Machine Learning (2022) 111:123–145
https://doi.org/10.1007/s10994-021-06058-8

Learning from interpretation transition using differentiable logic programming semantics

Kun Gao[1] · Hanpin Wang[1,2] · Yongzhi Cao[1] · Katsumi Inoue[3]

Received: 3 September 2020 / Revised: 24 July 2021 / Accepted: 6 August 2021 /
Published online: 14 September 2021

Abstract
The combination of learning and reasoning is an essential and challenging topic in neuro-symbolic research. Differentiable inductive logic programming is a technique for learning a symbolic knowledge representation from either complete, mislabeled, or incomplete observed facts using neural networks. In this paper, we propose a novel differentiable inductive logic programming system called differentiable learning from interpretation transition (D-LFIT) for learning logic programs through the proposed embeddings of logic programs, neural networks, optimization algorithms, and an adapted algebraic method to compute the logic program semantics. The proposed model has several characteristics, including a small number of parameters, the ability to generate logic programs in a curriculum-learning setting, and linear time complexity for the extraction of trained neural networks. The well-known bottom clause positionalization algorithm is incorporated when the proposed system learns from relational datasets. We compare our model with NN-LFIT, which extracts propositional logic rules from retuned connected networks, the highly accurate rule learner RIPPER, the purely symbolic LFIT system LF1T, and CILP++, which integrates neural networks and the propositionalization method to handle first-order logic knowledge. From the experimental results, we conclude that D-LFIT yields comparable accuracy with respect to the baselines when given complete, incomplete, and mislabeled data. Our experimental results indicate that D-LFIT not only learns symbolic logic programs quickly and precisely but also performs robustly when processing mislabeled and incomplete datasets.

Keywords Machine learning · Differentiable inductive logic programming · Explainability · Neuro-symbolic method · Learning from interpretation transition

Editors: Nikos Katzouris, Alexander Artikis, Luc De Raedt, Artur d'Avila Garcez, Sebastijan Dumančić, Ute Schmid, Jay Pujara.

The comments and suggestions of the editors and the anonymous reviewers are gratefully acknowledged.

✉ Hanpin Wang
whpxhy@pku.edu.cn

Extended author information available on the last page of the article

1 Introduction

Human cognition has been successfully modelled by machine learning. Moreover, learning human-readable knowledge from a large amount of data using machines is an easy and fast way for people to understand the facts hidden in the data. However, real datasets, especially in biology, are observed in an incomplete or noisy manner. Hence, extracting high accuracy and reliable knowledge is a difficult and important task in the field of machine learning. Inductive logic programming (ILP) was first proposed by Muggleton (1991) as a combination of inductive learning and logic programming techniques, and it has gained widespread attention in the machine learning community. Given a set of observations (examples), the ILP framework induces a logic program that explains all examples. Differentiable ILP systems, first introduced by Evans and Grefenstette (2018) adopt neural networks to automatically learn declarative theories hidden in the data. Compared with purely symbolic ILP methods, differentiable ILP systems are robust to incomplete or mislabeled data. Hence, differentiable ILP systems can help people cognize from these data in a more effective way.

The learning from interpretation transition (LFIT) approach proposed by Inoue et al. (2014) is an important sub-field of ILP. When interpreting a logic program as a state transition system (Inoue 2011; Inoue and Sakama 2012), a Herbrand interpretation represents the current state of the world, and a logic program P specifies how to define the next state of the world as a Herbrand interpretation through the immediate consequence operator (also called the T_P operator) (Van Emden and Kowalski 1976; Apt et al. 1988). The learning setting of LFIT is as follows: given a set of pairs of input and output Herbrand interpretations (I, J) such that $J = T_P(I)$ as positive examples, the goal is to induce a normal logic program P that realizes the given transition relations (Inoue et al. 2014).

In this paper, we propose a differentiable LFIT system called D-LFIT, which uses neural networks to learn the symbolic logic programs from Boolean network datasets (Kauffman et al. 1993) and relational datasets. For Boolean networks, it has been observed that the T_P operator of a normal logic program P precisely captures the synchronous updates of the corresponding Boolean network, where each node corresponds to a ground atom and its regulation function corresponds to the set of ground rules that have the atom in their heads (Inoue 2011). Given an observed interpretation pair (I, J) that corresponds to a transition of the gene activity profile at a time step such that $J = T_P(I)$, the LFIT of a normal logic program P corresponds to inferring a set of regulation rules describing the Boolean network. For relational datasets, to build a first-order logic program to describe the association between a relation and other relations, we utilize a propositionalization method (Kramer et al. 2001) called bottom clause positionalization (BCP) (França et al. 2014) to generate a set of bottom clauses G in the first-order form. With most literals, BCP generates a bottom clause, which can be considered to be a hypothesis for each observed relational example. Each predicate in a bottom clause is called a first-order feature. By regarding the bottom clause as a propositional logic program, we regard an input Herbrand interpretation I as a set of first-order features from the body of a bottom clause and an output Herbrand interpretation J as a set of first-order features from the head of the bottom clause, where I and J satisfy $J = T_G(I)$. Because the numbers of bottom clauses and the first-order features in the bodies of the bottom clauses are both large, the LFIT process of a normal logic program P in a relational dataset corresponds to inferring simpler and more legible logic programs describing the relational datasets.

To implement D-LFIT, we first define low-rank embeddings for logic programs and their interpretations in vector space. To compute the logic program semantics, we adapt an algebraic method that was first proposed by Sakama et al. (2018) to implement the deductive reasoning of logic programs in vector space. Through the semantics, discrete logical operations are transferred into a series of matrix computations. Then, we devise neural networks to match the logical inference defined by the semantics and search for the optimal embeddings of the logic programs through optimization algorithms. During the learning processes, we reduce the search space and boost the learning speed through curriculum learning: the system consolidates what it learns in one episode, stores it as background knowledge, and reuses it in subsequent episodes (Bengio et al. 2009). After the learning processes, we can extract symbolic logic programs directly from the embeddings, which means that the time complexity for extracting rules is linear. In experiments, we evaluated the proposed method on four Boolean network datasets, three relational datasets, and their incomplete and mislabeled data. The results demonstrate the advantages of our model.

In short, D-LFIT can learn human-readable knowledge from real-world data through neural networks. In addition, D-LFIT enables humans to understand how neural networks make decisions; for instance, by predicting a unique output interpretation according to the input interpretation. A better understanding of the way in which neural networks derive their decisions could also make neural networks more transparent and develop our understanding of the relationship between black-box models such as neural networks and human-readable models such as logic programs.

Our main contribution is two-fold:

- We adapt the algebraic method to compute logic program semantics and devise a practical algorithm for learning logic programs.
- We demonstrate that the proposed approach performs faster and more precisely on incomplete, mislabeled datasets when compared with various strong baselines.

The rest of the paper is organized as follows. We present an overview of related concepts in Sect. 2. In Sect. 3, we extend the algebraic method to enable it to compute the logic program semantics and describe the details of the proposed framework. In Sect. 4, we measure the robustness of the proposed framework on mislabeled and incomplete data. In addition, we evaluate the accuracy of logic programs generated by our model and several baselines. We review related ILP systems in Sect. 5 and compare them with the proposed method. In Sect. 6, we conclude the paper and describe further research plans.

2 Background

In this section, the notions used in logic programs, ILP, Boolean networks, and propositionalization are reviewed. This section also introduces the notations used throughout the paper.

2.1 Logic programs and ILP

A normal logic program includes several notions. Moreover, a normal logic program is a finite set of rules that satisfies the following form:

$$h \leftarrow l_1 \wedge l_2 \wedge \cdots \wedge l_n, \tag{1}$$

where the l_i's are literals and h is the head atom. In first-order logic programs, an atom a is a tuple $p(\mathbf{x})$, where p is a predicate and $\mathbf{x}$ is its argument. An argument is either a variable or a constant. A ground atom in first-order logic is an atom with no variables, and the set of ground instances of all rules in a first-order logic program P is denoted as $ground(P)$. In propositional logic programs, an atom a is a Boolean variable. The literal l_i can be an atom a_i or its negation $\neg a_i$. Given a logic program P, the set of all atoms (ground atoms) in P is called a Herbrand base and is denoted as B_P. A logic program P is a definite program if there are no negations of atoms in each rule of the program. Given rule r in the form (1), we denote the head of r as $head(r) = \{h\}$ and the body of r as $body(r) = \{l_1, \dots, l_n\}$. We also represent the body atoms appearing in the body of r positively and negatively as $body^+(r) = \{a_1, \dots, a_n\}$ and $body^-(r) = \{\neg a_1, \dots, \neg a_n\}$, respectively. In particular, if $n = 0$ in r, then we call the corresponding ground head atom h in r a fact, and its Boolean value is always *True*.

