretability of the learned hypothesis. Since all the weights are fuzzy values, even though the learned hypothesis is still a parameterized symbolic function, it does not necessarily represent a valid Boolean formula. To achieve an interpretable result we add a small penalty as described in (3.15). We also add a few more state constraints to make sure the learned representation follow our presumed grid notations. The learned action predicate is found as:

$$\text{move(X, Y)} \leftarrow \text{moveable(X, Y)},\ \neg\text{lower(X, Y)}$$

$$\text{move(X, Y)} \leftarrow \text{moveable(X, Y)},\ \text{isBlue(Y)}$$

$$\text{lower(X, Y)} \leftarrow \text{posV(X, Z)},\ \text{posV(Y, T)},\ \text{lessthan(Z, T)}$$

Details of the experiment is given in Appendix B.

5.3.2 GridWorld Experiment

We use the GridWorld environment introduced in [77] for this experiment. This environment is consisted of a 12×12 grid with keys and boxes randomly scattered. It also have an agent, represented by a single dark gray square box. The boxes are represented by two adjacent colors. The square on the right represents the box's lock type whose color indicates which key can be used to open that lock. The square on the left indicates the content of the box which is inaccessible while the box is locked. The agent must collect the key before accessing the box. When the agent has a key, provided that it walks over the lock box with the same color as its key, it can open the lock box, and then it must enter to the left box to acquire the new key which is inside the left box. The agent cannot get the new key prior to successfully opening the lock box on the right side of the key box. The goal is for the agent to open the gem box colored as white. We consider two difficulty levels. In the simple scenario, there is no (dead-end) branch. In the more difficult version, there can be one branch of dead end. An example of the environment and the branching

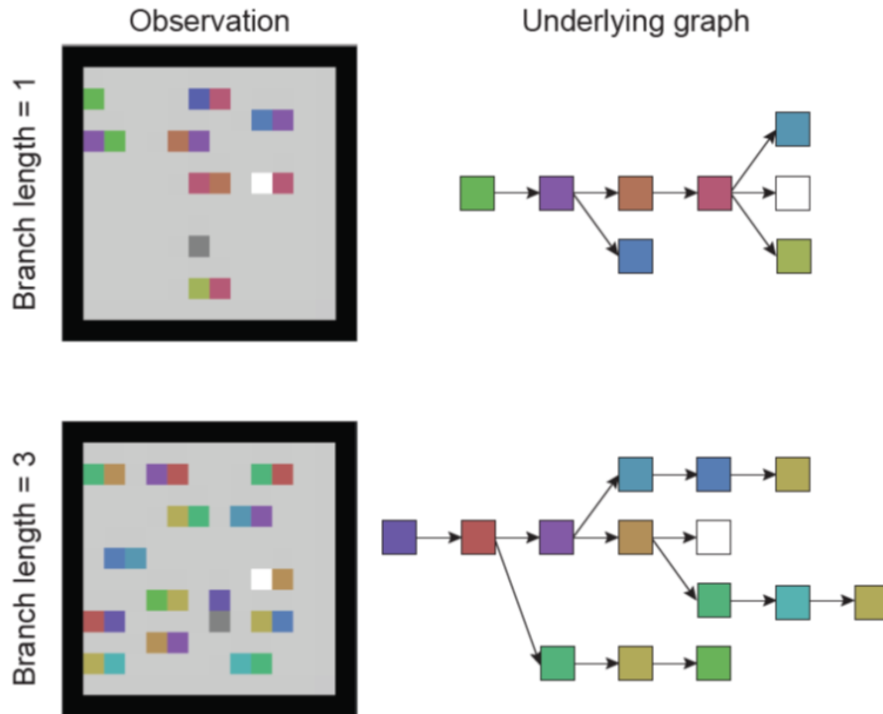


Figure 5.7: GridWorld environment [77]

scenarios is depicted in Fig. 5.7. This is a very difficult task involving complex reasoning. Indeed, in the original work it was shown that a multi agent A3C combined with a non-local learning attention model could only start to learn after processing 5×10^8 episodes. To make this problem easier to tackle, we modify the action space to include the location of any point in the grid instead of directional actions. Given this definition of the problem, the agent's task is to give the location of the next move inside the rectangular grid. Hence, the dimension of the action space is $144 = 12 \times 12$. For this environment, we define the predicates `color(X,Y,C)`, where $X, Y \in \{1, \dots 12\}, C \in \{1, \dots, 10\}$ and `hasKey(C)` to represent the state. Here, variables X, Y denote the coordinates, and the variable C is for the color. Similar to the BoxWorld game, we included a few auxiliary predicates such as `isBackground(X,Y)`, `isAgent(X,Y)` and `isGem(X,Y)` as part of the background knowledge. The representational power of ILP allows us to incorporate our prior knowledge about the problem into the model. As such we can include some higher level auxiliary helper

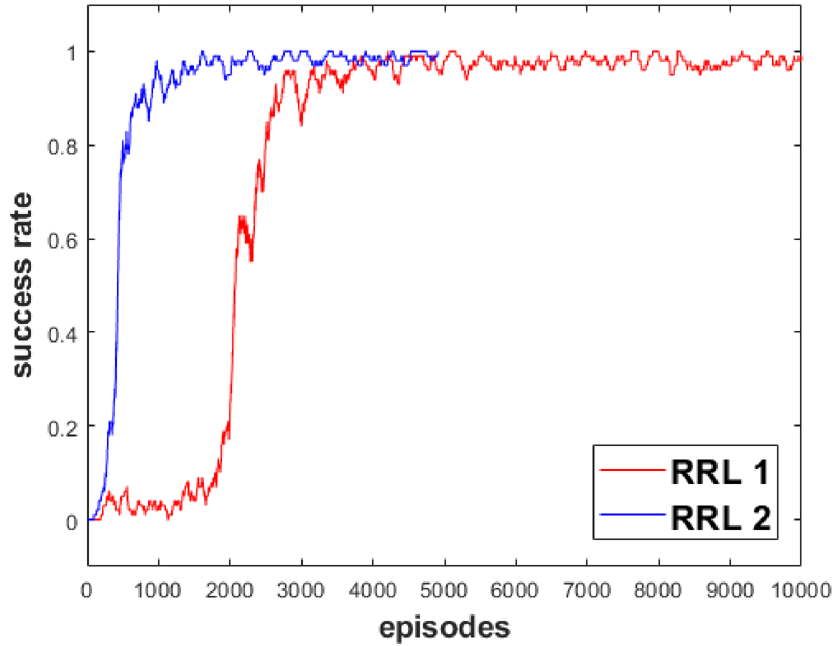


Figure 5.8: Effect of background knowledge on learning GridWorld

predicates such as :

$$\text{isItem(X, Y)} \leftarrow \neg\text{isBackground(X, Y)}, \neg\text{isAgent(X, Y)}$$

$$\text{locked(X, Y)} \leftarrow \text{isItem(X, Y)}, \text{isItem(X,Z)}, \text{inc(Y, Z)}$$

where predicate `inc(X,Y)` defines increments for integers (i.e., `inc(n,n+1)` is true for every integer n). The list of all auxiliary predicates used in this experiment as well as the parameters of the neural networks used in this experiment are given in Section C. Similar to previous experiments we consider two models, an A2C agent as the baseline and our proposed RRL model using the ILP language described in Section C. We listed the number

Table 5.1: Number of training episodes required for convergence

model	Without Branch	With Branch
proposed RRL	700	4500
A2C	$> 10^8$	$> 10^8$

of episodes it takes to converge in each setting in Table 5.1. As the results suggest, the

proposed approach can learn the solution in both settings very fast. On the contrary, the standard deep A2C was not able to converge after 10^8 episodes. This example restates the fact that incorporating our prior knowledge regarding the problem can significantly speed up the learning process.

Further, similar to the BoxWorld experiment, we study the importance of our background knowledge in the learning. In the first task (RRL1), we evaluate our model on the non-branching task by enforcing the action to include the $isItem(X, Y)$ predicate. In RRL2, we do not enforce this. As shown in Fig. 5.8, RRL1 model learns 4 times faster than RRL2. Arguably, this is because, enforcing the inclusion of $isItem(X, Y)$ in the action hypothesis reduces the possibility of exploring irrelevant moves (i.e., moving to a location without any item). Details of the experiment is given in Appendix C.

5.3.3 Relational Reasoning

Combining dNL-ILP with standard deep learning techniques is not limited to the RRL settings. In fact, the same approach can be used in other areas in which we wish to reason about the relations of objects. To showcase this, we consider the relational reasoning task involving the Sort-of-CLEVR [41] dataset. This dataset (See Fig. 5.4b) consists of 2D images of some colored objects. The shape fo each object is either a rectangle or a circle and each image contains up to 6 objects. The questions are hard-coded as fixed-length binary strings. Questions are either non-relational (e.g, "what is the color of the green object?") or relational (e.g., "what is the shape of the nearest object to the green object?"). In [41], the authors combined a CNN generated feature map with a special type of attention based non-local network in order to solve the problem. We use the same CNN network and similar to the GridWorld experiment, we learn the state representation using predicate `color(X,Y,C)` (the color of each cell in the grid) as well as `isCircle(X,Y)` which learn if the shape of an object is circle or not. Our proposed approach reaches the accuracy of 99% on this dataset compared to the 94% for the non-local approach presented in [41].

The details of the model and the list of predicates in our ILP implementation is given in Appendix D.

5.3.4 Asterix Experiment

In Asterix game (an Atari 2600 game), the player controls a unit called Asterix to collect as many objects as possible while avoiding deadly predators (see Fig. 5.9). Even though it is a rather simple game, many standard RL algorithms score significantly below the human level. for example, as reported in [84], DQN and A3C achieve the scores of 3170 and 22140 on average, respectively even after processing hundreds of millions of frames. Here, our goal is not to outperform the current approaches. Instead, we demonstrate that by using a very simple language for describing the problem, an agent can achieve scores in the range of 30K-40K with only a few thousands of training scenes. For this game we consider the active part of the scene as a 8x12 grid. For simplicity, we consider only 4 types of objects; agent, left to right moving predator, right to left moving predator and finally the food objects. The dimension of the input image is 128x144. We use 4 convolutional layers with strides of [(2,3),(2,1),(2,2),(2,2)] and kernel size of 5 to generate a feature map of size 8x12x48. By applying 4 fully connected layers with `softmax` activation function we learn the groundings of the predicates corresponding to the four types of objects, i.e., `O1(X,Y)`,...,`O4(X,Y)`.

The complete list of predicates for this experiment is listed in Appendix E.

We learn 5 nullary predicates corresponding to the 5 direction of moves (i.e., no move, left,right,up,down) and use the same policy gradient learning approach as before. The notable auxiliary predicates in the chosen language are the 4 helper predicates that define bad moves. For example, `badMoveUp()` states that an upward move is bad when there is a predator in the close neighborhood of the agent and in the top row. Similarly, `badMoveLeft()` is defined to state that when a predator is coming from left side of an agent, it is a bad idea to move left.

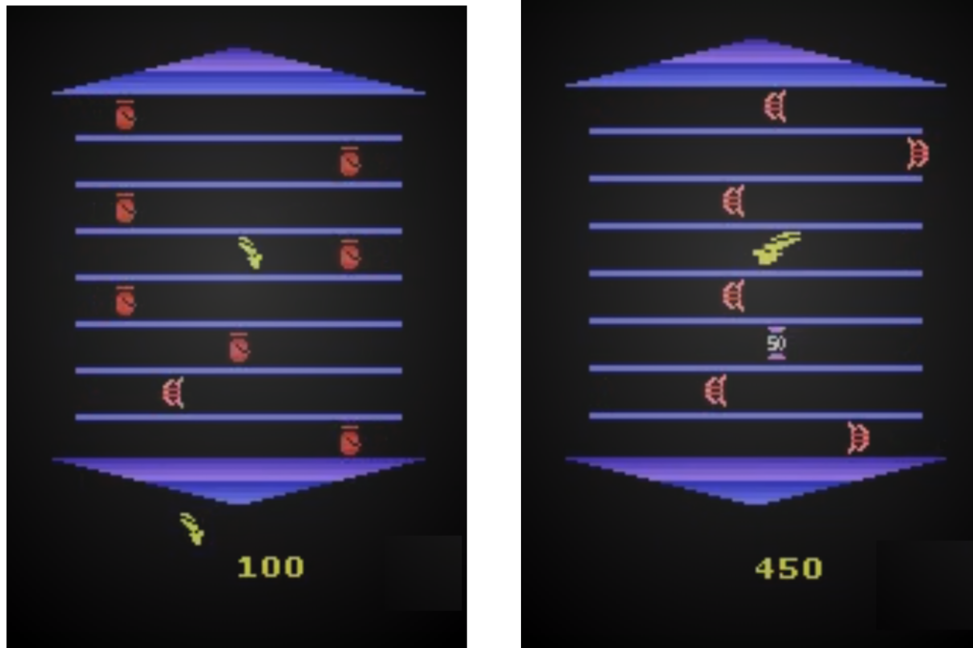


Figure 5.9: Asterix game environment example

However, given the complexity of the scene and the existence of some overlappings between the objects, learning the representation of the state is not as easy as the previously explored experiments. To help the agent learn the presumed grid presentation, we provide a set of labeled scene (a semi-supervised approach) and we penalize the objective function using the loss that is calculated between these labels and the learned representation. Fig. 5.10 shows the learning curve for two cases of the using 20 and 50 randomly generated labeled scenes. In the case of 50 provided labels, the agent finally learns to score around 30-40K each episodes. Please note that we did not include all the possible objects that are encountered during later stages of the game and we use a simplistic representation and language just to to demonstrate the application of RRL framework in more complex scenarios.

