Learning Logic Programs from State Transitions Using Neural Networks

by
PHUA Yin Jun

Dissertation

submitted to the Department of Informatics
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

The Graduate University for Advanced Studies, SOKENDAI
March 2022

Abstract

Learning from Interpretation Transition (LFIT) is an unsupervised learning algorithm which learns the dynamics just by observing state transitions. LFIT algorithms have mainly been implemented in the symbolic method, but they are not robust to noisy or missing data. Recently, research works combining logical operations with neural networks are receiving a lot of attention, with most works taking an extraction-based approach where a single neural network model is trained to solve the problem, followed by extracting a logic model from the neural network model. However, most research work suffer from the combinatorial explosion problem when trying to scale up to solve larger problems. A lot of the invariances that hold in the symbolic world are not getting utilized in the neural network field.

In this thesis, we will introduce a novel technique for obtaining symbolic knowledge by deep learning. We will also introduce our contribution in which we verified that our technique was sound. We show that our method works by performing experiments in various different commonly used benchmarks. We will then be introducing several techniques that will allow the model to scale up to a larger problem space than can be handled previously. First, we will introduce a technique that will allow us to exploit the symbolic invariance that exists in the LFIT framework. Namely, the sequential ordering of the input should be invariant. This allows the model to be more efficient in its use of training data. Next, we will introduce a technique to allow further scaling up of the model to a larger problem space. We will reuse the same output nodes for multiple different meanings to reduce the exponential explosion. Then, we will also introduce some techniques that, due to some of the previous techniques, made the model much more difficult to train. We will also perform some experiments to show the effectiveness of our approach. Lastly, we will show some extensions of our method to other notions of dynamics.

Contents

Listing of figures

List of Tables

Acknowledgments

Starting my PhD in the year 2019 was perhaps the most unfortunate of timings. The world was quickly swept in 2020 and most of 2021 by the COVID19 pandemic. Events and conferences were cancelled, stay at home mandates were imposed, and there were many difficulties in regards to carrying on with research work. However despite all these difficulties, I am grateful for being surrounded by people who were willing to help me in the toughest of times, despite facing their own hardships.

First and foremost, I would like to thank my supervisor Professor Katsumi Inoue, who provided all the support and help required. I am grateful for the freedom to pursue my own ideas and the encouragement, even when some of the ideas did not come to fruition. I also thank the opportunity to participate in many wonderful conferences, and the connections to many wonderful people whom I may have not been able to meet if not for him.

I am also grateful to be a part of the Inoue lab, which has a wonderful atmosphere. My time in the lab has been most enjoyable, despite the various different research topics that we all had.

Most of all, I would like to thank my family, my brother, my sister and especially my parents for their gracious support. 2020 and 2021 were the most difficult times all of us were facing, yet despite being far apart, the emotional support has been truly helpful. For which without it, I will not have been able to complete this work. Thank you.

1

Introduction

Recent advancements in machine learning and deep learning [32] have brought to the world an unprecedented level of artificial intelligence (AI) boom. Ranging from something as simple as image recognition [47] [62], natural language translations [10], speech recognition [5], to self autonomous vehicle driving [96]. These were thought to be nearly impossible a little more than a decade ago, but has since become something approachable to reality in a relatively short amount of time [6]. It is difficult to overstate the achievements [82] [19] [45] of these various tasks for a method as simple as gathering data and letting the machine do the crunching.

Meanwhile, there are also tasks that are seemingly simple for humans, but are exceedingly difficult for these techniques [81]. Certain image recognition are easily fooled by bit manipulation, despite an image looking literally the same to human eyes [60] [87]. There are also various accountability issues [14] where the decision for an algorithm to categorize something can affect a person's life, yet the algorithm can simply offer no explanation beyond the numbers that it calculated [30] [18].

A purely symbolic method, however, does not work as we've observed since the first AI winter. Purely symbolic methods suffer from the issue of robustness [89] and thus are less applicable to real world usage.

On the other hand, the AI community has rode the wave of deep learning and have achieved feats that are worthy of applause, but it is clear that, as more important problems need to be tackled, pure deep learning alone is no longer sufficient. Explainable AI (XAI) [3] [76] is one such emerging field in trying to solve the problem of deep learning being not really explainable [77]. Some experts also call this the interpretability problem [31]. An alternative name that is also widely used is the academic community is interpretable AI.

Multiple techniques have been developed in light of the XAI movement. Currently, most of the efforts are focused on interpreting the trained model, hence the name "interpretable AI" [17]. Most of these techniques probe the model to figure out where the model changes its decisions, or attempting visualization of so called "features" to figure out the priorities learned by the model [31] [61].

Unfortunately, application of these techniques are still fairly limited in the real world. Everyday, as algorithms are making decisions on social media and affecting people's lives, transparency is still no where to be found. As financial institutions also begin to use AI techniques to automate risk decisions, there are real issues where some people getting locked out of our financial systems with no clear explanation [53].

Some issues with these interpretable techniques include being only applicable to applications where visualization is meaningful [83] [98], or applicable only when features fed into the models are understandable. Evidently this is usually not the case with real world applications. There seems to be a trade-off between being able to obtain interpretable model and accurate model [44], although no empirical studies have yet been performed. Models that started out interpretable are not able to achieve an acceptable level of performance in the real world, whereas models that are less interpretable achieve stunning accuracy. However it is only in the edge cases where these models breakdown, and when they do they can have profound effects on someone's life, yet nobody can explain how that happened [54].

There is however an alternate technique, bringing back techniques from the 1950s, in combining the symbolic logic AI with the current deep learning AI.

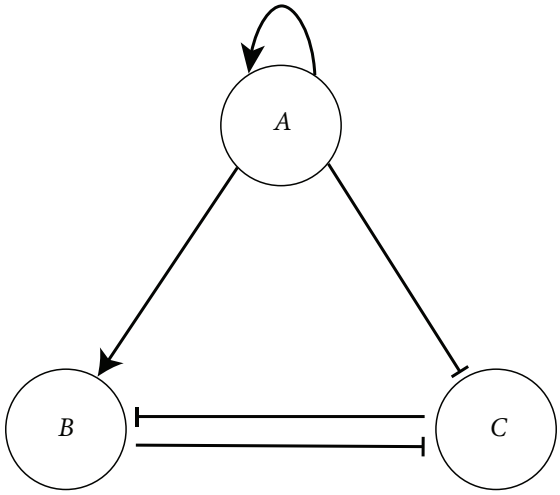


Figure 1.1: Boolean Network

This field, dubbed the Neural-Symbolic AI (NSAI) is not a recent trend [22] [13] [34], though it has certainly garnered some attention in light of the XAI movement. The main focus of NSAI research is to allow machines to perform higher-order thinking functions, like the human brain [48]. But it so happens that symbolic is interpretable and explainable as well. Thus there are some hopes that the combination of neural network and symbolic AI can lead to a transparent AI model [9].

This thesis attempts to propose a method in line with the spirit of NSAI. Solving the entire interpretability issue is a scope way beyond this thesis [17]. In this thesis, the focus lies on trying to refine an NSAI technique in a much narrower sense. In particular, the framework for which this thesis works on is the learning of system dynamics.

Boolean networks [46], first proposed by Kauffman as random models, to model gene regulatory networks are very simple at first sight. A boolean network describes the relationship between different variables, whereby the relationships define the values of the variables. In a dynamic system, the values of the variables change at each timestep. The collective values of the variables at any timestep is called a state.

Learning from Interpretation Transition (LFIT) is an algorithm that learns explainable rules of a dynamic system [41]. Given a series of state transitions

from the observed dynamic system, the LFIT algorithm outputs a normal logic program (NLP) which realizes the given state transitions. An NLP is fully equivalent to a boolean network [39] [42]. Therefore, LFIT can also learn boolen networks. LFIT has been applied in various fields, including biology [71] and multi-agent settings [57].

LFIT has been mainly implemented in two different methods, the symbolic method [70] and the neural network method [29]. The symbolic method uses logical operations and symbolic manipulation to learn and induce logic programs. Each state transition is taken as an example and the algorithm applies logical operation to ensure that every example is covered. This guarantees that the resulting NLP is consistent with the state transitions. The symbolic method is relatively scalable. Most research has also been done in the symbolic area. There are various advancements like dealing with multivalued systems [73], and general semantics [68] [69] that have not yet been applied to neural networks. However, there are also significant limits when utilizing logical operations. One of the limitations is that the logical operations employed by symbolic LFIT algorithms lack ambiguous notations. This means that any error or noise present in the data is reflected directly in the output, synonymous to the garbage in, garbage out problem.

On the other hand, since neural networks are known to be robust to noise [74] [80], neural network methods have been developed as an attempt to solve the ambiguous value issue. However there are also roadblocks with the neural network method. The first and simplest method developed, NN-LFIT [29] trains a neural network that models the system being studied. The weights of the neural network are then pruned based on a heuristic, and non-zero weights are determined as connections between the input nodes and the output nodes. Whilst simplistic and scales relatively well, this technique suffers from the usual machine learning suspect, namely overfitting.

First idea for extending NN-LFIT came by dealing with delays [65], which have already been implemented in the symbolic algorithm [72]. Most component in a biological system do not change their states in lockstep. Introducing a time delay component is typically one way to allow for the model to express such idea. Extending NN-LFIT to recurrent neural networks (RNN) however was not simple. RNNs, having loops in between them, have a much more complex inter-connection. Furthermore, as RNNs get unfolded to longer and longer time, training becomes very difficult due to the exponential gradient problem.

It became apparent that the simplistic NN-LFIT approach is fairly limited in its extensions.

The issue with NN-LFIT is that, mainly by fiddling with the neural network itself, for every advancements made in the field of deep learning, a new technique has to be devised in order to adapt the advancement. Obviously by employing current state-of-the-art techniques like batch normalization [43] and dropout [85], the characteristic of the neural network is bound to change so much that a simplistic approach like NN-LFIT is no longer feasible.

In this thesis, we will first cover the necessary background knowledge in chapter 2. These knowledge include both logic programming and neural network related background.

Our attempt at adapting neural network to LFIT came in the form of δLFIT [65]. δLFIT took a different approach from NN-LFIT, wherein NN-LFIT only trains one neural network per logic program, δLFIT trains one neural network to learn multiple logic programs. NN-LFIT takes a state as input and outputs the next state as prediction, while δLFIT takes a sequence of state transitions and predicts the NLP that explains the provided state transitions. Where NN-LFIT is a problem of modelling a dynamic system with neural network, δLFIT is about learning the semantics of dynamic systems, the relationship between their observed state transitions and the logical rules that explain them. The approach δLFIT took required a new idea. The LFIT algorithm, in its essence, is classifying whether a variable is true or false given the current state. This translates to an interesting concept where by the LFIT algorithm can be thought of as a classification problem. Neural networks have long claimed the superiority in the classification problem. Also by not relying in any way the internals of the network itself, it is now possible to employ state-of-the-art techniques in training the neural network. An overall flow of the learning method employed by δLFIT is depicted in figure 1.2. This work is described in chapter 3.

However, with the different approach that δLFIT has taken, many different problems have arise. The combinatorial nature of symbolic and the way δLFIT is structured architecturally leads to a combinatorial explosion problem. We have devised several techniques to curb the explosion, and we have also focused particularly in the invariance that is present in the symbolic world to further help this issue. This work is described in chapter 4.

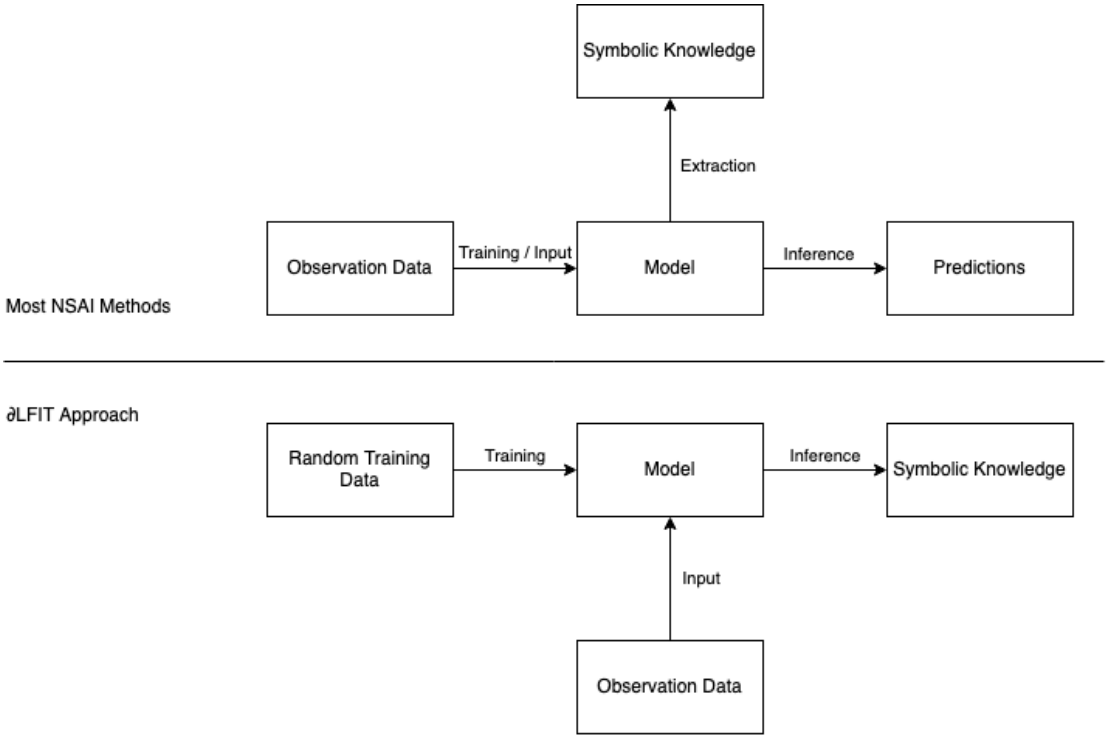


Figure 1.2: The difference in learning method between most NSAI approaches and δLFIT, δLFIT+

We then extend our work to various other improvements that have been made in the symbolic LFIT field. Namely dealing with systems with delays and also various different transition semantics. This work is described in chapter 5.

Then, we will be describing some of the related works in the field and the relationship of our work in chapter 6. Lastly, we will close out the thesis with a summary of our contributions and some future outlook in chapter 7.

2

Background

In this chapter, we will introduce the necessary background knowledge required to understand the contributions in this thesis.

The work done in this thesis touches 2 major areas, symbolic AI and neural networks. Both are seminal AI techniques and are the foundation of 2 separate AI booms.

This work forms a bridge between the 2 separate methods, and thus necessitates a brief coverage of both areas. On the symbolic side, we will cover the basics for logic programming, the LFIT framework for which this work is based on, and the symbolic algorithm for LFIT. On the neural network side, we will be covering first the basis of neural networks, then some techniques that are utilized in this thesis. Lastly, we will be covering the basics of δLFIT, which will be necessary in order to understand this thesis.

2.1 Logic Programming

Logic programming is a type of declarative programming based on formal logic. A logic program is typically a set of sentences written in logical form, expressing facts and rules about a problem domain. Rules are written in the form of clauses such as the following:

$$A \leftarrow A_1 \wedge A_2 \wedge \ldots \wedge A_m \tag{2.1}$$

where A and A_i are propositional atoms. Here, A is the head of the rule and $A_1, A_2, \ldots, A_m$ together with $\wedge$, which is the *logical and* operation forms the body of the rule. This rule, in plain English, can be read as "A is true if all of $A_1, A_2, \ldots, A_m$ are true".

For any rule R of the form (2.1), the head of R, denoted as $h(R)$ is defined as $h(R) = A$. The body of R, denoted as $b(R)$ is defined as the set of atoms in the body, $b(R) = A_1, A_2, \ldots, A_m$.

Facts are rules that have no body, and are written in the form:

$$A \leftarrow$$

which states that A is true.

A logic program P is a set of rules in the form (2.1). All atoms that appear in a logic program P, is called the Herbrand base $\mathcal{B}$. An Herbrand interpretation I is a subset of $\mathcal{B}$, which is an assignment of truth values to the atoms in the Herbrand base. An interpretation is *inconsistent* if $\bot \in I$, otherwise I is called *consistent*.

Given a rule R and an interpretation I, if $b(R) \subseteq I$ implies $h(R) \in I$, then the interpretation I is said to satisfy the rule R. If a consistent interpretation I satisfies all rules of a logic program P, I is called the model of P.

2.1.1 Normal Logic Program

For most practical applications, the negation of an atom has to be considered. In classical logic, the negation of an atom a can be written as $\overline{a}$ and is under-

stood as "a is false". In logic programming however, it is very difficult to disprove a statement (i.e. to infer that a is false). The knowledge we have about our world may simply be insufficient to decide that. Therefore, the notion of *negation as failure* [20] is introduced. Under the *negation as failure* viewpoint, a is assumed to be false if it cannot be shown true in our knowledge base. This is also known as the *closed world assumption*.

A normal rule is of the form:

$$A \leftarrow A_1 \wedge A_2 \wedge \ldots \wedge A_m \wedge \neg A_{m+1} \wedge A_{m+2} \wedge \ldots \wedge \neg A_n \tag{2.2}$$

where A and A_i are propositional atoms, and $\neg a$ is the negation of the atom a. The atoms $A_1, A_2, \ldots, A_m$ are referred to as the positive body, denoted as $b^+(R) = \{A_1, A_2, \ldots, A_m\}$, while the atoms $A_{m+1}, A_{m+2}, \ldots, A_n$ are the negative body, denoted as $b^-(R) = \{A_{m+1}, A_{m+2}, \ldots, A_n\}$. A logic program composed of a set of normal rules are called a normal logic program (NLP).

2.1.2 Inductive Logic Programming

Logic programs can be handwritten, but there is no reason to leave it at that. A method of producing logic programs, called the Inductive Logic Programming (ILP) [58] is an approach to machine learning which produces logic program as its result. ILP algorithms work on examples, both positive examples and negative examples. The algorithm will try to deduce a logic program that both confirms the positive examples and infirm the negative examples.

ILP algorithms mainly rely on generalization and specialization to deduce logic programs. Generalization and specialization are a kind of dual notion, where generalization is an inductive operation while specialization is a deductive operation.

A generalization operator maps a conjunction of clauses S onto a set of minimal generalizations of S. A *minimal generalization* G of S is a generalization of S such that S is not a generalization of G, and there is no generlization G' of S such that G is a generalization of G' [59]. To make this easier to understand, consider the following example.

Example 1 *Consider the following clauses* $C_1 = (a)$, $C_2 = (a \wedge b)$, *and* $C_3 =$

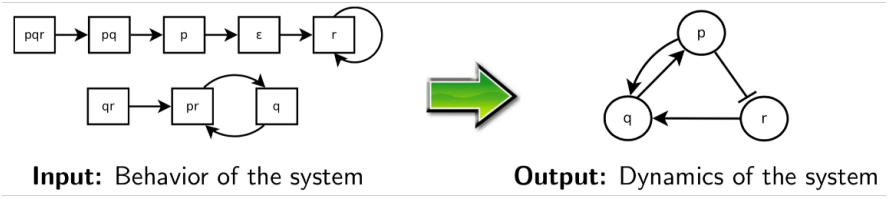


Figure 2.1: Left: input of state transitions, right: output of logic program

$(a \wedge b \wedge c)$. *In this case,* C_2 *is a minimal generalization of* C_3*, but* C_1 *is* ***not*** *a minimal generalization of* C_3*. This is because for* C_1*, there exists a much more minimal generalization for* C_3 *that is* C_2*.*

A specialization operator maps a conjunction of clauses G onto a set of maximal specializations of G. A *maximal specialization* S of G is a specialization of G such that G is not a specialization of S, and there is no specialization S' of G such that S is a specialization of S'. Consider the following example.

Example 2 *From example 1,* C_3 *is a maximal specialization of* C_2*, but* C_3 *is* ***not*** *a maximal specialization of* C_1*. For the similar reason that because for* C_3*, there exists a much more maximal specialization for* C_1 *that is* C_2*.*

2.2 LFIT

The main goal of LFIT [41] is to learn an NLP describing the dynamics of the observed system. The overall behavior of the LFIT algorithm is shown in figure 2.1. To describe the dynamics of a changing system with respect to time, we can use time as an argument. In particular, we will consider the state of an atom A at time t as $A(t)$. Thus, we can rewrite the rule of form (2.2) into a dynamic rule as follows:

$$A(t+1) \leftarrow A_1(t) \wedge A_2(t) \wedge \ldots \wedge A_m(t) \wedge \neg A_{m+1}(t) \wedge A_{m+2}(t) \wedge \ldots \wedge \neg A_n(t) \tag{2.3}$$

which means that, if $A_1, A_2, \ldots, A_m$ is true at time t and $A_{m+1}, A_{m+2}, \ldots, A_n$ is false at time t, then the head A will be true at time $t+1$. By describing a normal rule in the form (2.3), we can simulate the state transition of a dynamical system with the T_P operator [90] [7].

It is important to note that, even though t is expressed as a parameter in the rule, $t+1$ only ever appears on the left hand side while t strictly appears only on the right. They can subsequently omitted and be considered as a propositional rule, just like rule 2.2.

Also note that the literal on the left, A can appear on the right as well. Consider the following rule

$$A(t+1) \leftarrow A(t)$$

In terms of describing the dynamics of the system, we want the A on the left and A on the right to represent the same thing, say a particular gene. However, they represent different values in different times. Therefore this is not a cyclic rule. Since LFIT only learns rule that determine the next state of the system, there would never be any cyclic rule involved in the algorithm.

In LFIT, we are most concern about the transition of states between 2 adjacent timesteps. If an interpretation I reflects the state of a system at time t, then the state of the system at $t+1$, J is also an interpretation. In this case, we can denote (I, J) as the state transition of the dynamic system from I to J.

Given a rule R of form 2.3 and a state transition (I, J), R is *consistent* with (I, J) if and only if $b^+(R) \subseteq I$ and $b^-(R) \cap I = \emptyset$ implies $h(R) \in J$. The concept of *consistency* can be further expanded to a set of state transitions. If R is consistent with every state transition in set E, then R is consistent with E. Further, a logic program P is consistent with E if every rule $R \in P$ is consistent with E.

In section 2.1.2, we defined the generalization and specialization of clauses. Here, we can similarly define the generalization and specialization of rules. To apply these operations to rules, we will consider rules that share the same head and operate on their bodies.

A rule R_s is a *maximal specialization* of a rule R, if $h(R_S) = h(R)$ and $b(R_S)$ is a maximal specialization of $b(R)$. Given two rules R_1 and R_2 where $h(R_1) = h(R_2)$ and R_1 subsumes R_2, i.e. $b(R_1) \subseteq b(R_2)$ [70]. Let l_i be the i^{th} literal of $b(R_2)$, then the *least specialization* of R_1 over R_2 is defined as follows:

$$\begin{aligned} ls(R_1, R_2) &= \{h(R_1) \leftarrow (b(R_1) \wedge \neg b(R_2))\} \\ &= \{(h(R_1) \leftarrow (b(R_1) \wedge \overline{l_i})) \mid l_i \in b(R_2) \setminus b(R_1)\} \end{aligned}$$

Given an NLP P, a rule R and a set of rules $S \subseteq P$ that subsumes R, the *least specialization* of P by R can be written:

$$ls(P, R) = (P \setminus S) \cup (\bigcup_{R_P \in S} ls(R_P, R))$$

A *minimal generalization* of a rule R is then, conversely a rule R_g if $h(R_g) = h(R)$ and $b(R_g)$ is a minimal generalization of $b(R)$.

The LFIT algorithm is defined to be an algorithm which, when given the input of a set of state transitions E and an initial NLP P_0, produces a result of an NLP P such that P is consistent with E.

The symbolic method of implementing LFIT can utilize either the generalization operator or the specialization operator. The generalization method starts out with the least general rule and then generalizing for each example provided. The specialization method, on the other hand, starts with the most general rule and then specializing the rules for each example. It turns out that the generalization method produces a lot of rules. This is, in most cases, not what we want. The specialization method on the other hand, only considers the minimal conditions and thus produces far fewer rules.

2.2.1 Learning Prime Implicant Conditions

The specialization method learns all minimal conditions that imply a variable being true in the next state. Minimal conditions are easier to deal with in various situations, given the smaller amount of computation required. Therefore it is often desirable to learn only the minimal conditions that can describe a particular system.

To define the minimality of an NLP P, the notion of prime implicant is used. P, an NLP learned by the LFIT algorithm is considered to be minimal, if the body of each rule is a prime implicant to infer its head. Given a formula ϕ, an implicant of C is prime, if and only if none of its proper subset $S \subset C$ is also an implicant of ϕ [40]. This means that C is the most minimal clause that covers the same truth table as ϕ.