An interpretation I of a normal logic program P is a subset of B_P. Given a normal logic program P and $B_P = \{p_1, \dots, p_n\}$, an interpretation vector (Sakama et al. 2018; Nguyen et al. 2018) is a vector $\mathbf{v} = (v_1, \dots, v_n)^{\mathrm{T}} \in \{0, 1\}^n$. An interpretation vector represents an interpretation $I \subseteq B_P$. If $v_i = 1$, then the truth value of p_i is *True*, and $p_i \in I$. Otherwise, the truth value of p_i is *False*, and $p_i \notin I$. Given an interpretation vector $\mathbf{v}$, we define $row_i(\mathbf{v}) = \mathbf{v}[i]$. A mapping called the immediate consequence operator $T_{PP} : 2^{B_{PP}} \rightarrow 2^{B_{PP}}$ for a propositional logic program PP is defined as follows:

$$T_{PP}(I) = \{head(r) \mid r \in PP,\ body^+(r) \subseteq I,\ body^-(r) \cap I = \emptyset\}.$$

For a first-order logic program FP, the immediate consequence operator $T_{FP} : 2^{B_{FP}} \rightarrow 2^{B_{FP}}$ is defined as follows:

$$T_{FP}(I) = \{head(r) \mid r \in ground(FP),\ body^+(r) \subseteq I,\ body^-(r) \cap I = \emptyset\}.$$

In ILP, a problem can be defined as a tuple $(\mathscr{B}, \mathscr{P}, \mathscr{N})$. Set $\mathscr{B}$ is the set of facts and a set of clauses called background knowledge in the form (1). Sets $\mathscr{P}$ and $\mathscr{N}$ are the sets of positive and negative examples, respectively. Given this setting, the goal of an ILP is to construct a logic program (hypothesis) P that explains all the examples. The ILP tasks can be formally expressed as follows:

$$\mathscr{B},\ P \models e^+,\ \forall e^+ \in \mathscr{P} \quad , \quad \mathscr{B},\ P \not\models e^-,\ \forall e^- \in \mathscr{N}.$$

2.2 Boolean networks

A Boolean network is a pair $N = (V, F)$, where $V = \{v_1, \dots, v_n\}$ is a set of finite nodes (n is the number of nodes), and $F = \{f_1, \dots, f_n\}$ is the corresponding set of Boolean functions. Each single state of the Boolean network is regarded as $(v_1(t), ..., v_n(t))^T$, and the state transitions are regarded as $v_i(t+1) = f_i(v_{i_1}(t), \dots, v_{i_k}(t))$, where $v_{i_1}, \dots, v_{i_k}$ are the input nodes of v_i. Each state of N at time step t is $S(t) = (v_1(t), \dots, v_n(t))$. The trajectory of N is a sequence of states obtained by a series of state transitions. As $|V|$ is finite, every trajectory always reaches some attractor (Kauffman et al. 1993; Inoue 2011), which is either a fixed point or a periodic oscillation. Additionally, for each $v_i \in V$, its Boolean function $f_i(v_{i_1}(t), \dots, v_{i_k}(t))$ is equal to a set of propositional rules in the form (1), and we can convert any Boolean functions into a set of normal logic rules $\pi(N)$ in three steps. First, each Boolean function is transferred to a formula in the disjunctive normal form (DNF) $\bigvee_{j=1}^{l_i} B_{i,j}^t$, where $B_{i,j}^t$ is a conjunction of literals at time t. Second, we generate l_i rules with v_i^{t+1} as the head and $B_{i,j}^t$ as the body for each $j = 1, ..., l_i$. Finally, we delete all time arguments from every literal in the above rules (Inoue

2011). Then, we can simulate the trajectory of N from any state $S(0)$ by the orbit of the interpretation $I^0 = \{v_i \in V | v_i(0) \text{ is true}\}$ with respect to the $T_{\pi(N)}$ operator, i.e., $I^{t+1} = T_{\pi(N)}(I^t)$ for $t \geq 0$ (Inoue et al. 2014). Hence, devising a system in LFIT to learn a normal logic program P that matches the behaviors of the corresponding Boolean network is feasible.

2.3 Propositionalization and bottom clauses

Propositionalization transfers a relational database into an attribute–value table amenable to propositional learners (Kramer et al. 2001). Propositionalization algorithms use facts and examples to find distinctive first-order features. Bottom clauses are the most specific clauses, which can be regarded as boundaries on the hypothesis search space, first introduced by Muggleton (1995) as a part of the Progol system. Bottom clauses are built from one random positive example, facts, and language bias, which defines how a single rule is constructed.

There are two steps in the BCP algorithm: The first step is bottom clause generation. The BCP proposed by França et al. (2014) generates bottom clauses for all examples and maintains their semantic meaning. The BCP algorithm extends the original bottom clauses algorithm to learn the most specific clauses from both positive and negative examples, facts, and language bias.

Each example e_i in $\mathscr{E}$, where $\mathscr{E} = \mathscr{P} \cup \mathscr{N}$, is sent to the adapted bottom clause generation algorithm proposed by Tamaddoni-Nezhad and Muggleton (2009) to generate the corresponding bottom clause set $\mathscr{E}_\perp$. In the adapted algorithm, the same hash function that transfers examples to clauses is shared among all examples to maintain consistency among variable associations. The hash function also allows negative examples to have bottom clauses. In the second step, the BCP regards all bottom clause literals as first-order features and adds all features to feature table F. Then, BCP uses the following algorithm to transfer the examples into their embeddings:

1. Let $|F|$ be the number of the elements in F
2. For each bottom clause $\perp_{e_i} \in \mathscr{E}_\perp$ do
   (a) Create a numerical vector v_i of size $|F|$ and with 0 in all positions
   (b) For each position corresponding to a body literal of $\perp_{e_i}$, change its value to 1
   (c) Add v_i to $\mathscr{E}_v$;
3. Return $\mathscr{E}_v$.
4. Associate a label 1 to v_i if e_i is a positive example, and 0 otherwise;

Consider a motivating example called the *motherInLaw* relationship from a family relationship dataset (Muggleton and De Raedt 1994): $\mathscr{F} = \{mother(mom1, daughter1), wife(daughter1, husband1), wife(daughter2, husband2)\}$, $\mathscr{P} = \{motherInLaw(mom1, husband1)\}$, and $\mathscr{N} = \{motherInLaw(daughter1, husband2)\}$. The BCP algorithm analyzes each example and generates the following bottom clause set (the depth of the variable is set to 1):

$$E_\perp = \{motherInLaw(A, B) : -mother(A, C), wife(C, B);$$
$$\sim motherInLaw(A, B) : -wife(A, C)\}.$$

Then, the BCP generates vector (1, 1, 0) with label 1 for the example *motherInLaw(mom*1, *husband*1) and vector (0, 0, 1) with label 0 for the example *motherInLaw*(*daughter*1, *husband*2).

3 Methods

In this section, we first introduce the details of the algebraic method used to compute the logic program semantics and then present the proposed D-LFIT framework. The algebraic method to compute the logic program semantics first proposed by Sakama et al. (2018), embeds definite programs into matrix space and transfers the discrete logic inference into continuous linear algebra computation. In this paper, we extend the algebraic method to compute the normal logic program semantics, which makes it possible to differentiate the embeddings of logic programs using optimization methods and implement them with neural networks.

The proposed framework can be divided into two modules: a meta-info learner and an interpretation learner. Each module consists of two processes, an inference process and a backpropagation process. The first module, the meta-info learner, produces highly concentrated information about the logic program representation and delivers the meta-information to the second module. The meta-information consists of the number of different literals in the rules with the same head variable, and that information is connected with the maximal number of rules with the same head variable in the logic program. When the second module, the interpretation learner, receives the meta-information, it begins to search for comprehensive and interpretable logic rules in a reduced space.

3.1 Extended algebraic method to compute the logic program semantics

In this subsection, we provide the definitions needed to extend the algebraic method to compute the logic program semantics. To this end, we describe how to transfer the symbolic logic programs into matrices and the immediate vector consequence operator D_P corresponding to the logic program P.

First, we define a form of normal logic program called the same head variable (SHV) program. An SHV program P is a normal logic program such that all rules in P have the same variable in their heads. Therefore, a normal logic program consists of one or several SHV programs. Then, we stipulate that the body of each rule in a normal logic program is in the DNF. We construct the body of such a rule by connecting all the bodies in one SHV program using disjunction and setting the head of the rules with the same head across the SHV program to one head atom. Hence, any rules in the normal logic program have different head atoms.

Then, we extend the interpretation vector used to interpret the input to describe the input state of both the atoms and their negations in a normal logic program. We append the opposite Boolean value of each element in the original input interpretation vector $\mathbf{v} = (v_1, \dots, v_n, 1 - v_1, \dots, 1 - v_n)^{\mathrm{T}} \in \{0, 1\}^{2n}$. Then, we define the immediate vector consequence operator D_P as follows:

Definition 1 (Immediate vector consequence operator) Considering a normal logic program P has m rules and $B_P = \{p_1, \dots, p_n\}$. The immediate vector consequence operator D_P is an operator mapping in the vector space. Given an interpretation vector $\mathbf{v} = (v_1, \dots, v_n, 1 - v_1, \dots, 1 - v_n)^{\mathrm{T}} \in \{0,1\}^{2n}$, $D_P(\mathbf{v}) \in \{0,1\}^m$ is defined as follows:

1. $D_P(\mathbf{v})[k] = 1$, if $(p_k \leftarrow p_{j_1} \wedge \dots \wedge p_{j_s} \wedge \neg p_{j_{s+1}} \wedge \dots \wedge \neg p_{j_u})$ $\wedge \dots \wedge \neg p_{j_u}) \in P$ and $\mathbf{v}[j_1] = \dots = \mathbf{v}[j_s] = 1$ and $\mathbf{v}[j_{s+1}] = \dots = \mathbf{v}[j_u] = 0$,
2. $D_P(\mathbf{v})[k] = 0$, otherwise.

Obviously, $D_P(\mathbf{v})$ is the output interpretation vector corresponding to output interpretation $T_P(I)$, where $\mathbf{v}$ is the input interpretation vector corresponding to input interpretation I.