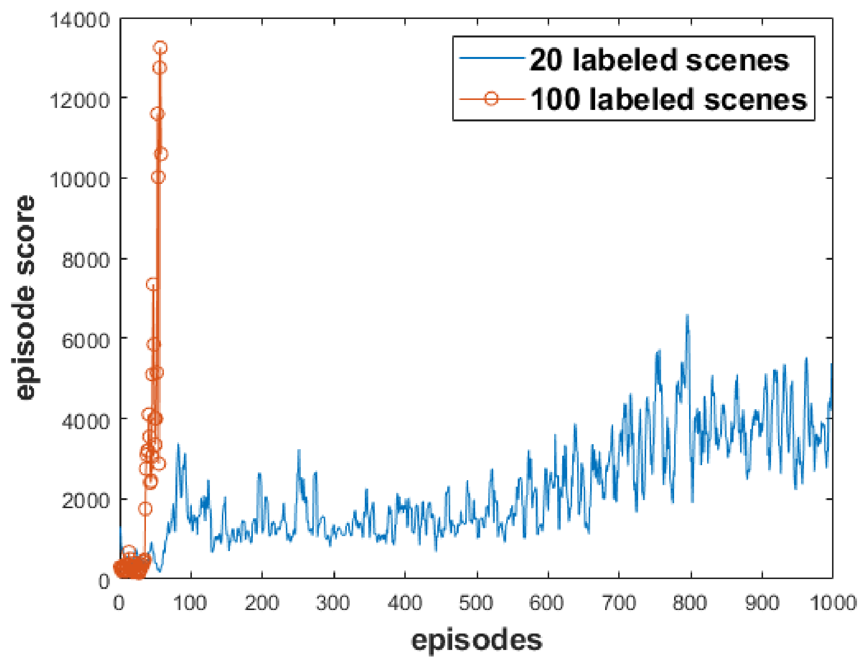


Figure 5.10: Score during training in Asterix experiment

CHAPTER 6

DECODING LDPC CODES OVER BINARY ERASURE CHANNELS VIA DNL

6.1 Introduction

It is known that Maximum Likelihood (ML) decoding over a BEC channel can be reduced to solving a set of linear equations; which can be implemented with the complexity $O(\nu^3)$ for ν erased bits in a general case. However, in most practical applications this approach is infeasible even for moderate length codes. As a result, some forms of iterative algorithms (based on a message passing algorithm) are often used to obtain a sub-optimal solution for decoding Low Density Parity Check (LDPC) codes. These iterative algorithms can perform very close to the optimal ML decoding bounds for sufficiently long LDPC codes. However, for short length codes, they do not perform as well. Moreover, in real-time applications, it is desirable to decode the messages in batches to increase the speed. As such, Deep Neural Networks (DNN), if successful in learning how to decode, with their suitable structures for batch processing in parallel can prove to be a viable alternative.

In decoding, we have virtually unlimited training examples. Since learning a decoder is basically a supervised learning task, the application of DNN in designing an efficient decoder may seem trivial at first. However, there exist several challenges. Even though we have as many training examples as we need, there are 2^m different possible codewords for a message of length m; which is prohibitive even for a rather short message (e.g., ~ 100 bits). Thus, the deep network should be able to learn the decoding algorithm instead of learning just the non-linear approximation as in Multi-Layer Perceptron (MLP) layers. In most deep algorithm learning tasks, auto-regressive models such as Recurrent Neural Networks (RNNs) are deployed which allow for recursion through intermediate results, and consequently learning some kinds of algorithms to solve the problem in an iterative fashion. For example in [42], the authors successfully employed RNNs to design a very efficient

decoder for convolutional and turbo codes over additive white Gaussian noise (AWGN) channels and in [43] the authors proposed a framework for soft decoding of linear codes of arbitrary length. However, as was shown in [85], typical neural network layers often fail to learn the exact logical flow in discrete (Binary) arithmetic tasks. This is specially true for decoding LDPC codes over BEC. In this Chapter, we study the application of the dNL networks in learning a decoder for LDPC codes over BEC channels. The main idea is to formulate an iterative message passing scheme for decoding LDPC codes in which, standard RNN cells are replaced by a multi layer design from the dNL networks. At the end of this chapter we also discuss another possibility of using dNL framework for decoding LDPC codes. We will show how the maximum likelihood approach for solving the parity check equations can be directly formulated via the use of dNL and in particular the differentiable XOR neurons.

6.2 Message Passing Decoding

In most practical LDPC decoders, a variant of the iterative algorithm is often used. These iterative algorithms are mainly based on the Message Passing (MP) algorithm. In MP, a unique message from each variable node is sent to every connected check node in the forward path. Likewise in the backward path, every check node computes a proper response message which is sent back to each variable node it is connected to, and hence the name message passing. Under certain simplifying assumptions, one can view the messages in the MP algorithm as the belief (i.e., estimated probabilities) of the model regarding the values of each particular variable node in the graph. Our objective is to employ deep neural networks to learn an iterative (recurrent) algorithm by exploiting the structure of the parity check matrix. While by iterating between variable nodes and check nodes, the model learns some forms of the message passing scheme, we make no probabilistic assumptions and we let the network learn an efficient iterative algorithm. This allows (at least in theory) for learning a more complicated model with possible performance improvements over the

belief propagation scheme. Let $\mathbf{H} \in \{0,1\}^{r\times n}$ be the parity check matrix for a length n regular LDPC code with parameters (d_v, d_c), i.e., $\mathbf{H}$ has d_v and d_c non-zero elements in each column and row, respectively. We assign vector $V_i \in \mathbb{R}^{dim_v}$, $i \in \{1,\ldots,n\}$ to each variable node and similarly vector $C_j \in \mathbb{R}^{dim_c}$, $j \in \{1,\ldots,r\}$ to each check node. Corresponding to the ith variable node at the time stamp t where $t \in \{1,\ldots,t_{max}\}$, we define two sets of functions $\mathscr{F}_i^{(t)} : \mathbb{R}^{d_c\times dim_v} \mapsto \mathbb{R}^{dim_c}$ (from the variable node to the check node) and $\mathscr{G}_i^{(t)} : \mathbb{R}^{d_v\times dim_c} \mapsto \mathbb{R}^{dim_v}$ (from the check node to the variable node). Here, dim_v and dim_c denote the dimensions of the representation used in variable nodes and check nodes, respectively. In soft decoding using belief propagation, the erasure bits are usually represented by a real valued scalar (0.5). However, we choose the one-hot representations of depth 3 for each variable node, which is more suitable for our model. In our representation, the three bit positions in the one-hot vector are correspond to values of '0', '1', and 'e' (erasure), respectively. For example, if the bit value is '0', the vector is $[1,0,0]$. The dimension dim_c of check-node vectors is a design parameter of the model which controls the amount of information that can be sent from each check node to variable nodes. We formulate the recursive formula for each variable node v_i at time stamp $t+1$ as:

$$v_i^{(t+1)} = \mathscr{G}_i^{(t)}\big(Vec(\,\mathbb{C}_i^{(t+1)}\,)\big) \quad \text{, where} \tag{6.1}$$

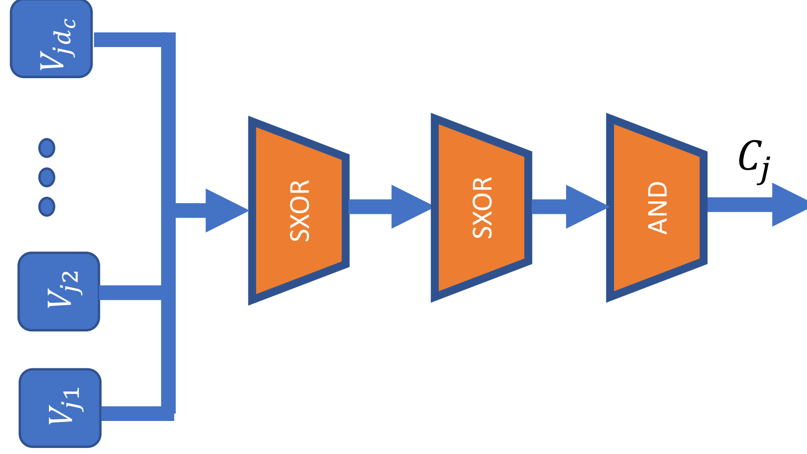
$$\mathbb{C}_i^{(t+1)} = \{C_j^{(t+1)} | H(j,i) = 1,\ j \in \{1,\ldots,r\}\,\} \tag{6.2}$$

also,

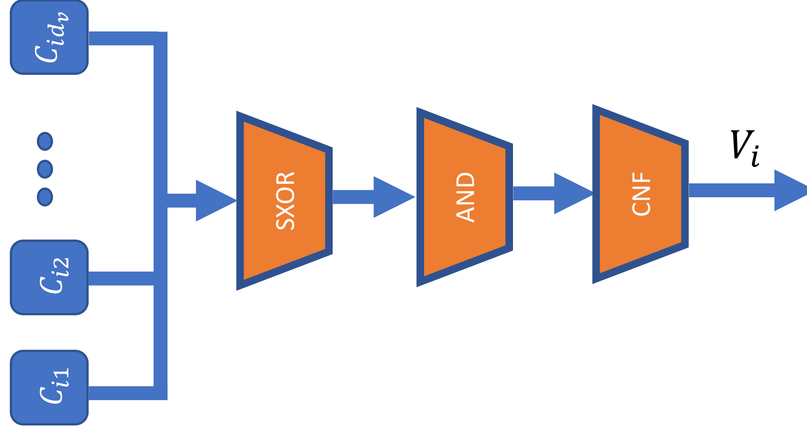
$$C_j^{(t+1)} = \mathscr{F}_i^{(t)}\big(Vec(\,\mathbb{V}_{i,j}^{(t)}\,)\big) \quad \text{, where} \tag{6.3}$$

$$\mathbb{V}_{i,j}^{(t)} = \{V_{i'}^{(t)} | H(j,i') = 1,\ i' \in \{1,\ldots,n\}\,\} \tag{6.4}$$

where $Vec(\mathcal{S})$ generates a vector by concatenates all the vectors in set $\mathcal{S}$. Further, we can exclude the intrinsic information (the information that each node already possesses about



(a) Function $\mathscr{F}(.)$ from a variable node to a check node



(b) Function $\mathscr{G}(.)$ from a check node to a variable node

Figure 6.1: Logical functions used in forward/backward operations

itself) as in the BP algorithm, i.e.

$$\mathbb{V}_{i,j}^{(t)} = \{V_{i'}^{(t)} | H(j,i') = 1,\ i' \in \{1,\ldots,n\}, i' \neq i\} \tag{6.5}$$

However, our experiments later showed that although using extrinsic information slightly increases the speed of the convergence, it has no significant effect on the network performance and the model can learn the desired algorithm regardless. we design the functions $\mathscr{F}_i^{(t)}(\cdot)$'s and $\mathscr{G}_i^{(t)}(\cdot)$'s as it is depicted in Fig. 6.1.

For training, we define the loss function at time t as the average `softmax` cross-entropy between the estimated values of the variable nodes and the correct codeword vector, both represented as one-hot vectors of length n and depth 3, i.e.

$$\mathcal{L}^{(t)} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{b=1}^{3} x_{ib}\log(v'^{(t)}_{ib}) \quad , \quad v'_i = softmax(v_i) \tag{6.6}$$

We may consider these options for designing the forward and backward function sets (i.e. (6.4) and (6.2)):

- Option 1: Use the same two functions $\mathscr{F}(\cdot)$ and $\mathscr{G}(\cdot)$ for all the time stamps and variable nodes, i.e. sharing the weights across both dimensions.
- Option 2: Share functions across the time dimension but use different functions $\mathscr{F}_i(\cdot)$'s and $\mathscr{G}_i(\cdot)$ for each variable node.
- Option 3: Do not share at all, i.e., choose different functions $\mathscr{F}_i^{(t)}(\cdot)$'s and $\mathscr{G}_i^{(t)}(\cdot)$'s for different time stamps and different variable nodes.

Although Option 1 may seem to be the most rigid and inflexible of all, our experimental results surprisingly show that its convergence speed is the fastest among the other options. While it requires a significantly fewer number of trainable parameters, it offers the best overall performance specially for the case of regular LDPC codes. For irregular LDPC codes, Option 2 provides more flexibility over the first option and performs better. Sharing weights across the time dimension facilitates learning a repetitive algorithmic task. In particular, instead of training the model using a large number of iterations (time stamps), we can train using only a few iterations and let the model learn a recursive algorithm. Consequently, both Options 1 and 2 provide good generalizations. Moreover, it allows to limit the number of iterations (to increase the training speed and reduce complexity) in the training phase while running the model for many more iterations in the test time to obtain much improved results. On the other hand, although Option 3 performs well when tested for the exact number

of training iterations, its accuracy cannot be improved by iterating beyond the number of training iterations because the weights are not shared across time. As such, Option 3 performs relatively poor.

When the number of iterations in the training phase is small (i.e. $t_{max} < 3$), we can train the model by minimizing $\mathcal{L}^{(t_{max})}$. However, when the number of iterations is large, this approach usually leads to no convergence or very slow convergence specially at the beginning of the training. To cope with this problem, in the beginning of training, we must include the loss functions from all the time stamps but then gradually remove the effect of the earlier time stamps in the loss function:

$$\mathcal{L} = \sum_{t=1}^{t_{max}} \alpha^{(t_{max}-t)} \mathcal{L}^{(t)} \tag{6.7}$$

where the value of α is reduced from 1 to 0 gradually as we progress in the training.

6.3 Experiments

We use Gallagher algorithm[86] for creating random parity check matrices used in our experiments. In order to avoid over-fitting, instead of repeating the training samples, for each training batch we create a new set of randomly generated codewords in which each bit is erased independently with the probability ϵ. In all the experiments we use batch size of 100 codewords and we train the model using ADAM [63] optimizer with the learning rate of 0.003. The dimensions of XOR and AND (conjunction) layers in designing $\mathscr{F}_i^{(t)}$'s and $\mathscr{F}_i^{(t)}$'s are set to 100 and we use $dim_c = 30$.

6.3.1 Performance

Fig. 6.3 depicts the decoding performance in terms of the average number of decoding errors per bit (BER) and also the average number of block (frame) errors (FER) for a half-rate regular $(3, 6)$ LDPC code of length 48, and compare it to the results of the belief propagation

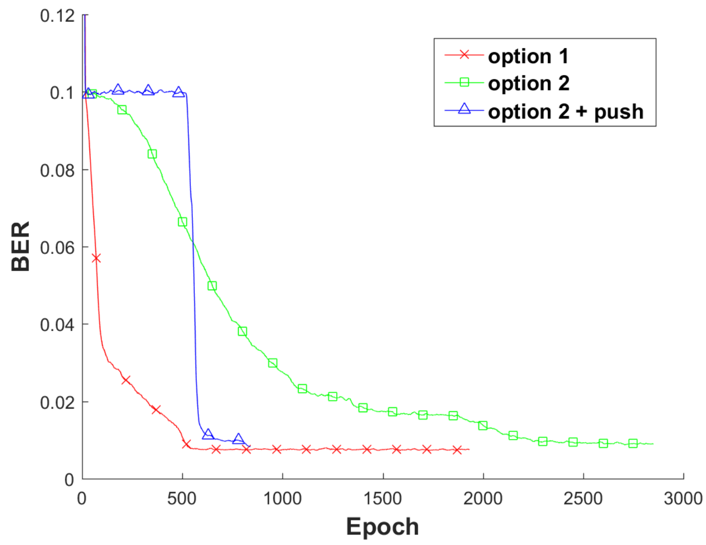
(BP) algorithm. Here, we train the model (by sharing all functions according to Option 1) using only 3 iterations (i.e. $t_{max} = 4$). After the training finished (model converged), we test the model using 10 iterations for the total of 100k randomly selected testing codewords. For the BP algorithm, we evaluate the performance using 50 iterations. As it is apparent from Fig. 6.3, our proposed model outperforms the BP algorithm for the (3,6) regular LDPC code. While the results are shown for a typical randomly generated code, we obtained the same level of performance improvements for all the test cases. As we increased the code length from 48 to values above 132, we noticed that while our model achieves its optimal results using significantly fewer iterations than BP, the performance gap between the two methods reduces for higher lengths.