Given a rule R and a set of state transitions E such that R is consistent with

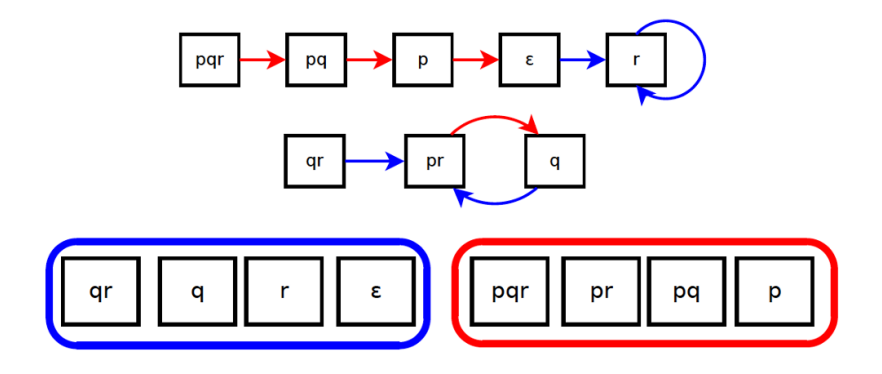


Figure 2.2: How LFIT learns, reds are negative examples and blues are positive examples

E. $b(R)$ is a *prime implicant condition* of $h(R)$ if there does not exist another rule R' that subsumes R and is consistent with E. To simplify further discussion, we also call R a *prime rule* of E. Given that the definition of a prime rule is a rule that is not subsumed by any other rules, the most general prime rule is the fact, i.e. a rule with an empty body.

An NLP P is a *prime NLP* for a set of state transitions E, if P realizes E and all rules of P are prime rules for E. The set of all prime rules of E is the *complete prime NLP* of E. This is illustrated in figure 2.2, the NLP must realize all blue transitions while not realizing the red transitions. Blue and red are colored with respective to r.

A simple naïve method of obtaining the complete prime NLP for a given set of transitions is to do a brute force search. By starting from the most general rules (i.e. facts), we can then generate all maximal specific specialization at each step and keep only the first ones that are consistent with E. However this method requires traversing all possible rules and thus is not computationally efficient.

This algorithm relies heavily on specialization. Starting from the most general rule, each state transition provides negative examples that allow the algorithm to perform specialization to each of the rules it has already learned. By doing this, the output of this algorithm is guaranteed to be a complete prime NLP of the input. The complete algorithm is described in algorithm 1.

The algorithm starts out with putting in facts for each atom in the Herbrand

Algorithm 1: LFIT: Learning the complete prime NLP P

Inputs : a set of atoms $\mathcal{B}$, $E \subseteq 2^{\mathcal{B}} \times 2^{\mathcal{B}}$
Output: an NLP P such that $J = T_P(I)$ holds for any $(I, J) \in E$
$P := \emptyset$;
foreach $A \in \mathcal{B}$ **do**
  $P = P \cup \{A.\}$;
end
while $E \neq \emptyset$ **do**
  Pick $(I, J) \in E$;
  $E := E \setminus \{(I, J)\}$;
  foreach $A \in \mathcal{B}$ **do**
    if $A \notin J$ **then**
      $R^I_A := A \leftarrow \bigwedge_{B_i \in I} B_i \wedge \bigwedge_{C_j \in (\mathcal{B} \setminus I)} \neg C_j$;
      $P :=$`Specialize(`P, R^I_A`)`;
    end
  end
end

base $\mathcal{B}$ of the input. Then each state transition $(I, J) \in E$ is analyzed. For each variable A that **is not present** in J, LFIT infers an anti-rule R^I_A:

$$R^I_A = A \leftarrow \bigwedge_{B_i \in I} B_i \wedge \bigwedge_{C_j \in \mathcal{B} \setminus I} \neg C_j$$

This rule is constructed such that I infers A, but since A is not present in J, this rule is not consistent with the input. This is done specifically so that once the algorithm performs a specialization of P by R^I_A, none of the rules in P will subsume R^I_A, which means that none of the rules in P will infer A given I, thus P is now consistent with I with respect to A. The specialization algorithm is described in algorithm 2.

To perform specialization, all rules $R_P \in P$ that subsume R^I_A is first extracted. The least specialization of each rule R_P is generated by creating a rule for each literal in R^I_A. Each of the created rules contain all of the literals from R_P, in addition to the negation of the literals in R^I_A. Then any rules that are not subsumed already by P is added to P. Now P is an NLP that is consistent with the transition (I, J), while also being a complete prime NLP. Once all the state transitions in E has been analyzed, P is a complete prime NLP of E.

Algorithm 2: Specialize(P, R): specialize P so that P does not subsume R

Inputs : an NLP P and a rule R
Output: the maximal specialization P that does not subsume R
conflicts := $\emptyset$;
foreach $R_P \in P$ **do**
  if $b(R_P) \subseteq b(R)$ **then**
    conflicts = conflicts $\cup R_P$;
    $P = P \setminus R_P$;
  end
end
foreach $R_c \in$ *conflicts* **do**
  foreach $l \in b(R)$ **do**
    if $l \notin b(R_c)$ *and* $\bar{l} \notin b(R_c)$ **then**
      $R'_c = (h(R_c) \leftarrow (b(R_c) \cup \bar{l}))$;
      if P *does not subsume* R'_c **then**
        $P = P \setminus$ all rules subsumed byR'_c;
        $P = P \cup R'_c$;
      end
    end
  end
end
return P

Table 2.1 shows the execution trace of LFIT, where the state transition $pqr \rightarrow pq$ represents the state transition $(\{p, q, r\}, \{p, q\})$. Literals that are introduced by least specialization is represented in bold, while rules that are subsumed after specialization are stroked out. This algorithm begins with the most general set of prime rules, which is $P = \{p., q., r.\}$. Based on the transition (pqr, pq), the algorithm infers the anti rule $r \leftarrow p \wedge q \wedge r$. This rule is subsumed by $r.$, therefore a least specialization operation has to be done. By performing the least specialization we get $ls(r., r \leftarrow p \wedge q \wedge r) = \{r \leftarrow \neg p, r \leftarrow \neg q, r \leftarrow \neg r\}$. By combining the rules obtained from the least specialization and the rules originally in P, and also removing the rule that subsumed the anti-rule, we can obtain an NLP that is now consistent with the transition (pqr, pq). The result of this is shown in the cell next to initialization in table 2.1.

Next, based on the transition (pq, p), the algorithm infers two anti-rules: $q \leftarrow p \wedge q \wedge \neg r$ and $r \leftarrow p \wedge q \wedge \neg r$. These are subsumed by $q.$ and $r \leftarrow \neg r$ respectively. $q.$ is replaced by $\{q \leftarrow \neg p, q \leftarrow \neg q, q \leftarrow r\}$ after performing least specialization. On the other hand, the least specialization of the second inferred anti-rule produces two new rules $r \leftarrow \neg p \wedge \neg r$ and $r \leftarrow \neg q \wedge \neg r$. However these rules are subsumed by rules $r \leftarrow \neg p$ and $r \leftarrow \neg q$ that are already in P respectively. Thus, the least specialization of the second anti-rule only resulted in the specialization of the rule $q.$ and the deletion of $r \leftarrow \neg r$.

The same steps are repeated until (q, pr), where a special case occurs. Based on this transition, the algorithm infers the rule $q \leftarrow \neg p \wedge q \wedge \neg r$, which is subsumed by P with rule $R : q \leftarrow \neg p \wedge q \wedge \neg r$. Unfortunately, the rule that P already has includes all the atoms that exists in the Herbrand base, $\|b(R)\| = \|\mathcal{B}\|$ and therefore cannot be further specialized. In this case, rule R is simply removed from the NLP P.

2.3 Neural Network and Deep Learning

The artificial neural network (ANN) is a machine learning prediction algorithm inspired by the human brain. ANN was originally developed in 1957 [75], but has only seen daylight in recent years thanks to the improvements in parallel computation technology, that made ANN practical and accessible. Although ANNs are inspired by the biological neural function of the brain, their mechanisms and functions are anything but.

An example of an ANN structure is the feed-forward neural network (FNN). An FNN, as shown in Figure 2.3, consists of multiple artificial neurons, connected in layers, in a way such that information only flows one way. An FNN typically includes three types of layers: one input layer, one or more hidden layers, and one output layer. The input layer usually denotes the dimension of the input to the FNN, with the number of neurons equals the dimension of the input. The hidden layers are usually responsible for the computation. The output layers consist of activation functions, to give meaningful purposes to the computation performed in the hidden layers. An FNN with at least one hidden layer is said to be able to function as a universal approximator, i.e. can compute almost any function [37].

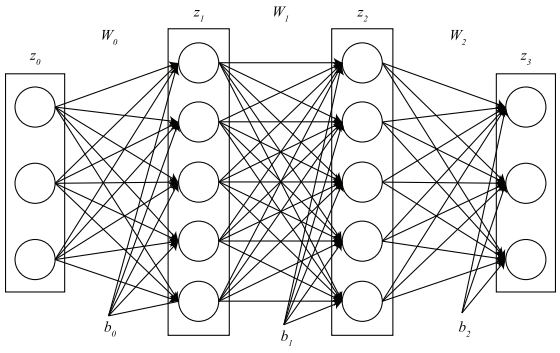


Figure 2.3: A feed-forward neural network.

Algorithm 3: Computation of an FNN

$z_0 := x$;
for $i = 1$ *to* $l + 1$ **do**
  $x_i := W_{i-1} z_{i-1} + b_i$;
  $z_i := e(x_i)$;
end
$z := z_l$;
return z;

An FNN with l hidden layers is parametrized by l+1 weight matrix $(W_0, \ldots, W_l)$ and $l + 1$ bias vectors $(b_0, \ldots, b_l)$. Given an input x, the output of the FNN z is computed as described in algorithm 3.

In algorithm 3, $e(\cdot)$ is the activation function, such as the element-wise sigmoid function $\text{sigmoid}(x_j) = 1/(1+\exp(-x_j))$. With this construct, it is possible to model Boolean functions with FNNs.

2.3.1 Backpropagation

To make FNN approximate a function, all parameters have to be carefully chosen. However, doing so by hand will not be practical. Trying to solve the parameters analytically is also very difficult. Thus, the *backpropagation algorithm* [35] was introduced as a simple and effective solution to finding the appropriate parameters iteratively. Traditionally, gradient descent was used as the optimiza-

tion method. Gradient descent is known to work well enough in practice with the proper configuration (also called hyperparameters), even though it can be quite time-consuming and is also not guaranteed to find the global minimum.

To perform the backpropagation algorithm, a pair of input vector and the corresponding desired output vector, also known as a training example, is prepared. The input vector is feed through the FNN. The parameters of the FNN is assumed to be initialized to a certain value, usually random. The output of the FNN is then compared to the desired output from the training example, and is evaluated with a loss function defined by the user. Usually a small value of the loss function indicates that the FNN is behaving closer to the desired function. Therefore, the loss function can be thought of as a distance measurement between the current function of the FNN and the desired function.

The gradient of the loss function is then calculated. Gradients for each layer in the hidden layer is also subsequently computed by using the chain rule of derivates. Each parameter is then updated by subtracting a portion of the gradients that were calculated. The amount to subtract is called the *learning rate*, which can either be fixed throughout the whole training process or be dynamic. After all parameters have been updated, the algorithm will repeat the whole process again by using a different training example. This whole process can be repeated until the parameters converge, i.e. does not change by a significant amount after updating.

It is not uncommon to have huge amount of training examples to train the network with. Such method of updating the parameters after each training example costs not only a huge amount of computation, but can also lead to huge amount of inefficiency due to gradients constantly changing directions and not leading towards convergence. Thus, another method of training called *mini-batch learning* [51] was introduced in order to overcome these problems. In mini-batch learning, a subset of training examples are taken together and used to calculate gradients before updating the parameters. The gradients are aggregated with average and thus provides more stability in terms of the direction.

2.3.2 Differentiable Programming

The allmightyness of neural networks through backpropagation has sparked a new form of programming paradigm, called the differentiable programming [52]. Compared to neural networks which only use matrix multiplication and an activation function, in differentiable programming a program can be defined with any differentiable mathematical operator. A loss function is then defined and a gradient descent algorithm will then optimize the program to minimize the loss function.

Stemming from this paradigm, many different "neural network" architectures have been proposed. Many of them with no resemblence to the original simplistic artificial perceptron. The first prominent example of this is the Long-Short Term Memory (LSTM) [36]. LSTM came from the need to deal with sequence inputs, yet the simplistic RNN was difficult to use given the gradient explosion problem [12]. LSTM was a breakthrough, "gates" were defined within the network to allow dynamic control of how much the inputs or previous states should affect the current internal states. Many revolutionary architectures have also since been proposed [11] [79].

2.3.3 Layer Normalization

As neural networks get deeper, training became more computationally expensive and difficult. As parameters increase, the risk of overfitting also increase. To counteract the risk of overfitting, more training data will be provided and thus neural network models took more time to train and converge.

A technique called batch normalization [43] was introduced at first, that takes the average and variance of a mini-batch during training and normalize the mini-batch. This allowed feed forward neural networks to converge quickly. However, it wasn't clear how this technique can be applied to recurrent neural networks.

Thus layer normalization [8] was introduced that can be applied not just to feed forward neural networks, but in fact can be applied to any neural network

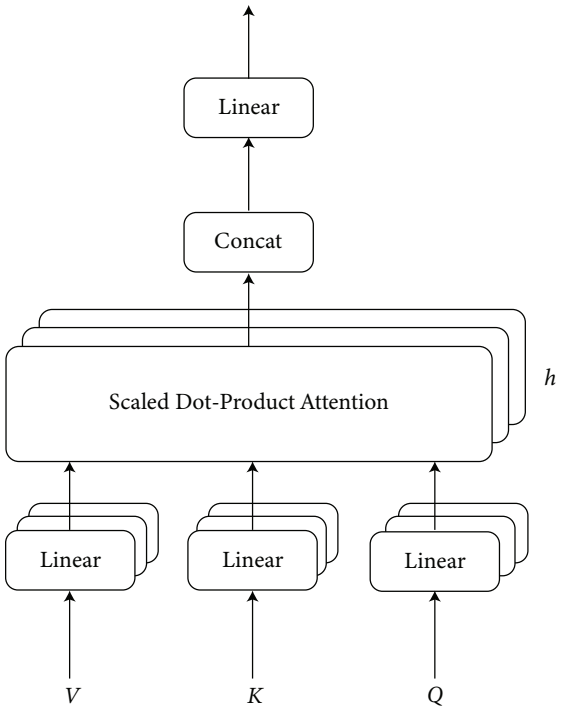


Figure 2.4: Multi-head Attention.

architecture. Layer normalization is defined as follows

$$\text{LayerNorm} = \gamma \frac{x - \text{E}[x]}{\sqrt{\text{Var}[x] + \epsilon}} + \beta \tag{2.4}$$

Layer normalization apples normalization to each training instance instead of a mini-batch, thus getting rid of inter instance dependencies. This has allowed neural networks to converge even quicker and is a major driving force behind recent deep learning advancements.

2.3.4 Attention

The next advancement came in the form of transformer [91]. Recurrent neural networks enforce sequence ordering with their layered structure, with the left most input in the sequence taking the longest route to the output. This, however disallows information to flow back in time, particularly when later infor-

mation can inform about details on past information. This is especially problematic in the natural language domain, as words typically can change meaning depending on what comes next.

The transformer model attempts to solve this by laying all the inputs flat and allow an "attention" mechanism to identify which part of the inputs are of relative importance to the output. Position of the inputs are provided as positional encoding.

The attention mechanism used in transformer is also called the Scaled Dot-Product Attention. Given a matrix Q, and two other matrices K and V, the attention is computed as follows

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \tag{2.5}$$

where d_k is the dimension of K.

Most models don't use a single attention, as it has been found to be beneficial to perform multiple linear projections to multiple dimensions. With multi-head attention, the model can attend to different information concurrently with different representations.

The multi-head attention is calculated as follows

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \tag{2.6}$$

$$\text{where head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{2.7}$$

The transformer is an encoder-decoder architecture that primarily utilizes the attention mechanism. The transformer model revolutionized machine translation and various other natural language processing tasks.

2.3.5 Set Transformer

Set transformer [49] is an architecture that deals with unordered sequence, namely a set. There might be questions as to why come up with another model when, if you take off the positional encoding in the transformer, you should be able to get non-positional inputs. The reason however, is fairly simple. An architecture

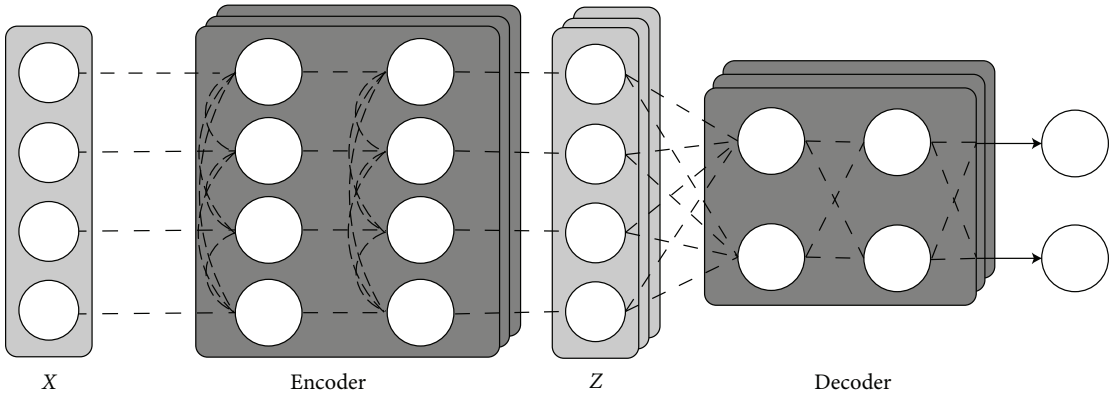


Figure 2.5: The Set Transformer Architecture.

processing a set has a requirement to be *permutation invariant*, that is whether element 1 and element 3 are in their own positions or swap places, the result of the computation should be the same. This is however, not the case with a vanilla transformer. A vanilla transformer even without positional encoding, is still sensitive to the input ordering.

To ensure permutation invariance, we will have to first pool the inputs. If the pooling is permutation invariant, then we can guarantee that subsequent operations will also be permutation invariant. One simple pooling operations is the max function, which takes the largest number from a vector. This is easily provable to be permutation invariant as the largest number will be the same no matter where it appears in the vector.

The set transformer is just like a transformer, consisting of an encoder and a decoder. The difference is that both encoder and decoder attend to their own inputs respectively to produce activation.

The set transformer introduces two permutation invariant blocks, set attention block (SAB) and induced set attention block (ISAB). The basic building block for these blocks is the multi-head attention block (MAB) which can be defined as follows

$$\begin{aligned} \text{MAB}(X, Y) &= \text{LayerNorm}(H + \text{rFF}(H)) \\ \text{where } H &= \text{LayerNorm}(X + \text{MultiHead(X,Y,Y)}) \end{aligned} \tag{2.8}$$

rFF is any row-wise feed forward network. The SAB is defined as

$$\mathrm{SAB}(X) = \mathrm{MAB}(X, X) \tag{2.9}$$

From this equation, it is evident that SAB takes a set and performs self-attention between elements in the set. This produces the permutation invariance with respect to the ordering of elements in X. However, SAB is quadratic in time complexity and may be fairly expensive when there is high number of elements. ISAB is thus introduced, which first reduces the dimension of the set before performing the attention. These lower dimension elements are called the inducing point I. With m inducing points, ISAB can be defined as

$$\begin{aligned} \mathrm{ISAB}_m(X) &= \mathrm{MAB}(X, H) \\ \text{where } H &= \mathrm{MAB}(I, H) \end{aligned} \tag{2.10}$$

Set transformer introduced the pooling by multi-head attention (PMA) layer, which is the permutation invariant layer. PMA has k learnable seed vectors. PMA is defined as

$$\mathrm{PMA}_k(Z) = \mathrm{MAB}(S, \mathrm{rFF}(Z)) \tag{2.11}$$

since SAB and ISAB are already permutation invariant, one might think that adding PMA is unnecessary. However the authors show that having a pooling layer helps in modeling explaining-away.

The encoder for the set transformer can use either stacks of SAB or ISAB. The decoder puts the output of the encoder through a layer of PMA, followed by SABs. The overall architecture is depicted in figure 2.5.

1. Initialization	$p.$ $q.$ $r.$	6. $r \to r$	$p \leftarrow q.$ $p \leftarrow \boldsymbol{p} \neg r.$ ~~$p \leftarrow \boldsymbol{q} \wedge r.$~~ $q \leftarrow \neg p \wedge q.$ $q \leftarrow \boldsymbol{p} \wedge r.$ $q \leftarrow \boldsymbol{q} \wedge r.$ $r \leftarrow \neg p.$ $r \leftarrow \neg q \wedge r.$
2. $pqr \to pq$	$p.$ $q.$ $r \leftarrow \neg \boldsymbol{p}.$ $r \leftarrow \neg \boldsymbol{q}.$ $r \leftarrow \neg \boldsymbol{r}.$	7. $qr \to pr$	$p \leftarrow q.$ $p \leftarrow p \wedge r.$ $q \leftarrow p \wedge r.$ $q \leftarrow \neg p \wedge q \wedge \neg \boldsymbol{r}.$ ~~$q \leftarrow \boldsymbol{p} \wedge q \wedge r.$~~ $r \leftarrow \neg p.$ $r \leftarrow \neg q \wedge r.$
3. $pq \to p$	$p.$ $q \leftarrow \neg \boldsymbol{p}.$ $q \leftarrow \neg \boldsymbol{q}.$ $q \leftarrow r.$ $r \leftarrow \neg p.$ $r \leftarrow \neg q.$ ~~$r \leftarrow \neg \boldsymbol{p} \wedge \neg r.$~~ ~~$r \leftarrow \neg \boldsymbol{q} \wedge \neg r.$~~	8. $pr \to q$	$p \leftarrow q.$ ~~$p \leftarrow p \wedge \boldsymbol{q} \wedge r.$~~ $q \leftarrow p \wedge r.$ $q \leftarrow \neg p \wedge q \wedge \neg r.$ $r \leftarrow \neg p.$ ~~$r \leftarrow \neg \boldsymbol{p} \wedge \neg q \wedge r.$~~
4. $p \to \epsilon$	$p \leftarrow \neg \boldsymbol{p}.$ $p \leftarrow \boldsymbol{q}.$ $p \leftarrow \boldsymbol{r}.$ $q \leftarrow \neg p.$ $q \leftarrow r.$ ~~$q \leftarrow \neg \boldsymbol{p} \wedge q.$~~ ~~$q \leftarrow q \wedge \boldsymbol{r}.$~~ $r \leftarrow \neg p.$ ~~$r \leftarrow \neg \boldsymbol{p} \wedge q.$~~ $r \leftarrow \neg q \wedge \boldsymbol{r}.$	9. $q \to pr$	$p \leftarrow q.$ $q \leftarrow p \wedge r.$ $r \leftarrow \neg p.$
5. $\epsilon \to r$	$p \leftarrow q.$ $p \leftarrow r.$ ~~$p \leftarrow \neg p \wedge \boldsymbol{q}.$~~ ~~$p \leftarrow \neg p \wedge \boldsymbol{r}.$~~ $q \leftarrow r.$ $q \leftarrow \neg p \wedge \boldsymbol{q}.$ ~~$q \leftarrow \neg p \wedge \boldsymbol{r}.$~~ $r \leftarrow \neg p.$ $r \leftarrow \neg q \wedge r.$		

Table 2.1: Execution trace of LFIT

3

δLFIT

δLFIT [65] is an end-to-end differentiable implementation of the LFIT algorithm. δLFIT takes a sequence of state transitions and outputs the logic program that best explains the transitions. Comparing to previous neural network methods that implement LFIT, δLFIT differs in that it views the LFIT problem as the classification problem. Namely, classifying whether a rule exists or doesn't exist within the given state transitions.

In this chapter, we will first investigate whether neural networks are capable of identifying logic programs. Different logic programs produce different transitions, but similarly the same logic program can produce different transitions depending on the initial state. Thus we want to understand whether (1) neural networks can identify different transitions that came from the same logic program, and (2) neural networks can identify transitions coming from different logic programs.

In the second half of this chapter, we will introduce the δLFIT algorithm. δLFIT takes a series of state transition as an input and outputs the logic program that explains the state transition.

However, before we are able to use neural networks to classify rules, we want to ensure that neural networks can properly classify state transitions. To do this, we propose a model [66] that is trained in such a way to learn a representation from which classification can be derived from.

In this model, we don't learn the representation directly. An optimal representation is obtained by training the model on a separate task, mainly the regression of the next state. Intuitively, by learning the regression on the next state, the NNs can focus on learning the abstract features of the particular system.

On a high level, we want to abstract the information represented in the logical space, into a more compact linear space. By abstracting and thus avoiding the need to deal with information in the lower level logical space, we are able to handle the fuzziness and ambiguity of the data. This model thus can be thought of as first encoding information in the logical space, into the representation space. We then perform the T_P operator, that will give us the next state, in the representation space. Once we obtain the next state in the representation space, we can then map it back into the logical space.