Next, we define two embedding methods, one representing normal logic programs and the other representing SHV programs. We call the embedding of an SHV program an SHV matrix, and we define it as follows.

Definition 2 (SHV matrix) For an SHV program P, which has l different rules and the head variable is p_h for all rules, each rule in P is indexed with a unique number i to locate it in the corresponding row in the matrix. Let $B_P = \{p_1, \dots, p_n\}$ be the Herbrand base of SHV program P. SHV program P is represented by a matrix $\mathbf{M}_P^{SHV} \in [0,1]^{l \times 2n}$, such that for each element $a_{ij} (1 \le i \le l, 1 \le j \le 2n)$,

1. if $p_h \leftarrow p_{j_1} \wedge \dots p_{j_s} \wedge \neg p_{j_{s+1}} \wedge \dots \wedge \neg p_{j_u}$ is in P, and the rule is indexed using number i, then

   (a) $a_{ij_k} = \frac{1}{u}\ (1 \le k \le s)$;
   (b) $a_{i(j_k+n)} = \frac{1}{u}\ (s+1 \le k \le u)$;

2. if $p_h \leftarrow$ is in P, and the rule is indexed using number i, then $a_{ih} = 1$;
3. $a_{ij} = 0$, otherwise.

Here, $\mathbf{M}_P^{SHV}$ is called the SHV matrix of P. Then, we define $row_i(\mathbf{M}_P^{SHV}) = \mathbf{M}_P[i, :]$ and $col_j(\mathbf{M}_P^{SHV}) = [:, j]$.

In SHV matrix $\mathbf{M}_P^{SHV}$, the i-th $(1 \le i \le l)$ row corresponds to the i-th rule in the SHV program. For $1 \le j \le n$, the j-th column corresponds to the literal p_j appearing in the body of a rule in the SHV program, and the $(n+j)$-th column corresponds to the literal $\neg p_j$ appearing in the body of a rule in an SHV program. When a rule in an SHV program contains u literals in its body, each literal is considered to have the truth value $\frac{1}{u}$. As a special case, each fact $p_h \leftarrow$ in P is represented as the tautology $p_h \leftarrow p_h$ in $\mathbf{M}_P^{SHV^u}$.

Example 1 Consider an SHV program P with three rules and $B_P = \{p, q, r\}$. Logic program P and its SHV matrix $\mathbf{M}_P^{SHV}$ are expressed as follows:

$$P : \begin{array}{l} p \leftarrow p \wedge q \wedge \neg r \\ p \leftarrow r \wedge \neg p \\ p \leftarrow \end{array}, \quad \mathbf{M}_P^{SHV} = \begin{array}{cccccc} \frac{1}{3} & \frac{1}{3} & 0 & 0 & 0 & \frac{1}{3} \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{array}$$

Before we introduce the embeddings for normal logic programs, we define the notion of a conjunction set as follows and present an example in Example 2.

Definition 3 (Conjunction set) Given a rule r in a normal logic program and $head(r) = p_i$, the *conjunction set* of a literal l in r is denoted as C_{l,p_i} and is defined as follows:

$$C_{l,p_i} = \{L_k | l \in L_k, 1 \le k \le n\},\ \text{if}\, p_i \leftarrow con(L_1) \vee con(L_2) \vee \cdots \vee con(L_n), \tag{2}$$

where con is a mapping from $\mathbb{L}$ to $\mathbb{F}$. The elements in the domain $\mathbb{L}$ are sets whose elements are literals, and the elements of codomain $\mathbb{F}$ are the conjunctions of those literals.

Example 2 Given literal sets $L_1 = \{p, q, \neg r\}$ and $L_2 = \{p, q\}$, we then have $con(L_1) = p \wedge q \wedge \neg r$ and $con(L_2) = p \wedge q$. Given a rule $p \leftarrow (p \wedge q \wedge \neg r) \vee (p \wedge q)$, we obtain $C_{\neg r,p} = \{\{p, q, \neg r\}\}$ and $C_{q,p} = \{\{p, q, \neg r\}, \{p, q\}\}$.

We now define the embedding for a normal logic program called a normal matrix.

Definition 4 (Normal matrix) For an arbitrary normal logic program P with m rules and $B_P = \{p_1, \dots, p_n\}$, P is represented by a matrix $\mathbf{M}_P^{NOR} \in [0,1]^{m \times 2n}$ such that for each element $a_{ij} (1 \le i \le m, 1 \le j \le 2n)$,

1. if $p_i \leftarrow (l_{j_1} \wedge \cdots \wedge l_{j_{u_1}}) \vee \cdots \vee (l_{j_1} \wedge \cdots \wedge l_{j_{u_l}})$ is in P, $1 \le i, j_k \le n$, and $1 \le k \le max(u_1, \dots, u_l)$, then
   (a) $a_{ij_k} = max(\frac{1}{|L_1|}, \dots, \frac{1}{|L_s|})$, where $L_1, \dots, L_s \in C_{p_{j_k}, p_i}$
   (b) $a_{i(j_k+n)} = max(\frac{1}{|L_1|}, \dots, \frac{1}{|L_s|})$, where $L_1, \dots, L_s \in C_{\neg p_{j_k}, p_i}$
2. if $p_i \leftarrow$ is in P, then $a_{ii} = 1$;
3. $a_{ij} = 0$, otherwise.

Here, $\mathbf{M}_P^{NOR}$ is called the normal matrix of P.

Given a normal matrix $\mathbf{M}_P^{NOR}$, the i-th $(1 \le i \le m)$ row corresponds to the Boolean variable p_i dominating the head of the DNF format rule. For $1 \le j \le n$, the j-th column corresponds to the literal p_j appearing in the body of the DNF format rule, and the $(n+j)$-th column corresponds to the literal $\neg p_j$ appearing in the body of the DNF format rule. Given a rule r in a normal logic program P with $head(r) = p_h$, each literal l_i in r is considered to have the truth value $max(\frac{1}{|L_1|}, \dots, \frac{1}{|L_s|})$, where $L_i \in C_{l_i, p_h}$. Specifically, each fact $p_i \leftarrow$ in P is represented as the tautology $p_i \leftarrow p_i$ in $\mathbf{M}_P^{NOR}$. We give an example of translating a normal logic program into its normal matrix as follows:

Example 3 Consider a normal logic program P with three rules and $B_P = \{p, q, r\}$. Logic program P and its normal matrix $\mathbf{M}_P^{NOR}$ are as follows:

$$P : \begin{array}{l} p \leftarrow (p \wedge q \wedge \neg r) \vee (p \wedge \neg q) \\ q \leftarrow p \wedge r \\ r \leftarrow \end{array}, \quad \mathbf{M}_P^{NOR} = \begin{array}{c} (p) \\ (q) \\ (r) \end{array} \begin{array}{cccccc} \frac{1}{2} & \frac{1}{3} & 0 & 0 & \frac{1}{2} & \frac{1}{3} \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{array}$$

We present some properties of the SHV matrices and normal matrices. First, we find that the number of non-zero entries in each row of the normal matrices is equal to the number of different literals in the rules. Second, all the values of the non-zero entries in one row of an SHV matrix are equal. Third, a single normal matrix can embed both normal logic programs and SHV programs. However, an SHV matrix only embeds one SHV program. We can represent normal logic programs through multiple two-dimensional SHV matrices. Finally, the SHV embedding method is a bijective mapping, which means that we can infer a unique symbolic SHV program according to the numbers, values, and positions of the non-zero entries from each row of an SHV matrix. However, we cannot infer a unique symbolic normal logic program from a normal matrix.

3.2 Framework details

In this subsection, we first briefly describe about the inference process and backpropagation process. Then, we provide a comprehensive description of the two modules, the meta-info learner and the interpretation learner. When defining each module, we describe the deductive reasoning in each module and how we convert this reasoning into inference processes, that is, differentiable processes corresponding to deductive reasoning, to learn the symbolic logic program through optimization methods. Finally, we describe learning logic programs from relational datasets in detail, the algorithm of D-LFIT, and the complexity of the D-LFIT algorithm.

The inference process implements the deductive reasoning in the D-LFIT framework. In each module, the trainable matrices for P, denoted as $\overline{\mathbf{M}}_P$, have different definitions. Therefore, we design distinct inference processes in the two modules. We explain the details of both inference processes in Sects. 3.2.1 and 3.2.2, respectively. In contrast, the backpropagation process is the same for both modules. It calculates the loss values and uses the stochastic gradient descent algorithm to adjust the parameters w in the trainable matrix $\overline{\mathbf{M}}_P$.

$$w = w - \alpha \cdot \nabla_w \text{loss}(w),$$

where α is the learning rate. In the general settings of the neural networks, the loss function is a binary cross-entropy function, defined as follows:

$$H(\mathbf{v}, \overline{\mathbf{v}}) = -\frac{1}{m}\left(\sum_{k=1}^{m} \mathbf{v}[k] \cdot \log(\overline{\mathbf{v}}[k]) + (1 - \mathbf{v}[k]) \cdot \log(1 - \overline{\mathbf{v}}[k])\right), \tag{3}$$

where m is the number of the different head atoms in the normal logic program, and $\mathbf{v}$ and $\overline{\mathbf{v}}$ are the labeled Boolean values of the head atoms and the predicted Boolean values of the head atoms, respectively.