6.3.2 Generalizations

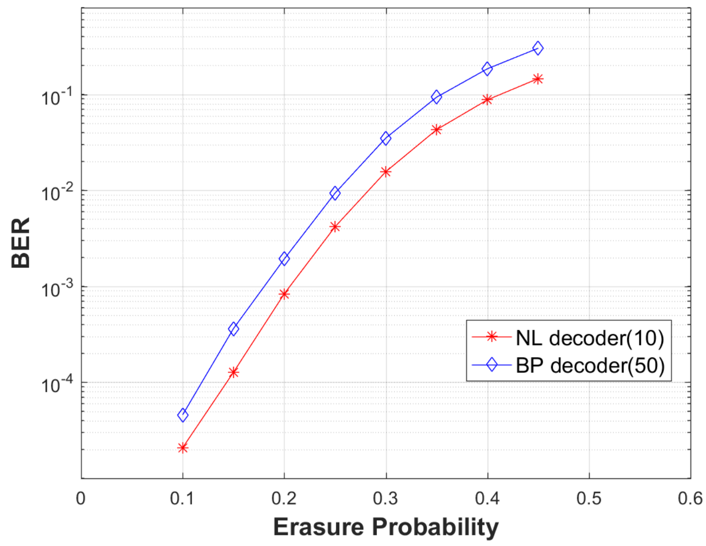
In most of approaches based on deep neural networks, while the models are usually able to outperform the classical approaches, the generalization is limited and the models must be re-trained for each particular different settings. In contrast, our proposed decoder model using a recurrent neural logic layer scheme is able to generalize very well under various settings and situations for which it has not been trained. To demonstrate this, we randomly generate three half-rate codes with parity check matrices H_1 and H_2 of code length 48 and H_3 of code length 132. We trained the model using only H_1 for erasure probability $\epsilon = 0.2$. Then, we tested the model generalization. Specifically, our model can generalize well for 3 variations:

1. Erasure Probability: We trained the model using only one specific erasure probability and evaluate its performance under different erasure probability settings. As it is apparent from the first row of Table 6.1, the decoder performance when tested for $\epsilon = 0.3$ does not depend on whether the model has been trained on $\epsilon = 0.3$ or $\epsilon = 0.2$.

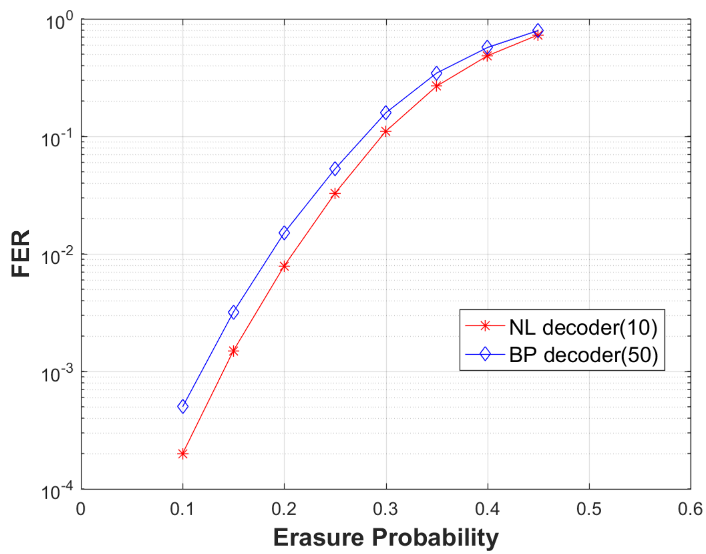
2. Parity Check Matrix: More importantly, our model is not dependent on the specific code that is used in the training as long as we are using codes with similar (d_v, d_c)



(a) Convergence



(b) Bit Error



(c) Block Error

Figure 6.2: Performance comparison between the neural logic decoder and BP

parameters. As it is depicted by the second row of Table 6.1, the model that has been trained for H_1 still provides very good performance when it is used for evaluating the code associated with H_2.

3. Code Length: This means that we can train the model using a smaller code size and the model can generalize well under larger code lengths if other parameters of the code remains the same. Since the dimension of the learnable functions only depends on the weight degrees (i.e., (d_v, d_c)) of the LDPC code, the same model can be used in different code lengths. In particular, as the third row of Table 6.1 shows the performance of the model trained on a $(48, 24)$ LDPC code when tested on a $(132, 66)$ code is only slightly less than the optimal performance for a model that is trained specifically for the latter code.

Table 6.1: Evaluating the generalization performance of the model

Train Setting	Test Setting	Test Generalization Performance (BER)	Test Optimal Performance (BER)
$H_1, \epsilon = .2$	$H_1, \epsilon = .3$	0.01480	0.01476
$H_1, \epsilon = .2$	$H_2, \epsilon = .3$	0.00891	0.00832
$H_1, \epsilon = .2$	$H_3, \epsilon = .3$	0.0014	0.0013

6.3.3 dNL vs MLP

To compare the performance of MLP and dNL we design the feed-forward functions $\mathscr{F}$ and $\mathscr{G}$ first using a 3 layers MLP architecture (LDPC-MLP) and for the second model using dNL architecture (LDPC-dNL). Table 6.2 summarizes the layers dimension and type/activation functions used in each model. We tried various settings and dimensions and number of layers and picked comparable settings in terms of the number of weights used in each model (see Table. 6.2). In both models, we generated random codewords for a regular LDPC(3,6) code of length 48 for each training epoch and set the number of message passing iteration to 2. ($t_{max} = 2$). For the test data, we ran the trained model for many more iterations to see how much each model has generalized and learned the iterative algorithm. We used erasure

probability of $\epsilon = 0.2$ in training and we tested the models for different erasure probabilities. Fig. 6.3 depicts the performance of each model in terms of bit error probability (BER) of the two models for $\epsilon = 0.4$. As one may expect, the model based on MLP performs better and generate lower BER for the $t = 2$ since the models were trained using the same setup and MLP is in general more flexible and can reduce the cost function more efficiently. However, increasing the number of iterations in test time not only does not improve the accuracy for MLP based model, it even degrades the performance for $t > 3$. On the other hand, the model based on dNL performs only moderately for $t = 0$, however, as the number of iterations increases, the performance improves significantly as one might expect from a message passing algorithm. In other words, the rigid structures of the dNL architecture while may seem limiting on the surface, it allows for learning discrete algorithms much more efficiently and can generalize better.

Table 6.2: Design parameters for functions $\mathscr{F}$ and $\mathscr{G}$

Function	Dimensions	Type/Activation
MLP : $\mathscr{F}$	100,30	relu,sigmoid
MLP : $\mathscr{G}$	100,100,60	relu,relu,sigmoid
dLN : $\mathscr{F}$	100,30	XOR,CONJ
dLN : $\mathscr{G}$	200,100,6	XOR,CONJ,DISJ

6.4 ML decoding of LDPC over BEC via dNL

It can be shown that the decoding of the LDPC codes over erasure channels such as BEC can be reduced to solving a set of linear equations in GF(2). For a received codeword of length n with ν erased bits, this problem can be solved by methods such as Gauss Jordan elimination using GF(2) arithmetic. Let's consider a regular LDPC code of length (n, k) with parameters (d_v, d_c). The set of k parity check equations for a received codeword $\mathbf{x}$ can

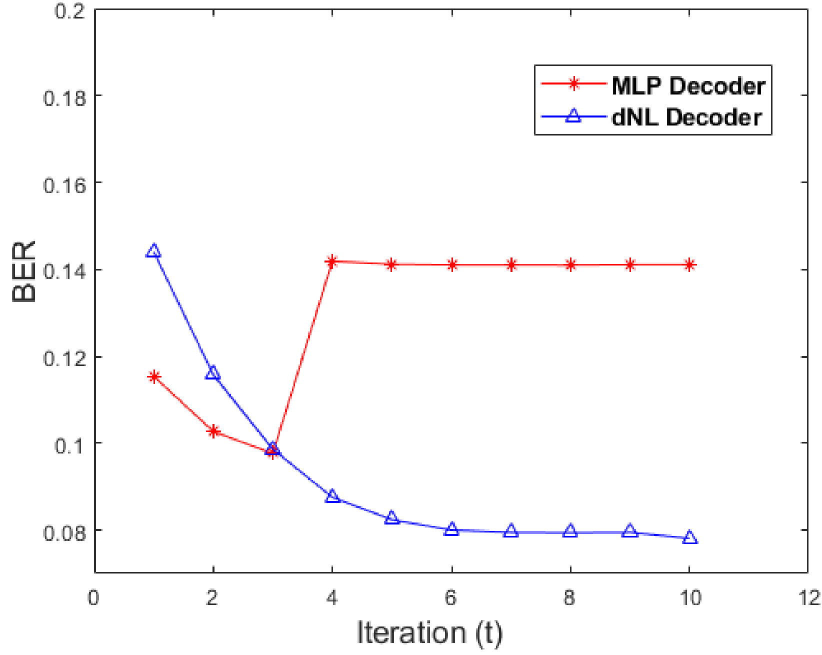


Figure 6.3: Performance comparison between the dNL and MLP for iteratively decoding LDPC codes over BEC channel

be expressed as :

$$x_{i1} \oplus x_{i2} \oplus \cdots \oplus x_{id_c} = 0 \text{ for } i = 1, \ldots, k$$
$$\text{where } x_{ij} \in \{x_1, \ldots, x_n\} \tag{6.8}$$

Authors in [87] among others have shown that a set of nonlinear equations can be effectively solved using neural networks by method of gradient descent. However, we cannot directly use this method for solving the equation (6.8). In Chapter 1.1, we show that by introducing a set of mask vectors comprised of the $\{-1, 1\}$ elements, we can transform the parity check equation from GF(2) into a real domain equation of degree d_c. More precisely, we showed that for a vector $\mathbf{y}$ of length $l = 2p$, we can define p mask vectors $\{\mathbf{M}_1, \ldots, \mathbf{M}_p\}$ such that

:

$$\mathbf{M}_1 = \{M_{11}, \ldots, M_{1l}\} \ , M_{1i} = \begin{cases} 1\ , \ \text{if } i < p \\ \\ -1\ , \ \text{else} \end{cases} \tag{6.9}$$

$$\mathbf{M}_j = \mathbf{M}_{j-1} \lll 1 \, \text{for } j \in \{2, 3, \ldots, p\} \tag{6.10}$$

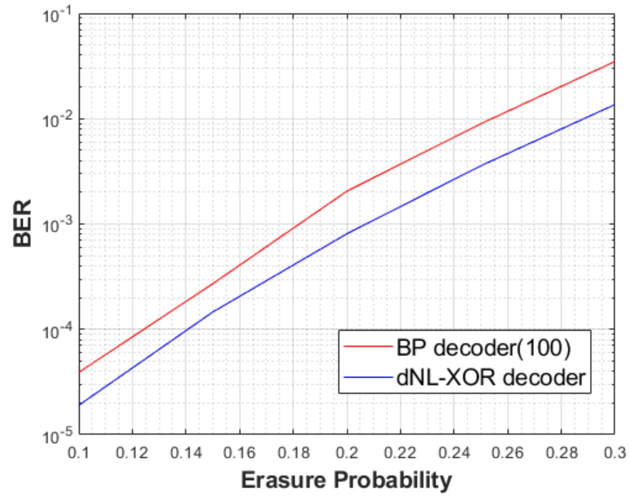
where $\lll 1$ denotes a one element left circular shift operation. As such, the GF(2) equation of $y_1 \oplus y_2 \oplus \cdots \oplus y_l = 0$ is translated to real domain as:

$$\prod_{i=1}^{p} (M_i \odot \mathbf{y}) = 0 \tag{6.11}$$

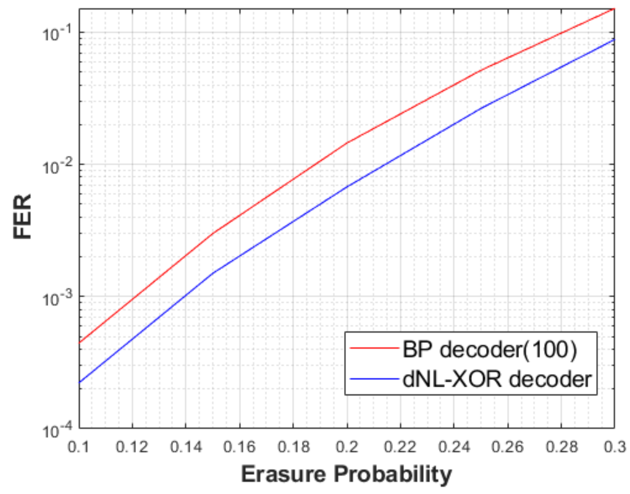
By combining the results of (6.11) and (6.8), we now have shown that the parity check equations can be transformed into k equation of degree d_v in real domain. For any received codewords $\boldsymbol{c}$ of length n with ν erased symbol, we also have a set of $n - \nu$ equations of the form $x_i - c_i = 0$. Let V_j be any of the $k + n - \nu$ described equations , similar to [87] we can define a cost function of the form :

$$\mathcal{L} = \sum_{i=1}^{k+n-\nu} \frac{1}{2} V_j^2 \tag{6.12}$$

and by methods such a gradient descent solve this erasure equation iteratively. Further, because we know the solution of any of the erased bits would be from the set of 0 and 1 elements, we can add a penalty term to encourage the weight vectors to be close to either 0 or 1 to avoid unwanted solutions to the nonlinear set of equations. To compare the performance of the proposed decoder (dNL-XOR decoder) to the standard BP algorithm, once again we use a (3,6) regular LDPC code of length 48. As it is shown in Fig. 6.4, the proposed dNL-XOR decoder outperforms the BP algorithm and reaches the accuracy of between 2 to 3 times of the BP decoder.