The model is defined as calculating the following

$$\vec{v}_{t+1} = f_{\text{decode}}(f_{\text{representation}}(M'_k, L_0) \times f_{\text{encode}}(M_k)^\top) \tag{3.1}$$

where M'_k can be equal or different from M_k, L_0 is an initial representation, $f_{\text{decode}} : \mathbb{R}^d \mapsto \{0,1\}^{|\mathcal{B}|}$, $f_{\text{representation}} : \{0,1\}^{|\mathcal{B}|\times k} \times \mathbb{R}^{d\times d} \mapsto \mathbb{R}^{d\times d}$, $f_{\text{encode}} : \{0,1\}^{|\mathcal{B}|\times k} \mapsto \mathbb{R}^d$, d is the dimension of the learned representation. All of f_{decode}, $f_{\text{representation}}$ and f_{encode} are differentiable, $\times$ represents the matrix-vector multiplication, and therefore this entire model is end-to-end differentiable.

Learning is performed by minimizing the mean squared loss defined as follows

$$\text{loss} = \frac{1}{|\mathcal{B}|}\sum_{i=1}^{|\mathcal{B}|}(v_{t+1,i} - y_i)^2$$

where y_i is the true label.

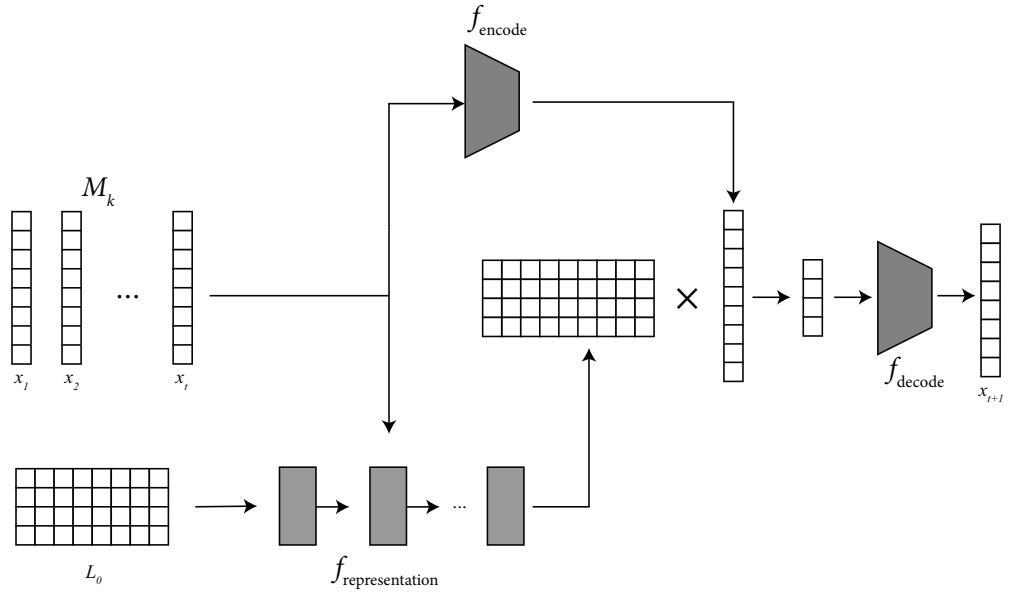


Figure 3.1: A visualization of the model

The f_{decode} function converts state vectors in the representation space, back into the logical space. The f_{encode} function does the opposite of converting state vectors in the logical space into the representation space.

The $f_{\text{representation}}$ function takes 2 parameters, the past k states that provides the abstract feature to extract from, and an initial representation that is the *a priori* knowledge. With these 2 parameters, the function outputs a matrix that is the representation of the dynamics of the system.

Implementation-wise, all of the above functions can be implemented with any non-linear function with parameters that can be trained by performing gradient descent. In our implementation, the f_{decode} and the f_{encode} functions are implemented with a multi-layer perceptron. The $f_{\text{representation}}$ function is implemented with LSTM (Long-Short Term Memory).

3.1.1 Experiments

To train the model, we used a randomly generated dataset. The model's goal is to extract features for *any* given system. The constraint of the system is such

that the system can be described by an NLP. Therefore, it is to our advantage that we can randomly generate a huge amount of data based on that constraint, and then train the model to extract common features of systems with such constraint.

Data generation works by first randomly generating an NLP, then generating state transitions that start from all possible initial states. However, if we purely randomly generate an NLP, we might not be able to generate *good quality* NLPs that allows the model to learn. Therefore, we limit ourselves to only generating NLPs with several properties. First, the body of each rules in the NLP should not be too long. We perform a random exponential cut-off for the length of the body, so there are cases where the body is long, but that should not be too often. Second, the state transitions generated from the NLP needs to have high variation. State transitions that are in the middle of an attractor are excluded from the training data. If there are too few state transitions from the NLP, then the NLP is also excluded from the training data.

During the experiment performed below, we generated 30,000 different NLPs. Then from each NLP, we obtained a maximum of 500 samples, each sample containing state transition for 10 timesteps. This gave us 150,000 samples to train the model.

Based on our experience, generating training data that has sufficient variance hugely affects the performance of the model. When we tried purely randomly generating NLPs and their corresponding states, we got transitions that are zero for most of the time or are constantly at the same state, and the model was unable to learn any useful features.

We tested the model on 4 LFIT benchmarks. These are the same benchmarks that were also used in [41] and [70].

We generated a series of 10 transitions from all possible initial states for each benchmark. The model is then asked to predict the next state based on these 10 transitions. We first ran all the benchmarks without providing any background knowledge, that is supplying the $\mathbf{0}$ matrix for L_0 in equation 3.1. Next, for predictions that are wrong by 1 or more variables, we obtained new L matrix by calculating $f_{\text{representation}}$ by supplying different state transitions than M_k, and then using that as L_0 for M_k to calculate the predictions again. The results are shown in table 3.1. We calculated the mean absolute error (MAE) as the metric

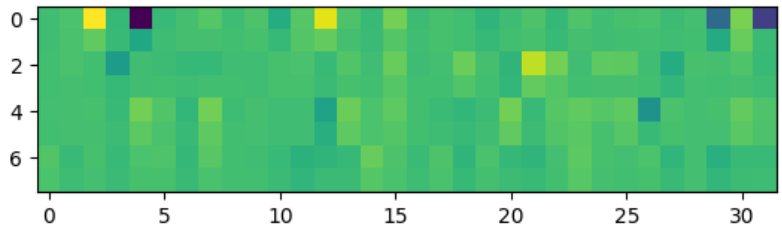
Method	Mammalian (10)	Fission (10)	Budding (12)	Arabidopsis (16)
Without L_0	0.224	0.063	0.218	0.146
With L_0	0.184	0.062	0.199	0.128
Fuzzy data (25%)	0.209	0.062	0.206	0.134
Fuzzy data (50%)	0.243	0.072	0.215	0.238
Error (10%)	0.223	0.081	0.198	0.157
Error (20%)	0.249	0.101	0.211	0.201
Error (30%)	0.287	0.129	0.227	0.230
Error (40%)	0.334	0.182	0.250	0.251
Error (50%)	0.379	0.250	0.288	0.263
Error (60%)	0.418	0.324	0.318	0.274
Error (70%)	0.435	0.363	0.341	0.280
Error (80%)	0.466	0.406	0.353	0.300
Error (90%)	0.490	0.469	0.355	0.317

Table 3.1: The MAE of the prediction performed by the model on 4 separate benchmarks.

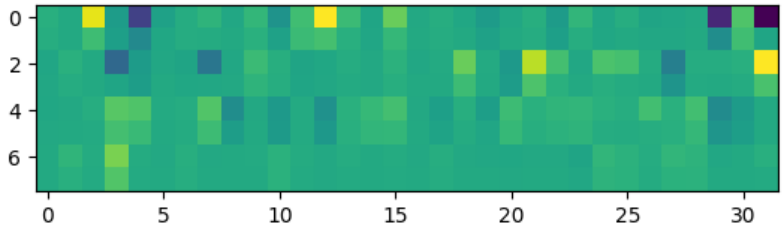
for accuracy. An MAE of 0.2 for a 10 variable benchmark means that the model predicted 2 of the 10 variables wrong. As can be seen from the results, we were able to improve the predictions when providing L_0, except in the case of the fission benchmark where the model was already doing very well.

Next, we added fuzziness to the data by mapping each element in the state vector $\{0, 1\} \mapsto [0, 1]$. When a particular variable is 1, the value is fuzzed into a range of $[0.5, 1]$, and when it is 0 it is mapped into the range $[0, 0.5]$. The results of this is indicated by the row fuzzy data (50%). For fuzzy data (25%), we mapped 1 to the range of $[0.75, 1]$ and 0 to $[0, 0.25]$. In this experiment, whenever the model made wrong predictions, we provided L_0 that is calculated based on discrete error-free data. This is done under the assumption that in a real world scenario, the prior knowledge provided is usually considered as a fact. Therefore we did not do any fuzziness test on L_0.

We then tested the model's ability to handle erroneous data. For 10%, we flipped 10% of the variable states that we provide as input to the model. As can be seen from table 3.1, by providing prior knowledge to the model we are able to maintain fairly high accuracy for data with errors up to 50%.



(a) Logic program representation A



(b) Logic program representation B

Figure 3.2: Two logic program representations learned from different state transitions from the fission benchmark

3.1.2 Discussion

In figure 3.2, we show 2 separate logic program representations learned from 2 distinct state transitions that came from the fission benchmark. We can see that both of them share very similar features, and can confirm that the model did indeed manage to extract some high level features based on the input state transitions. With this, we can be confident that neural networks have the ability to learn and classify logic programs.

3.2 Rule Classification

For δLFIT, the classes that it is trying to classify are the logical rules. Every combination of logical rules in a particular Herbrand base is assigned a class to which δLFIT can perform classification on.

To do this, a mapping from every logical rule has to be built. Given Herbrand base $\mathcal{B}$, if every single permutation of the form (2.3) has to be considered, the

possibilities are infinitely many. From the infinitely many rules, we first want to reduce them to a manageable set.

The following operations are defined

Definition 1 (Simplification of Rules) *A rule can be simplified according to the following operations:*

- $a \wedge a$ *is simplified to* a
- $\neg a \wedge \neg a$ *is simplified to* $\neg a$
- $a \wedge \neg a$ *and* $\neg a \wedge a$ *is simplified to* $\bot$

where a *is an atom.*

With these operations, we can define a minimal rule.

Definition 2 (Minimal Rule) *A rule is considered to be minimal, if its logical formula cannot be simplified further.*

Note that the ordering of the atoms within in the rule is insignificant in the above definition. Both $a \wedge b$ and $b \wedge a$ are equally minimal.

By limiting to only minimal rules, we now know that the length of a rule body is finite. In particular, the maximum length a rule can take is equal to $\|\mathcal{B}\|$ since the same atom cannot appear twice in the same rule. With this constraint, it is now possible to list out all possible rules a system can have.

Given an Herbrand base $\mathcal{B}$, we define a function $\tau(\mathcal{B})$ generates a finite ordered set that contains all possible minimal rules. In a classification scenario, we want to know which class maps to which rule. To ease this mapping, a deterministic approach is defined to map each rule to an index, which corresponds to a class.

Before explaining the approach to map the rules, first we will have to define several terms that we will be using throughout.

Definition 3 (Length of a rule) *The length of a rule $R \in \tau(\mathcal{B})$ is defined as $\|b(R)\|$.*

Definition 4 (Index of element in ordered set) *Let S be an ordered set, the index of element $e \in S$, is defined as $\sigma_S(e) = \|S_{<e}\|$, where $\sigma_S : S \mapsto \mathbb{N}$ and $S_{<e} = \{x \mid x < e, x \in S\}$.*

Definition 5 (Ordered Herbrand Base) *The ordered Herbrand base $\mathcal{B}_o$ contains the same elements as $\mathcal{B}$ except each element has an ordered relation $<$.*

The relation $<$ on $\mathcal{B}_o$ can be defined arbitrarily, but in most cases, the lexicographical ordering is the most convenient and conventional. Therefore this is the one that we will also be utilizing throughout this thesis.

Now, we will split all the rules in $\tau(\mathcal{B}_o)$ by their rule lengths. This will give us the basis to quickly and deterministically calculate the index of any given rule. Consider a set of rules $\tau_l(\mathcal{B}_o)$ which only contains rules of length less than or equal to l, where $\tau_l(\mathcal{B}_o) = \{R \mid \|b(R)\| \leq l, R \in \tau(\mathcal{B}_o)\} \subseteq \tau(\mathcal{B}_o)$. The number of rules in $\tau_l(\mathcal{B}_o)$ can be given by the following formula:

$$\|\tau_l(\mathcal{B}_o)\| = \begin{cases} 1 & \text{if } l = 0, \\ \|\tau_{l-1}(\mathcal{B}_o)\| + \binom{n}{l} \times 2^l & \text{if } l > 0. \end{cases} \tag{3.2}$$

where $n = \|\mathcal{B}_o\|$ is the number of elements in the Herbrand base and $\binom{n}{k}$ represents the binomial coefficient.

Next, consider the ordered set $\widetilde{\tau}_l(\mathcal{B}_o) = \{R \mid \|b(R)\| = l, R \in \tau(\mathcal{B}_o)\}$ containing all the rules R that are exactly of length l. The ordered relation for $\widetilde{\tau}_l$ is defined by first ordering the negation by marking the negative literals as 1s and positive literals as 0s, then next by ordering based on $\mathcal{B}_o$. By marking negative literals and positive literals to 1s and 0s, we can map each rule to a binary number, e.g. $\{a, b, \neg c\}$ maps to $(0, 0, 1)$, $\{\neg a, \neg b, c\}$ maps to $(1, 1, 0)$. regarding the left-most digit as the least significant bit and the right-most as the most significant bit, we can denote it as 100_2, which can be considered as 4 in decimal. Another example $\{\neg a, \neg b, c\}$ maps to $(1, 1, 0)$ which is 011_2 and can be considered as 3 in decimal. Next, we look at each atom in the rule and order them according to

$\mathcal{B}_o$, which in this case we choose the lexicographical ordering. In this relation, $\{a, b\} < \{a, c\} < \{\neg a, b\} < \{\neg b, c\} < \{\neg a, \neg c\}$.

With these tools, we can finally calculate the index for any arbitrary rule. We define the indexing function $\sigma_{\tau(\mathcal{B}_o)} : \tau(\mathcal{B}_o) \mapsto \mathbb{N}$. The index of a rule R is simply as follows:

$$\sigma_{\tau(\mathcal{B}_o)}(R) = \|\tau_{l-1}(\mathcal{B}_o)\| + \sigma_{\widetilde{\tau}_l(\mathcal{B}_o)}(R)$$

where $l = \|b(R)\|$ is the length of the rule R. The basic principle here is that rules are sorted first by their length, and then by their position within the same rule lengths. Thus, by adding up all the rules that are less than themselves, and then looking at the position within the same length, we would get the global position of the particular rule.

Example 3 *Consider an ordered Herbrand base $\mathcal{B}_o = \{a, b, c\}$, where the ordered relation is the alphabetical ordering. The full table of $\sigma_{\tau(\mathcal{B}_O)}(R)$ is listed as in table 3.2.*

3.3 Logic Program Encoding

Since neural networks only deal with matrices and vectors, we will have to encode logic programs into matrices. The encoding employed by δLFIT puts the rule head on the row, and all the minimal rules $\tau(\mathcal{B}_o)$ on the column.

For example, consider a system with the Herbrand base $\mathcal{B} = \{a, b\}$, and an NLP as follows:

$$\begin{aligned} a(t+1) &\leftarrow a(t) \wedge b(t) \\ a(t+1) &\leftarrow \neg b(t) \\ b(t+1) &\leftarrow \neg a(t) \end{aligned} \tag{3.3}$$

The above NLP can be encoded into the matrix below:

$$\begin{array}{c} \\ a \\ b \end{array} \begin{array}{c} \begin{array}{ccccccccc} \{\} & \{a\} & \{b\} & \{\neg a\} & \{\neg b\} & \{a, b\} & \{\neg a, b\} & \{a, \neg b\} & \{\neg a, \neg b\} \end{array} \\ \left[\begin{array}{ccccccccc} 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \end{array}$$

Note that heads with multiple rules like a, which are disjoins in the NLP semantics, can be encoded by marking their respective elements as 1, therefore in the a row there are 2 columns that are marked as 1. Therefore this is not a one-hot encoding.

3.4 Model

By viewing the LFIT problem as a classification problem, the loss can be calculated as the cross entropy of the correct labelling of each rule with regard to the predictions of the system. The model can then be trained by minimising the loss with stochastic gradient descent.

The model is defined by the number of variables in the system, i.e. the size of the Herbrand base. For example, by building out and training a model that uses 3 variables, the model will be able to provide predictions for any 3 variable logic programs. But in order to obtain predictions for 4 variable systems, a new model that is built with that in mind has to be retrained.

δLFIT consists of an LSTM and a feed-forward network. The LSTM transforms the input sequence into a feature vector, the feed-forward network then classifies and predicts the rules based on the feature vector. The model structure is depicted in figure 3.3.

The LSTM is trained by BPTT based on variable length, and the outputs

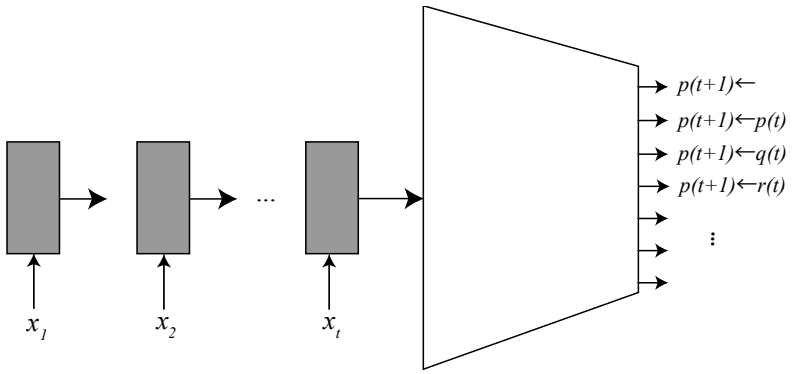


Figure 3.3: The δLFIT Architecture.

are the elements of $\tau(\mathcal{B}_O)$. Each output node is given an index and is mapped according to the indexing function σ.

3.5 Loss Function

For each rule r, given the state transition sequence S, δLFIT outputs the conditional probability:

$$p(r|S)$$

The aim is to ensure that each rule r that appears within the logic program P to have $p(r \mid S) = 1$ and those that don't appear in the logic program to have $p(r \mid S) = 0$. In other words, when all data Λ is sampled, (r, S) will hopefully match the actual label r. Therefore, the idea is to minimize the expected negative log likelihood:

$$\text{LFIT loss} = -\mathbb{E}_{(r,S)\sim\Lambda}(r \times \log p(r \mid S) + (1-r) \times \log(1 - p(r \mid S)))$$

An auxilary loss function, called the subsumption loss is also added to aid in training. The subsumption relation between different rules can be used to penalize the model as, assuming minimal NLP, subsumed rules should not appear in the prediction. For ever rule in $\tau(\mathcal{B})$ a softmax cross-entropy is added. The loss is defined as follows

$$\text{Subsumption Loss}_{\tau(\mathcal{B}_O)} = -\sum_{r\in\tau(\mathcal{B}_O)} (r \times \log p(r \mid S) + (1-r) \times \log(1 - p(r \mid S)))$$

3.6 Generating Training Data

Due to the generality of δLFIT, the model is very difficult and expensive to train as well. In order to be able to predict any arbitrary logic program, with the limitation of the Herbrand base, the training data will have to be sampled sufficiently across every logic program possibility. In particular, because δLFIT is essentially a classifier, datapoint for every class has to be provided.

Gathering good training data is usually the most important aspect in statistical machine learning [88]. In terms of the δLFIT task, we are attempting to solve a classification problem. Thus a training data for that allows identification for each separate rule is required. Unfortunately, this is very difficult to obtain from real world data. Therefore, the training data that we are going to use will be artificially generated.

For every rule $R \in \tau(\mathcal{B})$, we generate a corresponding training data set. Based on the input output of the model, this training data set consists of pairs of (T, P) where T is the state transitions and P is the corresponding logic program. Here, P is a logic program that contains the rule R, and is required that any other rules in P not subsume R.

To generate this training data set, first a logic program P_o is constructed. P_o must contain the rule R for which we are constructing the training data set for. Next, a random initial state is picked and a series of state transition T is generated based on P_o. Next, based on the state transition generated a logic program P_l is learned based on algorithm 1. If R is present in P_l, then the data pair (T, P_l) is accepted, otherwise it is rejected and a next initial state is picked. An extra step of running algorithm 1 is performed to ensure that the state transitions provided has sufficient information to learn the rule R.

The above process is repeated for every rule $R \in \tau(\mathcal{B})$. Therefore, for a 3-variable system we will have 81 sets of training data. For a 4-variable system, there will be 324 sets, etc.

3.7 Experiments

We first verify that δLFIT is capable of producing the expected logic programs when given a series of state transition. Next, we perform several experiments that focus mainly on the ability to handle noisy data and also erroneous data.

δLFIT is implemented in Tensorflow [1]. We prepared several experiments to verify the performance of δLFIT. As far as we are aware, there are no other works that perform similar tasks, therefore we were not able to do any comparison. Instead, we will show the precise results that we obtained in our experiments.

3.7.1 Hyperparameters

These hyperparameters were not tuned by a lot, mainly because the hyperparameters that we chose worked well for most experiments. The following describes some of the hyperparameters chosen:

- Batch size: 750.
- Gradient descent optimizer: Adam, learning rate is set to 0.01.
- All parameters are initialized randomly by the default initializer in Tensorflow r1.5.
- Dropout rate of 0.8.
- L_2-regularization for the parameters.
- Input is a series of 15 state transitions.

Other neural network specific hyperparameters varies from experiments to experiments, so we will describe them as we describe our experiments.

3.7.2 Experimental Methods

We generated 50,000 (T, P) pairs for each training data set. The model is then trained for 2 epochs on this data sets while attempting to minimize the loss. Each step, a mini-batch is sampled across all of these data sets randomly. This mini-batching should give the process a stochastic element and helps to escape local minima.

After 2 training epochs, δLFIT produces the predicted logic programs given a series of state transitions. To validate this result, we take the initial state from the series of state transitions that was given as input, and run the T_P operator on the predicted program. The generated state transitions is then compared with the original state transitions.

Once δLFIT has finished training, we get the probability of the rules. To turn it into a human-readable logic program, we just have to take for each $p(R) \geq \theta$, where θ is a threshold, use the reverse lookup $\sigma^{-1}_{\tau(\mathcal{B}_O)}(R)$ and combine them.

We tested δLFIT on 4 boolean networks taken from [86]. Two of them are 3-variable systems and one of them is a 5-variable system. We could only test on small systems currently, mainly due to memory constraints because larger systems require a larger neural network architecture. Another factor is that the training data generation takes too long to be practical beyond 5 variables.

For 3-variable systems, we used the following hyperparameters:

- LSTM hidden units: 50.
- Output of LSTM is a 6 dimension vector.
- Feed-forward neural network is 2 layers with 10,000 hidden neurons each.

Whereas for the 5-variable system, we used the following hyperparameters:

- LSTM hidden units: 500.
- Output of LSTM is a 10 dimension vector.
- Feed-forward neural network is 4 layers with 8,000, 5,000, 5,000 and 2,000 hidden neurons respectively.

First, we fed δLFIT discrete, error-free data. From the predicted logic program that was obtained from δLFIT, we attempted to reproduce the same state transition sequence. Then, we calculated the mean squared error (MSE) between the original input sequence and the generated sequence.

Next, we test δLFIT with fuzzy data. We map the values of the transition being fed into δLFIT from $1 \rightarrow [0.5, 1]$ and $0 \rightarrow [0, 0.5]$ randomly on a normal distribution. Note that both ranges include 0.5, this is done deliberately to test the model's robustness. Another thing to note is that δLFIT was not retrained specifically to deal with fuzziness. The same parameters that were used for the previous experiments were used for the fuzziness test as well.

The full results for the experiments we have performed is detailed in table 3.3. In general, we notice that δLFIT was not able to perform as well when the data is fuzzy, except for Raf, which has several attractors. We provide some analysis into why this is the case in section 3.8.

In this section, we will focus on the 3-node (a) and 5-node experiments in detail.

3-node (a)

This is a toy network that contains 3 nodes. The boolean network is shown as in figure 3.4, and its corresponding logic program is described as below:

$$
\begin{aligned}
v_1(t+1) &\leftarrow \neg v_1(t) \wedge \neg v_2(t).\\
v_1(t+1) &\leftarrow \neg v_1(t) \wedge \neg v_3(t).\\
v_2(t+1) &\leftarrow \neg v_1(t) \wedge v_2(t).\\
v_3(t+1) &\leftarrow \neg v_1(t).\\
v_3(t+1) &\leftarrow v_2(t).
\end{aligned}
$$

This network has two attractors and one steady state. We plugged every possible initial state (2^3 states) and produced a series of 15 state transitions for every initial state. These series are then fed into δLFIT to obtain the corresponding logic program. After δLFIT predicted the logic program, we then used the predicted logic program to generate a series of 15 state transitions. Those transitions are then compared with the original transitions.