3.2.1 Meta-info learner

A meta-info learner generates meta-information, which helps the interpretation learner reduce the search space and learn the symbolic logic program effectively. To define the deductive reasoning function in the meta-info learner, we use a threshold function, the proposed normal matrix, and the extended interpretation vector. The threshold function is defined as follows: for input element i, if $i \geq 1$, then we have $\theta(i) = 1$; otherwise, we have $\theta(i) = 0$. The deductive reasoning is then defined as follows:

$$\mathbf{v}_o = \theta(\mathbf{M}_P^{NOR}\mathbf{v}_i), \tag{4}$$

where $\mathbf{M}_P^{NOR} \in [0,1]^{m\times 2n}$ is a single normal matrix embedded in the logic program. Here, m is the number of rules and n is the number of atoms in Herbrand base B_P. Vectors $\mathbf{v}_i$ and $\mathbf{v}_o$ are the input and output interpretation vectors, respectively.

In particular, the deductive function plays the same role as the vector consequence operator D_P. To implement the inference process in the neural networks, we need to determine a differentiable function to replace the θ function in Eq. (4). The sigmoid function is a common activation function in neural networks. Not only is it a differentiable function, but the sigmoid function has a behavior at $x = 0$ that is similar to the behavior of the θ function at $x = 1$. In addition, to make our activation function similar to the θ function, we use the following sigmoid-like function in the meta-info learner:

$$\phi = \frac{1}{1 + e^{-\alpha x}},$$

where $\alpha \geq 1$. Parameter α controls the slope of the activation function. A larger α indicates that the activation function is closer to the θ function, but it tends to cause exploding gradients when x is close to zero, and vanishing gradients when x is far from zero. Hence, a larger α makes it harder to train the neural network. Similarly, we set a trainable normal matrix $\overline{\mathbf{M}}_P^{NOR} \in [0,1]^{m\times 2n}$ to learn the normal matrix of the normal logic program P. We define the inference process in Eq. (5) and the loss function in Eq. (6) in the first module as follows:

$$\overline{\mathbf{v}}_o = \phi(\overline{\mathbf{M}}_P^{NOR}\mathbf{v}_i - \mathbf{1}), \tag{5}$$

$$\text{loss} = \lambda_1 \cdot H(\mathbf{v}, \overline{\mathbf{v}}) + \lambda_2 \cdot \sum_{c=1}^{m}\sum_{b=1}^{2n} \overline{\mathbf{M}}_P^{NOR}[c,b]. \tag{6}$$

The optimized target in this module minimizes the distance between the label interpretation vector $\theta(\mathbf{M}_P^{NOR}\mathbf{v})$ and predicted vector $\phi(\overline{\mathbf{M}}_P^{NOR}\mathbf{v} - \mathbf{1})$. According to the ranges of trainable parameters in normal matrices, the inference process in this module, and the definition of the loss function, the values of the parameters in the trainable normal matrices searched by the neural networks may be larger than the corresponding parameters in the normal matrices. In addition, the values of the unimportant parameters in trainable normal matrices are trained to zeros. Hence, the number of non-zero parameters in each row of the trainable normal matrix $\overline{\mathbf{M}}_P^{NOR}$ is equal to the number of non-zero elements in each row of the normal matrix embedding for logic program P. As stated above, the number of non-zero elements in the k-th row of the normal matrix is equal to the number of different literals denoted as L_{p_k} in the rules whose head atom is p_k. Moreover, the generated normal matrix $\overline{\mathbf{M}}_P^{NOR}$ in this module includes the errors. We use min-max feature scaling (also known as data normalization) to normalize the elements in $\overline{\mathbf{M}}_P^{NOR}$ and set a threshold value to filter the valid elements. Then, we count the number of valid elements in $row_k(\overline{\mathbf{M}}_P^{NOR})$ to determine the number of literals L_{p_k} in the rules with head p_k. Although we cannot extract the precise symbolic logic program directly from the generated normal matrix $\overline{\mathbf{M}}_P^{NOR}$, the value of $L_{p_k} (1 \leq k \leq m)$ is a key knowledge and is regarded as the meta-information. We transfer this knowledge to the interpretation learner to extract the symbolic logic program precisely and effectively.

3.2.2 Interpretation learner

An interpretation learner uses the meta-information $L_{p_k} (1 \leq k \leq m)$ to learn how logic variables and their negation are connected by the operators of logical conjunction and disjunction. Before defining the deductive reasoning and inference process in the second module, given a head atom $p_k (1 \leq k \leq m)$, we show a proposition stating the relationship between the number of different literals L_{p_k} and the number of rules for an SHV program P with p_k as the head atom for all rules.

Proposition 1 *Given a set of literals B containing n elements, and an SHV program P whose body literals are chosen from B, the number of different rules in the SHV program P is less than or equal to the number of $\lfloor \frac{n}{2} \rfloor$-combinations from the set B, represented as $C(n, \lfloor \frac{n}{2} \rfloor)$.*

Proof For all rules in the SHV program, we also merge all the bodies of those rules into a single rule r and convert the body of r in the DNF. Each clause, connected by operator of logical disjunction in the DNF formula, is a conjunction of different literals. Hence, the maximum number of clauses equals the maximum number of different rules in the SHV program. Considering the relationship of logical entailment between different clauses, no two clauses have a logical entailment relationship if and only if all clauses have the same number of literals. Hence, we distribute different combinations of literals from B into each clause so that no two of clauses have a logical entailment relationship. In conclusion, the maximum possible number of clauses in rule r is the maximum number of combinations from the set B, represented as $C(n, \lfloor \frac{n}{2} \rfloor)$. Thus, $C(n, \lfloor \frac{n}{2} \rfloor)$ is the maximal number of different rules in the SHV program. □

To extract the symbolic logic program, we utilize multiple SHV matrices to embed the target logic program. We consider that a normal logic program with m rules and an SHV program P_k corresponding to the rule in the normal logic program with head atom p_k $(1 \leq k \leq m)$ has L_{p_k} different literals. Then, the SHV matrix for P_k is defined as $\mathbf{M}_{P_k}^{SHV} \in [0, 1]^{C(L_{p_k}, \lfloor \frac{1}{2} \times L_{p_k} \rfloor) \times 2n}$, where n is the number of ground atoms in the Herbrand base B_P, and $C(L_{p_k}, \lfloor \frac{1}{2} \times L_{p_k} \rfloor)$ is the maximal number of rules in P_k according to Proposition 1. We revise the definition of deductive reasoning in the second module as follows:

$$\mathbf{v}_o[k] = \bigvee_{c=1}^{C(L_{p_k}, \lfloor \frac{L_{p_k}}{2} \rfloor)} \theta\left(\sum_{b=1}^{2n} \mathbf{M}_{P_k}^{SHV}[c, b] \cdot \mathbf{v}_i[b] \right), \ 1 \leq k \leq m, \tag{7}$$

The successive Boolean value for Boolean variable p_k is a disjunction of the products of the input interpretation vector and rows in matrix $\mathbf{M}_{P_k}^{SHV}$. To implement the inference process in the interpretation learner, we utilize the same approach that transfers the θ function into the sigmoid-like function described in Sects. 3.2.1. In addition, fuzzy logic semantics must be applied to differentiably compute the operator of logical disjunction (*or*). There are several methods that represent the disjunction in fuzzy logic. In our setting, we need a fuzzy semantics *or* that satisfies the following properties:

- Commutativity: $or(x, y) = or(y, x)$
- Associativity: $or(or(x, y), z) = or(x, or(y, z))$

- Monotonicity: $x_1 \leq x_2$ implies $or(x_1, y) \leq or(x_2, y)$, and $x_1 \leq x_2$ implies $or(y, x_1) \leq or(y, x_2)$
- Identity element: $or(x, 0) = x$
- Zero element: $or(x, 1) = 1$

The semantics satisfying the above properties include the Gödel t-norm: $or(x, y) = max(x, y)$, Łukasiewicz t-norm: $or(x, y) = min(1, x + y)$, and product t-norm: $or(x, y) = 1 - (1 - x) \cdot (1 - y)$. In Gödel t-norm and Łukasiewicz t-norm semantics, it is easy to obtain a zero gradient (Evans and Grefenstette 2018). Hence, the loss of the gradient information in the Gödel t-norm and Łukasiewicz t-norms makes the backpropagation process unfeasible. We hence use the product t-norm as the semantics of the fuzzy disjunction in the interpretation learner as follows:

$$x_1 \vee x_2 \vee \cdots \vee x_n \Rightarrow 1 - (1 - x_1)(1 - x_2) \dots (1 - x_n).$$

Accordingly, we implement the deductive reasoning defined in Eq. (7) into the differentiable inference process and let the dimensions of the trainable matrix $\overline{\mathbf{M}}_{P_k}^{SHV}$ be consistent with those of $\mathbf{M}_{P_k}^{SHV}$ in Eq. (7). We define the inference process and loss function of the second module in Eqs. (8) and (9), respectively.

$$\bar{\mathbf{v}}_o[k] = 1 - \prod_{c=1}^{C\left(L_{p_k}, \left\lfloor \frac{L_{p_k}}{2} \right\rfloor\right)} \left(1 - \phi\left(\sum_{b=1}^{2n} \overline{\mathbf{M}}_{P_k}^{SHV}[c, b] \cdot \mathbf{v}_i[b] - 1\right)\right), \ 1 \leq k \leq m. \tag{8}$$

$$\text{loss} = \sum_{k=1}^{m} \left[\lambda_1 \cdot H(\mathbf{v}[k], \bar{\mathbf{v}}[k]) + \lambda_2 \cdot \sum_{c=1}^{C\left(L_{p_k}, \left\lfloor \frac{L_{p_k}}{2} \right\rfloor\right)} \left(1 - \sum_{b=1}^{2n} \overline{\mathbf{M}}_{P_k}^{SHV}[c, b]\right)\right]. \tag{9}$$