(a) Bit Error



(b) Block Error

Figure 6.4: Performance comparison between the dNL-XOR decoder and BP

CHAPTER 7

INDEPENDENT COMPONENT ANALYSIS USING VARIATIONAL AUTOENCODER FRAMEWORK

7.1 Introduction

One of the biggest challenges in unsupervised machine learning is finding nonlinear features from unlabeled data. In the general nonlinear case, there has been some successful applications of the deep neural networks in recent years. Multi-layer deep belief networks, Restricted Boltzman Machine (RBM) and most recently, variational autoencoders are some of the most promising new developments. Independent component analysis (ICA) is one of the most basic and successful classical tools in unsupervised learning which has been studied extensively during the early years of 2000s. For the simplest case of linear ICA without the presence of noise, this problem has been solved and the conditions for the identifiability of independent components has been well established [45]. However, for the more general form of the problem where the independent features (sources) are mixed using nonlinear transforms and specifically in the presence of noise, there has been little success. In recent years, the problem of learning ICA from the nonlinear mixtures (and similar related problems) has been addressed in works such as [46, 47]. In particular, in[47] the Adversarial Nonlinear Independent Component Analysis (Anica) algorithm was proposed which is an unsupervised learning method based on Generative Adversarial Networks (GAN)[48]. This method has shown to outperform the state of the art for the linear and nonlinear source separation tasks. Anica uses a simple non-probabilistic encoder/decoder network to obtain the components and the independence among components is encouraged via a GAN objective. This adversarial network compares the samples from the encoder to the output of a trainable generative model or alternatively to the samples obtained from the approximate product of marginals. Despite presenting good results, Anica has some

shortcomings. The main problem is the one common among most GAN based approaches. It is usually very tricky to find a balance between the two objectives (corresponding to the discriminator and generator network) for a GAN network. As such, this method requires many trials and extensive parameter search to find the right balance for weights. Further, the loss function does not always correspond to the independence and hence, the accuracy of finding independent sources fluctuates almost constantly. As such, there is no obvious method for picking the right answer in an unsupervised manner from the values of the loss function. Thus, the reported maximum correlation found from the training is not a good indicator for the performance of unsupervised method. Furthermore, the ICA model used in Anica does not consider an additive noise. Thus, as we will also show by experiments, Anica is not robust to noise and its performance drops significantly for noisy mixtures.

The problem of linear ICA in the presence of noise has been studied in [49, 50, 51, 52, 53] among others. Some of the most successful approaches (e.g. [52, 53]) use the Mean-Field variational Bayesian framework to solve the linear ICA in a Bayesian setting. However, these approaches are usually very slow and have very limited applicability even for the linear case. In this chapter, we adopt the well established Variational AutoEncoder (VAE)[88] framework to formulate the problem of nonlinear ICA in the presence of additive independent Gaussian noise. We use a technique similar to the one proposed in [47] to obtain approximate samples from the product of marginals. Further, we introduce an auxiliary objective which encourages the independence of the components without the need to train a separate GAN network. In the next section we will formulate the problem and propose a new algorithm for solving nonlinear noisy ICA.

7.2 Proposed Method

We consider the nonlinear ICA in the presence of Gaussian noise. We formulate the problem as follows: The observed measurements are $M-$dimensional signals $\mathbf{x}^{(t)}$, $t = 1, \dots N$. Similarly, let $\mathbf{s}^{(t)}$, $t = 1, \dots N$ be N data points from M mutually independent sources

$s_j, j = 1, \ldots, M$. We assume the measurement signals $\mathbf{x}^{(t)}$ be the instantaneous nonlinear mixing of the source components $s_j^{(t)}$ corrupted by an independently distributed white Gaussian noise $\mathbf{e}$,

$$\mathbf{x}^{(t)} = \mathcal{F}(\mathbf{s}^{(t)}) + \mathbf{e}^{(t)}, \tag{7.1}$$

where $\mathcal{F} : \Re^M \rightarrow \Re^M$ is a nonlinear mixing function. Even though we defined the problem using the same number of components for the source and the mixture, it is easy to extend to the more general form as well. In most probabilistic frameworks for solving ICA, a family of super Gaussian distributions is used for the independent components. In our model, to increase the flexibility, we use Gaussian mixture distribution to model the probability density function of each independent components, i.e.

$$p(\mathbf{s}) = \prod_{i=1}^{M} p(s_i) \text{ where, } p(s_i) \sim \sum_{j=1}^{K^i} \pi_j^i p(s_i|\mu_j^i, \sigma_j^i),$$

$$p(s_i|\mu_j^i, \sigma_j^i) = \frac{1}{\sqrt{2\pi{\sigma_j^i}^2}} e^{-\frac{1}{2}\frac{(s_i-\mu_j^i)^2}{{\sigma_j^i}^2}},$$

where K^i is the number of components of i mixture and π_j^i's are the mixing proportions.

The linear case of the noisy ICA problem has been investigated in [53] among others. Because of the intractability of the posterior, in these approaches, the approximate inference is usually performed via variational mean-field techniques. However, such methods are very slow and inefficient since there is a need for an iterative inference scheme such as Markov Chain Monte Carlo (MCMC) per data point. Ideally, we would like to estimate the source components $\mathbf{s}$ given the observed measurements $\mathbf{x}$ via maximum marginal likelihood method, i.e.:

$$\hat{\Theta} = \underset{\Theta}{\text{argmax}} \, \log p(\mathbf{X}) = \underset{\Theta}{\text{argmax}} \sum_{t=1}^{N} \log p(\mathbf{x}^{(t)}) \tag{7.3}$$

Assuming a latent variable model, the variational lower bound for $p(\mathbf{x})$ can be written as:

$$\log p(\mathbf{x}) \geq \mathbb{E}_{q(\mathbf{s}|\mathbf{x})}\big[\log p(\mathbf{x}, \mathbf{s}) - \log q(\mathbf{s}|\mathbf{x})\big] = \mathcal{L}(\mathbf{x}; \Theta), \tag{7.4}$$

where Θ indicates the parameters of the model. In [88], it was shown that by using the Stochastic Gradient Variational Bayes (SGVB) algorithm, the approximate inference can be performed efficiently by maximizing the variational lower bound $\mathcal{L}$ via the reparameterization trick. In this framework, the approximate posterior $q(\mathbf{x}|\mathbf{s})$ (which is usually parameterized via neural networks) has to meet two requirements: 1) It should be easy (efficient) to obtain samples from. 2) It should allow for reparameterization trick [88]. In practice, usually a distribution from 'location-scale' family and particularly the Gaussian distribution is used to model the approximate posterior. The SGVB algorithm enables us to tackle the noisy ICA problem as stated in (7.1), much more efficiently than the classical variational mean-filed techniques. In the following we show how to adopt the VAE framework for the task of identifying independent components from a noisy nonlinear mixture.

7.2.1 Encoder

In the VAE framework, the prior distribution is usually assumed to be a parameter-less standard Normal distribution and the posterior is approximated with a multivariate Gaussian distribution. However, since in our model, the prior can be an arbitrary distribution (approximated with mixtures of Gaussian), we need the posterior approximation $q(\mathbf{s}|\mathbf{x})$ to be very close to the actual posterior $p(\mathbf{s}|\mathbf{x})$. One way to accomplish this, is to use the framework of normalizing flows (e.g. Normalizing Flows[89], Inverse autoregressive flows [90]) which provides a general strategy for flexible variational inference of posteriors over latent variables. In these approaches, a simple base distribution is modified using an easily invertible transformation. In our experiments though, this method did not yield any noticeable improvements and in many cases degraded the performance. Alternatively, we use a mixture of Gaussian latent variable scheme which is proved to be a viable solution for the

proposed ICA framework. In particular, by introducing the latent variable y, we define the variational model such that:

$$q(y, \mathbf{s} \,|\, \mathbf{x}) = q(y \,|\, \mathbf{x}) \, q(\mathbf{s} \,|\, y, \mathbf{x}) \tag{7.5a}$$

$$q_\phi(y \,|\, \mathbf{x}) = Cat(y | \pi_\phi(\mathbf{x})) \tag{7.5b}$$

$$q_\phi(\mathbf{s} \,|\, \mathbf{x}, y) = \mathcal{N}\big(\mathbf{s} \,\big|\, \mu_\phi(y, \mathbf{x}), diag(\mathbf{\Sigma}_\phi(y, \mathbf{x})\big), \tag{7.5c}$$

where ϕ indicates the set of all the variational parameters. Similarly, θ indicates the set of all generative parameters. Further, Cat is short for categorical distribution. It is worth nothing that we incorporate the latent variable y only for the variational approximation model and for the generative model we assume $\mathbf{x}$ only depends on $\mathbf{s}$ and $\mathbf{e}$ as we will see next.

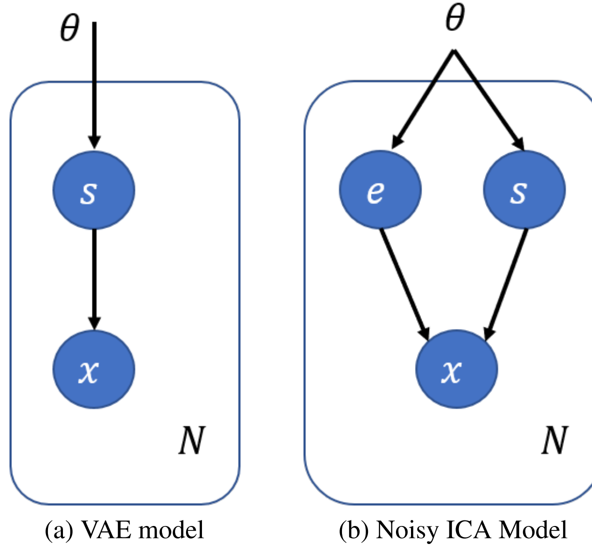
7.2.2 Decoder

In a typical VAE framework, the generative model is usually defined as $p(\mathbf{x}) = p_\theta(\mathbf{s}) p_\theta(\mathbf{x}|\mathbf{s})$ (see Fig. 7.1a). In this framework, $p_\theta(\mathbf{x}|\mathbf{s})$ (usually referred to as the decoder) is assumed to be a function of the latent variable. For example, in cases where $\mathbf{x}$ is continuous data, $p_\theta(\mathbf{x}|\mathbf{s})$ is modeled via a multivariate Gaussian distribution whose covariance matrix and mean vector are neural functions of the latent variable. However, the generative model used in our proposed noisy ICA (see Fig.7.1b) is different and we assume the observation vector $\mathbf{x}$ is generated by two independent random processes $\mathbf{s}$ and $\mathbf{e}$. Further, we have:

$$Pr(\mathbf{x} = \mathbf{x}^{(i)} | \mathbf{s} = \mathbf{s}^{(i)}) = Pr(\mathbf{x} = \mathcal{F}(\mathbf{s}^{(i)}) + \mathbf{e}^{(i)} | \mathbf{s} = \mathbf{s}^{(i)}) \tag{7.6a}$$

$$= Pr(\mathbf{e} = \mathbf{x}^{(i)} - \mathcal{F}(\mathbf{s}^{(i)})), \tag{7.6b}$$

where $\mathcal{F}$ is the nonlinear mapping in (7.1). In the VAE framework, this can be interpreted as learning a posterior distribution whose mean is a function of latent variable. However, other parameters (for example the covariance matrix in the case of multivariate Gaussian



(a) VAE model (b) Noisy ICA Model

Figure 7.1: Graphical Models used in VAE vs Noisy ICA

distribution) are trainable but are independent of the latent variables, i.e.

$$p_\theta(\mathbf{x}|\mathbf{s}) = \mathcal{N}(\mathbf{x}|\mu = \mathcal{F}(\mathbf{s}), \mathbf{\Sigma} = \Sigma_{e_\theta}),$$

where Σ_{e_θ} is the covariance of additive Gaussian noise. The above feature is a very important distinction of our model compared to the traditional VAE. Indeed, as we will show in the experiments, if we do not restrict the covariance Σ_{e_θ} to be fixed for all the data points and let it be a function of latent variables (as in VAE), the model completely fails to learn independent components.

7.2.3 Learning Algorithm

Given the proposed probabilistic framework, the variational lower-bound of the model can be expressed as:

$$\mathcal{L}(\mathbf{x}, y) = -\mathbb{E}_{q_\phi(y,\mathbf{s}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{s}) + \log p_\theta(\mathbf{s}) - \log q_\phi(y, \mathbf{s}|\mathbf{x})] \tag{7.7a}$$

$$\mathcal{L}(\mathbf{x}) = -\sum_y q_\phi(y|\mathbf{x})\big[\mathcal{L}(\mathbf{x}, y)\big] \tag{7.7b}$$

Even though we thus far have not explicitly enforced the independence among latent components, our experiments show that simply by minimizing the variational objective $\mathcal{L}(\mathbf{x})$, in most cases our model is able to find the independent components in a variety of tasks. However, in some cases, specially in the presence of added noise, the model may fail to converge. Further, in some cases, the model converges to a good result but at some point it diverges from the desired objective even though the loss function is still decreasing. As suggested by [47], we obtain approximate samples from the product of marginals, i.e. $\prod_{i=1}^{M} p(s_i)$, by uniformly sampling from the model's estimate of the s (maximum likelihood estimate). In this method, assuming we obtain a sample $(s_1^{(i)}, \ldots, s_M^{(i)})$ from the joint distribution of the posterior, i.e., $p(s_1, \ldots, s_M)|\mathbf{x}^{(i)})$, an approximate sample from the marginal distribution $p(s_i)$ can be obtained by discarding all other variables from the joint sample. In summary, this resampling process involves:

- Let $\mathbf{S} \in \Re^{K \times M}$ be K samples from the joint distribution where $K > M$.
- Let $\mathbf{u}$ be a vector of M indices uniformly sampled from $\{1, \ldots, K\}$.
- Construct the resampled $\hat{\mathbf{s}}$ such that $\hat{s}_i = \mathbf{S}_{u_i,i}$.

To ensure the independence, we need to have $\prod_{i=1}^{M} p(s_i) = p(s_1, \ldots, s_M)$. Although, all the samples and the resampled versions are obtained from a conditional distribution, if we select a very large set of samples (in our experiments we use all available samples from the training set), the samples can be approximately considered to be generated from the prior distribution $p_\theta(\mathbf{s})$. As such, we define an independence objective $\mathcal{L}_{ind}$ as:

$$\mathcal{L}_{ind} = \frac{1}{N} \left| \sum_{i=1}^{N} \log p_\theta(\mathbf{s} = \mathbf{s}^{(i)}) - \sum_{i=1}^{N} \log p_\theta(\mathbf{s} = \hat{\mathbf{s}}^{(i)}) \right| \tag{7.8}$$

We cannot combine this objective with the variational objective since the independence criterion requires a much larger batch size. Further, we can optimize this objective separately and less frequently than the variational objective and by excluding the parameters of the

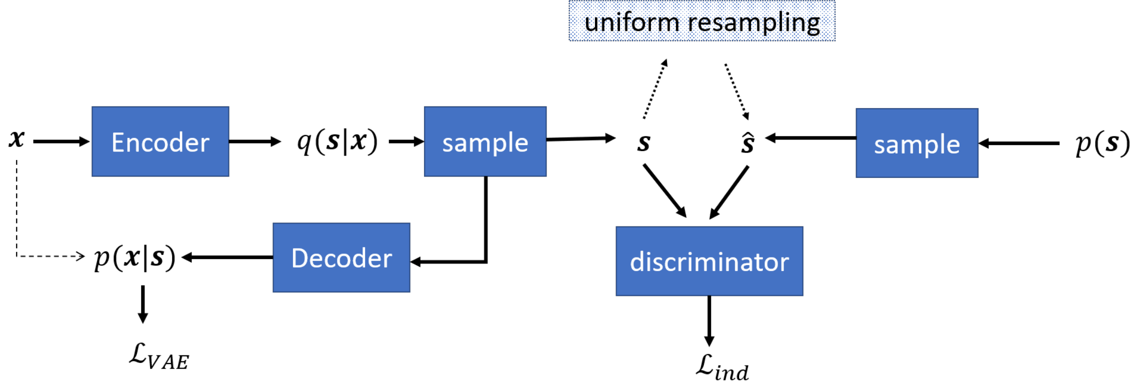


Figure 7.2: Block diagram of the noisy ICA model

prior distribution from the gradient. Algorithm 2 summarizes the proposed method for learning the nonlinear noisy ICA.