In this particular task, the mean squared error (MSE) for all the state transitions were 0.095. More precisely, out of 8 possible series of state transitions (based on the initial state), there were 2 series of state transitions for which the predicted logic program couldn't reproduce accurately. For those that were ac-

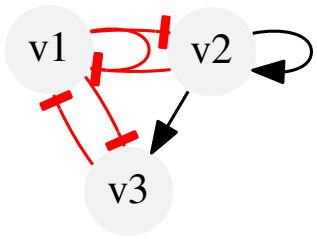


Figure 3.4: 3-node toy network with monotone edges and two attractors

curate, δLFIT predicted the following logic program:

$$
\begin{aligned}
v_1(t+1) &\leftarrow v_2(t).\\
v_1(t+1) &\leftarrow \neg v_1(t).\\
v_1(t+1) &\leftarrow \neg v_3(t).\\
v_2(t+1) &\leftarrow v_2(t).\\
v_2(t+1) &\leftarrow \neg v_1(t) \wedge v_3(t).\\
v_2(t+1) &\leftarrow v_1(t) \wedge \neg v_3(t).\\
v_3(t+1) &\leftarrow v_2(t).\\
v_3(t+1) &\leftarrow \neg v_1(t).\\
v_3(t+1) &\leftarrow \neg v_3(t).
\end{aligned}
$$

Notice that even though the predicted logic program is longer than the original logic program, none of the rules subsume each other. We attribute this to the subsumption loss that is added to the loss function.

On the other hand, out of those that weren't correct, one of them is the at-

tractor, where the state is always 011. The predicted logic program was:

$$
\begin{aligned}
v_1(t+1) &\leftarrow v_1(t).\\
v_2(t+1) &\leftarrow v_3(t).\\
v_3(t+1) &\leftarrow .\\
v_3(t+1) &\leftarrow \neg v_2(t).
\end{aligned}
$$

Here, we notice that all the rules regarding v_3 on the head is wrong. If the predicted logic program was able to create rules that made sure v_3 is true, either by $v_3(t+1) \leftarrow v_3(t)$, $v_3(t+1) \leftarrow v_2(t)$ or $v_3(t+1) \leftarrow \neg v_1(t)$, then the logic program will be correct.

Another incorrect prediction worth noting, is when the initial state was 111. The original series was $111 \to 001 \to 101 \to 000 \to 101 \to \ldots$. The predicted logic program was:

$$
\begin{aligned}
v_1(t+1) &\leftarrow v_2(t).\\
v_1(t+1) &\leftarrow \neg v_1(t).\\
v_1(t+1) &\leftarrow v_1(t) \wedge \neg v_3(t).\\
v_2(t+1) &\leftarrow \neg v_1(t) \wedge v_2(t).\\
v_2(t+1) &\leftarrow v_2(t) \wedge \neg v_3(t).\\
v_3(t+1) &\leftarrow \neg v_1(t).\\
v_3(t+1) &\leftarrow \neg v_3(t).
\end{aligned}
$$

This logic program produced the series $111 \to 100 \to 101 \to 000 \to 101 \to \ldots$. Note that before entering the attractor, the predicted logic program transitioned to 100 instead of 001.

When we fed the fuzzy state transitions, δLFIT got 3 out of the 8 sequences correct. The correct predicted logic program is as follow:

$$
\begin{aligned}
v_1(t+1) &\leftarrow v_2(t).\\
v_1(t+1) &\leftarrow \neg v_1(t).\\
v_3(t+1) &\leftarrow v_1(t).\\
v_3(t+1) &\leftarrow \neg v_3(t).
\end{aligned}
$$

Note that no rules for v_2 was predicted. This is because in these 3 sequences, the value for v_2 has always been 0.

We then look at one of the incorrect predictions. The given sequence was $001 \to 101 \to 000 \to 101 \to \ldots$. The predicted logic program was:

$$
\begin{aligned}
v_1(t+1) &\leftarrow v_2(t).\\
v_1(t+1) &\leftarrow \neg v_1(t).\\
v_1(t+1) &\leftarrow \neg v_3(t).\\
v_3(t+1) &\leftarrow \neg v_3(t).
\end{aligned}
$$

Based on this logic program, the generated sequence was $001 \to 100 \to 101 \to 000 \to 101 \to \ldots$. In this case, δLFIT missed the $v_3(t+1) \leftarrow v_1(t)$ rule which had been predicted for the other correct sequences.

5-NODE

This is a toy network that contains 5 nodes. The boolean network is shown as in figure 3.5, and its corresponding logic program is described as below:

$$
\begin{aligned}
v_1(t+1) &\leftarrow v_2(t) \wedge \neg v_3(t) \wedge \neg v_4(t) \wedge v_5(t).\\
v_1(t+1) &\leftarrow v_2(t) \wedge v_3(t) \wedge v_4(t) \wedge \neg v_5(t).\\
v_2(t+1) &\leftarrow v_2(t) \wedge v_3(t) \wedge v_4(t) \wedge \neg v_5(t).\\
v_3(t+1) &\leftarrow v_1(t) \wedge v_2(t) \wedge v_4(t).\\
v_3(t+1) &\leftarrow v_1(t) \wedge \neg v_2(t) \wedge \neg v_4(t).\\
v_4(t+1) &\leftarrow v_2(t) \wedge \neg v_3(t) \wedge \neg v_4(t) \wedge v_5(t).\\
v_4(t+1) &\leftarrow v_2(t) \wedge v_3(t) \wedge v_4(t) \wedge \neg v_5(t).\\
v_4(t+1) &\leftarrow \neg v_2(t) \wedge v_4(t).\\
v_5(t+1) &\leftarrow \neg v_1(t) \wedge v_2(t) \wedge v_3(t).
\end{aligned}
$$

This network has 3 steady states. Similar to the other experiments, We plugged every possible initial state (2^5 states) and produced a series of 15 state transitions for every initial state. These series are then fed into δLFIT to obtain the corresponding logic program. After δLFIT predicted the logic program, we then used the predicted logic program to generate a series of 15 state transitions. Those transitions are then compared with the original transitions.

For the accurate logic programs that δLFIT was able to predict, it looked like

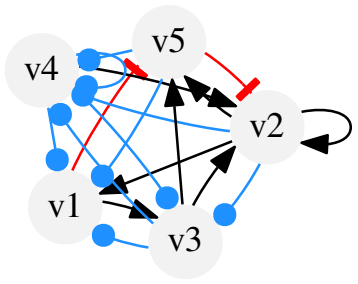


Figure 3.5: 5-node toy network with 3 steady states

this:

$$
\begin{aligned}
&v_1(t+1) \leftarrow v_2(t). && v_1(t+1) \leftarrow v_3(t).\\
&v_1(t+1) \leftarrow v_4(t). && v_1(t+1) \leftarrow v_5(t).\\
&v_2(t+1) \leftarrow v_3(t). && v_2(t+1) \leftarrow v_5(t).\\
&v_2(t+1) \leftarrow v_1(t) \wedge v_4(t). && v_2(t+1) \leftarrow \neg v_1(t) \wedge v_2(t).\\
&v_2(t+1) \leftarrow \neg v_1(t) \wedge v_3(t). && v_2(t+1) \leftarrow \neg v_2(t) \wedge v_3(t).\\
&v_2(t+1) \leftarrow \neg v_2(t) \wedge v_4(t). && v_2(t+1) \leftarrow v_2(t) \wedge \neg v_3(t).\\
&v_2(t+1) \leftarrow v_3(t) \wedge \neg v_3(t). && v_3(t+1) \leftarrow v_2(t).\\
&v_3(t+1) \leftarrow v_3(t). && v_3(t+1) \leftarrow \neg v_1(t) \wedge v_3(t).\\
&v_3(t+1) \leftarrow v_3(t) \wedge \neg v_4(t). && v_3(t+1) \leftarrow v_3(t) \wedge \neg v_5(t).\\
&v_4(t+1) \leftarrow \neg v_2(t) \wedge v_4(t). && v_4(t+1) \leftarrow v_2(t) \wedge \neg v_3(t).\\
&v_4(t+1) \leftarrow v_2(t) \wedge \neg v_5(t). && v_5(t+1) \leftarrow v_3(t).\\
&v_5(t+1) \leftarrow v_1(t) \wedge v_4(t). && v_5(t+1) \leftarrow \neg v_3(t) \wedge v_4(t).
\end{aligned}
$$

Eventhough δLFIT produced much more rules than the original logic program, every state transition tested was correct.

On the other hand, for those that were incorrect, we mostly observed that the state transition sequence that were given mostly involved steady states.

Next, when we fed fuzzy transitions, δLFIT predicted logic programs that were correct, but much shorter than when discrete error-free transitions were given. The correctly predicted logic program is as follows:

$$
\begin{aligned}
&v_1(t+1) \leftarrow v_2(t).\\
&v_1(t+1) \leftarrow v_3(t).\\
&v_1(t+1) \leftarrow v_4(t).\\
&v_1(t+1) \leftarrow v_5(t).\\
&v_2(t+1) \leftarrow v_5(t).\\
&v_3(t+1) \leftarrow v_2(t).\\
&v_3(t+1) \leftarrow v_4(t).\\
&v_3(t+1) \leftarrow \neg v_1(t) \wedge v_3(t).\\
&v_4(t+1) \leftarrow v_3(t) \wedge v_5(t).\\
&v_4(t+1) \leftarrow \neg v_1(t) \wedge v_4(t).\\
&v_5(t+1) \leftarrow \neg v_3(t) \wedge v_4(t).
\end{aligned}
$$

3.8 Discussion

Our aim is to utilize neural network's ability to handle ambiguous data to produce models that fully explain the system that we are trying to observe. While we were able to show that when passing discrete data, δLFIT was able to perform very well. We were not able to get δLFIT to handle ambiguous and fuzzy data as well. We attribute this to the fact that we did not train our neural networks to learn to handle fuzziness in the data. We speculate that adding fuzziness to the training data should be able to help δLFIT generalize better to ambiguity.

We were also not able to scale the neural network beyond 5 variables. Mainly because the amount of expressiveness required beyond that point far exceeds the computation capability we currently possess. Aside from having more computing power, we would also like to think that improvements to the area of multiclass multilabel tasks could help in this area.

Also, another thing to note is that on all 4 of the experiments we have per-

formed, δLFIT was not able to predict the accurate logic program when the given state transition sequence is within an attractor. However, we can see that the performance improved by a little bit when fuzzy data was given for Raf, which has more attractors than the other boolean networks. We attribute this to the low variance nature of an attractor as an input. The neural network was not able to classify properly when given the same uniform input. However, when fuzzy data was given, there was fluctuation within the state transitions and therefore led to improvements in performance.

	$\tau(\mathcal{B}_o)$
$l = 0$	$0 \to \{\}$
$l = 1$	$1 \to \{a\}$ $2 \to \{b\}$ $3 \to \{c\}$ $4 \to \{\neg a\}$ $5 \to \{\neg b\}$ $6 \to \{\neg c\}$
$l = 2$	$7 \to \{a, b\}$ $8 \to \{a, c\}$ $9 \to \{b, c\}$ $10 \to \{\neg a, b\}$ $11 \to \{\neg a, c\}$ $12 \to \{\neg b, c\}$ $13 \to \{a, \neg b\}$ $14 \to \{a, \neg c\}$ $15 \to \{b, \neg c\}$ $16 \to \{\neg a, \neg b\}$ $17 \to \{\neg a, \neg c\}$ $18 \to \{\neg b, \neg c\}$
$l = 3$	$19 \to \{a, b, c\}$ $20 \to \{\neg a, b, c\}$ $21 \to \{a, \neg b, c\}$ $22 \to \{\neg a, \neg b, c\}$ $23 \to \{a, b, \neg c\}$ $24 \to \{\neg a, b, \neg c\}$ $25 \to \{a, \neg b, \neg c\}$ $26 \to \{\neg a, \neg b, \neg c\}$

Table 3.2: Full index of rules for $\mathcal{B}_o = \{a, b, c\}$

Boolean Network	MSE (Discrete)	MSE (Fuzzy)
3-node (a)	0.095	0.137
3-node (b)	0.054	0.057
Raf	0.253	0.217
5-node	0.142	0.147

Table 3.3: The MSE for the state transitions generated by the predicted logic programs.

4

δLFIT+

We will introduce our third contribution of this thesis in this chapter. δLFIT+ is an improvement upon δLFIT's main shortcoming, scalability. δLFIT was only able to deal with systems up to 5 variables. This is due to the combinatorial explosion problem. Recall that δLFIT assigns an output node to each logical rule. Even considering only minimal rules, the number of output nodes scale by $n3^n$. Most NSAI techniques in fact, suffer from the combinatorial explosion problem one way or the other.

In δLFIT the issue is particularly severe because of the way the model learns. It is not just that the number of output nodes increase, that leads to the number of parameters subsequently memory space increasing. δLFIT is a classification algorithm at its heart, the number of classes to classify scales exponentially, therefore the amount of training data required for convergence also scales exponentially. The δLFIT method is fortunate in that training data is generated and not obtained through other means, however more data means more computational time required.

In broader terms, when we looked at various other NSAI techniques, we found

that symmetries or invariants that exist in the symbolic world are woefully under-appreciated. There are some symmetries that have been implicitly built in as part of the model, like in δLFIT, considering minimal rules where the position of the atoms within the rules are pre-defined, is one such symmetry. But most symmetries were not actively exploited.

We demonstrate our technique to exploit invariance in the symbolic world, in the field of NSAI. In hopes that our techniques will inspire other works, which may address the NSAI scalability problems.

4.1 Problems with δLFIT

Before introducing the δLFIT+, we would like to rehash the issues facing δLFIT which inspired this work.

One of the largest issues, as noted above, is the scalability issue. In particular, we faced issue trying to instantiate a model with 7 variables, the GPU that we have been using had 12GB VRAM yet it was insufficient. Of course it would have been possible to train such a model if we had access to a huge cluster of compute, however the model itself had other flaws that such an endeavour is possibly not worth.

Another issue is with regards to symbolic invariances. While we have taken account of the invariance in terms of the positioning of atoms within logical rules, it is still insufficient. In particular, δLFIT is sensitive to the ordering of sequence in the input data, which means that in order to be able to train properly it will be ncessary to provide all different permutations as training data. As an example, consider the following input

$$1 : (0, 0, 1) \rightarrow (1, 1, 0) \rightarrow (0, 0, 1)$$
$$2 : (1, 1, 0) \rightarrow (0, 0, 1) \rightarrow (1, 1, 0)$$

it should be immediately obvious that sequence 1 and sequence 2 are both equivalent. However, due to the way neural networks work, the hidden state of the LSTM in δLFIT will be different in both cases and thus it will affect the prediction.

Finally, δLFIT uses an LSTM, which takes a sequence as input data. In a

dynamic system, it is possible for certain states to never appear in a sequence of transitions for certain initial states. As an example, consider the following NLP

$$\begin{aligned} p(t+1) &\leftarrow q(t). \\ q(t+1) &\leftarrow p(t) \wedge r(t). \\ r(t+1) &\leftarrow \neg p(t). \end{aligned} \tag{4.1}$$

This NLP has 2 disjunct sequence of transitions. The first, starting with pqr as the initial state, the transitions are as follows

$$pqr \rightarrow pq \rightarrow p \rightarrow \epsilon \rightarrow r \rightarrow r \rightarrow \ldots$$

Notice that not all state configurations appear. Those states appear in the following sequence

$$qr \rightarrow pr \rightarrow q \rightarrow pr \rightarrow \ldots$$

These 2 sequences will never cross over. Since δLFIT can only take one sequence as input, it will be ignorant of the other sequence and thus will have to produce predictions only based on the first sequence. This will, no doubt, penalize the performance of the model.

4.2 Input Sequence Invariance

First, we will introduce an improvement we've made that will solve both the input sequence being disjoint issue and one of the symbolic variance issues. We will restructure the inputs such that it is no longer a continuous sequence, and instead it will be a set of state transitions. This is much more in-line with the symbolic LFIT algorithms, and also allow the model to get information about disjoint sequences.

In δLFIT, assuming the sequence length as l_s, the dimension of the input to the LSTM was $n \times l_s$, where n is the number of variables or the size of the Herbrand base. This has some implications on the model itself. First, it was unable to convert or learn a much more suitable representation for use with the input states. Next, the input itself scaled by a factor of n, which contributes to the overall memory usage.

Previous State	$\theta_p = 0$	$\theta_p = 1$
ϵ	0	8
p	1	9
q	2	10
pq	3	11
r	4	12
pr	5	13
qr	6	14
pqr	7	15

Table 4.1: Full state input for a logic program with $\mathcal{B} = \{p, q, r\}$

4.2.1 Encoding States

In δLFIT+, we use a much more compact form of input. In the synchronous 1-step dynamic system setting, we are generally only interested on a particular variable's state, 1 or 0, provided a prior state. For example, the LFIT algorithm described in section 2.2.1 only learns from transitions where in the next state the variable is 0. Therefore, in δLFIT+, we propose an encoding of a state transition for a particular variable into a number, which will then be mapped to a learnable embedding.

Given a set of state transitions E, and the variable of interest x, we define $E_x = \{(I, \theta_x) \mid I \in 2^{\mathcal{B}}\}$, where $\mathcal{B}$ is the Herbrand base. $\theta_x = 1$ when $x \in T_P(I)$ and $\theta_x = 0$ when $x \notin T_P(I)$. Therefore, E_x will contain all positive examples of x with $(I, 1)$ and all negative examples of x with $(I, 0)$. Next, we will define a function $\zeta_0 : 2^{\mathcal{B}} \mapsto \mathbb{N}$ that maps I to a natural number. We first take I and represent it as a vector $v \in \{0, 1\}^n$. Each element in v is 0 if it is not in the interpretation and 1 if it is. Note however, that the index position of each atom within v is order-dependent. This can then be interpreted as a binary number.

One state transition of (I, θ_x) can then be mapped to a number with the following function

$$\zeta_1(I, \theta_x) = (\theta_x + 1) \times \zeta_0(I) \tag{4.2}$$

Thus, the input to δLFIT+ is a set of $\{\zeta_1(\zeta_0(I), \theta_x) \mid (I, \theta_x) \in E_x\}$. An example for the list of possible inputs are shown in table 4.1.

Consider the example NLP in (4.1). The full state transition input for p in

this NLP will look like $E'_p = \{15, 11, 1, 0, 4, 14, 5, 10\}$, which corresponds to $E_p = \{(pqr, 1), (pq, 1), (p, 0), (\epsilon, 0), (r, 0), (qr, 1), (pr, 0), (q, 1)\}$. Note that the full $\|E_x\|$ will always be 2^n as each of the combination of $\{p, q, r\}$ should appear.

Based on this, we know for an n-variable system, there will be 2^n different prior states. Each state can transition to 2 different possibilities. And since each variable is independent, there are n different combinations of these. Therefore for an n-variable system, the input space of the δLFIT+ algorithm looks like this

$$(2^{2^n})^n \tag{4.3}$$

As can be imagined, even for a small n this is quickly intractable. The implication for this is that, even for small values of n, the input space can grow really huge.

n	Input Space	Number
2	$(2^{2^2})^2$	256
3	$(2^{2^3})^3$	1.68×10^7
4	$(2^{2^4})^4$	1.84×10^{19}
5	$(2^{2^5})^5$	1.46×10^{48}
6	$(2^{2^6})^6$	3.94×10^{115}

Table 4.2: The input space for each n

Seeing that the number of atoms in the universe is estimated to be 10^{80}, we cross that threshold by $n = 6$. Thus it is not possible to collect every possible logic program for the training data, and instead a good sampling method is required. We will discuss our random sampling method in section 4.9.

4.2.2 State Ordering Invariance

Now we would like to input the set E'_p obtained above into the model. Conventionally, RNNs are utilized for processing sequences. Recent works however heavily prefer the transformer model [91]. The transformer, however is not permutation invariant, it does not lend well to the notion that the input is a set. As described previously, this will require us to train the model with every single permutation, which increases the training time.

Therefore, we will instead use the set transformer. In addition to that, instead of passing the state numbers directly into the input of set transformer, we will pass it through with an embedding. This will allow the model to learn a much more suitable representation space for the transitions. Also recall that, in (2.8), the set transformer computes the self-attention for the inputs. In particular, based on (2.5), QK^T is the pairwise relationship between the matrix Q and K. Since in set transformer, referring to both (2.9) and (2.10), $Q = K = X$, this will compute the pairwise relationship between each of the state transition that is given to the model.

4.3 Reducing Number of Output Nodes

Next, we look into the scalability issue in δLFIT. In δLFIT, every possible minimal rule is mapped to a single output node. This means that the size of the neural network scales directly to the number of possible minimal rules. These possible minimal rules in turn scales exponentially to the number of variables in the Herbrand base. This means that the size of the neural network scales exponentially with the number of variables. As explained, this was already an issue at a number of variables as lowly as 6. Most real world applications however, range from 10 variables, and up to 100k for genes. δLFIT is nowhere near in this regard.

4.3.1 Reuse by Rule Head

Recall that the number of output nodes for δLFIT is $n3^n$. The first thing we can do is get rid of the factor n. The n to the left in $n3^n$ represents the head of the rule. We can move from asking δLFIT to also predict which rule head it should output, to telling the model which rule head we are currently interested in. This means that instead of assigning each combination of the rule head and rule body to the output nodes, now the output nodes only need to be the possibilities of all rule bodies.

Nodes	0	1	2	3	4	5
$l=0$	$\{\}$	-	-	-	-	-
$l=1$	$\{a\}$	$\{b\}$	$\{c\}$	$\{\neg a\}$	$\{\neg b\}$	$\{\neg c\}$
$l=2$	$\{a,b\}$	$\{a,c\}$	$\{b,c\}$	$\{\neg a,b\}$	$\{\neg a,c\}$	$\{a,\neg b\}$
$l=3$	$\{a,b,c\}$	$\{\neg a,b,c\}$	$\{a,\neg b,c\}$	$\{\neg a,\neg b,c\}$	$\{a,b,\neg c\}$	$\{\neg a,b,\neg c\}$
Nodes	6	7	8	9	10	11
$l=0$	-	-	-	-	-	-
$l=1$	-	-	-	-	-	-
$l=2$	$\{\neg b,c\}$	$\{a,\neg c\}$	$\{b,\neg c\}$	$\{\neg a,\neg b\}$	$\{\neg a,\neg c\}$	$\{\neg b,\neg c\}$
$l=3$	$\{\neg a,b,\neg c\}$	$\{a,\neg b,\neg c\}$	$\{\neg a,\neg b,\neg c\}$	-	-	-

Table 4.3: Mapping of output nodes to rule bodies with various lengths

4.3.2 Reuse by Rule Length

However this is still 3^n output nodes, which is still exponential. Here, we devise another strategy to reduce the number of outputs. Observing that the next logical partitioning for the rule bodies are the length of the rule, we will again move the length of the rule from the outputs to the input, allowing us to specify what the outputs mean by controlling the input. In particular, we can specify say rule length to be 1 in the input, and then we interpret the output for the respective rules of length 1.

Recall again table 3.2, in which we list all the rules and their lengths separately. By noting that $l = 1$ containing 6 rules, $l = 2$ containing 12 rules, and $l = 3$ containing 8 rules, we can observe that the maximum number of output nodes we require is 12 nodes. This is less than half of the 27 required if without the head, and far fewer than the 81 required for δLFIT. We show the mapping of the output nodes to the corresponding rule in table 4.3.

Output node 0 corresponds to 3 different rules depending on l, which are $\{\}$, $\{a\}$, $\{a, b\}$ and $\{a, b, c\}$ respectively. As can be seen, not all output nodes are assigned a rule body. For those without an assignment, we will just ignore them in the output.

To figure out the number of output nodes required for different number of variables in the Herbrand base, first consider the set $\widetilde{\tau}_l(\mathcal{B}_o)$ which contains all rules that are equal to length l. From (3.2), the cardinality of $\widetilde{\tau}_l(\mathcal{B}_o)$ can be calculated from the following

$$\|\widetilde{\tau}_l(\mathcal{B}_o)\| = \binom{n}{l} \times 2^l \tag{4.4}$$

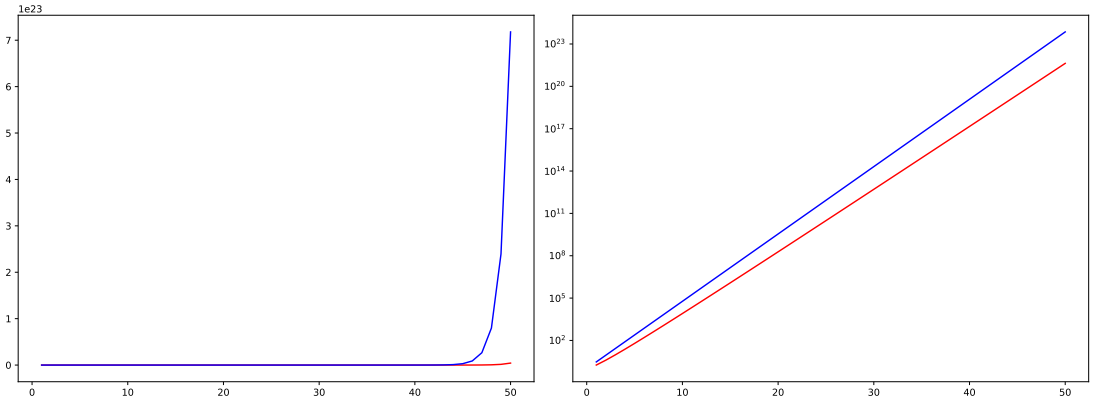


Figure 4.1: Plot of 3^n vs $\binom{n}{n/2} \times 2^{n/2}$, left graph being linear scale and right graph being logarithmic scale

Thus the number of output nodes is obtainable based on the following equation

$$n_o = \max\{\|\widetilde{\tau}_l(\mathcal{B}_o)\| : l = 0 \dots \|\mathcal{B}\|\}$$

Given that $\binom{n}{l}$ is the largest when $l = \lceil \frac{n}{2} \rceil$, assuming n is even to ease the calculation, we can rewrite the equation in (4.4) into the following

$$\binom{n}{n/2} \times 2^{\frac{n}{2}} \tag{4.5}$$

As n approaches infinity, we can see that $\lim_{n\to\infty} \frac{2^{n/2}}{\binom{n}{n/2}} = 0$, therefore we know that the binomial term dominates and the number of output node is scaling by factors of $\binom{n}{n/2}$. It is difficult to calculate the upper bound for this, but since $\sum_k \binom{n}{k} = 2^n$, therefore we can say that worst case this will be $O(2^n)$. The characteristics of this is still exponential, but it is much less than $O(3^n)$ of δLFIT. In practice though, the difference gained is substantial. Figure 4.1 shows the the number of output nodes for δLFIT and δLFIT+. It can be observed that even though in logarithmic scale, the new method only trails the old method by a little bit, in the linear scale graph we can see that the gains are actually substantial, as there is at least 1 or 2 order of magnitude difference.