According to the ranges of parameters in the trainable SHV matrices as well as the definitions of the inference process and loss function, neural networks fit the sum of the parameters in each row of $\overline{\mathbf{M}}_{P_k}^{SHV}$ to one. Hence, the optimal solution searched for by neural networks corresponds to the SHV matrices of the logic program. The number of rows $C(L_{p_k}, \lfloor \frac{L_{p_k}}{2} \rfloor)$ in trainable matrix $\overline{\mathbf{M}}_{P_k}^{SHV}$ is the maximal number of possible rules in SHV matrix P_k. Hence, the trained rows may be duplicates if the real number of rules is less than the maximal number. After generating all of $\overline{\mathbf{M}}_{P_k}^{SHV}$, its elements are floating-points values ranging from zero to one, and we use the min-max feature scaling function, taking the elements in $\overline{\mathbf{M}}_{P_k}^{SHV}$ as inputs. Then, we set a small threshold to filter out the insignificant elements and extract the symbolic logic program according to the position information of those elements in the trained $\overline{\mathbf{M}}_{P_k}^{SHV}$ $(1 \leq k \leq m)$.

Because the mapping from logic programs to SHV matrices is bijective, we can add background knowledge to the inference process or implement curriculum learning in the interpretation learner. Hence, in the interpretation learner, we add a curriculum learning strategy: once we find some rules are correct, we embed them into the corresponding SHV matrices and fix them in the following episodes. The curriculum learning strategy helps the interpretation learner generate logic programs faster and more precisely.

3.2.3 Learning from relational data using interpretation transition

To use the LFIT setting to perform deductive and inductive inferences on the relational datasets, we propose a method for generating pairs of interpretation transitions from the relation datasets. In the LFIT setting, because the bottom clauses G can be regarded as a hypothesis describing the relational datasets, we set I as an input interpretation including the ground atoms from the body of $ground(G)$ and J as an output interpretation including the ground atoms from the head of $ground(G)$, where for I and J, $J = T_G(I)$ holds. In addition, the generated first-order features from the BCP and the bottom clauses can be regarded as propositional atoms and propositional logic programs, respectively. Hence, the Herbrand base for the bottom clauses includes all first-order features in the bottom clauses. Then, depending on the defined embeddings of the examples and their labels, the pairs of interpretation transitions can be generated as follows: for the input interpretation corresponding to the example e_i, we let the Boolean values of the first-order features appearing in the body of bottom clauses be the same as the values of corresponding bits in the embedding of the example e_i, and we set the Boolean values of the remaining first-order features in the head of the bottom clauses to zero. For the output interpretation corresponding to an example e_i, we let the Boolean values of the head first-order features in the bottom clauses be the same as the label of the example e_i, and we set the Boolean values of the rest of first-order features in the body of the bottom clauses to zero. Using the *motherInLaw* example in Sect. 2.3, according to the two bottom clauses generated from the family relationship example, two pairs of interpretation transitions are generated: $(I_1, J_1) = (\{mother(A, C), wife(C, B)\}, \{motherInLaw(A, B)\})$ and $(I_2, J_2) = (\{wife(A, C)\}, \{\})$.

In addition, we present an example to illustrate the architecture of D-LFIT in Fig. 1. In the example, D-LFIT learns a logic program with two head atoms and $B_p = \{p, q\}$. Through the inference and backpropagation processes, the meta-info learner generates the normal matrix representing the target symbolic logic program and the meta-information consists of $L_p = 2$ and $L_q = 1$. The interpretation learner receives the meta-information $L_{p,q}$ and sets the number of rows in the two SHV matrices to $C(2, 1) = 2$ and $C(1, 0) = 1$, respectively. Hence, the maximal number of rules with head variable p is equals 2, and the maximal number of rules with head variable q equals 1. Through the inference and propagation processes, the interpretation learner learns the

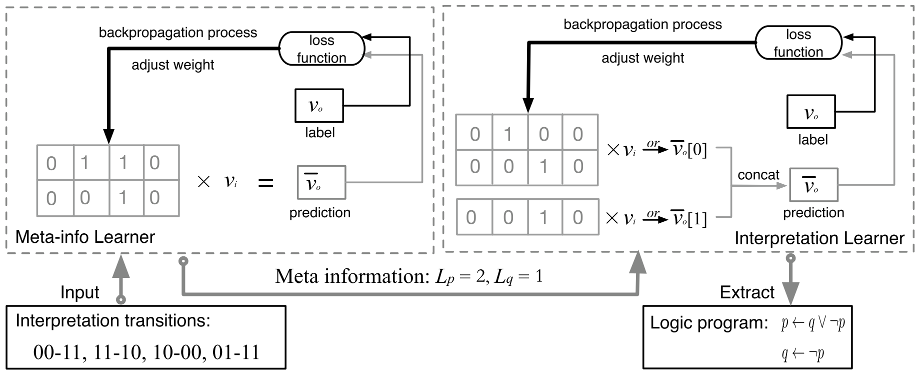

Fig. 1 An example depicting the architecture of D-LFIT

parameters in the two SHV matrices. Consequently, we extract the specific rules from these two SHV matrices as the generated normal logic program.

3.2.4 D-LFIT algorithm

In this section, we describe the algorithm of D-LFIT and analyze the space and time complexity of the algorithm. The whole process of D-LFIT is described in Algorithm 1.

Algorithm 1 The architecture of D-LFIT.

```
1: procedure D-LFIT($\mathbf{v}_i$, $\mathbf{v}_o$)    ▷ Set pairs of interpretation vectors as the input of D-LFIT
2:     Meta-info = META-INFO LEARNER($\mathbf{v}_i$, $\mathbf{v}_o$)
3:     $P$ = INTERPRETATION LEARNER ($\mathbf{v}_i$, $\mathbf{v}_o$, meta-info)
4:     return $P$    ▷ P is the generated symbolic normal logic program
1: procedure META-INFO LEARNER($\mathbf{v}_i$, $\mathbf{v}_o$)
2:     Set an empty normal matrix $\overline{\mathbf{M}}_P^{NOR} \in \mathbb{R}^{m\times 2n}$ and training times $N$
3:     while $epoch \le N$ do
4:         Backpropagation($\mathbf{v}_o$,Inference($\mathbf{v}_i$,$\overline{\mathbf{M}}_P^{NOR}$))
5:         $epoch = epoch + 1$
6:     $L_P$ = [], $k$ = 1    ▷ Use $L_P$ to store the meta-information for each rule in the normal logic program $P$
7:     while $k \le m$ do
8:         $L_P$.append(Count( $row_k(\overline{\mathbf{M}}_P^{NOR})$))    ▷ Count the valid number in each row of $\overline{\mathbf{M}}_P^{NOR}$ as the
   meta-information
9:         $k = k + 1$
10:    return $L_P$
1: procedure INTERPRETATION LEARNER($\mathbf{v}_i$, $\mathbf{v}_o$, $L_P$)
2:     Set $P$ as an empty set
3:     Set empty SHV matrices $\overline{\mathbf{M}}_{P_k}^{SHV} \in [0,1]^{C(L_{p_k}, \lfloor \frac{L_{p_k}}{2} \rfloor)\times 2n}$, $1 \le k \le m$ , and training times $N$
4:     while $epoch \le N$ do
5:         Backpropagation($\mathbf{v}_o$,Inference($\mathbf{v}_i$,$\overline{\mathbf{M}}_{P_k}^{SHV}$)), $1 \le k \le m$
6:         $epoch = epoch + 1$
7:         P += Extract($\overline{\mathbf{M}}_{P_k}^{SHV}$), $1 \le k \le m$    ▷ Extract the rule from each SHV matrix
8:         if rule $p_{k'}$'s accuracy is 1 then    ▷ Add the correct generated rules as BK dynamically
9:             Embed $p_{k'}$ to SHV matrix $\overline{\mathbf{M}}_{P_{k'}}^{SHV}$ and fix its parameter in the following training process
10:    $P$ += Extract($\overline{\mathbf{M}}_{P_k}^{SHV}$), $1 \le k \le m$
11:    return $P$
```

Then, we analyze the space complexity when the algorithm is running and the time complexity when extracting the symbolic programs from the trained SHV matrices. We assume that the maximal number of different literals among all rules of a normal logic program is L_{max}. According to the dimensions of normal matrices and SHV matrices, the space complexity is $O(n \times m \times C(L_{max}, \lfloor \frac{L_{max}}{2} \rfloor))$. Remarkably, L_{max} is much less than n in common Boolean networks; hence, D-LFIT can handle most Boolean network datasets. Considering the time complexity, given the logic program embeddings and the algebraic method to compute the logic program semantics, we translate the symbolic logic programs by checking the valid values in the trained SHV matrices. Hence, the time complexity when extracting the logic programs is $O(n)$, where n is the number of entries in the trained SHV matrices.