Alternative, we can use an adversarial objective similar to Anica algorithm. In our experiments both approaches resulted in similar performance. Fig. 7.2 shows the overall diagram of the proposed algorithm.

7.3 Experiments

We have implemented[1] the proposed noisy ICA model using the Python and Tensorflow [91] library. In all the experiments, we set the number of components for the source mixtures to 10 and for the posterior mixture we use 6 components. In most of the experiments, we

[1]The source code for the experiments is available at `https://github.com/apayani/NonLinearNoisyICA`

Algorithm 2: Noisy nonlinear ICA (VAE-nICA)

Data: $\mathbf{X} \in \Re^{N \times M}$
Result: $\mathrm{S} \in \Re^{N \times M}$
1 **for** $iter = 1$ *to* $iter_{max}$ **do**
2 Sample small batch $\mathbf{X}_{small}$ from X
3 update ϕ and θ to minimize $\mathcal{L}$
4 Sample large batch $\mathbf{X}_{large}$ from X
5 obtain S as the mean of posterior
6 Obtain $\hat{\mathrm{S}}$ by uniform and independent resampling from S
7 update $\theta \setminus \theta_{prior}$ to minimize $\mathcal{L}_{ind}$

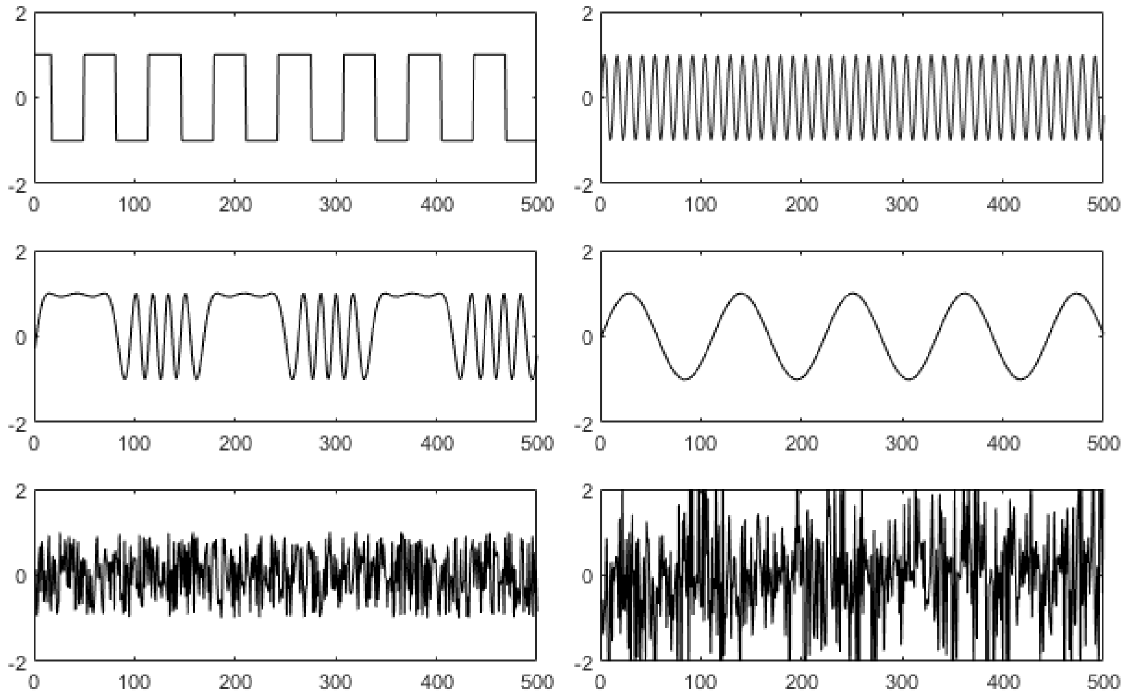


Figure 7.3: Six synthetic sources used in our experiments

found out that changing the number of components has little effect on the overall score but using more components usually increases the speed of convergence. We trained the model using ADAM [59] optimizer and we chose different learning rates for the two objectives; 0.001 for the $\mathcal{L}$ and 0.0001 for the $\mathcal{L}_{ind}$. In addition to the independence objective (7.8 we also implemented a few similar objectives as well as the GAN objective used in [47]. In most cases, the proposed variational noisy ICA approach (VAE-nICA) could read an optimal solution even without the use of any independence objective. However, in cases where we were dealing with noisy measurements, the independence objectives were beneficial. We use batch size of 500 samples and we perform the experiments using the same synthetic data that was used in [47]. This dataset contains 6 independent sources as shown in Fig. 7.3. We use 3 different models for creating the mixtures:

- **Linear Mixture**: we form a linear mixing matrix by randomly drawing its elements from $[-5, .5]$) and we apply a linear transformation.

- **Post Non-Linear (PNL) Mixtures**: this mixture is formed by applying a hyperbolic

tangent activation function to the output of a linear mixture.

- **Multi-Layer Nonlinear (MLP) Mixtures**: formed by cascading multiple (two) PNL mixtures.

Finally, similar to the [47], we report the maximum correlation score to evaluate the performance of each ICA algorithm.

7.3.1 VAE-nICA vs VAE

As we stated in previous section, imposing the noisy ICA model would lead to a restriction in formation of the decoder. In particular, for the conditional Gaussian encoder, unlike the VAE, we have to learn the parameters of the covariance matrix, independent of the latent variables. To demonstrate the importance of this simple distinction, we modify our model to allow for learning the covariance matrix of the $p(\mathbf{x}|\mathbf{s})$ as a function of s similar to the VAE model. In Fig. 7.4, we have compared the performance of the VAE-nICA to the VAE model. The VAE-nICA easily converges to the correct solution. In contrast, the VAE model fails to converge for this simple linear mixing problem.

7.3.2 Performance

Table 7.1, compares the performance of our proposed method to Anica as well as FastICA for the three different mixture tasks. In all cases the proposed VAE-nICA outperforms the other two.

7.3.3 Robustness to Noise

Fig. 7.5 shows the performance scores of Anica and VAE-nICA over training data. In this experiment, we evaluated the performance of each model for the linear separation problem in two cases; 1) without noise 2) with a white Gaussian noise ($\sigma = 0.05$). Although we have not shown all the epochs, for the noiseless case, Anica reaches the optimal solution after many more epochs. However, for the noisy case, it starts to learn but at some point

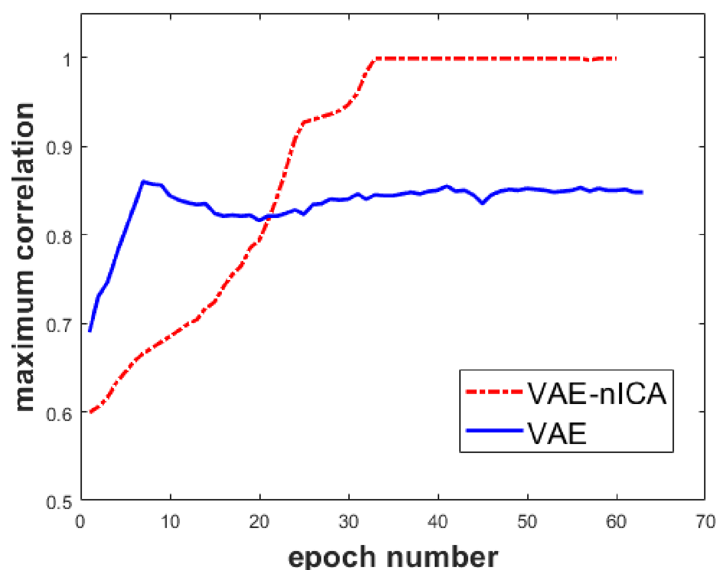


Figure 7.4: Comparing the performance of standard VAE vs VAE-nICA

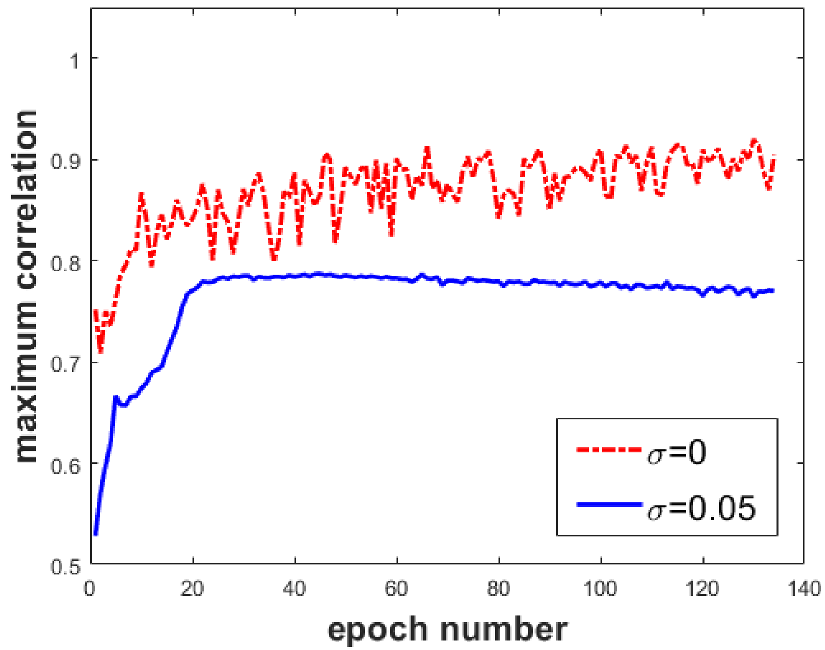
Table 7.1: Maximum correlation results for the 3 different types of mixings; Linear, PNL and MLP.

Method	Linear	PNL	MLP
Anica	0.9990	0.9793	0.9673
FastICA	0.9998	0.8327	0.9173
VAE-nICA	**0.9999**	**0.9869**	**0.9834**

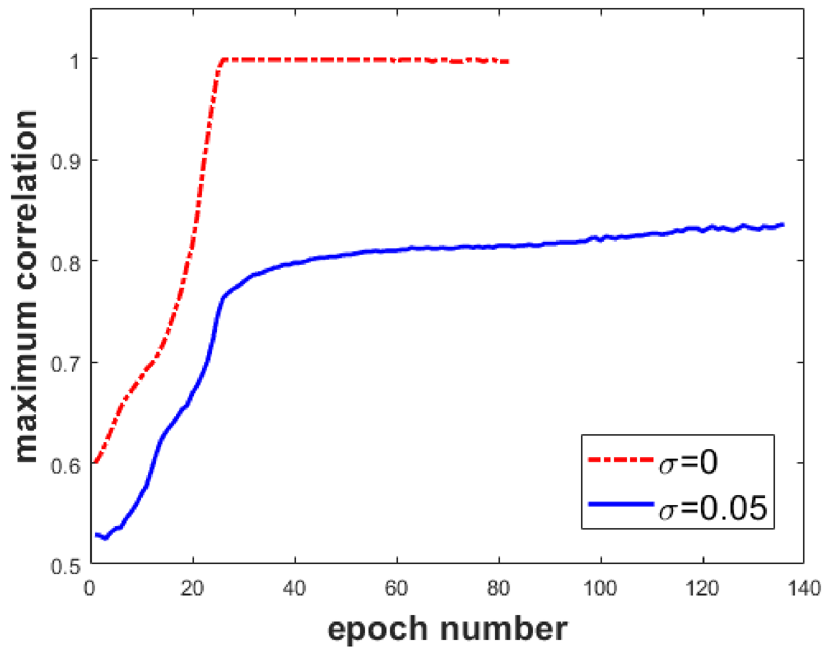
the model diverges. In contrast, VAE-nICA reaches the optimal solution much faster in the noiseless case. Further, for the noisy case, VAE-nICA outperforms the Anica. Table 7.2, compares the overall best scores obtained by the two methods for the 3 different levels of noise.

Table 7.2: Maximum correlation results for the linear mixing problem in the presence of white Gaussian noise with three different σ_e.

Method	$\sigma_e^2 = 0.0$	$\sigma_e^2 = 0.03$	$\sigma_e^2 = 0.05$
Anica	0.99	0.82	0.78
VAE-nICA	0.99	**0.91**	**0.85**



(a) Anica



(b) VAE-nICA

Figure 7.5: Comparing the performance of VAE-nICA vs Anica in the presence of noise

CHAPTER 8
CONCLUSION

8.1 Summary of Achievements

In this dissertation a novel paradigm for learning algorithmic and discrete tasks via Deep Neural Networks (DNN) was introduced. The aim of this paradigm is to bridge the gap between the standard DNNs and the explanatory AI disciplines such as logic programming. While DNNs offer impressive performance, their learning is usually implicit and is hard to decipher. On the other hand, logic programming provides easy to interpret hypothesises but has a limited scope and cannot be easily applied to the more complex domains such as images and other multidimensional data. At the heart of this proposed paradigm is a new neural network model for learning and representing Boolean functions in an explicit manner. Unlike traditional multi-layer neural networks, the differentiable Neural Logic network (dNL) was specifically designed for learning Boolean functions. In this dissertation, the applications of dNL in various domains, including inductive logic programming and relational reinforcement learning, as well as discrete algorithmic tasks such as decoding LDPC codes over binary erasure channels was studied. In the following, the contributions of the thesis are summarized.

In Chapter 2, we introduced the basic design of the differentiable Neural Logic (dNL) networks. In particular, we demonstrated how by introducing the concept of membership weights and by using a minimal number of training weights, the logical conjunction and disjunction functions can be implemented as multiplicative neurons. These novel neurons can be stacked and arranged in a multi-layer design, similar to the perceptron neurons to learn and represent the Boolean DNF (Disjunctive Normal Form) and CNF (Conjunctive Normal Form) functions. We further introduced a novel multiplicative neuron for learning and representing the exclusive-OR logical function and we proved the correctness of its

mathematical model. Further, via experiments such as learning arbitrary Boolean DNF functions, XOR logic, binary arithmetic and Grammar verification, we showed its superior performance to the standard neural networks in certain Boolean and discrete algorithmic tasks.