4.4 Subsumed Label Smoothing

By introducing the rule length sharing technique, we face another problem. If we only consider minimal logic programs, the majority of the dataset will only contain negative labels. To illustrate this, consider again the NLP in (4.1). This NLP does not contain any rules with length of 3. Therefore, there are no positive labels for length 3. In fact, this will happen more frequently as the number of variables increase, as such we will have a dataset with largely negative labels and almost no positive labels.

This is obviously problematic, as if more than 80% of the dataset is just 0s, the neural network could just memorize the output as 0 and still achieve 80% accuracy. Since this is just an instance of overfitting, we will utilize a technique, known as label smoothing, to attempt to regularize the neural network and avoid this issue.

In δLFIT there was an auxilary loss function that penalizes the model when outputting subsumed rules, here instead we are encouraging subsumed rules. This is due to the different characteristics in these models. The output prediction of δLFIT are treated as is, and thus it will be more beneficial if subsumed rules are not predicted in the first place. δLFIT+ however does not have the picture of the full output, as the model only goes length by length. Therefore it will not be beneficial for the model to be aware of subsumption and be penalized for nothing from its perspective.

We apply label smoothing by randomly sampling subsumed labels at each mini-batch. It is quite redundant to apply smoothing to every rule that is subsumed, since if we have a rule at $l = 1$, about half of the rules at $l = 3$ will be subsumed. This will provide no immediate insight for the neural network.

To compute the smoothing to apply for each label, we first have to compute all the rules that are subsumed by the label. Assume that we have a label that is rule length of 2, we know that there will be no rule for which $l = 1$ will be subsumed by the label. As this is also problematic, we also extend it such that rules that subsume the labels are also qualified. A matrix of $3^n \times 3^n$ is constructed, with each row and column representing a rule body, and the matrix is filled with 0 being not subsumed and 1 being subsumed. For ease of computation, we define $\{\}$ as being subsumed by all rules. An example of the matrix is

shown as follows for $n = 2$

$$\begin{array}{c} \begin{array}{ccccccccc} \{\} & \{a\} & \{b\} & \{\neg a\} & \{\neg b\} & \{a,b\} & \{\neg a,b\} & \{a,\neg b\} & \{\neg a,\neg b\} \end{array} \\ \left(\begin{array}{ccccccccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \end{array}\right) \begin{array}{l} \{\} \\ \{a\} \\ \{b\} \\ \{\neg a\} \\ \{\neg b\} \\ \{a,b\} \\ \{\neg a,b\} \\ \{a,\neg b\} \\ \{\neg a,\neg b\} \end{array} \end{array}$$

An immediately observable pattern is that, aside from $\{\}$, every rows and columns add up to 4, which is half of 2^3.

As an example, suppose we have an NLP as in (3.3). To demonstrate that we can handle cases with more than 2 rules per head, we will investigate the case with a and $l = 2$. The label in this case is $\{a, b\}$ and $\{\neg b\}$. We will first gather columns that are $l = 2$, which is 4 of the right columns. We then look up the $\{a, b\}$ and $\{\neg b\}$ rows, to get the following matrix

$$\begin{array}{c} \begin{array}{cccc} \{a,b\} & \{\neg a,b\} & \{a,\neg b\} & \{\neg a,\neg b\} \end{array} \\ \left(\begin{array}{cccc} 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \end{array}\right) \begin{array}{l} \{\neg b\} \\ \{a,b\} \end{array} \end{array}$$

We then add up the columns to get a vector of $(1, 0, 1, 1)$ which can now be used to apply the subsumption factor. Let μ be the subsumption factor, the target labels for a and $l = 2$ was initially just $(1, 0, 0, 0)$, after applying the subsumption factor, the target will now look like $(1 + \mu, 0, \mu, \mu)$. In (3.3), we can also see that for b and $l = 2$, the target label is $(0, 0, 0, 0)$ which is the problematic part. However by applying the subsumption factor, the target label will now be $(0, \mu, 0, \mu)$.

4.5 Label Imbalance

If we look again at table 4.3, it is clear that some nodes are used more than others. For example, node 0 is used 4 times, nodes 1 through 5 is used 3 times,

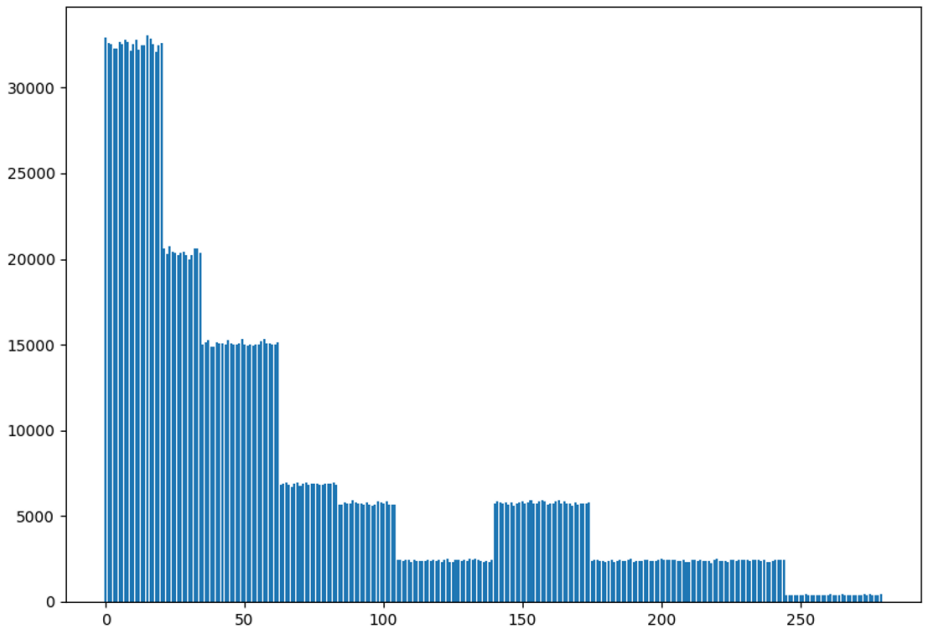


Figure 4.2: The imbalance in label distribution in the training data

while nodes 9 to 11 are only used once. The imbalance in the assignation is also apparent in the training data.

The distribution of the positive labels within the training data is shown in figure 4.2. To counteract this imbalance, we apply some weights to the class while training. The weight are the ratio of negative examples to positive examples.

The weighted loss function can be calculated as

$$\frac{1}{C \cdot N} \sum_{c=1}^{C} \sum_{n=1}^{N} -w_{n,c}[p_c y_{n,c} \cdot \log \sigma(x_{n,c}) + (1 - y_{n,c}) \cdot \log(1 - \sigma(x_{n,c}))]$$

where N is the number of labels in the training data, C is the total number of rules or classes, and p_c is the positive weight applied to each rule. p_c can be ob-

tained by

$$p_c = \frac{N_c^n}{N_c^p}$$

where N_c^n is the number of negative examples for class c and N_c^p is the number of positive examples for class c.

4.6 Network Architecture

The overall architecture is as depicted in figure 4.3. The neural network takes as input a set of state transitions, the variable index and the length of the rules to produce predictions for. The set of state transitions is as described in section 4.2. Variable index is an integer going from $0 \dots n-1$. While rule length is an integer going from $0 \dots n$.

Each input goes through a layer of trainable embedding, which converts the integers into a learnable representation space. Integer inputs are first converted into a one-hot vector, which is then multiplied by a learnable matrix which will provide the embedded vector, followed by a ReLU activation [4] and a LayerNorm

$$\text{Embedding}(x) = \text{LayerNorm}(\text{ReLU}(\text{Onehot}(x) \cdot M_e))$$

where $\text{Onehot}(x)$ converts the integer x into a one-hot vector and M_e is the matrix for the embedding space.

Next, both variable index and rule length are put through to a combination of a feed forward network and an "Add & Norm" layer. These layers behave like a residue layer, allowing the neural network to learn a much more suitable representation for each of the embeddings, while also retaining some of the representations in the embedding itself. This layer can be obtained by the following

$$\text{ResidualFF}(x) = \text{LayerNorm}(\text{rFF}(x) + x)$$

Then, the results computed from the variable index and the output of the set transformer based on the set of state transitions are concatenated into a singular vector. This vector is then passed through a residual feed forward layer.

Algorithm 4: δLFIT+ algorithm to output NLP P

Inputs : a set of atoms $\mathcal{B}$, set of state transitions $E \subseteq 2^{\mathcal{B}} \times 2^{\mathcal{B}}$
Output: an NLP P
$P := \emptyset$;
foreach $x \in \mathcal{B}$ **do**
  $E := \{(I, \theta_x) \mid (I, J) \in E, \theta_x = \{0, 1\}\}$;
  $E_x := \{\zeta_1(\zeta_0(I), \theta_x) \mid (I, \theta_x) \in E_x\}$;
  $R_x := \emptyset$;
  for l *in* $0 \ldots \|\mathcal{B}\|$ **do**
    $R_{x,l} := \delta\text{LFIT+}(E_x, x, l)$;
    foreach $R \in R_{x,l}$ **do**
      $R_x =$ `Specialize` (R_x, R);
    end
  end
  foreach $R \in R_x$ **do**
    $P =$ `Specialize` (P, R);
  end
end

Next, this concatenated vector and the embedding obtained from the rule length input are applied through a multi-head attention layer. With the query being the concatenated vector and the embedding from the rule length being the key and value self-attention. This allows the network to compute the pair-wise element relationship between the rule length and the obtained vector, and to attend to the components that are crucial for predicting the rules.

Following the multi-head attention, the resulting vector is passed through 2 layers of residual feed-forward network. Finally, the result is passed through a signmoid activation layer, and the probability for each rule is obtained.

4.7 The Algorithm

Since one inference pass will not generate the logic program, we will need to run the inference pass of the model multiple times in order to obtain the logic program. The algorithm for this is described in algorithm 4.

Given a set of variables $\mathcal{B}$ and a set of state transitions E, the algorithm outputs an NLP P. First, the algorithm initializes P into the empty set. There are no rules at first. Then, for each variable in the Herbrand base, the algorithm predicts the rules that the specific variable is the rule head.

At each variable, we convert the set of state transitions in the form of $\{(I, J) \mid I, J \in 2^{\mathcal{B}}\}$ to the form of $\{(I, \theta_x)\}$. This method is described in detail in section 4.2.1. We first convert the transition (I, J) into (I, θ_x) where θ_x is whether or not x is present in the next state J.

Then, from 0 to $\|\mathcal{B}\|$, which is the maximum rule length for minimal rules, we will probe δLFIT+ model to ask for predictions. δLFIT+ will return the predictions in terms of probability for each of the rules in their respective lengths. These predictions are then sorted by model confidence score and the top k rules are selected.

These rules that are selected by δLFIT+ are then added to the set of rules that have x being the head of the rule. This is done all while maintaining R_x to be minimal. Since R_x is a set of rules, which is equivalent to a logic program, we can use the algorithm described in algorithm 1 to perform the specialization.

Finally, after we've finished iterating over all lengths, we will incorporate the rules into the logic program P. Once again, we will add the rules one by one, all while maintaining the logic program to be minimal.

4.8 Recovering the Logic Program

As the output obtained from the δLFIT+ is a vector of probabilities, with each column representing the probability of the rule for the respective column, we will need a way to restore this into a logical rule. The indexes are in numeric form, so we are looking for an algorithm to map $\mathbb{N} \mapsto R$. The reverse has already been described in section 3.3, which maps logical rules to indexes.

Here, we will describe an algorithm that does an inverse lookup for the index to logical rule. This algorithm, which we name it $\sigma^{-1}_{\tau(\mathcal{B}_O)}(R)$, performs several calculations to figure out the length of the rule and the exact composition of the rule body.

The algorithm is described in algorithm 5. We first check if the index is larger than 0. $x = 0$ is the trivial case of $R = \{\}$. We then have to find the l length of the rule. The length of the rule can be found by iterating through all the lengths, finding the first index of that particular length, and checking if the first index of l is larger than x. If it is, then we know that the length of the rule is $l - 1$.

Once we know the length of the rule, we want to know the position that rule lies in within the same length. This allows us to find the combination of atoms in the body, and whether they are positive or negative literals. We first initialize the rule to be the first rule in its length. This will just be all positive literals in the order that $\mathcal{B}_O$ is in. For example, the first rule for $l = 2$ will be $\{a, b\}$.

Next, we increment the rule one by one up until we get to the index. The IncrementRule function takes care of figuring out which variables to go next. The algorithm for IncrementRule is described in algorithm 6.

Finally, we will assign the positive and negatives to the variables based on the index and their position. This can be known by doing a bitwise & operation, to figure out if the bit at position i is 0 or 1. If it is 1, then we want to negate the variable in that position.

Once this is done, we will have the rule body.

4.8.1 Incrementing a Rule

In this section, we will describe the algorithm that allows us to obtain the combination of atoms for a particular rule index. This algorithm is described in algorithm 6.

First we will check if the position of the rule l to increment is the same as the length of the Herbrand base, which is the trivial case. In this case, the rule will just contain all of the variables in their specific ordering and there is nothing to increment to.

Then we attempt to increment the variable in the position l. If the index of the incremented variable exceeds the number of variables in the Herbrand base, we will first attempt to increment the variable at position $l - 1$. Then, note that

within the rule, the ordering of the variable is strictly inline with the ordering in $\mathcal{B}_O$. Therefore, the variable index in position l will never be smaller than the variable index in position $l-1$. Therefore, once we incremented the variable at $l-1$, we can set the variable at l to be the same as the variable at $l-1$, then increment the variable at l.

Otherwise, we will set the variable at l to be the next variable of itself and return.

4.9 Training Data Generation

Similar to δLFIT, due to the generality of δLFIT+, it is infeasible to obtain sufficient data from the real world and treat it as training data. Fortunately, the data that we require are fairly structured and known. In particular, in the LFIT domain, valid NLPs are known ahead of time. If we also limit ourselves to the deterministic, single-step semantics, then it is obvious that it will be possible to generate all data for use with training.

The training data consists of the set of state transitions, being the input and the set of rules which is the logic program, being the target. Given $E = \{(I, T_P(I)) \mid I \in \mathbb{N}\}$ where P is the logic program, the training dataset is $\{(E, P)\}$ for many different E and P, ensuring that every P is consistent with their respective E.

We will describe the data generation for a single datapoint in the following. To generate the whole training dataset, these steps will just be repeated for the number of datapoints required.

1. Generate random rule

   We first pick a variable from the Herbrand base to be the rule head. We then sample $\{0, 1, 2\}$, each with equal probability, from a binomial distribution for each variable. When it is 0, the variable is not added to the body. When it is 1, the variable is added as a positive literal. When it is 2, the variable is added as a negative literal.

2. Add rule to logic program

For each random rule generated, we will compare to rules that are already added to the logic program. If the new rule is subsumed by other rules in the logic program, it will not be added. For each variable in the Herbrand base, we will generate at least 1 rule. Each subsequent rule generated has a 0.5^r probability to be the last rule generated for that specific variable head, where r is the number of rules already in the logic program with the same variable head.

3. Generate state transitions from the generated logic program

   Since there are no guarantee that the logic program is actually minimal, even though we check for rule subsumptions, we will do another pass to obtain the minimal logic program. We will run the T_P operator to obtain transitions from all states, which is 2^n states. These transitions are then recorded as the input of the training data as E.

4. Run the symbolic LFIT algorithm to obtain the minimal logic program

   Based on the transitions that we obtained from the previous step, we run the LFIT algorithm described in section 2.2.1. This algorithm will give us the minimal logic program to learn from. This is then recorded as the target of the training data as P. Note that the initial randomly generated logic program is thrown away and not used to train the model.

Based on the large space that is the possible logic programs, we can run the data generation process in parallel and be relatively confident that there will not be any duplicates within the dataset.

4.10 Experiments

In this section, we will describe the experiments that we have performed to validate the effectiveness of our proposed methods.

4.10.1 Experimental Method

We trained separate variants of the δLFIT+ model with different number of variables. The number of variables that the model is trained with is important

because it decides the number of outputs the δLFIT+ has. It is possible, for example, to learn a 3-variable system with a 5-variable system model, but it is not possible to learn a 7-variable system with a 5-variable model. The number of variables that the model is trained with, is denoted by $\delta\text{LFIT}+^n$, where n is the number of variables.

The network was implemented in PyTorch [64] and then trained with the Adadelta [97] optimizer. After training for 10 epochs, the network is evaluated by feeding data from the boolean networks. We then produce state transitions from the predicted logic program, and calculated the mean squared error between the state transitions from the predicted logic program and the boolean network.

During training, we only focus on one of the variables and one particular rule length for each data point. The selection for the variable is randomized within each batch, but the selection for the rule length is fixed for each batch, instead randomizing between different batches. This is done primarily for computational efficiency reasons. For each data point, we pick 80% of the states from all possible states for training. E.g., for 5 variables there are $2^5 = 32$ possible states, thus we pick 25 state transitions for training. These 25 states are picked randomly each epoch. Also for each epoch, we only train a single rule length l and on a particular variable as the rule head x for 1 datapoint. The combination of l and x is selected randomly. Therefore, at least $n(n + 1)$ epochs need to be trained to at least go through every available datapoint once. Validation is performed on data point withheld from the training process. The results are taken only from 1 run of the experiment, due to the lengthiness in the process of training, and also when we did multiple runs on the smaller systems, we did not observe significant variance in the results.

4.10.2 Baseline

Baseline results are shown in table 4.4. δLFIT wasn't able to deal with more than 5 variables therefore the results are omitted for 7 variable networks. The top number next to δLFIT+ represents the number of variables that was trained on. $\delta\text{LFIT}+^3$ means that the specific network was trained with 3 variables, and thus have 12 output nodes. Networks trained with more variables have more output nodes and thus can deal with larger boolean networks, together with

smaller networks.

We can observe that δLFIT+ performance is generally a little worse than δLFIT. However, δLFIT+ can generate predictions for 7 variables and above. Only in the boolean network Raf, which δLFIT struggled, was δLFIT+ beat in terms of performance.

4.10.3 Regular Transformer

In this experiment, we replaced the set transformer in the network architecture to a regular transformer. We retrained the model with the same hyperparameters and the same dataset. The results are shown in table 4.5. Results are taken by randomizing the order of the state transitions and taking the average. The drop in performance is easily observable across the board.

4.10.4 Rule Length Sharing

We show the difference in number of parameters with rule length sharing compared to without rule length sharing in figure 4.4. Without this, the number of parameters start exploding from 14 variables. However, with rule length sharing this effect is delayed until 16 variables. Contrasting with the state of the art natural language processing model like GPT-3, which has 175 billion parameters [16], there are still some leeway in terms of network architecture, however it will require state of the art hardware and large amount of computational time to actually train it. We did not perform any experiments to validate the difference in terms of accuracy because it will require a significant restructuring in the training process. We think that the restructuring is sufficiently impactful that the difference in accuracy that we will be measuring will not just be the effect of sharing rule lengths, but also the difference in the entire training process.

4.10.5 Without Label Smoothing

In this experiment, we trained the network without applying subsumed label smoothing. The results are shown in table 4.5 with the results for $\delta\text{LFIT}+^{5}_{\neg S}$

being one without label smoothing applied. No clear performance difference was found with the 3 variable networks, but the network that trained without label smoothing performed significantly worse with the 5 node boolean network.

4.10.6 Training Data Availability

Based on (4.3), we do know the entire problem space for δLFIT+ for any given n-variable system. Here, we do some experiment to figure out how much of the problem space given to δLFIT+ will affect its convergence. Experiments are performed for 3-variable systems, and hyperparameters are constant throughout all experiments. Each experiment is repeated 3 times. The results of this experiment are shown in table 4.6. We can observe that as more training data becomes available, the accuracy of the model improves.

4.10.7 Missing Data

In this experiment, we randomly redact state transitions from the input. Each experiment is performed 3 times and the averages are recorded. The results are shown in table 4.7. We can observe that as a general trend, as more data becomes available the accuracy improves.

4.10.8 Baseline: 3-node-a

This is a toy network with 3 variables. The boolean network constructed is shown in figure 4.5. Its corresponding logic program is written as

$$
\begin{aligned}
v_1(t+1) &\leftarrow \neg v_1(t) \wedge \neg v_2(t).\\
v_1(t+1) &\leftarrow \neg v_1(t) \wedge \neg v_3(t).\\
v_2(t+1) &\leftarrow \neg v_1(t) \wedge v_2(t).\\
v_3(t+1) &\leftarrow \neg v_1(t).\\
v_3(t+1) &\leftarrow v_2(t).
\end{aligned}
$$

This network has two attractors and one steady state. The predicted logic

program by δLFIT+ is shown as below

$$
\begin{aligned}
v_1(t+1) &\leftarrow \neg v_1(t) \wedge \neg v_2(t).\\
v_1(t+1) &\leftarrow \neg v_1(t) \wedge \neg v_3(t).\\
v_2(t+1) &\leftarrow v_2(t).\\
v_2(t+1) &\leftarrow \neg v_1(t) \wedge \neg v_3(t).\\
v_3(t+1) &\leftarrow v_2(t) \wedge \neg v_3(t).\\
v_3(t+1) &\leftarrow \neg v_1(t) \wedge \neg v_2(t).\\
v_3(t+1) &\leftarrow \neg v_1(t) \wedge \neg v_3(t).
\end{aligned}
$$

Note that there are a few rules that disagree with the original logic program. We validate the accuracy of the prediction based on the state transitions that can be obtained by the logic programs. The difference in the state transitions produced by the respective logic programs are shown in table 4.8.

4.11 Discussions

In general, we observed that there is a drop in accuracy across the board as the outputs get reused. δLFIT+ performed poorer on the other networks, except in the boolean network Raf, where δLFIT struggled because the same state was repeated. We also observed that δLFIT+^{3} performing better than δLFIT+^{5} for 3 variable networks. Memory efficiency has also improved from δLFIT, which allowed us to easily train a network with 7 variables. However, generating training data for 8 variables and beyond required too much time with the amount of computational resource that we had, therefore we were unable to train the networks.

With the rule length sharing method proposed, we were able to cut the number of parameters in the model by half. This allowed us to use larger number of batch size during training to increase training speed. However with the reduced number of output nodes, this might have led to a potential loss in terms of accuracy, as now each output node represents multiple rules, compared to one output node one rule in δLFIT.

Subsumed label smoothing did not have any visible impact on smaller networks with 3 variables, but with 5 variables the effects were visible. In partic-

ular, with the same hyperparameter, we observed that δLFIT+$^5_{\neg S}$ was overfitting, meaning training loss decreased very rapidly while validation loss remained high. Adding subsumed label smoothing helped apply some regularization to δLFIT+^{5}.

Training data availability seems to have a substantial effect on the model accuracy. For $n = 3$, it is still feasible to sample a substantial amount of the problem space to provide as training data. However, starting from $n = 4$, attempting to sample even 1% of the training data requires huge amount of computational time. It might have been possible to further improve on δLFIT+ accuracy, should there be a better means of sampling the problem space. For example, by ensuring that each rule has a clear example for the model to learn.

We can also see that missing data has an effect on the accuracy. Although once given more than half of the transitions, the model seems to provide an acceptable prediction for the logic program. There were huge variance for each run on the experiments here, and we can speculate that it is due to some state transition providing more information than others. It might be interesting to further investigate on whether there are some main transitions that could define the logic program, such that when those transitions are included the model is always able to perform better predictions than other transitions.

δLFIT+ took 1 day to generate and train for 3 variable systems, while it took 5 days to generate and train for 7 variable systems. Compared to other LFIT methods, the runtime for δLFIT+ is significantly higher. However, most methods only focus on learning a single system whereas δLFIT+ learns a general n-variable system. The advantage for this is that, in real world applications, novel systems that are being investigated often have only small amount of observation data available. A technique that only works with the obtained observation data will overfit, whereas δLFIT+ can avoid the overfitting problem.