4 Experimental results

To validate the accuracy and efficiency of our model, several Boolean networks of different sizes, the networks for control of flower morphogenesis in *Arabidopsis thaliana* (Chaos et al. 2006), budding yeast cell cycle regulation (Li et al. 2004), fission yeast cell cycle regulation (Davidich and Bornholdt 2008), and mammalian cell cycle regulation (Fauré et al. 2006), were used as benchmarks in this study. In addition, to check the learning ability of D-LFIT in relational datasets, we tested it on the relational datasets including Mutagenesis (Srinivasan et al. 1994), UW-CSE (Davis et al. 2005), and Alzheimers-amine (King et al. 1995). We used BCP to transfer positive examples and negative examples into their bottom clauses. Then, we embedded the bottom clauses into the corresponding vector and label the vector according to the class of the example. According to these bottom clause vectors and their labels, we then used D-LFIT to learn clause embedding and extract a simpler first-order logic program.

In this study, the complexity of a Boolean network dataset refers to the number of nodes in the Boolean network and the number of total data instances in the dataset. The complexity of a relational dataset is the number of first-order features generated by the BCP algorithm and the number of bottom clauses generated by the BCP algorithm. The complexities for all benchmarks are listed in Table 1. In addition, the learning abilities of D-LFIT on the corresponding incomplete data and mislabeled data were tested in this study. The learning abilities include deductive reasoning ability, as reflected by the mean squared error (MSE) and inductive reasoning ability, as reflected by the accuracy of the rules generated from D-LFIT. The MSE is calculated between the predicted output interpretations from the meta-info learner and the label output interpretations. The accuracy is calculated according to the logic programs generated from interpretation learner. For the accuracy of the logic programs describing the relational datasets, because the Boolean values of the predicates in the body of the bottom clauses are always zero in the output interpretations, only one significant rule whose head predicate is the same as the head first-order feature in the bottom clauses is considered. The accuracy of the logic programs describing the Boolean network datasets is calculated by averaging the accuracy of each rule. The accuracy of a single rule is defined as follows:

$$\text{accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{10}$$

where TP, TN, FP, and FN are the true positives, true negatives, false positives, and false negatives, respectively, in the confusion matrix of the rule.

For the Boolean networks datasets, we compared the MSE with NN-LFIT, which is based on the neural networks proposed by Gentet et al. (2016) and Tourret et al. (2017). We compared the accuracy of the logic rules with LF1T, which is based on the purely symbolic ILP method proposed by Inoue et al. (2014), the WEKA (Witten et al. 2016) JRip implementation of RIPPER (Cohen 1995), which is a fast and accurate purely symbolic rule learner algorithm, and NN-LFIT. For relational datasets, we compared the accuracy of the proposed method with that of JRip and CILP++, proposed by França et al. (2015). CILP++ combines a three-layer feedforward neural network and the BCP algorithm to perform deduction tasks on relational datasets. All experiments described in this section were performed on a 6-core Intel Xeon Gold 6142.

4.1 Handing incomplete data

In the real world, data are often incomplete, especially in biology, so we tested D-LFIT on multiple split datasets of the Boolean network datasets and relational datasets. To obtain the split datasets, we used different split rates from 10 to 100%. After obtaining the split datasets, we computed the values of MSE and accuracy for two repeats of 5-fold cross-validation experimental setting. Figure 2a shows the accuracy and MSE on the split fission dataset. These results lead to two conclusions: First, when the training data consist of at least 40% of the total data, the MSE floats around the optimal value, and the error becomes negligible. Second, the accuracy of the logic program becomes stable when the split rates of the dataset are greater than 45%.

We compared the MSE and accuracy of the proposed method with those of the other baselines. We particularly care about the performance on the small sizes of datasets; hence, the experimental datasets include datasets that are 10% and 20% of the whole data. We also conducted experiments on half of datasets and whole datasets. The results of NN-LFIT, LF1T, JRip, and the proposed model are compared in Table 1. The accuracy of the logic programs on the relation datasets and the results of CILP++ are compared in Table 2. In the paper, we used run out of time (ROT) to stand that the baselines exceeded the maximal running time, and omitted the values of MSE on the purely symbolic algorithms including LFIT and JRip. In all tables of the paper, the results in bold are the best ones. According to Table 1, our proposed model obtains a slightly lower accuracy and higher MSE than NN-LFIT on the fission and mammalian datasets. However, our model generated more accurate logic programs than LF1T and JRip on most split datasets. Because the rule extraction process in NN-LFIT and LF1T is computationally expensive, the running time when extracting rules from the budding and *Arabidopsis* datasets exceeded the limit (5 hours). However, our model runs fast because it directly maps the symbolic logic programs from the trained SHV matrix. According to Table 2, D-LFIT also obtains results comparable to those of CILP++.

In addition, we tested the maximal number of nodes in a Boolean network that our model can execute under the same experimental settings. We manually generated a Boolean network with n nodes; then, we used all data instances in the datasets to train D-LFIT. We found that D-LFIT can handle a Boolean network with 20 nodes and

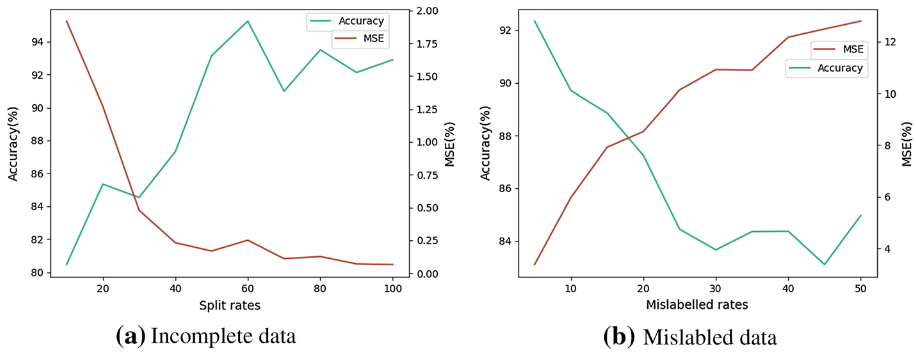

Fig. 2 Mean accuracy of the logic program and the MSE of the predicted Boolean value with respect to different split rates and mislabeling rates of the fission dataset

Table 1 Comparison of MSE (%) and accuracy (%) on partial datasets with different split rates

Datasets (Complexity)	Model	The split rates			
		10%	20%	50%	100%
Fission (10, 2^{10})	NN-LFIT	**1.2**, **98.80**	**0.2**, **99.80**	**0.00**,**99.92**	**0.00**,99.87
	D-LFIT	1.91, 80.45	1.27, 85.33	0.17, 93.13	0.06, 92.89
	LF1T	–, 76.85	–,77.03	–,77.15	– ,**100**
	JRip	–,79.14	–,78.05	–,80.04	–, 78.47
Mammalian (10, 2^{10})	NN-LFIT	**1.3**, **96.6**	**0.4** , **94.35**	**0.00**,**99.89**	**0.00**,**99.91**
	D-LFIT	1.73, 71.67	1.21 ,75.9	0.79, 80.09	0.52,82.84
	LF1T	–,76.01	–,76.48	–,76.73	91.56
	JRip	– ,77.84	–,75.44	–,76.41	–, 74.66
Budding (12, 2^{12})	NN-LFIT	ROT	ROT	ROT	ROT
	D-LFIT	1.03, **71.96**	0.43,**71.39**	0.15,**70.5**	0.09,**76.52**
	LF1T	ROT	ROT	ROT	ROT
	JRip	–,67.97	–,68.55	–, 67.91	–,68.32
Arabidopsis (15, 2^{15})	NN-LFIT	ROT	ROT	ROT	ROT
	D-LFIT	0.57, **84.35**	0.51, **86.83**	0.48, **88.56**	0.45, **89.70**
	LF1T	ROT	ROT	ROT	ROT
	JRip	–, 68.84	–, 69.00	–, 68.79	–,68.67
Mutagenesis (1111, 188)	D-LFIT	**77.78**	**94.44**	**88.88**	**83.33**
	JRip	58.37	59.46	65.97	66.45
UW-CSE (601, 1614)	D-LFIT	**75.00**	**78.12**	**78.44**	**79.44**
	JRip	70.18	70.81	73.85	74.35
Alzheimers-amine (1084, 686)	D-LFIT	58.42	**60.29**	**63.24**	**67.65**
	JRip	**58.74**	57.66	57.14	54.81

Table 2 Comparison of the accuracy (%) of rules obtained by CILP++

Model	Datasets		
	Mutagenesis	UW_CSE	Alzheimer-amine
CILP++	77.72	**81.98**	**78.70**
D-LFIT	**83.33**	79.44	67.75

2^{20} training instances in the corresponding dataset under the stipulated experimental environment.

4.2 Handing mislabeled data

In this section, we present the results obtained by D-LFIT on mislabeled training data. For the Boolean network datasets, the mislabeled training data were generated by changing the Boolean value of the input interpretations at random positions. For the relational datasets, we stochastically changed the labels of the examples. Then, we compare the MSE and accuracy with those obtained on the original clean test datasets.

In this experiment, we set the mislabeling rates from 5% to 50%. After obtaining the mislabeled datasets, we computed the MSE and accuracy values under two repeats of 5-fold cross-validation experiments. Figure 2b shows the accuracy and MSE achieved by the model in the fission dataset at different mislabeling rates. From Fig. 2b, we conclude that the accuracy decreases as the mislabeling rate increases and stabilizes when the mislabeling rates are larger than 30%. The results of the performance of the proposed method on mislabeled data with respect to those of other baselines are shown in Table 3. The results reveal the following: First, the proposed method and NN-LFIT obtain comparable results on the fission dataset. Second, D-LFIT performs better than other baselines on all datasets except for the Arabidopsis dataset with 50% mislabeled data.