In Chapter 3, we focused on the explanatory aspect of the proposed dNL networks. In particular, we demonstrated that by exploiting the explicit representation of Boolean functions in dNL, we can formulate the ILP as a satisfiability problem. We showed that by using the fuzzy relaxation of this satisfiability problem, we can translate ILP into a differentiable neural networks which can be optimized via standard gradient based learning techniques. It is well known that the need for using program templates to generate a limited set of viable candidate clauses in forming the predicates is the key weakness in all the past ILP systems, severely limiting the solution space of a problem. In this chapter, we showed that unlike past solvers, the proposed differentiable neural ILP solver (dNL-ILP) can learn arbitrary complex predicates without the need for specifying any rule template. Using various experiments we showed that the proposed dNL-ILP outperforms past algorithms for learning algorithmic and recursive predicates. In particular, we demonstrated that the flexibility of dNL-ILP solvers makes it possible to learn very complex recursive formula for algorithmic tasks such as sorting.

In Chapter 4 we examined the application of dNL-ILP in learning from uncertain and ambiguous data. We demonstrated that unlike past neural solvers such as dILP [27], dNL-ILP is scalable to handle large scale problems and is capable of learning from uncertain and ambiguous relational data. In particular, we showed that dNL-ILP outperforms the state of the art ILP solvers in classification tasks for relational datasets including benchmark tasks such as Mutagenesis, Cora and IMDB. Moreover, via experiments we showed that the differentiable aspect of dNL-ILP allows for combining standard neural network with dNL-ILP to handle continuous data. We provided two approaches to handle continuous data. In particular, we showed that we can learn the decision boundaries using an end-to-end

framework to translate continuous data into a series of binary predicates. Alternatively, we showed that using the dNL-ILP framework we can learn the distribution of continuous data and we successfully applied this concept in the DREAM4 gene regulatory network experiment.

Unlike past ILP solvers, the dNL-ILP is a differentiable neural network that can be combined with other neural network layers and can be used in an end-to-end neural network design. In Chapter 5, we proposed a novel deep Relational Reinforcement Learning (RRL) model based on the differentiable dNL-ILP that can effectively learn relational information from image. We showed how this model can take the expert background knowledge and incorporate it into the learning problem using appropriate predicates. The differentiable ILP allows for an end-to-end optimization of the entire framework for learning the policy in RRL. We examined the performance of the proposed RRL framework using environments such as BoxWorld and GridWorld. In particular, we demonstrated that by employing this novel approach, we can solve some problems involving very complex reasoning orders of magnitude faster than the state of the art reinforcement learning techniques such as A2C.

In Chapter 6, we studied the generalization performance of the dNL by considering their application in decoding LDPC codes over binary erasure channels. In this chapter, we formulated a general message passing algorithm and we employed functions consisting of the elements from dNL as building blocks for transmitting messages between variable nodes and check nodes. We demonstrated that the design via dNL leads to superior generalization compared to the DNNs counterpart. We further exploited the differentiable exclusive-OR function of dNL (i.e., dNL-XOR) to directly solve the parity check equations via transforming it into a differentiable set of equations. The consequent equations are then solved via standard gradient techniques.

Finally, in Chapter 7, we studied the application of variational autoencoders in learning the Independent Component Analysis (ICA). ICA is one of the most basic and successful classical tools in unsupervised learning which has many applications in various fields.

While for the linear case there is an extensive body of research and a variety of successful algorithms exist, there are very few effective algorithms that can tackle the problem for the general nonlinear case in presence of noise. In this chapter, we formulated the general non-linear ICA with the additive Gaussian noise via employing a variational autoencoder framework. The properties of the new proposed ICA framework was analyzed. In particular, we demonstrated that it has superior performance and is more robust to the noise compared the state of the art algorithms such as Anica [47].

8.2 Future Research Directions

8.2.1 Incorporating Human Preference and Safe AI

Traditional RL depends on a well defined reward function that fully describes the objective of a problem. This approach has been successfully applied in various game scenarios (e.g. Atari games) where the reward is simply defined by the success at the end of an episode (game) or by the scores gained during the game. Unfortunately, many tasks involve complex goals for which it is difficult to define a reward function. This difficulty has raised concerns about misalignment between our values (preferences) and the objectives of the RL systems [92, 93, 94]. A viable but costly approaches have been proposed by [95, 96, 97, 98, 99]; using feedbacks from human overseers to define/modify the reward function. Apart from the cost, these approaches have various limitations. For example, they do not necessarily cover all the cases of preferences since rare phenomena may not be discovered during most of the training episodes. Moreover, in the context of safe AI, the above approaches do not prevent the agent from doing something harmful during the training or after. In other words, an intermediately learned policy executed by a robot (or a self-driving car) may break the system or harm the environment. Another example is Microsoft's chatbot which reproduced thousands of offensive tweets before being taken down. In general, the goal of safe reinforcement learning is to maximize system performance while minimizing number of safety constraint violations during the learning and deployment. Safe RL has been studied

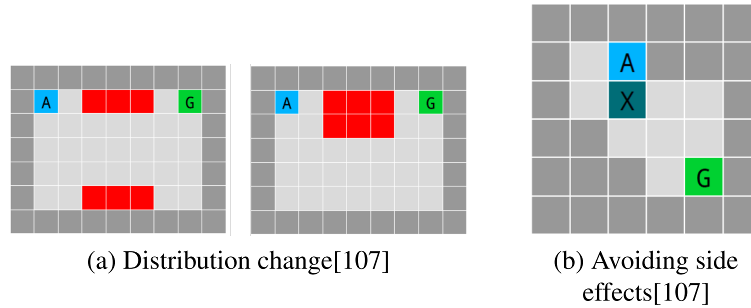
in the past [100, 101, 102, 103, 104, 105, 106] primarily using two types of approaches:

- Modifying the optimization criterion with a safety constraint.
- Incorporating external safety knowledge to avoid dangerous states in exploration strategies.

However, these solutions are inadequate and are applicable to the systems with very simple dynamics.

Given the expressiveness and explicit form of the relational language, our ILP based RRL provides a natural platform for learning from human preferences. To clarify the above concept, consider an abstract example where the human preference is defined as following. The agent must take Action 1 if the current state is 1. Further, if the current state is 2, the agent must take action 2. For all other states, the agent needs to learn the correct actions according to the desired goal. To solve the above learning problem, traditional approaches would need supplementary overseers feedbacks to enforce for human preferences in states 1 and 2, which is not only costly for implementation (in general learning problems) but also possibly lacking some of the human preferred cases (e.g., the less frequent cases). In contrast, in our ILP based RRL framework, we can use a relational program such as 'Action1 $\leftarrow$ (state=1) or (learnedAction=1)', and also 'Action2 $\leftarrow$ (state=2) or (learnedAction=2)'; where the `learnedAction` is the target predicate that our ILP program learns. As such, the above clauses will enforce the agent to take action 1 or action 2 when the current detected states are 1 or 2, respectively. Otherwise (i.e., in other states), the agent will act according to the learned predicate `learnedAction`. This abstract example demonstrates how by using the explicit representation of an state, we could coerce the agent into learning specific type of actions instead of indirectly learning such preferences from the ambiguous reward functions.

In Chapter 5, we explained that the proposed RRL framework using dNL-ILP solver is capable of incorporating biases through the use of language. As such, we will investigate



(a) Distribution change[107]

(b) Avoiding side effects[107]

Figure 8.1: Grid World environments for safe AI

the application of the proposed framework in designing safe AI and also incorporating human preferences through the use of predicate logic. We will explore various ways of avoiding certain states, or taking particular actions based on the current state of the system through designing appropriate predicate language to account for such circumstances. For the evaluation, we may use the basic framework of the Grid World environment. As in [107], we can evaluate several problems related to safe-AI and human preferences by tweaking this simple environment and defining different objectives. Fig. 8.1 shows two environments designed for evaluating the impacts of distribution change and avoiding side effects. In Fig. 8.1a, the agent (i.e., denoted by A) is tasked to reach the goal (G) while avoiding obstacles (red lava). In this task, the distribution of the lava on the grid in testing is different from the training. The agent will be successful in testing only if it could learn the concept of hazardous lava as blocks for red colored boxes instead of just relying on avoiding certain locations based on the distribution of lava in the training data. Fig. 8.1b shows an environment that evaluates the agent's ability to avoid side effects. Here, the side effect is quantified by relocation of the box X while agent A is approaching its goal G. The agent may choose the shortest path by pushing the box X down. However, by choosing this path, the box X cannot be pushed back to its original location. Agent needs to push the box X to the right in order to push it back to the original position before heading to the goal. We will investigate these tasks and other similarly defined tasks to investigate our solutions to safe AI and incorporating human preferences via the proposed RRL framework.

Appendices

APPENDIX A

PROOF OF THEOREM 1

Proof. First consider that for the case where the number of $'1's$ in $\mathbf{x}$ is odd (i.e. $XOR(x) = 1$), none of the functions $f_i(\mathbf{x})$ can be equal to zero since the sum of odd number of elements from the set $\{-1, 1\}$ cannot be zero. Therefore, the statement in (2.7a) is true due to (2.7b). Now consider the case that $XOR(\mathbf{x})$ is zero. We must show that at least one of the k functions $f_i(\mathbf{x})$ would be equal to zero in this case. Let $M_i \in \{-1, 1\}^n$ be the vector of coefficients for $f_i(\mathbf{x})$ and s be the number of ones in the input vector $\mathbf{x}$. Further, for any f_i, let $n_i^{(1)}$ and $n_i^{(-1)}$ be the number of corresponding 1 and -1 coefficients that matches the positions of elements of '1' in vector $\mathbf{x}$. We notice that the sign of exactly two elements in M_i and M_{i+1} changes when we go from $f_i(\mathbf{x})$ to $f_{i+1}(\mathbf{x})$ and those signs remain unchanged in the next set of functions. As we have k functions and $s \leq 2k$, this would guarantee that the sign of the coefficients corresponding to '1' elements changes exactly once in the set of k functions. Thus, in one of the functions, let's say the one corresponding to the jth coefficient vector we would have $n_j^{(1)} = n_1^{(-1)}$ and $n_j^{(-1)} = n_1^{(1)}$ which means $f_j(\mathbf{x}) = -f_1(\mathbf{x})$. Since the difference between each consecutive f_i can be zero or ± 2, this guarantees that at some point one of the f_i's $(1 \leq i \leq j)$ should be equal to zero.

In the above arguments we assumed n is an even number. However, if n is an odd number, we can modify it to the $n + 1$ problem by appending an extra '0' entry to the input vector $\mathbf{x}$. Since '0' has no effect on the results of XOR, the above arguments still hold. □

APPENDIX B

BOXWORLD EXPERIMENT DETAILS

For the problem consists of n box, we need $n+1$ constants of type `box` (note that we consider the floor as one of the boxes in our problem definition). Additionally, we define numerical constants $\{0, \ldots, n\}$ to represent box coordinates using in posH and posV predicates. For the numerical constants we define orderings via extensional predicates `lt`/2 and `inc`/2. For the box constants, we define two extensional predicates `same/2` and `isBlue/1`. Here, by `p`/N we mean predicate `p` of arity N (i.e., `p` has N arguments). Since these are extensional predicates, their truth values are fixed in the beginning of the program via the background facts. For example, for predicate `inc`/2 which defines the increment by one for the natural numbers, we need to set these background facts the beginning: {`inc(0,1)`, `inc(1,2),...,inc(n-1,n)`}. Similarly, for the predicate `lt` (short for lessThan) this set includes items such as {`lt(0,1), lt(0,2),...,lt(n-1,n)`}. It is worth noting that introducing predicates such as `isBlue` for boxes does not mean we already know which box is the blue one (the target box that needs to be first box in the stack). This predicate merely provides a way for the system to distinguish between boxes. Since in our learned predicates we can only use symbolic atoms (predicates with variables) and in dNL-ILP implementation, no constant is allowed in forming the hypothesis, this method allows for referring to an specific constant in the learned action predicates. Here, for example, we make an assumption that box `a` in our list of constants corresponds to the blue box via the background fact `isBlue(a)`. Table B.1 explains all extensional predicates as well as those helper auxiliary predicates that was used in our BoxWorld program.

So far in our discussions, we have not distinguished between the type of variables. However, in the dNL-ILP implementation, the type of each variable should be specified in the signature of each defined predicate. This allows the dNL-ILP to use only valid combination

of variables to generate the symbolic formulas. In the BoxWorld experiment, we have two types of constants T_b and T_p referring to the box and numeric constants, respectively. In TableB.1, for each defined predicate, the list of variables and their corresponding types are given, where for example X(T_p) states that the type of variable X is T_p.

For clarity, Table B.1 is divided into two sections. In the top section the constants are defined. In the other section, 4 groups of predicates are presented: (i) state representation predicates that their groundings are learned from image, (ii) extensional predicates which are solely defined by background facts, (iii) auxiliary predicates and, (iv) the signature for the target predicate that is used to represent the actions in the policy gradient scheme. To learn the policy, we used discount factor of 0.7, and we use ADAM optimizer with learning rate of 0.002. We set the maximum number of steps for each episode to 20. To learn the feature map, we used two layers of CNNs with kernel size of 5 and strides of 3 and we applied fully connected layers with `softmax` activation functions to learn the groundings of the state representation predicates `posH` and `posV`.

Table B.1: ILP definition of the BoxWorld

Constants	Description	Values
T_b	box constants	{a, b, c, d, floor}
T_p	coordinate position constants	{0, 1, 2, ..., n}

Predicate	Variables	Definition
posH(X,Y)	X(T_b), Y(T_p)	learned from Image
posV(X,Y)	X(T_b), Y(T_p)	learned from Image
isFloor(X)	X(T_b)	isFloor(floor)
isBlue(X)	X(T_b)	isBlue(a)
isV1(X)	X(T_p)	isV1(1)
inc(X,Y)	X(T_p), Y(T_p)	inc(0,1), inc(1,2), ..., inc(n-1,n)
lt(X,Y)	X(T_p), Y(T_p)	lt(0,1), lt(0,2), ..., lt(n-1,n)
same(X,Y)	X(T_b), Y(T_b)	same(a,a), same(b,b), ..., same(floor,floor)
sameH(X,Y)	X(T_b), Y(T_b), Z(T_p)	posH(X,Z), posH(Y,Z)
sameV(X,Y)	X(T_b), Y(T_b), Z(T_p)	posV(X,Z), posV(Y,Z)
above(X,Y)	X(T_b), Y(T_b), Z(T_p), T(T_p)	sameH(X,Y), posV(X,Z), posV(Y,T), lt(T,Z)
below(X,Y)	X(T_b), Y(T_b), Z(T_p), T(T_p)	sameH(X,Y), posV(X,Z), posV(Y,T), lt(Z,T)
on(X,Y)	X(T_b), Y(T_b), Z(T_p),T(T_p)	sameH(X,Y), posV(X,Z), posV(Y,T), inc(T,Z)
isCovered(X)	X(T_b), Y(T_b)	On(Y,X), ¬isFloor(X)
moveable(X,Y)	X(T_b), Y(T_b)	¬isCovered(X), ¬isCovered(Y), ¬same(A,B), ¬isfloor(X), ¬on(X,Y)
move(X,Y)	X(T_b), Y(T_b), Z(T_p),T(T_p)	Action predicate that is learned via policy gradient

APPENDIX C
GRIDWORLD EXPERIMENT DETAILS

In the GridWorld experiment, we distinguish between constants used for vertical and horizontal coordinates as shown Table C.1. This reduces the number of possible symbolic atoms and makes the convergence faster. We also define 10 constants of type T_c to represent each of the 10 possible colors for the grid cells in the scene. The complete list of all the extensional and auxiliary predicates for this program is listed in Table C.1. In this task, to incorporate our knowledge about the problem, we define the concept of `item` via predicate `isItem(X,Y)`. Using this definitionm, an `item` represents any of the cells that are neither background nor agent. By incorporating this concept, we can further define higher level concepts via `locked` and `isLock` predicates which define the two adjacent items as a locked item and its corresponding key, respectively. The dimension of the observation image is 112x112x3. We consider two types of networks to extract grid colors from the image. In the first method, we apply three layers of convolutional network with kernel sizes of $[5, 3, 3]$ and strides of 2 for all. We use `relu` activation function apply batch normalization after each layer. The number of features in each layer are set to 24. We take the feature map of size $14 \times 14 \times 24$ and apply two layers MLP with dimensions of 64 and 10 (10 is the number of color constants) and activation functions of `relu` and `softmax`, to generate the grounding matrix G (the dimension of G is 14x14x10). As can be seen in Fig. C.1, the current key is located at the top left border. We extract the grounding for predicate `hasKey` from the value of $G[0, 0, :]$ and the groundings of the predicate `color` by discarding the border elements from G (i.e., $G[1..12, 1..12, :]$).