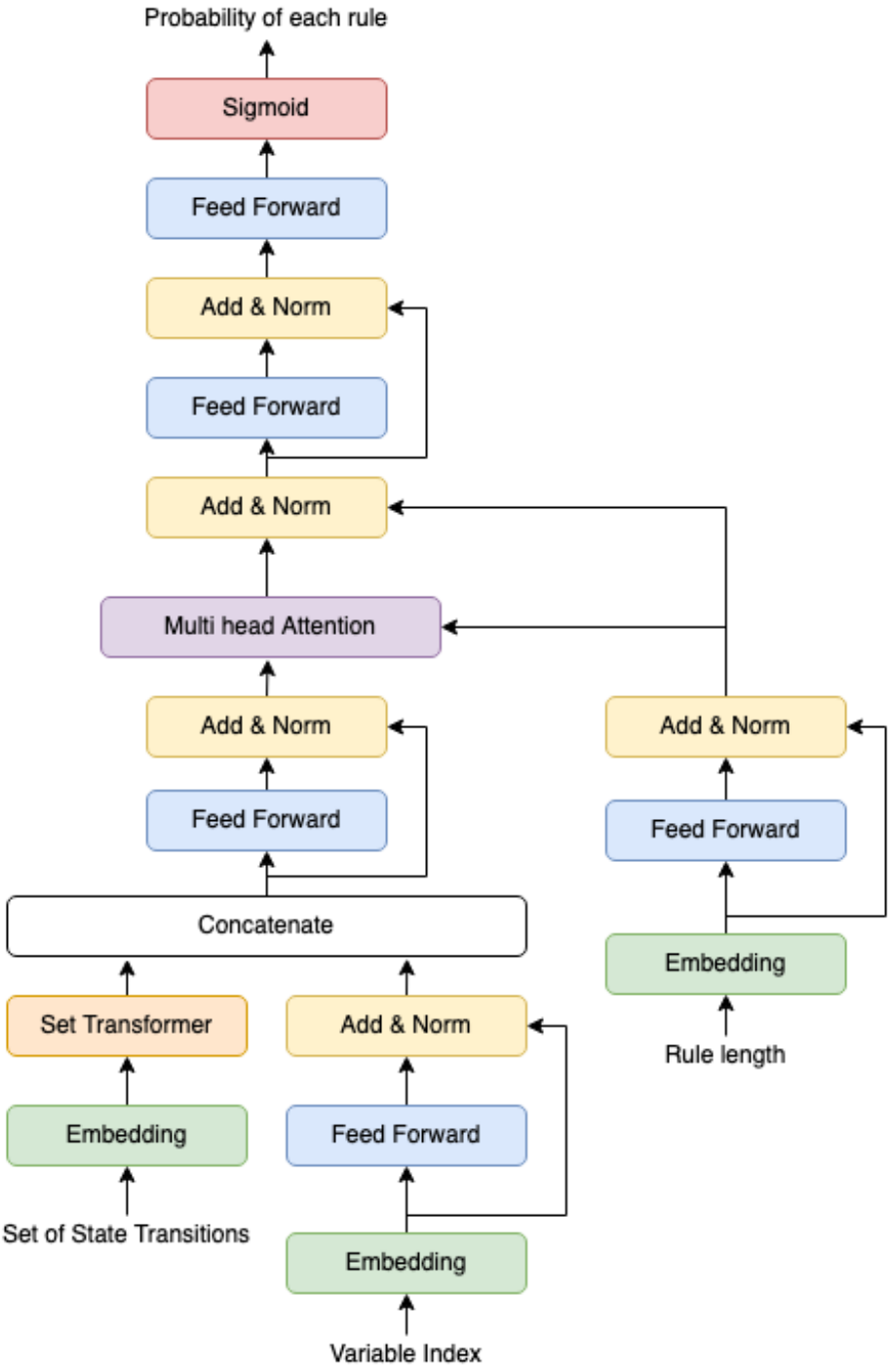


Figure 4.3: δLFIT+ architecture

Algorithm 5: $\sigma^{-1}_{\tau(\mathcal{B}_O)}(R)$: Reverse lookup a rule given an index

Input : $x \in \mathbb{N}$
Output: Rule R such that $\sigma_{\tau(\mathcal{B}_O)}(R) = x$
if $x > 0$ **then**
  $l = 1$, $x_r = 1$;
  for $i := 0$ **to** $\|\mathcal{B}\|$ **do**
    if $\|\tau_i(\mathcal{B}_O)\| > x$ **then**
      break
    end
    $x_r = \|\tau_j(\mathcal{B}_O)\|$;
    $l = i + 1$;
  end
end
$x_r = x - x_r$;
$c := \binom{\|\mathcal{B}\|}{l}$;
$R := \{\}$, $C := \mathcal{B}_O$;
for $i = 0$ **to** $l - 1$ **do**
  $a := C[0]$;
  $C = C \setminus a$;
  $R = R \cup \{a\}$;
end
for $i = 0$ **to** $x_r \mod c$ **do**
  `IncrementRule` $(R, l - 1)$;
end
for $i = 0$ **to** a **do**
  if $\lfloor \frac{x_r}{c} \rfloor \& (1 \ll i) \neq 0$ **then**
    $a := \sigma^{-1}_R(i)$;
    $R = (R \setminus \{l\}) \cup \{\overline{a}\}$;
  end
end
return R

Algorithm 6: IncrementRule(R, l): Increments a rule to the next rule

Inputs : R a rule body or a set of clauses, l the position to which the literal should be incremented

Output: R a rule body

```
if l < ||B_O|| then
    a = σ_{B_O}(R[l]) + 1;
    if a ≥ ||B_O|| then
        R = IncrementRule (R, l − 1);
        R[l] = R[l − 1];
        R = IncrementRule (R, l);
    end
    else
        R[l] = σ^{-1}_{B_O}(a);
    end
end
return R
```

Boolean Network (n)	δLFIT	δLFIT+^{3}	δLFIT+^{5}	δLFIT+^{7}
3-node-a (3)	0.095	0.271	0.271	0.271
3-node-b (3)	0.054	0.188	0.208	0.208
Raf (3)	0.253	0.188	0.208	0.208
5-node (5)	0.142	N.A.	0.278	0.325
7-node (7)	O.O.M.	N.A.	N.A.	0.223
WNT5A (7) [93]	O.O.M.	N.A.	N.A.	0.194

Table 4.4: The MSE for the state transitions generated by the predicted logic programs compared to δLFIT+.

Boolean Network	δLFIT+^{5}	δLFIT+$^5_{\neg T}$	δLFIT+$^5_{\neg S}$
3-node (a)	0.271	0.313	0.271
3-node (b)	0.208	0.292	0.208
Raf	0.208	0.271	0.208
5-node	0.278	0.375	0.378

Table 4.5: The MSE for the state transitions generated by the predicted logic programs with and without set transformers, and with and without label smoothing

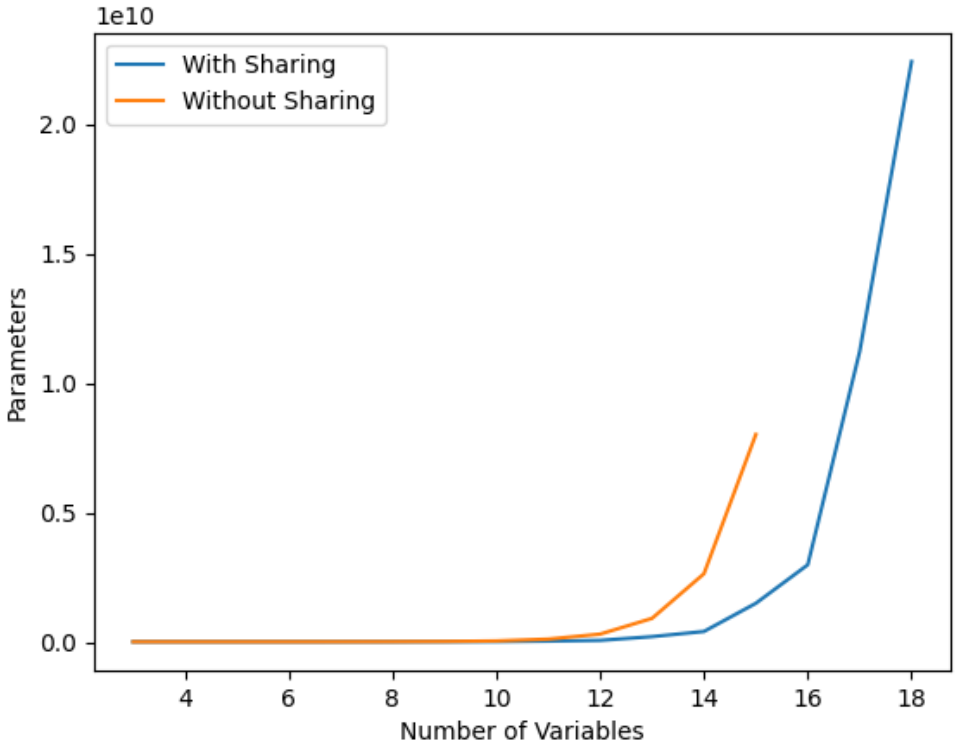


Figure 4.4: Number of parameters in 10 billions, of the network with rule length sharing compared to without rule length sharing

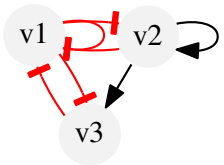


Figure 4.5: Boolean network of 3-node-a, a toy network with 3 nodes

Percentage	3-node (a)	3-node (b)	Raf	Average
0.1%	0.472	0.542	0.458	0.49
0.25%	0.5	0.444	0.43	0.458
0.5%	0.444	0.486	0.445	0.458
0.75%	0.472	0.375	0.486	0.444
1%	0.472	0.597	0.333	0.468
5%	0.292	0.431	0.430	0.384
10%	0.236	0.403	0.347	0.329
15%	0.236	0.278	0.361	0.292
20%	0.264	0.264	0.25	0.259
25%	0.236	0.306	0.236	0.259
50%	0.222	0.319	0.208	0.25
75%	0.264	0.194	0.25	0.236

Table 4.6: Accuracy for different level of training data availability for 3-variable systems

	3-node (a)		3-node (b)		Raf	
Given	δLFIT+	NN-LFIT	δLFIT+	NN-LFIT	δLFIT+	NN-LFIT
1 (12.5%)	**0.431**	0.459	0.625	**0.472**	0.444	**0.417**
2 (25%)	**0.417**	0.542	**0.431**	0.458	**0.347**	0.389
3 (37.5%)	**0.319**	0.542	0.472	**0.292**	**0.319**	0.458
4 (50%)	**0.264**	0.5	0.403	0.403	**0.264**	0.417
5 (62.5%)	0.389	**0.333**	0.389	**0.292**	**0.306**	0.458
6 (75%)	**0.236**	0.375	0.222	**0.167**	**0.278**	0.333
7 (87.5%)	0.292	**0.083**	0.195	**0.083**	**0.236**	0.25
8 (100%)	0.208	**0.00**	0.083	**0.042**	**0.25**	0.333

Table 4.7: Accuracy for different amount of missing data for 3-variable systems

Prior State	Actual	δLFIT+
ϵ	$\{v_1, v_3\}$	$\{v_1, v_2, v_3\}$
$\{v_1\}$	ϵ	ϵ
$\{v_2\}$	$\{v_1, v_2, v_3\}$	$\{v_1, v_2, v_3\}$
$\{v_1, v_2\}$	$\{v_3\}$	$\{v_2, v_3\}$
$\{v_3\}$	$\{v_1, v_3\}$	$\{v_1, v_3\}$
$\{v_1, v_3\}$	ϵ	ϵ
$\{v_2, v_3\}$	$\{v_2, v_3\}$	$\{v_2\}$
$\{v_1, v_2, v_3\}$	$\{v_3\}$	$\{v_2\}$

Table 4.8: Predicted state transitions for the network 3-node-a

5

Extensions

In this chapter, we will introduce several extensions that can be applied to δLFIT+. In the LFIT literature, there are various extensions that extend the problem beyond synchronous semantics and boolean values. These extensions can also be applied to δLFIT+.

We will first introduce the basic ideas of these extensions to the LFIT framework, and then explain the methods that we applied δLFIT+ for the extensions.

5.1 Systems with Delays

Some real world application require that not all variables be updated at the same time. In particular, some of the effects of actions or events can appear after certain time points. For example, the mammalian circadian clock [21] is better expressed with delays. Another example is the DNA damage repair [2] system. Even in social interactions, past actions that extend beyond the previous timestep is certain to be influencing future decisions [63]. Therefore, to

capture these sorts of interactions in the model, delays can be introduced into the LFIT framework.

In the most vanilla LFIT, the algorithms deal with a Markov(1) system. Extending this idea, where Markov(k) means that the state of the system may depend on, up to k previous states. In this case, any k sequence of transitions there is a deterministic state at the $k+1$ timestep. Note that, given a state I at time step t, there may be one or more states that I will lead to.

5.1.1 LFkT

The LFIT algorithm that deals with delays is called Learning from k-step Transitions (LFkT). We will now extend the idea of Herbrand base $\mathcal{B}$ to capture the Markov(k) system dynamic. Given a logic program P, an Herbrand base $\mathcal{B}$ and a natural number k, the *timed Herbrand base* with period k is

$$\mathcal{B}_k = \bigcup_{i=1}^{k} \{v_{t-i} \mid v \in \mathcal{B}\}$$

Consider the following logic program in a Markov(2) system

$$a(t+1) \leftarrow b(t) \wedge b(t-1) \tag{5.1}$$
$$b(t+1) \leftarrow a(t-1) \wedge \neg b(t-1) \tag{5.2}$$

this means that the atom a will only be in the interpretation at timestep $t+1$, only if b is in both the timestep t and timestep $t-1$. Notice that we are looking up to 2 previous timesteps. On the other hand, atom b will be in the interpretation at timestep $t+1$, if only atom a is in the timestep $t-1$.

Under this framework, instead of state transitions, we will be dealing with traces of execution. A trace of execution T is defined as a finite sequence of states S, where $T = (S_0, S_1, \ldots, S_n)$. To figure out the sequences, we will also define the following function

$$\text{prev}(i, j, T) = \begin{cases} \emptyset & \text{if } i = 0 \text{ or } j = 0, \\ (S_{i-j-1}, \ldots, S_{i-1}) & \text{if } j+1 \leq i, \\ (S_0, \ldots, S_{i-1}) & \text{otherwise.} \end{cases}$$

where $n \geq 1$, $\forall i \in \mathbb{N}\ (i \leq n)$, $S_i \in 2^{\mathcal{B}}$. To ease some of the notations, we will also define the following

$$\text{prev}(i, T) = \text{prev}(i, n, T)$$
$$\text{next}(i', T) = S_{i'+1}$$

where $i' \in \mathbb{N}, i' < n$. We will denote $\|T\|$ to be the length of the trace, which is the number of **state transitions** within the sequence. A subtrace of size m of T, is a sub-sequence of m consecutive states within T. As an example, given $T_1 = (\{a\}, \{b\}, \{a\})$, then $\|T_1\| = 2$ and both $(\{a\}, \{b\})$ and $(\{b\}, \{a\})$ are subtraces of length 1 for T_1.

Next we consider the k-step interpretation transitions. Given an Herbrand base $\mathcal{B}$ and the corresponding timed Herbrand base $\mathcal{B}_k$, a k-step interpretation transition is a pair of intepretations (I, J) where $I \subseteq \mathcal{B}_k$ and $J \subseteq \mathcal{B}$.

As an example, given a trace $T = (\{a, b\}, \{b\}, \{a\})$, the interpretation transitions obtained are the following

1. $(\{a_{t-1}, b_{t-1}, b_t\}, \{a\})$, this corresponds to the entire trace
2. $(\{a_t, b_t\}, \{b\})$, this corresponds to the subtrace $(\{a, b\}, \{b\})$
3. $(\{b_t\}, \{a\})$, this corresponds to the subtrace $(\{b\}, \{a\})$

To explain this Markov(k) system, we define the consistency as follows. Given a logical rule R and a k-step interpretation transition (I, J), R is *consistent* with (I, J) if and only if $b^+(R) \subseteq I$ and $b^-(R) \cap I = \emptyset$ imply $h(R) \in J$.

5.1.2 Extending Input State Encoding

To apply δLFIT+ to delayed systems, we will extend the input state encoding. Given an Herbrand base $\mathcal{B} = \{p, q\}$ and period of $k = 2$, the timed Herbrand base is $\mathcal{B}_k = \{p_t, q_t, p_{t-1}, q_{t-1}\}$. The Herbrand base is basically extended by a factor of k, which means $\|\mathcal{B}_k\| = nk$. We will therefore just extend the encodings accordingly.

Previous State	$\theta_a = 0$	$\theta_a = 1$
ϵ	0	16
a_t	1	17
b_t	2	18
$a_t b_t$	3	19
a_{t-1}	4	20
$a_t a_{t-1}$	5	21
$b_t a_{t-1}$	6	22
$a_t b_t a_{t-1}$	7	23
b_{t-1}	8	24
$a_t b_{t-1}$	9	25
$b_t b_{t-1}$	10	26
$a_t b_t b_{t-1}$	11	27
$a_{t-1} b_{t-1}$	12	28
$a_t a_{t-1} b_{t-1}$	13	29
$b_t a_{t-1} b_{t-1}$	14	30
$a_t b_t a_{t-1} b_{t-1}$	15	31

Table 5.1: Extended state input for a logic program with $\mathcal{B}_k$

The ordering of the elements in ordered timed Herbrand base $\mathcal{B}_{k,O}$ can be determined arbitrarily as noted in section 3.2. For convenience though, we will first sort by the timing factor from $t, t-1, \ldots$ and then by lexicographical order.

We will thus extend the definition of both ζ_0 and ζ_1 to work with the timed Herbrand base. Redefining $E_x^k = \{(I, \theta_x) \mid I \in 2^{\mathcal{B}_k}\}$, and θ_x is still as the following

$$\theta_x = \begin{cases} 0 & \text{if } \forall (I,J) \in E \ (x \notin J), \\ 1 & \text{if } \exists (I,J) \in E \ (x \in J), \end{cases}$$

Next, we define the function $\zeta_0^k : 2^{\mathcal{B}_k} \mapsto \mathbb{N}$. An example of the mapping represented by ζ_0^k is listed in table 5.1.

As can be observed, one can simply treat the original Herbrand base as having 4 variables instead of 2, and perform the encoding. Consider the example NLP in (5.2). The full state transition input for a in this NLP can be written like $E'_a = \{6, 30, 26, 2, 8, 12, 4, 0\}$.

The input space for systems with delays, can be modified based on (4.3) to the following

$$(2^{2^{nk}})^n$$

For a system with n-variable and k delay, there are nk prior states. For each of these states there are 2 possible transitions, for each of the n variables. Considering (4.3) to be the special case of $k = 1$, it is clear that the input space with delays is much larger.

n	k	Input Space	Number
2	1	$(2^{2^2})^2$	256
2	2	$(2^{2^4})^2$	4.24×10^9
2	3	$(2^{2^6})^2$	3.40×10^{38}
3	1	$(2^{2^3})^3$	1.68×10^7
3	2	$(2^{2^6})^3$	6.28×10^{57}
3	3	$(2^{2^9})^3$	2.41×10^{642}

Table 5.2: The input space for each n and k

5.1.3 Adapting the δLFIT+ Algorithm

Comparing from the vanilla LFIT framework, in LFkT there are separate domains for $b(R)$ and $h(R)$. In particular, $b(R) \subseteq \mathcal{B}_k$ is a subset of the timed Herbrand base while $h(R) \in \mathcal{B}$. This means that in terms of what δLFIT+ cares, if $\|\mathcal{B}\| = 2$ and $k = 2$, we can train the model for 4 variables while only asking for rules with 2 variables as the head.

The algorithm in algorithm 4 can be subsequently modified to the algorithm described in 7.

The only modifications to this algorithm are the inputs, where we know have a set of state transitions E which have the timed Herbrand base as the states to transition from.

Algorithm 7: δLFIT+ algorithm to adapt to delayed systems

Inputs : a set of atoms $\mathcal{B}$ and another set of atoms $\mathcal{B}_k$, integer k, set of state transitions $E \subseteq 2^{\mathcal{B}_k} \times 2^{\mathcal{B}}$

Output: an NLP P

$P := \emptyset$;
foreach $x \in \mathcal{B}$ **do**
  $E := \{(I, \theta_x) \mid (I, J) \in E, \theta_x = \{0, 1\}\}$;
  $E_x := \{\zeta_1(\zeta_0(I), \theta_x) \mid (I, \theta_x) \in E_x\}$;
  $R_x := \emptyset$;
  for l *in* $0 \ldots \|\mathcal{B}_k\|$ **do**
    $R_{x,l} := \delta\text{LFIT+}(E_x, x, l)$;
    foreach $R \in R_{x,l}$ **do**
      $R_x =$ `Specialize` (R_x, R);
    end
  end
  foreach $R \in R_x$ **do**
    $P =$ `Specialize` (P, R);
  end
end

5.1.4 Recovering the Logic Program

We will also extend the reverse lookup for a rule given an index algorithm σ^{-1}. The only modifications required are places related to those which operated on $\mathcal{B}$, we will instead extend them to work on $\mathcal{B}_k$.

5.1.5 Other Details

Other parts of the δLFIT+ technique can be extended to cover delay systems with minimal modifications. In particular with regards to rule sharing, the basic principles are the same. A full example of the mapping for the output nodes is listed in table 5.3.

Due to the fact that the head of the rules are still in domain of $\mathcal{B}$, no further modifications are required on the input.

5.1.6 Experiments

In this section, we will describe the experiments that we have performed to validate our methods. The setup is largely similar to that of δLFIT+. We first generate random logic programs to obtain the training data, we then train the model and subsequently validate the results with a prior known boolean network.

Experimental Methods

We first generated training data for the $\mathcal{B}_k$ that we wanted to study. In this case, due to computational constraints, we picked 2 variables with $k = 2$. The model is trained based on this training dataset. We then performed experiment with the logic program given in (5.2) and recorded the results.

The network implementation is exactly the same as in chapter 4. The only modifications required are to adapt the input encoding to take into account the delays.

Results

In this experiment, we generated state transitions from the logic program in (5.2). The generated state transitions are then input into the model. The state transitions from the predicted logic program is then compared with the original logic program.

The predicted logic program by δLFIT+ is shown as below

$$
\begin{aligned}
a(t+1) &\leftarrow a(t) \wedge a(t-1) \wedge b(t-1) \\
a(t+1) &\leftarrow b(t) \wedge \neg a(t-1) \wedge \neg b(t-1) \\
b(t+1) &\leftarrow \neg a(t) \wedge \neg b(t) \wedge a(t-1) \\
b(t+1) &\leftarrow b(t) \wedge a(t-1) \wedge \neg b(t-1)
\end{aligned}
$$

The full difference in the state transitions produced by the respective logic programs are shown in table 5.4.

The MSE between the predicted state transitions and the actual state transitions is 0.313. Despite there being no common rules between the predicted logic program and the actual logic program, we can observe that most of the errors are from the prediction of variable a. There are only 2 errors with regards to variable b.

5.1.7 Discussions

We can observe that δLFIT+ can be applied to systems with delay without too many modifications. However due to the fact that the problem space growing large even for small variables and small amount of delays, it became infeasible to train for even a slightly larger network with a little bit of delays.

5.2 $\delta\text{GULA}_{\text{BOOL}}$

The original LFIT only considered the synchronous semantic, in which all variables are updated at the same time. Meaning at each transition, every rule from the logic program that can be applied, are applied simultaneously in 1 step to obtain the next state. Real world applications, however, demanded different kinds of update semantics.

5.2.1 GULA

The original GULA algorithm proposed in [69] deals with multivalued systems. Here, however, we do not want to investigate the multivalued properties of δLFIT+, therefore we will work under the assumptions of a boolean system. A multivalued system will be a generalized version of the boolean system.

First, we say that a *rule matches* an interpretation I, written as $R \cap I$ if $b(R) \subseteq I$. This means that rule R can be applied at the interpretation. Given 2 rules R and R', these rules *cross-match*, written as $R \cap R'$ if there exists $I \subseteq \mathcal{B}$ such that both R and R' matches, $R \cap I$ and $R' \cap I$. In this case, during state I, both R and R' can be applied.

Next, we say that a rule R realizes the transition (I, J), written as $I \xrightarrow{R} J$, if $R \cap I \wedge h(R) \in J$. This means that in the transition (I, J), rule R has been applied on state I to obtain the transition state J. A logic program P realizes the transition (I, J), written as $I \xrightarrow{P} J$, if $\forall l \in \mathcal{B}\ \exists R \in P\ (h(R) = l) \wedge (I \xrightarrow{R} J)$. If there is a rule R in the logic program P that realizes the transition from I to J, then we can say P realizes the transition.

For a set of transitions $E = \{(I, J) \mid I, J \subseteq \mathcal{B}\}$, we denote the set of interpretations that has a transition as $\gamma(E) = \{I \mid \exists (I, J) \in E\}$. Note that $\gamma(E) = \emptyset \Rightarrow E = \emptyset$, if there are no interpretations that have a transition, then the set of transitions should be empty as well.