5 Related work

The field of ILP has a long history and many sub-fields. LFIT, as a growing and important sub-field of ILP, has received increasing attention. LF1T, proposed by Inoue et al. (2014), was the first model to learn the logic programs representing the Boolean networks from traces of interpretation transitions. LF1T uses a purely logic framework to learn the

Table 3 Comparison of the MSE (%) and accuracy (%) on the mislabeled data with different mislabeling rates

Datasets	Model name	The rates of mislabeled data			
		5%	20%	35%	50%
Fission	NN-LFIT	3.25, **96.75**	10.56,**89.43**	15.14,**84.86**	17.74,82.25
	D-LFIT	**2.23**, 92.34	**8.53**, 87.25	**10.89**, 84.34	**12.08**, **84.96**
	LF1T	–,77.25	–,77.16	–,77.38	–,77.32
	JRip	– ,78.91	– ,78.54	–,78.55	– ,78.15
Mammalian	NN-LFIT	4.77, 79.72	16,78.11	20.98,79.01	23.20,74.49
	D-LFIT	**3.45**, **80.00**	**11.53**,**78.82**	**15.74**, **80.29**	**16.35**, **86.27**
	LF1T	–, 76.72	–,76,73	–,76.62	–,77.32
	JRip	–,74.21	–,74.45	– , 74.43	–,74.00
Budding	NN-LFIT	ROT	ROT	ROT	ROT
	D-LFIT	4.9, **76.42**	13.3, **74.28**	16.53,**73.41**	18.21,**74.71**
	LF1T	ROT	ROT	ROT	ROT
	JRip	– ,67.99	–,67.15	–,66.80	–,66.41
Arabidopsis	NN-LFIT	ROT	ROT	ROT	ROT
	D-LFIT	4.8 ,**81.46**	11.83,**76.59**	15.4 ,**70.28**	17.35,64.90
	LF1T	ROT	ROT	ROT	ROT
	JRip	–,68.27	–, 67.34	–,66.94	–,**66.65**
Mutagenesis	D-LFIT	**88.89**	**83.33**	**88.89**	**88.89**
	JRip	66.49	62.23	62.77	62.23
UW-CSE	D-LFIT	**73.49**	**72.67**	**73.29**	**73.29**
	JRip	72.80	67.35	66.04	66.23
Alzheimers-amine	D-LFIT	**63.24**	**63.24**	**60.29**	**58.83**
	JRip	48.35	49.42	49.27	50.87

propositional logic programs. Other well-known purely symbolic methods such as C4.5, proposed by Quinlan (1993) and Ripper, proposed by Cohen (1995), which is much faster than C4.5 and often provides more accurate logic rules, can also perform LFIT tasks. The shortcomings of these models are that they cannot handle incomplete and mislabeled data precisely. NN-LFIT, proposed by Gentet et al. (2016) and Tourret et al. (2017), uses neural networks to learn features from data. The NN-LFIT includes the initialization, constructive, and pruning processes. The initialization process constructs feedforward networks. The constructive process adds hidden neurons to the feedforward networks to decrease the errors. The pruning process removes the useless weights to decrease complexity when extracting the final symbolic logic rules. However, the last process is slow to run, especially on large Boolean networks. Phua et al. (2019) proposed a model using recurrent neural networks to predict unseen interpretations from the traces of interpretation transitions. Their model is able to learn features from background knowledge. However, this model is not interpretable because of the complexity of recurrent neural networks. δ-LFIT proposed by Phua and Inoue (2019) regards ILP as a classification problem. δ-LFIT embeds all possible logic programs into vectors and uses long short term memory (Hochreiter and Schmidhuber 1997) to learn the function mapping from a series of state transitions to the corresponding logic program embeddings. The shortcoming of their model is that the framework cannot learn logic programs with more than five Boolean variables under the same memory and runtime constraints as in our experimental environment. By contrast, D-LFIT has fewer parameters according to its definition; hence, D-LFIT is able to learn common logic programs with 20 nodes or less in our experimental setting.

For other systems combining neural networks and symbolic logic programs, there are some works that proves the feasibility to approximate the T_P operator through neural networks given a logic program (Hölldobler and Kalinke 1994; Hölldobler 1993; Hölldobler et al. 1999; Hitzler and Seda 2000; Hitzler et al. 2004; Seda and Lane 2004; Bader et al. 2005; Seda 2006). But the interpretability of the neural networks in these works is missing. Our model not only constructs a neural network to approximate the T_P operator from the dataset, but also extracts the symbolic rules from the trained neural network. The surveys presented by Bader et al. (2004) and Bader et al. (2005) state some important problems in the neuro-symbolic field, which include the representation and extraction of knowledge. Our model solves those two problems by designing an embedding method that connects neural networks to logic programs. Garcez and Zaverucha (2004) proposed a mapping algorithm that converts the arbitrary first-order symbolic logic rules into neural networks. CILP++ proposed by França et al. (2014) learns from relational data and inferences like the first-order logic programs with neural networks using the BCP algorithm. Subsequently, França et al. (2015) proposed an algorithm to extract symbolic rules from CILP++. The most significant difference between D-LFIT and CILP++ is the architecture of the neural network. We designed a logic program embedding method and revised a novel loss function of the neural networks to learn logic programs based on the embedding method. After learning the optimal embedding of a logic program through a neural network, the symbolic rules can be interpreted directly from the trained matrices in $O(n)$ time complexity in D-LFIT. By contrast, CILP++ uses standard three-layer neural networks to perform deduction. The extraction method from CILP++ uses a learned neural network as an oracle and through a set of examples, possibly distinct from the example set used for training the neural network, to recursively build a decision tree based on an information gain-based heuristic. NTPs proposed by Rocktäschel and Riedel (2016) performs first-order deductive inference task. However, D-LFIT focuses on performing inductive inference using neural networks. LRNN proposed by Šourek et al. (2018) uses first-order logic programs as templates to find underlying relational structures. The connection between LRNN and D-LFIT is that the logic programs generated by D-LFIT can be the input to LRNN. RelNN proposed by Kazemi and Poole (2018) learns the relations between predicates and formulas with specific formats. However, D-LFIT, as an inductive logic programming system, can generate relations between predicates and arbitrary DNFs consisting of literals. Wang and Cohen (2016) also proposed embeddings for first-order logic programs and searched for them through optimization methods. However, they did not give constraints on the embeddings of first-order logic programs. Hence, it is hard to extract the corresponding symbolic rules from the embeddings. Yang et al. (2017) made use of the connection between the matrix multiplication and logic inference. They built recurrent neural networks and attention vectors to perform the matrix computation. Evans and Grefenstette (2018) proposed ∂ ILP, which uses program templates to generate the set of candidate clauses and continuous semantics proposed by Serafini and Garcez (2016) to calculate the Boolean value for each fact. ∂ILP computes the distance between the predicted values and labels to differentiably optimize the weight of each candidate clause according to the loss value. Garcez et al. (2001) and Lehmann et al. (2008) proposed algorithms to extract the first-order symbolic logic rules from a pre-trained feedforward neural network. Evans et al. (2021) proposed the apperception engine, a purely symbolic algorithm to learn first-order logic rules to explain a sequence of sensory data in the first-order format. In contrast to their work, our model translates symbolic logic programs into embeddings according to the continuous semantics of the logic program. Hence, we can embed background knowledge into matrices. In addition, our model does not need logic program templates in advance, and we can interpret logic programs in linear time complexity from trained SHV matrices. Hence, curriculum learning can be applied: once the correct logic rules are obtained, we embed and fix them in the matrices in the following training process. This strategy reduces the search space for the following learning process.

6 Conclusion and future work

In this paper, we proposed D-LFIT, a framework that translates logic programs into embeddings, infers logical values through differentiable semantics of the logic programs, and searching for embeddings through an optimization algorithm. After learning the optimal embeddings of the logic programs, we can interpret them to obtain the corresponding symbolic logic programs with linear time complexity. Experimental results showed that the proposed model is robust, precise, and fast in the LFIT field. More specifically, because of the optimization methods, D-LFIT performs well on incomplete and mislabeled data. In addition, we set precise embeddings for logic programs and rigorous semantics to describe the deductive inference process. Hence, we can embed background knowledge into neural networks, extract symbolic logic programs from the trained matrix directly, and implement curriculum learning. In addition, we embedded rules with the same head into a two-dimensional matrix; hence, the model has a few parameters, and the learning processes are computationally cheap. In future work, to improve the accuracy of the first-order logic program, we will build more connections, such as sharing the same values between the parameters of the trainable matrix, to ensure that the first-order logic program satisfies both spatial unity and conceptual unity (Evans et al. 2021). We will also adapt our neural-symbolic model to handle other task settings, such as knowledge base completion.

Author Contributions Conceptualization: Kun Gao, Katsumi Inoue; Methodology: Kun Gao; Formal analysis and investigation: Kun Gao; Writing - original draft preparation: Kun Gao; Writing - review and editing: Katsumi Inoue, Kun Gao, Yongzhi Cao; Funding acquisition: Hanpin Wang, Yongzhi Cao, Katsumi Inoue; Resources: Hanpin Wang, Yongzhi Cao, Katsumi Inoue; Supervision: Hanpin Wang, Yongzhi Cao, Katsumi Inoue.