Alternatively, and because of the simplicity of the environment, instead of using CNNs, we may take the center of each box to directly create the feature map of size $14 \times 14 \times 3$ and then apply the MLP to generate the groundings. In our experiments we tested both

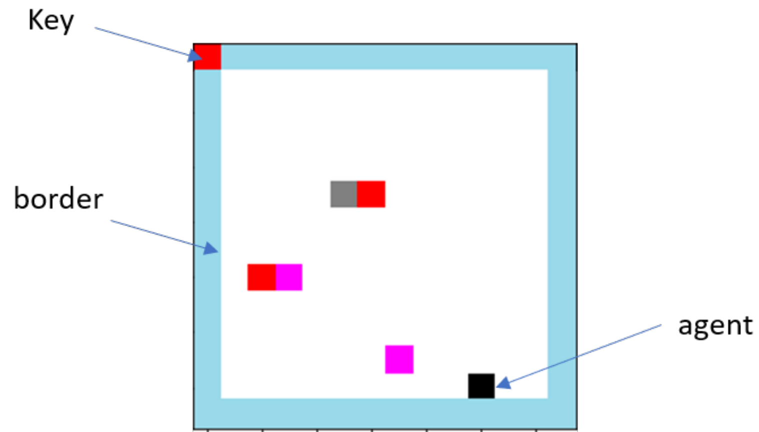


Figure C.1: An example GridWorld scene

scenarios. Using the CNN approach the speed of convergence in our method was around 2 times slower. For the A2C algorithm, we could not solve the problem using either of these approaches. We set the maximum number of steps in an episode to 50 and we use learning rate of .001. For our models, we use discount factor of 0.9 and for the A2C we tested various numbers in range of 0.9 to 0.99 to find the best solution.

Table C.1: ILP definition of the GridWorld

Constants	Description	Values
T_v	vertical coordinates	$\{0,1,2,\ldots,11\}$
T_h	horizontal coordinates	$\{0,1,2,\ldots,11\}$
T_c	cell color code	$\{0,1,2,\ldots,9\}$

Predicate	Variables	Definition
color(X,Y,Z)	X(T_v), Y(T_h), Z(T_c)	learned from Image
hasKey(X)	X(T_c)	Learned from Image
incH(X,Y)	X(T_h)	incH(0,1),...,incH(10,11)
isC0(X)	X(T_c)	isC0(0)
isC1(X)	X(T_c)	isC1(1)
isC2(X)	X(T_c)	isC2(2)
isBK(X,Y)	X(T_v), X(T_h), Z(T_c)	color(X,Y,Z), isC0(Z)
isAgent(X,Y)	X(T_v), X(T_h), Z(T_c)	color(X,Y,Z), isC1(Z)
isGem(X,Y)	X(T_v), X(T_h), Z(T_c)	color(X,Y,Z), isC2(Z)
isItem(X,Y)	X(T_v), X(T_h)	¬ isBK(X,Y), ¬ isAgent(X,Y),
locked(X)	X(T_v), X(T_h), Z(T_h)	isItem(X,Y), isItem(X,Z), incH(Y,Z)
isLock(X)	X(T_v), X(T_h), Z(T_h)	isItem(X,Y), isItem(X,Z), incH(Z,Y)
move(X,Y)	X(T_v), Y(T_h), Z(T_c)	Action predicate that is learned via policy gradient

APPENDIX D

RELATIONAL REASONING EXPERIMENT DETAILS

This task was introduced as a benchmark for learning relational information from images in [41]. In this task, the objects of two types (circle or rectangle) and in 6 different colors are randomly placed in a 4x4 grid and questions like *'is the blue object circle'* or *'what is the color of nearest object to blue'* should be answered. To formulate this problem via ILP, instead of using grid coordinates (vertical and horizontal positions as in past experiments) we consider a flat index as can be seen in Table D.1. As such, we will have 16 positional constants of type T_p to represent grid positions as well as 6 constants of type T_c to represent the color of each object. In the original problem in [41], a question is represented as binary vector of length 11. The first 6 bits represent the one-hot representation of the color of the object in question. The next 5 bits represent one of the five possible types of questions, i.e., *'is it a circle or a rectangle'* and *'the color of the nearest object?'* for example.

Table D.1: Flat index for a grid of 4 by 4 used in relational learning task

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

We define a nullary predicate for each of 11 possible bits in the question via predicates `isQ0()`,...,`isQ10()`. Similar to to the original paper we use 4 layers of CNNs with `relu` activation function and batch normalizations to obtain a feature map of size $4 \times 4 \times 24$. We use kernel size of 5 and strides of [3,3,2,2] for each of the layers. By applying fully connected layers to the feature map, we learn the groundings of predicates `color(X,Y,Z)`, `isObject(X,Y)` and `isCircle(X,Y)`. We define some auxiliary predicates as shown in Table D.2. For each of the 10 possible answers, we create one nullary predicate. The vector of size 10 that is created by the groundings of these 10 predicates (i.e., `isAnswer0()`,...)

are then used to calculate the cross entropy loss between the network output and the provided ground truth. We need to mention that in the definition of the auxiliary predicate `qa(X)`, we exploit our prior knowledge regarding the fact that the first 6 bits of the question vector correspond to the colors of the object. Without introducing this predicate, the convergence of our model is significantly slower. For example, while by incorporating this predicate we need around 30000 training samples for convergence, it would take than 200000 training samples without this predicate.

Table D.2: ILP definition of the relational reasoning task

Constants	Description	Values
T_p	Flat position of items in a 4 by 4 grid	$\{0,1,2,\ldots,15\}$
T_c	color of an item	$\{0,1,2,\ldots,5\}$

Predicate	Variables	Definition
isQ0()		Given as a binary value
...		...
isQ10()		Given as a binary value
color(X,Y)	X(T_p), Y(T_c)	Learned from Image
isCircle(X,Y)	X(T_p)	Learned from Image
isObject(X,Y)	X(T_p)	Learned from Image
equal(X,Y)	X(T_p), Y(T_p)	equal(0,0),...,incH(15,15)
lt(X,Y)	X(T_p), Y(T_p), Z(T_p)	true if distance between grid cells corresponding to X and Y is less than distance between X and Z (see Table D.1)
left(X)	X(T_p)	left(0),left(1),...,left(12),left(13)
right(X)	X(T_p)	right(2),right(3),...,right(14),right(15)
top(X)	X(T_p)	top(0),left(1),...,left(6),left(7)
bottom(X)	X(T_p)	bottom(8),left(9),...,left(14),left(15)
closer(X,Y,Z)	X(T_p), X(T_p), Z(T_p)	isObject(X), isObject(Y), isObject(Z), lt(X,Y,Z)
farther(X,Y,Z)	X(T_p), X(T_p), Z(T_p)	isObject(X), isObject(Y), isObject(Z), gt(X,Y,Z)
notClosest(X,Y)	X(T_p), X(T_p), Z(T_p)	closer(X,Z,Y)
notFarthest(X,Y)	X(T_p), X(T_p), Z(T_p)	farther(X,Z,Y)
qa(X)	X(T_p), Y(T_c)	isQ0(), color(X,Y), isC0(Y) isQ1(), color(X,Y), isC1(Y) isQ2(), color(X,Y), isC2(Y) isQ3(), color(X,Y), isC3(Y) isQ4(), color(X,Y), isC4(Y) isQ5(), color(X,Y), isC5(Y)
isAnswer0()	X(T_p), Y(T_p)	The learned hypothesis : Is answer is 0
...	...	...
isAnswer9()	X(T_p), Y(T_p)	The learned hypothesis : Is answer is 9

APPENDIX E

ASTERIX EXPERIMENT DETAILS

Table E.1: ILP definition of the Asterix experiment

Constants	Description	Values
T_v	vertical coordinates	$\{0,1,2,\ldots,8\}$
T_h	horizontal coordinates	$\{0,1,2,\ldots,12\}$

Predicate	Variables	Definition
O1(X,Y)	X(T_v), Y(T_h),	learned from Image : objects of type agent
O2(X,Y)	X(T_v), Y(T_h),	learned from Image : objects of type L2R predator
O3(X,Y)	X(T_v), Y(T_h),	learned from Image : objects of type R2L predator
O4(X,Y)	X(T_v), Y(T_h),	learned from Image : objects of type food
isV0(X)	X(T_v)	isV(0)
isV11(X)	X(T_v)	isV11(11)
isH0(X)	X(T_h)	isH0(0)
isH7(X)	X(T_h)	isH7(7)
incV(X,Y)	X(T_v), X(T_v)	incV(0,1), ...,incV(6,7)
ltH(X,Y)	X(T_h), X(T_h)	ltH(0,1),ltH(0,2), ...,ltH(11,12)
closeH(X,Y)	X(T_h), Y(T_h)	true if $\lvert X-Y\rvert \leq 2$
agentH(X)	X(T_v), Y(T_h)	O1(X,X)
agentV(X)	X(T_h), Y(T_v)	O1(Y,X)
predator(X,Y)	X(T_v), Y(T_h)	O2(X,X) O3(X,Y)
agent()	X(T_v)	agentV(X)
badMoveUp()	X(T_v), Y(T_h), Z(T_v), T(T_h)	O1(X,Y), predator(Z,T), incV(Z,X), closeH(Y,T)
badMoveDown()	X(T_v), Y(T_h), Z(T_v), T(T_h)	O1(X,Y), predator(Z,T), incV(X,Z), closeH(Y,T)
badMoveLeft()	X(T_v), Y(T_h), Z(T_h)	O1(X,Y), O2(X,Z), ltH(Z,Y), closeH(Z,Y)
badMoveRight()	X(T_v), Y(T_h), Z(T_h)	O1(X,Y), O3(X,Z), ltH(Y,Z), closeH(Z,Y)
moveUp()	X(T_v), Y(T_h), Z(T_v), T(T_h),	Action predicate that is learned via policy gradient
moveDown()	X(T_v), Y(T_h), Z(T_v), T(T_h),	Action predicate that is learned via policy gradient
moveLeft()	X(T_v), Y(T_h), Z(T_v), T(T_h),	Action predicate that is learned via policy gradient
moveRight()	X(T_v), Y(T_h), Z(T_v), T(T_h),	Action predicate that is learned via policy gradient
moveNOOP()	X(T_v), Y(T_h), Z(T_v), T(T_h),	Action predicate that is learned via policy gradient

REFERENCES

[1] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proceedings of the 25th international conference on Machine learning*, ACM, 2008, pp. 160–167.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[3] G. E. Dahl, D. Yu, L. Deng, and A. Acero, "Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition," *IEEE Transactions on audio, speech, and language processing*, vol. 20, no. 1, pp. 30–42, 2012.

[4] A. Graves, G. Wayne, and I. Danihelka, "Neural turing machines," *arXiv preprint arXiv:1410.5401*, 2014.

[5] M. Andrychowicz and K. Kurach, "Learning efficient algorithms with hierarchical attentive memory," *arXiv preprint arXiv:1602.03218*, 2016.

[6] A. Joulin and T. Mikolov, "Inferring algorithmic patterns with stack-augmented recurrent nets," in *Advances in neural information processing systems*, 2015, pp. 190–198.

[7] Ł. Kaiser and I. Sutskever, "Neural gpus learn algorithms," *arXiv preprint arXiv:1511.08228*, 2015.

[8] M. Minsky and S. A. Papert, *Perceptrons: An introduction to computational geometry*. MIT press, 2017.

[9] P. D. Wasserman, *Neural computing: theory and practice*. Van Nostrand Reinhold Co., 1989.

[10] B. Steinbach and R. Kohut, "Neural networks–a model of boolean functions," in *Boolean Problems, Proceedings of the 5th International Workshop on Boolean Problems*, 2002, pp. 223–240.

[11] S. Dzeroski, "Inductive logic programming in a nutshell," *Introduction to Statistical Relational Learning [16]*, 2007.

[12] A. Cropper and S. H. Muggleton, *Metagol system*, https://github.com/metagol/metagol, 2016. [Online]. Available: `https://github.com/metagol/metagol`.

[13] A. Cropper and S. H. Muggleton, "Logical minimisation of meta-rules within meta-interpretive learning," in *Inductive Logic Programming*, Springer, 2015, pp. 62–75.

[14] A. Tamaddoni-Nezhad, D. Bohan, A. Raybould, and S. Muggleton, "Towards machine learning of predictive models from ecological data," in *Inductive Logic Programming*, Springer, 2015, pp. 154–167.

[15] A. Cropper and S. H. Muggleton, "Learning efficient logical robot strategies involving composable objects," in *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[16] S. Hölldobler, Y. Kalinke, and H.-P. Störr, "Approximating the semantics of logic programs by recurrent neural networks," *Applied Intelligence*, vol. 11, no. 1, pp. 45–58, 1999.

[17] S. Bader, A. S. d. Garcez, and P. Hitzler, "Computing first-order logic programs by fibring artificial neural networks.," in *FLAIRS Conference*, 2005, pp. 314–319.

[18] K. Kersting, L. De Raedt, and T. Raiko, "Logical hidden markov models," *Journal of Artificial Intelligence Research*, vol. 25, pp. 425–456, 2006.

[19] L. De Raedt and K. Kersting, "Probabilistic inductive logic programming," in *Probabilistic Inductive Logic Programming*, Springer, 2008, pp. 1–27.