We will also define an operator $I \circ l$ where $I \subseteq \mathcal{B}$ is an interpretation and $l \in \mathcal{B}$ is an atom. $I \circ l$ is defined as follows

$$I \circ l = \begin{cases} I \setminus l & \text{if } l \in I, \\ I \cup \{l\} & \text{if } l \notin I. \end{cases}$$

essentially this operator says l must change value in the transition after I.

Let's now define what we mean by semantics of a dynamic system. A semantics is a function that maps a logic program P to a set of transitions $E \subseteq 2^{\mathcal{B}} \times 2^{\mathcal{B}}$ such that $\gamma(E) = \mathcal{B}$, meaning each interpretation is assigned a next state to transition to.

Using this definition of semantics, we now define the synchronous semantics. The synchronous semantics $\mathcal{T}_{\text{syn}}$ is defined as follows

$$\mathcal{T}_{\text{syn}} : P \mapsto \{(I, J) \in 2^{\mathcal{B}} \times 2^{\mathcal{B}} \mid J \subseteq \{h(R) \mid R \in P, R \cap I\}\}$$

In the synchronous semantics, all rules $R \in P$ that is consistent with I will take effect. We say that the transitions in E are synchronous when the following is true. First, $\gamma(E) = \mathcal{B}$, meaning all interpretations have a transition. Then, $\exists P$ such that $\mathcal{T}_{\text{syn}}(P) = E$ if and only if $\forall (I, J), (I, K) \in E$, $\forall L \in 2^{\mathcal{B}}$, $L \subseteq J \cup K \Rightarrow (I, L) \in T$. This says that if we observed I transitioning into 2 separate states J and K, the combination of both J and K, L is also a valid transition from state I.

Next, we will define the asynchronous semantics. Contrasting to the synchronous semantics, in the asynchronous semantics only one variable is allowed

to change their state at one time. The asynchronous semantics $\mathcal{T}_{\text{asyn}}$ is defined as follows

$$\mathcal{T}_{\text{asyn}} : P \mapsto \{(I, J \circ \{h(R)\}) \in 2^{\mathcal{B}} \times 2^{\mathcal{B}} \mid (R \in P) \wedge (R \cap I) \wedge (h(R) \notin I)\} \cup \{(I, J) \in 2^{\mathcal{B}} \times 2^{\mathcal{B}} \mid \forall R \in P \ (R \cap I \Rightarrow h(R) \in I)\}$$

We can say E is asynchronous, when $\exists P$ such that $\mathcal{T}_{\text{asyn}}(P) = E$, if and only if $\forall I, J \in 2^{\mathcal{B}}$, $I \neq J$, $((I, I) \in E \Rightarrow (I, J) \notin E) \wedge ((I, J) \in E \Rightarrow \|I \setminus J\| = 1)$. This says that, the transition from state I will be either state I itself, or another state J which differs from I by only 1 atom. The asynchronous semantics have a tendency to priorize atoms changing value, since this is the only immediate observable effect when applying the rules. Note that the semantics says only at most 1 variable can change, but nothing about how many rules shall be applied.

Lastly, we will define the general semantics. The general semantics is a general version of the asynchronous semantics. Any subset of the variable is allowed to change their values. The general semantics $\mathcal{T}_{\text{gen}}$ is defined as follows

$$\mathcal{T}_{\text{gen}} : P \mapsto \{(I, J \circ r) \in 2^{\mathcal{B}} \times 2^{\mathcal{B}} \mid r \subseteq \{h(R) \mid (R \in P) \wedge (R \cap I)\}\}$$

E is general, when $\exists P$ such that $\mathcal{T}_{\text{gen}} = E$, if and only if $\forall (I, J), (I, K) \in E, \forall L \in 2^{\mathcal{B}}$ $L \subseteq I \cup J \cup K \Rightarrow (I, L) \in E$. This says that, if we observe an interpretation I transitioning into 2 states J and K, another state L which can be a combination of J, K and I itself will also be a transition from I.

5.2.2 Extending δLFIT+

δLFIT+ can be trivially extended to support the general semantics. In fact, due to the probabilistic nature of δLFIT+, it already inherently supports all the other semantics.

The only thing to consider here is the input. Where in the synchronous semantics, there will only be one possible state for each prior state, in the general semantics there can be more than one. However, since δLFIT+ only considers a single variable at a time, the only thing that might be possible is that the variable either changes or doesn't change in the next timestep.

In the perspective of δLFIT+, given a prior state of $\{a, b\}$, focusing at the variable a, if an input of $(\{a, b\}, \{\})$ or $(\{a, b\}, \{b\})$ exists, then the model can know that a rule that makes a negative exists. And this is the only information that δLFIT+ will need to predict logic programs.

5.2.3 Experiments

We performed experiments by randomly picking rules to apply to each state transition, and asked δLFIT+ to predict the logic program. The logic program is then compared to the original full logic program.

Experimental Method

We generated training data based on the synchronous semantics, then trained the model based on that training data. The implementation was the same as described in chapter 4. We then tested the model on boolean networks taken from PyBoolNet. To test the difference in semantics, we generated the state transitions by randomly picking rules with probability of p. The state transitions are then given to the model to produce a logic program prediction. The logic program is then compared with the original boolean network by producing the state transitions based on the synchronous semantics. The experiments are repeated 3 times for each case and the averages are recorded.

Results

The results for this are shown in table 5.5. We can observe that as p increases, it becomes closer to the synchronous semantics and thus the model performs better.

Here, we'll show the example for the Raf network with 3 variables. The boolean network constructed is shown in figure 5.1. Its corresponding logic program is

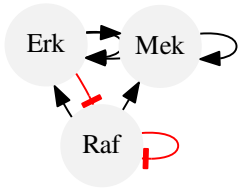


Figure 5.1: Boolean network of raf

written as

$$\begin{aligned}
\text{Erk}(t+1) &\leftarrow \text{Erk}(t) \wedge \text{Mek}(t) \\
\text{Erk}(t+1) &\leftarrow \text{Mek}(t) \wedge \text{Raf}(t) \\
\text{Mek}(t+1) &\leftarrow \text{Erk}(t) \\
\text{Mek}(t+1) &\leftarrow \text{Mek}(t) \wedge \text{Raf}(t) \\
\text{Raf}(t+1) &\leftarrow \neg\text{Erk}(t) \\
\text{Raf}(t+1) &\leftarrow \neg\text{Raf}(t)
\end{aligned}$$

The predicted logic program, after providing the state transitions with $p = 0.25$ probability of applying the rules, is shown as below

$$\begin{aligned}
\text{Erk}(t+1) &\leftarrow \neg\text{Erk}(t) \wedge \text{Raf}(t) \\
\text{Erk}(t+1) &\leftarrow \neg\text{Mek}(t) \wedge \text{Raf}(t) \\
\text{Erk}(t+1) &\leftarrow \text{Erk}(t) \wedge \neg\text{Raf}(t) \\
\text{Mek}(t+1) &\leftarrow \text{Erk}(t) \wedge \text{Mek}(t) \\
\text{Mek}(t+1) &\leftarrow \text{Erk}(t) \wedge \neg\text{Raf}(t) \\
\text{Mek}(t+1) &\leftarrow \neg\text{Mek}(t) \wedge \neg\text{Raf}(t) \\
\text{Raf}(t+1) &\leftarrow \neg\text{Raf}(t) \\
\text{Raf}(t+1) &\leftarrow \neg\text{Erk}(t)
\end{aligned}$$

In this case we can see there is a huge difference as δLFIT+ struggles a little bit to clearly predict the logic program. The MSE for this logic program was 0.292.

On the other hand, here is the logic program after providing the state transitions with $p = 0.75$ probability of applying the rules

$$\begin{aligned}
\text{Erk}(t+1) &\leftarrow \text{Erk}(t) \wedge \text{Mek}(t) \\
\text{Erk}(t+1) &\leftarrow \text{Mek}(t) \wedge \text{Raf}(t) \\
\text{Erk}(t+1) &\leftarrow \neg\text{Erk}(t) \wedge \text{Raf}(t) \\
\text{Mek}(t+1) &\leftarrow \text{Erk}(t) \wedge \text{Raf}(t) \\
\text{Mek}(t+1) &\leftarrow \text{Mek}(t) \wedge \text{Raf}(t) \\
\text{Raf}(t+1) &\leftarrow \neg\text{Raf}(t) \\
\text{Raf}(t+1) &\leftarrow \neg\text{Erk}(t) \wedge \neg\text{Mek}(t)
\end{aligned}$$

Here we can see a very good prediction from δLFIT+, where the rules for Erk are actually all correct, and for Raf there is only 1 state transition that disagrees. The MSE in this case was 0.167.

5.2.4 Discussions

We can observe that there is no necessity to train the model specifically in the various semantics as the generalization ability of the neural network already allows it to also adapt to the various semantics. This is also aided in particular to the way that we already encode the state transitions, by providing transitions for each variables individually, there is already inherently a generalizability in terms of semantics. Despite the fact that semantics focus on the rules that are applicable and not the variables that change within the transitions, the outcome, we can argue are the same.

Algorithm 8: $\sigma^{-1}_{\tau(\mathcal{B}_{k,O})}(R)$: Reverse lookup a rule given an index

Input : $x \in \mathbb{N}$
Output: Rule R such that $\sigma_{\tau(\mathcal{B}_{k,O})}(R) = x$
if $x > 0$ **then**
  $l = 1$, $x_r = 1$;
  for $i := 0$ **to** $\|\mathcal{B}_k\|$ **do**
    if $\|\tau_i(\mathcal{B}_{k,O})\| > x$ **then**
      break
    end
    $x_r = \|\tau_j(\mathcal{B}_{k,O})\|$;
    $l = i + 1$;
  end
end
$x_r = x - x_r$;
$c := \binom{\|\mathcal{B}_k\|}{l}$;
$R := \{\}$, $C := \mathcal{B}_{k,O}$;
for $i = 0$ **to** $l - 1$ **do**
  $a := C[0]$;
  $C = C \setminus a$;
  $R = R \cup \{a\}$;
end
for $i = 0$ **to** $x_r \mod c$ **do**
  `IncrementRule` $(R, l - 1)$;
end
for $i = 0$ **to** a **do**
  if $\lfloor \frac{x_r}{c} \rfloor \& (1 \ll i) \neq 0$ **then**
    $a := \sigma_R^{-1}(i)$;
    $R = (R \setminus \{l\}) \cup \{\overline{a}\}$;
  end
end
return R

Nodes	0	1	2	3
$l=0$	$\{\}$	-	-	-
$l=1$	$\{a_t\}$	$\{b_t\}$	$\{a_{t-1}\}$	$\{b_{t-1}\}$
$l=2$	$\{a_t, b_t\}$	$\{a_t, a_{t-1}\}$	$\{a_t, b_{t-1}\}$	$\{b_t, a_{t-1}\}$
$l=3$	$\{a_t, b_t, a_{t-1}\}$	$\{a_t, b_t, b_{t-1}\}$	$\{b_t, a_{t-1}, b_{t-1}\}$	$\{a_t, a_{t-1}, b_{t-1}\}$
$l=4$	$\{a_t, b_t, a_{t-1}, b_{t-1}\}$	$\{\neg a_t, b_t, a_{t-1}, b_{t-1}\}$	$\{a_t, \neg b_t, a_{t-1}, b_{t-1}\}$	$\{\neg a_t, \neg b_t, a_{t-1}, b_{t-1}\}$
Nodes	4	5	6	7
$l=0$	-	-	-	-
$l=1$	$\{\neg a_t\}$	$\{\neg b_t\}$	$\{\neg a_{t-1}\}$	$\{\neg b_{t-1}\}$
$l=2$	$\{b_t, b_{t-1}\}$	$\{\neg a_t, b_t\}$	$\{\neg a_t, a_{t-1}\}$	$\{\neg a_t, b_{t-1}\}$
$l=3$	$\{\neg a_t, b_t, a_{t-1}\}$	$\{\neg a_t, b_t, b_{t-1}\}$	$\{\neg a_t, a_{t-1}, b_{t-1}\}$	$\{\neg b_t, a_{t-1}, b_{t-1}\}$
$l=4$	$\{a_t, b_t, \neg a_{t-1}, b_{t-1}\}$	$\{\neg a_t, b_t, \neg a_{t-1}, b_{t-1}\}$	$\{a_t, \neg b_t, \neg a_{t-1}, b_{t-1}\}$	$\{\neg a_t, \neg b_t, \neg a_{t-1}, b_{t-1}\}$
Nodes	8	9	10	11
$l=0$	-	-	-	-
$l=1$	-	-	-	-
$l=2$	$\{\neg b_t, a_{t-1}\}$	$\{\neg b_t, b_{t-1}\}$	$\{\neg a_{t-1}, b_{t-1}\}$	$\{a_t, \neg b_t\}$
$l=3$	$\{a_t, \neg b_t, a_{t-1}\}$	$\{a_t, \neg b_t, b_{t-1}\}$	$\{a_t, \neg a_{t-1}, b_{t-1}\}$	$\{b_t, \neg a_{t-1}, b_{t-1}\}$
$l=4$	$\{a_t, b_t, a_{t-1}, \neg b_{t-1}\}$	$\{\neg a_t, b_t, a_{t-1}, \neg b_{t-1}\}$	$\{a_t, \neg b_t, a_{t-1}, \neg b_{t-1}\}$	$\{\neg a_t, \neg b_t, a_{t-1}, \neg b_{t-1}\}$
Nodes	12	13	14	15
$l=0$	-	-	-	-
$l=1$	-	-	-	-
$l=2$	$\{\neg b_t, a_{t-1}\}$	$\{\neg b_t, b_{t-1}\}$	$\{\neg a_{t-1}, b_{t-1}\}$	$\{\neg b_{t-1}, a_t\}$
$l=3$	$\{\neg a_t, \neg b_t, a_{t-1}\}$	$\{\neg a_t, \neg b_t, b_{t-1}\}$	$\{\neg a_t, \neg a_{t-1}, b_{t-1}\}$	$\{\neg b_t, \neg a_{t-1}, b_{t-1}\}$
$l=4$	$\{a_t, b_t, \neg a_{t-1}, \neg b_{t-1}\}$	$\{\neg a_t, b_t, \neg a_{t-1}, \neg b_{t-1}\}$	$\{a_t, \neg b_t, \neg a_{t-1}, \neg b_{t-1}\}$	$\{\neg a_t, \neg b_t, \neg a_{t-1}, \neg b_{t-1}\}$
Nodes	16	17	18	19
$l=0$	-	-	-	-
$l=1$	-	-	-	-
$l=2$	$\{\neg b_{t-1}, b_t\}$	$\{\neg b_{t-1}, a_{t-1}\}$	$\{\neg a_t, \neg b_t\}$	$\{\neg a_t, \neg a_{t-1}\}$
$l=3$	$\{a_t, b_t, \neg a_{t-1}\}$	$\{a_t, b_t, \neg b_{t-1}\}$	$\{a_t, a_{t-1}, \neg b_{t-1}\}$	$\{b_t, a_{t-1}, \neg b_{t-1}\}$
$l=4$	-	-	-	-
Nodes	20	21	22	23
$l=0$	-	-	-	-
$l=1$	-	-	-	-
$l=2$	$\{\neg a_t, \neg b_{t-1}\}$	$\{\neg b_t, \neg a_{t-1}\}$	$\{\neg b_t, \neg b_{t-1}\}$	$\{\neg a_{t-1}, \neg b_{t-1}\}$
$l=3$	$\{\neg a_t, b_t, \neg a_{t-1}\}$	$\{\neg a_t, b_t, \neg b_{t-1}\}$	$\{\neg a_t, a_{t-1}, \neg b_{t-1}\}$	$\{\neg b_t, a_{t-1}, \neg b_{t-1}\}$
$l=4$	-	-	-	-
Nodes	24	25	26	27
$l=0$	-	-	-	-
$l=1$	-	-	-	-
$l=2$	-	-	-	-
$l=3$	$\{a_t, \neg b_t, \neg a_{t-1}\}$	$\{a_t, \neg b_t, \neg b_{t-1}\}$	$\{a_t \neg a_{t-1}, \neg b_{t-1}\}$	$\{b_t, \neg a_{t-1}, \neg b_{t-1}\}$
$l=4$	-	-	-	-
Nodes	28	29	30	31
$l=0$	-	-	-	-
$l=1$	-	-	-	-
$l=2$	-	-	-	-
$l=3$	$\{\neg a_t, \neg b_t, \neg a_{t-1}\}$	$\{\neg a_t, \neg b_t, \neg b_{t-1}\}$	$\{\neg a_t, \neg a_{t-1}, \neg b_{t-1}\}$	$\{\neg b_t, \neg a_{t-1}, \neg b_{t-1}\}$
$l=4$	-	-	-	-

Table 5.3: Mapping of output nodes to rule bodies with various lengths

Prior State	Actual	δLFIT+
ϵ	ϵ	ϵ
a_t	ϵ	ϵ
b_t	ϵ	$\{a\}$
$a_t b_t$	ϵ	$\{a\}$
a_{t-1}	$\{b\}$	$\{b\}$
$a_t a_{t-1}$	$\{b\}$	$\{a\}$
$b_t a_{t-1}$	$\{b\}$	$\{b\}$
$a_t b_t a_{t-1}$	$\{b\}$	$\{a, b\}$
b_{t-1}	ϵ	ϵ
$a_t b_{t-1}$	ϵ	ϵ
$b_t b_{t-1}$	$\{a\}$	ϵ
$a_t b_t b_{t-1}$	$\{a\}$	ϵ
$a_{t-1} b_{t-1}$	ϵ	$\{b\}$
$a_t a_{t-1} b_{t-1}$	ϵ	ϵ
$b_t a_{t-1} b_{t-1}$	$\{a\}$	ϵ
$a_t b_t a_{t-1} b_{t-1}$	$\{a\}$	ϵ

Table 5.4: Predicted state transitions for the network with delays

	p	3-node-a	3-node-b	Raf
δLFIT+^{3}	$p = 0.125$	0.528	0.472	0.417
δLFIT+^{3}	$p = 0.25$	0.556	0.5	0.389
δLFIT+^{3}	$p = 0.375$	0.417	0.347	0.32
δLFIT+^{3}	$p = 0.5$	0.458	0.236	0.222
δLFIT+^{3}	$p = 0.625$	0.417	0.305	0.375
δLFIT+^{3}	$p = 0.75$	0.305	0.222	0.209
δLFIT+^{3}	$p = 0.875$	0.264	0.139	0.264

Table 5.5: The MSE for the state transitions generated by the predicted logic programs compared to δLFIT+ for general semantics.

Prior State	Actual	$p = 0.25$	$p = 0.75$
ϵ	{Raf}	{Mek, Raf}	{Raf}
{Erk}	{Mek, Raf}	{Erk, Mek, Raf}	{Raf}
{Mek}	{Raf}	{Raf}	{Raf}
{Erk, Mek}	{Erk, Mek, Raf}	{Erk, Mek, Raf}	{Erk, Raf}
{Raf}	{Raf}	{Erk, Raf}	{Erk, Raf}
{Erk, Raf}	{Mek}	{Erk}	{Mek}
{Mek, Raf}	{Erk, Mek, Raf}	{Erk, Raf}	{Erk, Mek}
{Erk, Mek, Raf}	{Erk, Mek}	{Mek}	{Erk, Mek}

Table 5.6: The difference state transitions for raf generated by the predicted logic programs compared to δLFIT+.

6

Related Work

We will provide a summary of some of the works that are most relevant to the work presented in this thesis. As noted before, this work presents a novel method of training a general neural network for many different problems. This has not yet been replicated in the field of NSAI.

Related works in the NSAI field can largely be categorized into different 2 approaches, the extraction-based approach and the integration-based approach. We will also introduce other neural-symbolic work which are not based in logic, namely the Neural Turing Machine and Neural-Programmer Interpreters. Then we will introduce NN-LFIT which is most similar to our work. And lastly, we will describe a new work that is done at the level that is similar to that of the LFIT framework itself.

6.1 Extraction-based Approach of NSAI

The extraction-based approach of NSAI concerns of training the model in the statistical space, and later attempting to retrieve symbolic knowledge from the trained statistical model. These approaches either heavily rely on expert knowledge to build out the initial neural network architecture base to extract from, or have serious limitations in their scalability.

6.1.1 Connectionist Approaches

One of the earliest methods of extracting symbolic knowledge from neural networks was demonstrated by d'Avila Garcez et al in [23]. In this work, which the authors named C-IL^{2}P, a background knowledge represented as logic programs are translated into a neural network with an algorithm. The translated neural network is then subsequently trained with examples, and then logical rules are extracted from it.

In [22], the authors proposed a much more general extraction algorithm, applicable not just to networks that were constructed in C-IL^{2}P. Further in [26] and subsequently [27] the authors further refined their extraction method. However, this still placed heavy constraints on the network the method was applicable to, and are sadly not applicable to deeper networks that employ modern state-of-the-art techniques to achieve higher performance.

Another extraction method, propsed in [50], utilizes interpretations to construct NLP. This method, again assumes a specific structure of a neural network that prohibits this method from being applied to more recent neural network structures.

6.1.2 Neural Markov Logic Networks

In [56], the authors introduced a model that are based on neural network, but which models a Markov Logic Networks. Given a knowledge base to train on, the Neural Markov Logic Networks can model the actual relationships between the entities within the knowledge base. Neural Markov Logic Networks can also

add new relationships to an incomplete knowledge base. Since this approach is more relational while δLFIT+ is more propositional, the approaches are not directly comparable. However this work lies more on the extraction line than the δLFIT+ approach.

6.1.3 δILP

The δ-ILP framework, introduced by Evans, et. al. in [24] incorporates a differentiable machine learning approach and the symbolic ILP approach. In the δ-ILP approach, the authors proposed a framework that uses differentiable methods to perform the ILP task at hand, while also simultaneously learning to produce the logic program that could solve the ILP task itself. In LFIT terms, this would be synonymous to a method that is differentiable all the way to predictions of the next state while also producing the logic program. However, in δLFIT+, we only aim to construct a differentiable method to learn any logic programs given any state transitions, while the prediction of the next states, that is the T_P operator is not within the differentiable method. Additionally, δ-ILP requires a language template and program template to significantly reduce the search space, whereas we don't impose such a requirement. This however may be an unfair comparison because in LFIT, the number of variables within the system can be known before hand. Whereas in ILP tasks, not knowing the predicates before hand could potentially explode the search space.

The core idea in this paper is to solve the ILP task by minimizing a loss using stochastic gradient descent. In this case, the loss is the cross entropy of the correct label with regard to the predictions of the system. Given an ILP task T, the probability of λ, which is the prediction, for a ground atom α is written as:

$$p(\lambda|\alpha, T)$$

This probability is computed by several differentiable operations and several non-differentiable operations. One of the differentiable operation, f_{infer} is where all the heavy-lifting takes place. After generating all the rules and clauses that are possible given the language template and program template, the f_{infer} operation performs a single step of forward chaining inference, in order to find the correct clause.

6.1.4 Relational Neural Machines

In [55], the authors introduced a method to learn and reason on first-order logic. This work is inline with δILP in that it integrates both the problem and also producing the logic program together. This is in contrast with δLFIT+ where no backpropagation learning for the problem itself is performed. It also differs in that the backpropagation learning has to be performed for each different problems whereas δLFIT+ doesn't require that.

6.1.5 LRNN

In [84], the authors introduced lifted relational neural network (LRNN). LRNN is a little bit different from the aforementioned works, in that LRNN starts with a logic program and constructs a neural network, to then subsequently find new relations within the knowledge base. It differs from our work in that it uses relational logic whereas LFIT only deals with proposition logic, and the method of integrating neural networks and symbolic are vastly different from δLFIT+.

6.1.6 NeuralLP

In [94], the authors introduced NeuralLP which are capable of learning rules based on a knowledge base. This work is similar to all the other approaches in that the neural network is trained to model the provided knowledge base, and then rules are extracted from within the trained network. This work however is applicable to relational logic.

6.2 Integration-based Approach of NSAI

The integration-based approaches, in contrast to the extraction-based approaches, are mostly about having separate statistical learning and symbolic algorithms. Since neural networks are good at dealing with real world data, these approaches utilize the neural networks to extract useful representations, which can then be used for the symbolic algorithms to do the reasoning.

6.2.1 NeurASP

In [95], the authors showed a method that combines neural network with a conventional ASP engine. The authors showed an interesting approach that allows the backpropagation of neural network to be guided by the symbolic reasoning engine. This work is mainly focused on bringing symbolic methods to deal with real world continuous data, such as image data. While δLFIT+ is partially concerned with interpretability and the opacity of neural networks.

6.2.2 NLProlog

In [92], the authors proposed NLProlog that integrates neural network and a Prolog solver. In this case the authors were concerned with attempting to solve Prolog problems that were described in natural language. To that end, the combination of neural network and symbolic was purely at an external level, where there is no cross over.