Funding This work was supported by the National Key R&D Program of China under Grants 2018YFC1314200 and 2018YFB1003904, and the National Natural Science Foundation of China under Grants 61772035, 61972005, and 61932001; this work was also supported in part by the NII international internship program and JSPS KAKENHI Grant Number JP17H00763.

Data availability https://drive.google.com/drive/folders/1SKB05Jka5jaJjfVi8LZmeo2smf0ujMeM?usp=sharing

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

Code availability https://drive.google.com/drive/folders/1SKB05Jka5jaJjfVi8LZmeo2smf0ujMeM?usp=sharing (Need to download the LF1T by Python)

References

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). San Mateo: Morgan Kaufmann.

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

Avila, A. S., Broda, K., & Gabbay, D. M. (2001). Symbolic knowledge extraction from trained neural networks: A sound approach. *Artificial Intelligence, 125*(1–2), 155–207.

Bader, S., Hitzler, P., & Hölldobler, S. (2004). The integration of connectionism and first-order knowledge representation and reasoning as a challenge for artificial intelligence. In *Proceedings of the third international conference on information* (pp. 22–33).

Bader, S., Hitzler, P., & Witzel, A. (2005). Integrating first-order logic programs and connectionist systems—a constructive approach. In *Proceedings of the IJCAI workshop on neural-symbolic learning and reasoning* (Vol. 5).

Bengio, Y., Louradour, J., Collobert, R., & Weston, J. (2009). Curriculum learning. In *Proceedings of ICML* (Vol, 382, pp. 41–48). New York: ACM Press.

Chaos, A., Aldana, M., Espinosa-Soto, C., Ponce de León, B., Arroyo, A. G., & Alvarez-Buylla, E. R. (2006). From genes to flower patterns and evolution: Dynamic models of gene regulatory networks. *Journal of Plant Growth Regulation, 25*(4), 278–289.

Cohen, W. W. (1995). Fast effective rule induction. In *Proceedings of ICML* (pp. 115–123). Elsevier.

Davidich, M. I., & Bornholdt, S. (2008). Boolean network model predicts cell cycle sequence of fission yeast. *PLoS ONE, 3*(2), e1672.

Davis, J., Burnside, E. S., Dutra, I. C., Page, D., & Costa, V. S. (2005). An integrated approach to learning Bayesian networks of rules. In *LNAI: Vol. 3720. Proc. ECML* (pp. 84–95). Berlin: Springer.

Evans, R., & Grefenstette, E. (2018). Learning explanatory rules from noisy data. *Journal of Artificial Intelligence Research, 61,* 1–64.

Evans, R., Hernández-Orallo, J., Welbl, J., Kohli, P., & Sergot, M. (2019). Making sense of sensory input. *Artificial Intelligence, 293,* 103438.

Fauré, A., Naldi, A., Chaouiya, C., & Thieffry, D. (2006). Dynamical analysis of a generic Boolean model for the control of the mammalian cell cycle. *Bioinformatics, 22*(14), e124–e131.

França, M. V. M., D'Avila Garcez, A. S., & Zaverucha, G. (2015). Relational knowledge extraction from neural networks. In *CEUR workshop proceedings* (Vol. 1583, pp. 11–12).

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

Gentet, E., Tourret, S., & Inoue, K. (2017). Learning from interpretation transition using feed-forward neural networks. In *CEUR workshop proceedings* (pp. 27–33).

Hitzler, P., & Seda, A. K. (2000). A note on the relationships between logic programs and neural networks. In *Proceedings of the 4th irish workshop on formal methods* (pp. 1–9).

Hitzler, P., Hölldobler, S., & Seda, A. K. (2004). Logic programs and connectionist networks. *Journal of Applied Logic, 2*(3), 273–300.

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

Hölldobler, S. (1993). *Automated inferencing and connectionist models*. Fakultät Informatik. Technische Hochschule Darmstadt. (Doctoral dissertation, Habilitationsschrift).

Hölldobler, S., Kalinke, Y., Hoelldobler, S., & Kalinke, Y. (1991). Towards a new massively parallel computational model for logic programming. In *ECAI'94 workshop on combining symbolic and connectioninst processing* (pp. 68–77).

Hölldobler, S., Kalinke, Y., & Störr, H. P. (1999). Approximating the semantics of logic programs by recurrent neural networks. *Applied Intelligence, 11*(1), 45–58.

Inoue, K. (2011). Logic programming for Boolean networks. In *Proceedings of IJCAI* (pp. 924–930). Menlo Park: AAAI Press.

Inoue, K., & Sakama, C. (2012). Oscillating behavior of logic programs. Correct reasoning-essays on logic-based AI in honour of Vladimir LifschitzIn E. Erdem, J. Lee, Y. Lierler, & D. Pearce (Eds.), *LNAI* (Vol. 7265, pp. 345–362). Berlin: Springer.

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

Kauffman, S. A. (1993). *The origins of order: Self-organization and selection in evolution*. Oxford: Oxford University Press.

Kazemi, S. M., & Poole, D. (2018). RelNN: a deep neural model for relational learning. In *Proceedings of AAAI* (pp. 6367–6375). AAAI press.

King, R. D., Srinivasan, A., & Sternberg, M. J. E. (1995). Relating chemical activity to structure: An examination of ILP successes. *New Generation Computing, 13*(3–4), 411–433.

Kramer, S., Lavrač, N., & Flach, P. (2001). Propositionalization approaches to relational data mining. *Relational Data Mining,* 262–291.

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

Li, F., Long, T., Lu, Y., Ouyang, Q., & Tang, C. (2004). The yeast cell-cycle network is robustly designed. *Proceedings of the National Academy of Sciences of the United States of America, 101*(14), 4781–4786.

Muggleton, S. (1991). Inductive logic programming. *New Generation Computing, 8*(4), 295–318.

Muggleton, S. (1995). Inverse entailment and Progol. *New Generation Computing, 13*(3–4), 245–286.

Muggleton, S., & De Raedt, L. (1994). Inductive logic programming: Theory and methods. *The Journal of Logic Programming, 19*(1), 629–679.

Nguyen, H. D., Sakama, C., Sato, T., & Inoue, K. (2018). Computing logic programming semantics in linear algebra. *International conference on multi-disciplinary trends in artificial intelligence* (pp. 32–48). Cham: Springer.

Phua, Y. J., & Inoue, K. (2019). Learning logic programs from noisy state transition data. *ILP* (pp. 72–80). Cham: Springer.

Phua, Y. J., Ribeiro, T., & Inoue, K. (2019). Learning representation of relational dynamics with delays and refining with prior knowledge. *If CoLoG Journal of Logics and their Applications, 6*(4), 695–708.

Quinlan, J. R. (1993). *C4.5: programs for machine learning*. San Francisco: Morgan Kaufmann.

Rocktäschel, T., & Riedel, S. (2016). Learning knowledge base inference with neural theorem provers. In *Proceedings of the 5th workshop on automated knowledge base construction* (pp. 45–50).

Sakama, C., Nguyen, H. D., Sato, T., & Inoue, K. (2018). Partial evaluation of logic programs in vector spaces. In *11th workshop on answer set programming and other computing paradigms*. Oxford, UK.

Seda, A. K., & Lane, M. (2004). On approximation in the integration of connectionist and logic-based systems. In *Proceedings of the third international conference on information* (pp. 297–300).

Seda, A. K. (2006). On the integration of connectionist and logic-based systems. *Electronic Notes in Theoretical Computer Science, 161*(1), 109–130.

Serafini, L., & Garcez, A. D. A. (2016). Logic tensor networks: deep learning and logical reasoning from data and knowledge. In *CEUR workshop proceedings* (Vol. 1768).

Šourek, G., Aschenbrenner, V., Železný, F., Schockaert, S., & Kuželka, O. (2018). Lifted relational neural networks: Efficient learning of latent relational structures. *Journal of Artificial Intelligence Research, 62,* 69–100.

Srinivasan, A., Muggleton, S., King, R. D., & Sternberg, M. J. E. (1994). Mutagenesis: ILP experiments in a non-determinate biological domain. In *LNAI: Vol. 237. Proc. ILP* (pp. 217–232). Berlin: Springer.

Tamaddoni-Nezhad, A., & Muggleton, S. (2009). The lattice structure and refinement operators for the hypothesis space bounded by a bottom clause. *Machine Learning, 76*(1), 37–72.

Tourret, S., Gentet, E., & Inoue, K. (2017). Learning human-understandable description oaf dynamical systems from feed-forward neural networks. *International symposium on neural networks* (pp. 483–492). Cham: Springer.

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

Wang, W. Y., & Cohen, W. W. (2016). Learning first-order logic embeddings via matrix factorization. In *Proceedings of IJCAI* (pp. 2132–2138).

Witten, I. H., Frank, E., Hall, M. A., & Pal, C. J. (2017). *Data mining: practical machine learning tools and techniques* (Fourth ed.). Morgan Kaufmann, ian imorint of Elsevier.

Yang, F., Yang, Z., & Cohen, W. W. (2017). Differentiable learning of logical rules for knowledge base reasoning. In *Proceedings of NIPS* (pp. 2320–2329).

Authors and Affiliations

Kun Gao[1] · Hanpin Wang[1,2] · Yongzhi Cao[1] · Katsumi Inoue[3]

Kun Gao
kungao@pku.edu.cn

Yongzhi Cao
caoyz@pku.edu.cn

Katsumi Inoue
inoue@nii.ac.jp

[1] Peking University, Beijing, China

[2] Guangzhou University, Guangzhou, China

[3] National Institute of Informatics, Tokyo, Japan