[20] S. Bader, P. Hitzler, and S. Hölldobler, "Connectionist model generation: A first-order approach," *Neurocomputing*, vol. 71, no. 13-15, pp. 2420–2432, 2008.

[21] M. Guillame-Bert, K. Broda, and A. d. Garcez, "First-order logic learning in artificial neural networks," in *The 2010 International Joint Conference on Neural Networks (IJCNN)*, IEEE, 2010, pp. 1–8.

[22] M. V. França, G. Zaverucha, and A. S. d. Garcez, "Fast relational learning using bottom clause propositionalization with artificial neural networks," *Machine learning*, vol. 94, no. 1, pp. 81–104, 2014.

[23] T. Rocktäschel and S. Riedel, "Learning knowledge base inference with neural theorem provers," in *Proceedings of the 5th Workshop on Automated Knowledge Base Construction*, 2016, pp. 45–50.

[24] F. Yang, Z. Yang, and W. W. Cohen, "A differentiable approach to inductive logic programming," 2016.

[25] J. Eisner, E. Goldlust, and N. A. Smith, "Dyna: A declarative language for implementing dynamic programs," in *Proceedings of the ACL 2004 on Interactive poster and demonstration sessions*, Association for Computational Linguistics, 2004, p. 32.

[26] L. Serafini and A. d. Garcez, "Logic tensor networks: Deep learning and logical reasoning from data and knowledge," *arXiv preprint arXiv:1606.04422*, 2016.

[27] R. Evans and E. Grefenstette, "Learning explanatory rules from noisy data," *Journal of Artificial Intelligence Research*, vol. 61, pp. 1–64, 2018.

[28] S. Muggleton, "Inverse entailment and progol," *New generation computing*, vol. 13, no. 3-4, pp. 245–286, 1995.

[29] H. Dong, J. Mao, T. Lin, C. Wang, L. Li, and D. Zhou, "Neural logic machines," 2018.

[30] R. Srikant and R. Agrawal, *Mining generalized association rules*, 1995.

[31] S. Džeroski, "Relational data mining," in *Data Mining and Knowledge Discovery Handbook*, Springer, 2009, pp. 887–911.

[32] M. Richardson and P. Domingos, "Markov logic networks," *Machine learning*, vol. 62, no. 1-2, pp. 107–136, 2006.

[33] A. K. Debnath, R. L. Lopez de Compadre, G. Debnath, A. J. Shusterman, and C. Hansch, "Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. Correlation with molecular orbital energies and hydrophobicity.," *Journal of medicinal chemistry*, vol. 34, no. 2, pp. 786–797, 1991, ISSN: 0022-2623. DOI: 10.1021/jm00106a046.

[34] C. Bryant, S. Muggleton, C. Page, M. Sternberg, *et al.*, "Combining active learning with inductive logic programming to close the loop in machine learning," in *AISB'99 Symposium on AI and Scientific Creativity*, Citeseer, 1999, pp. 59–64.

[35] S. Džeroski, L. De Raedt, and H. Blockeel, "Relational reinforcement learning," in *International Conference on Inductive Logic Programming*, Springer, 1998, pp. 11–22.

[36] S. Džeroski, L. De Raedt, and K. Driessens, "Relational reinforcement learning," *Machine learning*, vol. 43, no. 1-2, pp. 7–52, 2001.

[37] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[38] H. Blockeel and L. De Raedt, "Top-down induction of first-order logical decision trees," *Artificial intelligence*, vol. 101, no. 1-2, pp. 285–297, 1998.

[39] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[40] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[41] A. Santoro, D. Raposo, D. G. Barrett, M. Malinowski, R. Pascanu, P. Battaglia, and T. Lillicrap, "A simple neural network module for relational reasoning," in *Advances in neural information processing systems*, 2017, pp. 4967–4976.

[42] H. Kim, Y. Jiang, R. Rana, S. Kannan, S. Oh, and P. Viswanath, "Communication algorithms via deep learning," *arXiv preprint arXiv:1805.09317*, 2018.

[43] E. Nachmani, Y. Be'ery, and D. Burshtein, "Learning to decode linear codes using deep learning," in *Communication, Control, and Computing (Allerton), 2016 54th Annual Allerton Conference on*, IEEE, 2016, pp. 341–346.

[44] A. Payani and F. Fekri, "Decoding ldpc codes on binary erasure channels using deep recurrent neural-logic layers," in *Turbo Codes and Iterative Information Processing (ISTC), 2018 International Symposium On*, IEEE, 2018.

[45] A. Hyvärinen and E. Oja, "Independent component analysis: Algorithms and applications," *Neural networks*, vol. 13, no. 4-5, pp. 411–430, 2000.

[46] J.-i. Hirayama, A. Hyvärinen, and M. Kawanabe, "Splice: Fully tractable hierarchical extension of ica with pooling," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, JMLR. org, 2017, pp. 1491–1500.

[47] P. Brakel and Y. Bengio, "Learning independent features with adversarial nets for non-linear ica," *arXiv preprint arXiv:1710.05050*, 2017.

[48] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," *arXiv preprint arXiv:1511.06434*, 2015.

[49] T.-W. Lee, "Independent component analysis," in *Independent component analysis*, Springer, 1998, pp. 27–66.

[50] M. Davies, "Identifiability issues in noisy ica," *IEEE Signal processing letters*, vol. 11, no. 5, pp. 470–473, 2004.

[51] G. R. Naik and D. K. Kumar, "An overview of independent component analysis and its applications," *Informatica*, vol. 35, no. 1, 2011.

[52] M. Vosough, "Using mean field approach independent component analysis to fatty acid characterization with overlapped gc–ms signals," *Analytica chimica acta*, vol. 598, no. 2, pp. 219–226, 2007.

[53] P. A. Højen-Sørensen, O. Winther, and L. K. Hansen, "Mean-field approaches to independent component analysis," *Neural Computation*, vol. 14, no. 4, pp. 889–918, 2002.

[54] A. Payani and F. Fekri, "Unsupervised learning of independent components from a noisy and non-linear mixture via variational autoencoders," in *2019 IEEE 20th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*, 2019, pp. 1–5.

[55] A. Payani and F. Fekri, "Learning algorithms via neural logic networks," *submitted to ICML 2019*, 2019.

[56] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer networks," in *Advances in Neural Information Processing Systems*, 2015, pp. 2692–2700.

[57] W. Duch, "K-separability," in *International Conference on Artificial Neural Networks*, Springer, 2006, pp. 188–197.

[58] E. M. Iyoda, H. Nobuhara, and K. Hirota, "A solution for the n-bit parity problem using a single translated multiplicative neuron," *Neural Processing Letters*, vol. 18, no. 3, pp. 233–238, 2003.

[59] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014.

[60] M.-T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," *arXiv preprint arXiv:1508.04025*, 2015.

[61] K. Cho, B. Van Merriënboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches," *arXiv preprint arXiv:1409.1259*, 2014.

[62] M. Sundermeyer, R. Schlüter, and H. Ney, "Lstm neural networks for language modeling," in *Thirteenth annual conference of the international speech communication association*, 2012.

[63] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[64] M. Abadi, P. Barham, Chen, *et al.*, “Tensorflow: A system for large-scale machine learning,” in *12th {USENIX} Symposium on Operating Systems Design and Implementation*, 2016, pp. 265–283.

[65] Q.-T. Dinh, M. Exbrayat, and C. Vrain, “Generative structure learning for markov logic networks based on graph of predicates,” in *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

[66] L. De Raedt, A. Kimmig, and H. Toivonen, “Problog: A probabilistic prolog and its application in link discovery.,” in *IJCAI*, Hyderabad, vol. 7, 2007, pp. 2462–2467.

[67] T. Khot, S. Natarajan, K. Kersting, and J. Shavlik, “Learning markov logic networks via functional gradient boosting,” in *2011 IEEE 11th International Conference on Data Mining*, IEEE, 2011, pp. 320–329.

[68] T. Ribeiro, S. Tourret, M. Folschette, M. Magnin, D. Borzacchiello, F. Chinesta, O. Roux, and K. Inoue, “Inductive learning from state transitions over continuous domains,” in *International Conference on Inductive Logic Programming*, Springer, 2017, pp. 124–139.

[69] D. Dua and E. Karra Taniskidou, *UCI machine learning repository*, 2017. [Online]. Available: `http://archive.ics.uci.edu/ml`.

[70] A. Srinivasan, *The aleph manual*, 2001.

[71] F. Shakerin and G. Gupta, “Induction of non-monotonic logic programs to explain boosted tree models using lime,” *arXiv preprint arXiv:1808.00629*, 2018.

[72] D. Marbach, T. Schaffter, D. Floreano, R. J. Prill, and G. Stolovitzky, “The dream4 in-silico network challenge,” *Draft, version 0.3*, 2009.

[73] X. Zhang, K. Liu, Z.-P. Liu, B. Duval, J.-M. Richer, X.-M. Zhao, J.-K. Hao, and L. Chen, “Narromi: A noise and redundancy reduction technique improves accuracy of gene regulatory network inference,” *Bioinformatics*, vol. 29, no. 1, pp. 106–113, 2012.

[74] B. Yang, Y. Xu, A. Maxwell, W. Koh, P. Gong, and C. Zhang, “Micrat: A novel algorithm for inferring gene regulatory networks using time series gene expression data,” *BMC systems biology*, vol. 12, no. 7, p. 115, 2018.

[75] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, “Asynchronous methods for deep reinforcement learning,” in *International conference on machine learning*, 2016, pp. 1928–1937.

[76] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[77] V. Zambaldi, D. Raposo, A. Santoro, V. Bapst, Y. Li, I. Babuschkin, K. Tuyls, D. Reichert, T. Lillicrap, E. Lockhart, *et al.*, "Relational deep reinforcement learning," *arXiv preprint arXiv:1806.01830*, 2018.

[78] M. Van Otterlo, "A survey of reinforcement learning in relational domains," *Centre for Telematics and Information Technology (CTIT) University of Twente, Tech. Rep*, 2005.

[79] Z. Jiang and S. Luo, *Neural logic reinforcement learning*, 2019. arXiv: `1904.10729 [cs.LG]`.

[80] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu, and S. Jaiswal, "Graph2vec: Learning distributed representations of graphs," *arXiv preprint arXiv:1707.05005*, 2017.

[81] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *arXiv preprint arXiv:1711.00740*, 2017.

[82] A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu, *Stable baselines*, `https://github.com/hill-a/stable-baselines`, 2018.

[83] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, *et al.*, "Relational inductive biases, deep learning, and graph networks," *arXiv preprint arXiv:1806.01261*, 2018.

[84] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, *Rainbow: Combining improvements in deep reinforcement learning*, 2017. arXiv: `1710.02298 [cs.AI]`.

[85] A. Payani and F. Faramarz, "Learning algorithms via neural logic networks," in *submitted to Thirty-Third AAAI Conference on Artificial Intelligence (AAAI-19)*, 2019.

[86] R. Gallager, "Low-density parity-check codes," *IRE Transactions on information theory*, vol. 8, no. 1, pp. 21–28, 1962.

[87] G. Li and Z. Zeng, "A neural-network algorithm for solving nonlinear equation systems," in *Computational Intelligence and Security, 2008. CIS'08. International Conference on*, IEEE, vol. 1, 2008, pp. 20–23.

[88] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *arXiv preprint arXiv:1312.6114*, 2013.

[89] D. J. Rezende and S. Mohamed, "Variational inference with normalizing flows," *arXiv preprint arXiv:1505.05770*, 2015.

[90] D. P. Kingma, T. Salimans, R. Jozefowicz, X. Chen, I. Sutskever, and M. Welling, "Improved variational inference with inverse autoregressive flow," in *Advances in neural information processing systems*, 2016, pp. 4743–4751.

[91] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *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.

[92] N. Bostrom, *Superintelligence: Paths, dangers, strategies, reprint ed*, 2016.

[93] S. Russell, "Should we fear supersmart robots?" *Scientific American*, vol. 314, no. 6, pp. 58–59, 2016.

[94] D. Amodei, C. Olah, J. Steinhardt, P. Christiano, J. Schulman, and D. Mané, "Concrete problems in ai safety," *arXiv preprint arXiv:1606.06565*, 2016.

[95] R. Akrour, M. Schoenauer, M. Sebag, and J.-C. Souplet, "Programming by feedback," in *International Conference on Machine Learning*, vol. 32, 2014, pp. 1503–1511.

[96] P. M. Pilarski, M. R. Dawson, T. Degris, F. Fahimi, J. P. Carey, and R. S. Sutton, "Online human training of a myoelectric prosthesis controller via actor-critic reinforcement learning," in *2011 IEEE International Conference on Rehabilitation Robotics*, IEEE, 2011, pp. 1–7.

[97] L. El Asri, B. Piot, M. Geist, R. Laroche, and O. Pietquin, "Score-based inverse reinforcement learning," in *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, International Foundation for Autonomous Agents and Multiagent Systems, 2016, pp. 457–465.

[98] S. I. Wang, P. Liang, and C. D. Manning, "Learning language games through interaction," *arXiv preprint arXiv:1606.02447*, 2016.

[99] P. F. Christiano, J. Leike, T. Brown, M. Martic, S. Legg, and D. Amodei, "Deep reinforcement learning from human preferences," in *Advances in Neural Information Processing Systems*, 2017, pp. 4299–4307.

[100] J. Garcıa and F. Fernández, "A comprehensive survey on safe reinforcement learning," *Journal of Machine Learning Research*, vol. 16, no. 1, pp. 1437–1480, 2015.

[101] M. Sato, H. Kimura, and S. Kobayashi, "Td algorithm for the variance of return and mean-variance reinforcement learning," *Transactions of the Japanese Society for Artificial Intelligence*, vol. 16, no. 3, pp. 353–362, 2001.

[102] P. Geibel and F. Wysotzki, "Risk-sensitive reinforcement learning applied to control under constraints," *Journal of Artificial Intelligence Research*, vol. 24, pp. 81–108, 2005.

[103] T. M. Moldovan and P. Abbeel, "Safe exploration in markov decision processes," *arXiv preprint arXiv:1205.4810*, 2012.

[104] F. Berkenkamp, M. Turchetta, A. Schoellig, and A. Krause, "Safe model-based reinforcement learning with stability guarantees," in *Advances in neural information processing systems*, 2017, pp. 908–918.

[105] B. Eysenbach, S. Gu, J. Ibarz, and S. Levine, "Leave no trace: Learning to reset for safe and autonomous reinforcement learning," *arXiv preprint arXiv:1711.06782*, 2017.

[106] G. Dalal, K. Dvijotham, M. Vecerik, T. Hester, C. Paduraru, and Y. Tassa, "Safe exploration in continuous action spaces," *arXiv preprint arXiv:1801.08757*, 2018.

[107] J. Leike, M. Martic, V. Krakovna, P. A. Ortega, T. Everitt, A. Lefrancq, L. Orseau, and S. Legg, "Ai safety gridworlds," *arXiv preprint arXiv:1711.09883*, 2017.