6.3 Neural Turing Machines

For the purposes of comparing and constrasting the various methods, we will regard the turing machine, namely the von-Neumann architecture [15], as being in the same category of symbolic manipulation, eventhough it is not strictly logical. The neural turing machines, described in [33], shows a novel method of integrating an algorithm that has shown to work in the symbolic world, into neural network architecture. In the Turing machine, there are several parts that are integral, namely the ability to read/write to memory and the ability to compute.

Neural networks have long been proven to have the ability to compute, since it can approximate any arbitrary function [38], but however lacked the ability to randomly access memory as necessary. In the work done by [33] however, the authors showed a method to allow neural networks to randomly access memory, and learning to do so by backpropagation.

6.3.1 Neural-Programmer Interpreters

Following Neural Turing Machines, Reed et al introduced the Neural-Programmer Interpreters in [67]. In Neural-Programmer Interpreters, in addition to having random access to memory, the model also outputs a trace of its execution. The model itself learns how to execute and solve a problem by training on examples, but as it executes, the trace of execution that it outputs resembles an assembly program, which can then be interpreted.

This is similar to the δLFIT+ approach, in that they are asking the neural network to provide the symbolic knowledge directly as output. The main difference however, is that Neural-Programmer Interpreters work in the imperative programming sense, whereas in the logical world, especially in ILP, they are declarative and thus have no execution trace.

6.4 NN-LFIT

NN-LFIT can be categorized as an extraction-based approach in the field of NSAI, but because it is directly related to this thesis in that it is also based on the LFIT framework, it deserves a section on its own.

In [29], NN-LFIT starts by training a minimal neural network that learns the target system, by ensuring the inputs and outputs of the neural network matches those of the target system. Since it is the aim of this algorithm to produce the *minimal* neural network, the training process starts by one hidden neuron, that is a hidden layer of dimension 1. The dimension of the hidden layer is continuously expanded until the accuracy of the neural network drops on the validation set after a fixed number of training steps. Once the minimal number of hidden layer dimension is obtained, the algorithm proceeds to zero out the trained parameters. Each trainable parameter is set to 0 in a fixed order, until the accuracy of the neural network starts to drop again.

Now that a minimal neural network, with a sparse trained parameter, that could model the system is obtained, the algorithm proceeds to extract the corresponding logic program in an analytical way. The algorithm inspects the sub-network that is connected to each of the outputs, which correspond to the prediction of a variable in the next state. Thus, the sub-network could reveal

the relationship between the inputs of the neural network, which represent the present state of the variables, and the corresponding output for the specific variable that the algorithm is currently inspecting. After obtaining all the relationships for the variable, the algorithm constructs a truth table by probing the neural network for all input possibilities, and then obtaining the logic formula that represents the rule for the variable on the output.

The experimental results presented in [29] showed that the algorithm was able to achieve high performance despite reducing the amount of training data. However, no experimental results were presented on the effect of noise and errors in the training data. Besides, this algorithm requires a new neural network to be train each time a different system is to be learned.

6.5 D-LFIT

D-LFIT, proposed in [28], is similar in many ways to δLFIT and δLFIT+ in which a differentiable method is used to learn intepretation transitions. The main difference though, similar to that of NN-LFIT, is that training is performed separately for each logic program. Contrary to NN-LFIT however, D-LFIT does not use neural networks or perceptrons, in that it does not consist of the usual learnable parameter, bias and activation function. It performs backpropagation on a mathematical linearly implemented T_P operator. The T_P operator is implemented by multiplying matrices and applying non-linearity to recover the consequences in boolean space. Therefore, to learn a logic program that conforms to a certain input sequence, is then a matter of searching for the correct logic program matrix. D-LFIT does this by performing backpropagation, hence the name differentiable.

However, due to the nature of the encoding of the logic program matrix, recovering the full logic program is non-trivial. The authors of D-LFIT solved this by introducing meta-info learner, which helps inform the shape of the logic program to be learned. In particular, the meta-info learned by this learner consists of information such as the number of rules per variable as the head. With this information, an appropriate matrix is then constructed to fully recover the logic program.

This approach is different from δLFIT and δLFIT+, where in δLFIT+ the

goal is not to learn a single logic program but to learn a general model that distinguishes various different systems of logic programs. The approach taken by D-LFIT to reduce the search space, which is the meta-learner, is unfortunately not applicable to δLFIT+. This is because the combinatorial explosivity of δLFIT+ is in the architectural sense whereas D-LFIT's explosivity is in the learning sense.

6.6 AND/OR BN

In [78], the authors proposed the learning of AND/OR BN which is a subclass of boolean network consisting only of logical AND and OR. The algorithm proposed to learn this class of boolean network in the linear mathematical space. The learning algorithm proposed is similar to that of D-LFIT, in that the encoding of the logic program is derived from the same family.

Considering only AND or OR within the logic program will obviously have a huge impact in the scaling factor of δLFIT+. The authors managed to use the scaling factor to achieve learning of real world genome datasets, which are a considerably larger problem than what δLFIT+ is currently able to deal with. However, we can predict that even by only considering AND/OR BN, the scaling factor for δLFIT+ is still combinatorial. This is because δLFIT+'s inherent problem lies in the combination of the variables, and not in the combination of the logical operators within the rules. Since δLFIT has a simpler scaling factor, we can attempt to analyze what the scale factor will be if we only consider AND/OR BN. To recall, the scale factor for the normal case is $n \times 3^n$, where we consider each literal being absent, positive or negative. For AND/OR BN, this will be $2n \times 2^n$, where we will have separate rules for AND and OR, and each literal will be either present or absent. This is obviously, still exponential.

6.7 Apperception

In [25], the authors introduced an algorithm they called Apperception Engine. This algorithm is similar to LFIT and δILP in that it is trying to learn the rules given observed data. The authors defined and introduced the framework of apperception tasks, that not only takes into account the time-based dynamics

LFIT, but also spatial dynamics as well. While LFIT is strictly propositional, apperception tasks are first-oder logics and thus, in general, a much harder problem to solve. The authors introduced the algorithm, Apperception Engine that could solve the problems, but the algorithm needs to be provided a program template, which could be seen as prior knowledge, before solving the problem. In contrast to that, LFIT requires no templates and can learn logic programs without any prior knowledge.

7

Conclusion

In this thesis, we have investigated a method that will integrate the statistical machine learning and the symbolic machine learning method. In particular, we focused on the LFIT framework which learns the dynamics of a system based on the observation of its state transitions. We devised a method to directly employ the strength of statistical machine learning in classification to obtain symbolic knowledge. We have also shown several techniques that overcome some of the hurdles that have plagued the neural symbolic field, particularly the combinatorial explosion problem.

7.1 Summary of contribution

In chapter 3, we introduced the δLFIT algorithm, which is an algorithm that classifies logic programs based on the given state transitions. We first investigated whether neural networks are capable of classifying logic programs. Next, we described our neural network architecture that learns the logic program representation, and then performs classification of rules based on the representa-

tion. We described our method of enumerating through all possible rules. We show that our method works with clean and noisy data. We then show several examples of δLFIT algorithm in action.

In chapter 4, we introduced our techniques of enabling neural network to learn larger systems. We first introduced the integration of Set Transformer, that ensured different permutations of the state transitions do not affect the output. Next, we introduced a method of encoding state transitions, in a manner that allows the neural network to focus on what matters most, that is the positive examples and the negative examples for a particular system. We then introduced a technique to reduce the combinatorial factor that exists inherently in the neural network architecture, due to how we structure the problem. By reusing the output nodes for multiple different rule heads and rule body lengths, we were able to cut the combinatorial factor that allowed us to perform experiments on larger systems. Following that, we showed some technique to allow the model to converge better due to the improvements that were introduced above. We considered label smoothing with subsumed rules, and target weighting based on the imbalanced distribution of learning targets for the model.We then performed several experiments to validate the improvement techniques that we have introduced. Due to a pretty structured problem space, we were able to map out the entire problem space. However the problem space is really huge, and due to computational limitations it is infeasible to provide the entire problem space as training data. We therefore introduced our method of sampling the problem space. Based on this, we showed several experiments based on well-known Boolean network benchmarks, and showed the effectiveness of our methods. We have also investigated and found that covering more of the problem space increases the accuracy of the model, which is in-line with what is expected.

Finally in chapter 5, we also extended our techniques to systems that have delayed influences, and also to a much more general semantics. We showed that our techniques were effective, even with minimal modifications to the model. This is consistent with the expectation that neural networks are much better generalizers.

7.2 Future Directions

There are still several potentially promising directions that are worth investigating.

Logic program generation is a potentially interesting direction. The combinatorial explosion inherent in our method is due to the fixation of the output node and the requirement that all possible rules are being classified. If, instead of this, there is a technique that allows the neural network to directly output the logic program rule by rule, literal by literal, then this should solve the combinatorial explosion problem.

Variable permutation invariance is also an interesting direction to pursue. Given that the names of the variables are arbitrarily assigned by humans, the ordering in which variable goes where within the neural network is also pretty much arbitrary. By having an invariance that allows variables to freely permute both in the input states and the output logic programs will cut the problem space by a very significant margin.

Distilling logic programs will be another potential direction. As the number of variables increase, and the number of rules increase, the logic program becomes more complicated. There are some arguments that the interpretability property of a logic program will be lost when the logic program is simply too complicated for one to understand, as it is as opaque as a string of numbers. With our method of learning the entire problem space and getting the general semantics of all logic programs given the number of variables, it might be possible to have the neural network identify a *core part* of the logic program and ensure that the core part remains readable and understandable.

References

[1] Abadi, M., Agarwal, A., Barham, P., et al. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

[2] Abou-Jaoudé, W., Ouattara, D. A., & Kaufman, M. (2009). From structure to dynamics: frequency tuning in the p53–mdm2 network: I. logical approach. *Journal of theoretical biology*, 258(4), 561–577.

[3] Adadi, A. & Berrada, M. (2018). Peeking inside the black-box: a survey on explainable artificial intelligence (xai). *IEEE access*, 6, 52138–52160.

[4] Agarap, A. F. (2018). Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375*.

[5] Alam, M., Samad, M. D., Vidyaratne, L., Glandon, A., & Iftekharuddin, K. M. (2020). Survey on deep neural networks in speech and vision systems. *Neurocomputing*, 417, 302–321.

[6] Alom, M. Z., Taha, T. M., Yakopcic, C., Westberg, S., Sidike, P., Nasrin, M. S., Van Esesn, B. C., Awwal, A. A. S., & Asari, V. K. (2018). The history began from alexnet: A comprehensive survey on deep learning approaches. *arXiv preprint arXiv:1803.01164*.

[7] Apt, K. R., Blair, H. A., & Walker, A. (1988). Towards a theory of declarative knowledge. In *Foundations of deductive databases and logic programming* (pp. 89–148). Elsevier.

[8] Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016). Layer normalization. *arXiv preprint arXiv:1607.06450*.

[9] Bader, S. & Hitzler, P. (2005). Dimensions of neural-symbolic integration-a structured survey. *arXiv preprint cs/0511042*.

[10] Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

[11] Bashar, A. (2019). Survey on evolving deep learning neural network architectures. *Journal of Artificial Intelligence*, 1(02), 73–82.

[12] Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *Trans. Neur. Netw.*, 5(2), 157–166.

[13] Besold, T. R., d'Avila Garcez, A., Bader, S., Bowman, H., Domingos, P., Hitzler, P., Kuehnberger, K.-U., Lamb, L. C., Lowd, D., Lima, P. M. V., de Penning, L., Pinkas, G., Poon, H., & Zaverucha, G. (2017). Neural-symbolic learning and reasoning: A survey and interpretation.

[14] Binns, R. (2018). Algorithmic accountability and public reason. *Philosophy & technology*, 31(4), 543–556.

[15] Bowden, B. V. (1953). *Faster than thought: A symposium on digital computing machines*. Pitman Publishing, Inc.

[16] Brown, T. B., Mann, B., et al. (2020). Language models are few-shot learners. *CoRR*, abs/2005.14165.

[17] Carvalho, D. V., Pereira, E. M., & Cardoso, J. S. (2019). Machine learning interpretability: A survey on methods and metrics. *Electronics*, 8(8), 832.

[18] Char, D. S., Shah, N. H., & Magnus, D. (2018). Implementing machine learning in health care—addressing ethical challenges. *The New England journal of medicine*, 378(11), 981.

[19] Ciresan, D., Giusti, A., Gambardella, L., & Schmidhuber, J. (2012). Deep neural networks segment neuronal membranes in electron microscopy images. *Advances in neural information processing systems*, 25, 2843–2851.

[20] Clark, K. L. (1978). Negation as failure. In *Logic and data bases* (pp. 293–322). Springer.

[21] Comet, J.-P., Bernot, G., Das, A., Diener, F., Massot, C., & Cessieux, A. (2012). Simplified models for the mammalian circadian clock. *Procedia Computer Science*, 11, 127–138.

[22] d'Avila Garcez, A. S., Broda, K., & Gabbay, D. M. (2001). Symbolic knowledge extraction from trained neural networks: A sound approach. *Artificial Intelligence*, 125(1), 155–207.

[23] d'Avila Garcez, A. S. & Zaverucha, G. (1999). The connectionist inductive learning and logic programming system. *Applied Intelligence*, 11(1), 59–77.

[24] Evans, R. & Grefenstette, E. (2017). Learning explanatory rules from noisy data. *CoRR*, abs/1711.04574.

[25] Evans, R., Hernández-Orallo, J., Welbl, J., Kohli, P., & Sergot, M. (2021). Making sense of sensory input. *Artificial Intelligence*, 293, 103438.

[26] França, M. V., Zaverucha, G., & Garcez, A. S. d. (2014). Fast relational learning using bottom clause propositionalization with artificial neural networks. *Machine learning*, 94(1), 81–104.

[27] França, M. V. M., Garcez, A. S. d., & Zaverucha, G. (2015). Relational knowledge extraction from neural networks. In *CoCo@ NIPS*.

[28] Gao, K., Wang, H., Cao, Y., & Inoue, K. (2021). Learning from interpretation transition using differentiable logic programming semantics. *Machine Learning*, (pp. 1–23).

[29] Gentet, E., Tourret, S., & Inoue, K. (2016). Learning from interpretation transition using feed-forward neural network. In *Proceedings of ILP 2016, CEUR Proc. 1865* (pp. 27–33).

[30] Gilpin, L. H., Bau, D., Yuan, B. Z., Bajwa, A., Specter, M., & Kagal, L. (2018a). Explaining explanations: An overview of interpretability of machine learning. In *2018 IEEE 5th International Conference on Data Science and Advanced Analytics (DSAA)* (pp. 80–89).

[31] Gilpin, L. H., Bau, D., Yuan, B. Z., Bajwa, A., Specter, M., & Kagal, L. (2018b). Explaining explanations: An overview of interpretability of machine learning. In *2018 IEEE 5th International Conference on data science and advanced analytics (DSAA)* (pp. 80–89).: IEEE.

[32] Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT press.

[33] Graves, A., Wayne, G., & Danihelka, I. (2014). Neural turing machines. *arXiv preprint arXiv:1410.5401*.

[34] Hammer, B. & Hitzler, P. (2007). *Perspectives of neural-symbolic integration*, volume 77. Springer.

[35] Hecht-Nielsen, R. (1992). Theory of the backpropagation neural network. In *Neural networks for perception* (pp. 65–93). Elsevier.

[36] Hochreiter, S. & Schmidhuber, J. (1997). Long short-term memory. *Neural Comput.*, 9(8), 1735–1780.

[37] Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural Netw.*, 4(2), 251–257.

[38] Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5), 359–366.

[39] Inoue, K. (2011). Logic programming for boolean networks. In *Twenty-Second International Joint Conference on Artificial Intelligence*.

[40] Inoue, K. (2012). Dnf hypotheses in explanatory induction. In S. H. Muggleton, A. Tamaddoni-Nezhad, & F. A. Lisi (Eds.), *Inductive Logic Programming* (pp. 173–188). Berlin, Heidelberg: Springer Berlin Heidelberg.

[41] Inoue, K., Ribeiro, T., & Sakama, C. (2014). Learning from interpretation transition. *Machine Learning*, 94(1), 51–79.

[42] Inoue, K. & Sakama, C. (2012). Oscillating behavior of logic programs. In *Correct Reasoning* (pp. 345–362). Springer.

[43] Ioffe, S. & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning* (pp. 448–456).: PMLR.

[44] Ishibuchi, H. & Nojima, Y. (2007). Analysis of interpretability-accuracy tradeoff of fuzzy systems by multiobjective fuzzy genetics-based machine learning. *International Journal of Approximate Reasoning*, 44(1), 4–31.

[45] Jumper, J., Evans, R., Pritzel, A., Green, T., Figurnov, M., Ronneberger, O., Tunyasuvunakool, K., Bates, R., Žídek, A., Potapenko, A., et al. (2021). Highly accurate protein structure prediction with alphafold. *Nature*, 596(7873), 583–589.

[46] Kauffman, S. A. (1969). Homeostasis and differentiation in random genetic control networks. *Nature*, 224, 177–178.

[47] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, & K. Q. Weinberger (Eds.), *Advances in Neural Information Processing Systems*, volume 25: Curran Associates, Inc.

[48] Lamb, L. C., Garcez, A., Gori, M., Prates, M., Avelar, P., & Vardi, M. (2020). Graph neural networks meet neural-symbolic computing: A survey and perspective. *arXiv preprint arXiv:2003.00330*.

[49] Lee, J., Lee, Y., Kim, J., Kosiorek, A. R., Choi, S., & Teh, Y. W. (2018). Set transformer. *CoRR*, abs/1810.00825.

[50] Lehmann, J., Bader, S., & Hitzler, P. (2010). Extracting reduced logic programs from artificial neural networks. *Applied Intelligence*, 32(3), 249–266.

[51] Li, M., Zhang, T., Chen, Y., & Smola, A. J. (2014). Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 661–670).

[52] Liao, H.-J., Liu, J.-G., Wang, L., & Xiang, T. (2019). Differentiable programming tensor networks. *Physical Review X*, 9(3), 031041.

[53] Lipton, Z. C. (2018). The mythos of model interpretability: In machine learning, the concept of interpretability is both important and slippery. *Queue*, 16(3), 31–57.

[54] Marcus, G. (2020). The next decade in ai: four steps towards robust artificial intelligence. *arXiv preprint arXiv:2002.06177*.

[55] Marra, G., Diligenti, M., Giannini, F., Gori, M., & Maggini, M. (2020). Relational neural machines. *arXiv preprint arXiv:2002.02193*.

[56] Marra, G. & Kuželka, O. (2019). Neural markov logic networks. *arXiv preprint arXiv:1905.13462*.

[57] Martínez, D., Alenyà, G., Ribeiro, T., Inoue, K., & Torras, C. (2017). Relational reinforcement learning for planning with exogenous effects. *Journal of Machine Learning Research*, 18(78), 1–44.

[58] Muggleton, S. (1992). *Inductive logic programming*. Number 38. Morgan Kaufmann.

[59] Muggleton, S. & de Raedt, L. (1994). Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19-20, 629 – 679. Special Issue: Ten Years of Logic Programming.

[60] Nguyen, A., Yosinski, J., & Clune, J. (2015). Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 427–436).

[61] Olah, C., Satyanarayan, A., Johnson, I., Carter, S., Schubert, L., Ye, K., & Mordvintsev, A. (2018). The building blocks of interpretability. *Distill*, 3(3), e10.

[62] O'Mahony, N., Campbell, S., Carvalho, A., Harapanahalli, S., Hernandez, G. V., Krpalkova, L., Riordan, D., & Walsh, J. (2019). Deep learning vs. traditional computer vision. In *Science and Information Conference* (pp. 128–144).: Springer.

[63] Paczuski, M., Bassler, K. E., & Corral, Á. (2000). Self-organized networks of competing boolean agents. *Physical Review Letters*, 84(14), 3185.

[64] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. (2019). Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 8026–8037.

[65] Phua, Y. J. & Inoue, K. (2019). Learning logic programs from noisy state transition data. In *International Conference on Inductive Logic Programming* (pp. 72–80).: Springer.

[66] Phua, Y. J., Ribeiro, T., & Inoue, K. (2019). Learning representation of relational dynamics with delays and refining with prior knowledge. *FLAP*, 6(4), 695–708.

[67] Reed, S. & De Freitas, N. (2015). Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*.

[68] Ribeiro, T., Folschette, M., Magnin, M., & Inoue, K. (2021). Learning any memory-less discrete semantics for dynamical systems represented by logic programs. working paper or preprint.

[69] Ribeiro, T., Folschette, M., Magnin, M., Roux, O., & Inoue, K. (2018a). Learning dynamics with synchronous, asynchronous and general semantics. In *International Conference on Inductive Logic Programming* (pp. 118–140).: Springer.

[70] Ribeiro, T. & Inoue, K. (2015). Learning prime implicant conditions from interpretation transition. In *ILP 2015* (pp. 108–125). Springer.

[71] Ribeiro, T., Magnin, M., Inoue, K., & Sakama, C. (2015a). Learning delayed influences of biological systems. *Frontiers in Bioengineering and Biotechnology*, 2, 81.

[72] Ribeiro, T., Magnin, M., Inoue, K., & Sakama, C. (2015b). Learning multi-valued biological models with delayed influence from time-series observations. In *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)* (pp. 25–31).

[73] Ribeiro, T., Tourret, S., Folschette, M., Magnin, M., Borzacchiello, D., Chinesta, F., Roux, O., & Inoue, K. (2018b). Inductive learning from state transitions over continuous domains. In N. Lachiche & C. Vrain (Eds.), *Inductive Logic Programming* (pp. 124–139). Cham: Springer International Publishing.

[74] Rolnick, D., Veit, A., Belongie, S., & Shavit, N. (2018). Deep learning is robust to massive label noise.

[75] Rosenblatt, F. (1957). *The perceptron, a perceiving and recognizing automaton Project Para*. Cornell Aeronautical Laboratory.

[76] Samek, W., Montavon, G., Vedaldi, A., Hansen, L. K., & Müller, K.-R. (2019). *Explainable AI: interpreting, explaining and visualizing deep learning*, volume 11700. Springer Nature.

[77] Samek, W., Wiegand, T., & Müller, K.-R. (2017). Explainable artificial intelligence: Understanding, visualizing and interpreting deep learning models. *arXiv preprint arXiv:1708.08296*.

[78] Sato, T. & Kojima, R. (2021). Boolean network learning in vector spaces for genome-wide network analysis. In *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning Special Session on KR and Machine Learning*.

[79] Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., & Monfardini, G. (2008). The graph neural network model. *IEEE transactions on neural networks*, 20(1), 61–80.

[80] Seltzer, M. L., Yu, D., & Wang, Y. (2013). An investigation of deep neural networks for noise robust speech recognition. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on* (pp. 7398–7402).: IEEE.

[81] Silva, S. H. & Najafirad, P. (2020). Opportunities and challenges in deep learning adversarial robustness: A survey. *arXiv preprint arXiv:2007.00753*.

[82] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587), 484–489.

[83] Simonyan, K., Vedaldi, A., & Zisserman, A. (2013). Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv preprint arXiv:1312.6034*.

[84] Sourek, G., Aschenbrenner, V., Zelezny, F., Schockaert, S., & Kuzelka, O. (2018). Lifted relational neural networks: Efficient learning of latent relational structures. *Journal of Artificial Intelligence Research*, 62, 69–100.

[85] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1), 1929–1958.

[86] Streck, A., Siebert, H., & Klarner, H. (2016). PyBoolNet: a python package for the generation, analysis and visualization of boolean networks. *Bioinformatics*, 33(5), 770–772.

[87] Su, J., Vargas, D. V., & Sakurai, K. (2019). One pixel attack for fooling deep neural networks. *IEEE Transactions on Evolutionary Computation*, 23(5), 828–841.

[88] Sun, C., Shrivastava, A., Singh, S., & Gupta, A. (2017). Revisiting unreasonable effectiveness of data in deep learning era. *CoRR*, abs/1707.02968.

[89] Valiant, L. G. (2008). Knowledge infusion: In pursuit of robustness in artificial intelligence. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*: Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[90] Van Emden, M. H. & Kowalski, R. A. (1976). The semantics of predicate logic as a programming language. *Journal of the ACM (JACM)*, 23(4), 733–742.

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

[92] Weber, L., Minervini, P., Münchmeyer, J., Leser, U., & Rocktäschel, T. (2019). Nlprolog: Reasoning with weak unification for question answering in natural language. *arXiv preprint arXiv:1906.06187*.

[93] Xiao, Y. & Dougherty, E. R. (2007). The impact of function perturbations in Boolean networks. *Bioinformatics*, 23(10), 1265–1273.

[94] Yang, F., Yang, Z., & Cohen, W. W. (2017). Differentiable learning of logical rules for knowledge base reasoning. *arXiv preprint arXiv:1702.08367*.

[95] Yang, Z., Ishay, A., & Lee, J. (2020). NeurASP: Embracing neural networks into answer set programming. In *IJCAI* (pp. 1755–1762).

[96] Yurtsever, E., Lambert, J., Carballo, A., & Takeda, K. (2020). A survey of autonomous driving: Common practices and emerging technologies. *IEEE access*, 8, 58443–58469.

[97] Zeiler, M. D. (2012). ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701.

[98] Zeiler, M. D. & Fergus, R. (2014). Visualizing and understanding convolutional networks. In *European conference on computer vision* (pp. 818–833).: Springer.