Artificial Intelligence 331 (2024) 104108

Contents lists available at ScienceDirect

Artificial Intelligence

journal homepage: www.elsevier.com/locate/artint

# A differentiable first-order rule learner for inductive logic programming[☆]

Kun Gao[a,b], Katsumi Inoue[c], Yongzhi Cao[a], Hanpin Wang[d,a,*]

[a] Key Laboratory of High Confidence Software Technologies (MOE), School of Computer Science, Peking University, Beijing 100871, China
[b] Institute of High Performance Computing (IHPC), Agency for Science, Technology and Research (A*STAR), Singapore 138632, Singapore
[c] National Institute of Informatics, Tokyo 101-8430, Japan
[d] School of Computer Science and Cyber Engineering, Guangzhou University, Guangzhou 510006, China

## ARTICLE INFO

Dataset link: classic ILP datasets, Countries dataset, Unified Medical Language System (UMLS), Nations, Kinship, UW-CSE, Alzheimers-amine, WN18, WN18RR, and FB15KSelected. (Reference Data)

*Keywords:*
Machine learning
Differentiable inductive logic programming
Interpretability
Neuro-symbolic method

## ABSTRACT

Learning first-order logic programs from relational facts yields intuitive insights into the data. Inductive logic programming (ILP) models are effective in learning first-order logic programs from observed relational data. Symbolic ILP models support rule learning in a data-efficient manner. However, symbolic ILP models are not robust to learn from noisy data. Neuro-symbolic ILP models utilize neural networks to learn logic programs in a differentiable manner which improves the robustness of ILP models. However, most neuro-symbolic methods need a strong language bias to learn logic programs, which reduces the usability and flexibility of ILP models and limits the logic program formats. In addition, most neuro-symbolic ILP methods cannot learn logic programs effectively from both small-size datasets and large-size datasets such as knowledge graphs. In the paper, we introduce a novel differentiable ILP model called differentiable first-order rule learner (DFORL), which is scalable to learn rules from both smaller and larger datasets. Besides, DFORL only needs the number of variables in the learned logic programs as input. Hence, DFORL is easy to use and does not need a strong language bias. We demonstrate that DFORL can perform well on several standard ILP datasets, knowledge graphs, and probabilistic relation facts and outperform several well-known differentiable ILP models. Experimental results indicate that DFORL is a precise, robust, scalable, and computationally cheap differentiable ILP model.

## 1. Introduction

Nowadays, knowledge discovery is an important technique for people to acquire knowledge from large or complex realistic datasets. Relational data mining techniques construct human-readable representations from relational datasets. An explicit logic program, i.e., a set of Horn clauses, can be a clear explanation for complex and noisy relational facts in several domains such as business, biology, and medicine. Inductive logic programming (ILP) combines inductive learning and logic programming techniques to generate logic programs from examples [38]. Given a set of positive examples and a set of negative examples, an ILP system learns a logic program that entails all the positive examples but does not entail any of the negative examples [14]. ILP techniques

[☆] A preliminary version of this article appeared in the Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence (IJCAI-22), July 23-29, 2022, at Messe Wien, Vienna, Austria.
[*] Corresponding author at: Key Laboratory of High Confidence Software Technologies (MOE), School of Computer Science, Peking University, Beijing 100871, China.

*E-mail addresses:* kungao@pku.edu.cn (K. Gao), inoue@nii.ac.jp (K. Inoue), caoyz@pku.edu.cn (Y. Cao), whpxhy@pku.edu.cn (H. Wang).

https://doi.org/10.1016/j.artint.2024.104108
Received 6 June 2022; Received in revised form 23 February 2024; Accepted 25 February 2024

Available online 15 March 2024

typically include purely symbolic methods [39] and neuro-symbolic methods [12,14,61]. Purely symbolic ILP methods support lifelong learning [6] and have more interpretability [7]. However, purely symbolic approaches cannot deal with noisy data in realistic domains [14]. Neuro-symbolic ILP methods adopt neural networks, gradient-based methods, and basic concepts of ILP methods to learn logic programs in a differentiable manner [6]. By employing neural networks, neuro-symbolic ILP models can generate logic programs from noisy relational facts.

However, there are two limitations of the current neuro-symbolic ILP models: Firstly, the neuro-symbolic ILP models need strong language biases [6] such as logic templates [14,45] as input. A language bias constrains the syntax of learned logic programs, which includes the number of variables, the number of predicates, etc. Hence, a language bias consequently limits the search space for logic programs in ILP models [6]. Furthermore, logic templates can be regarded as a strong language bias, which defines the structures of rules that we expect to hold in a dataset. For example, the occurrences of predicates in learned logic programs [14] or the occurrence orders of variables in the bodies of the generated rules [45] are pre-defined in the logic templates. However, strong language biases may encounter two significant challenges: On the one hand, defining suitable logic templates reduce the usability and flexibility of ILP models, especially for users lacking domain knowledge. On the other hand, while logic templates reduce the search space for ILP models, they also constrain the syntactic formats of the learned logic programs. Consequently, the performance of ILP models can be negatively impacted by inappropriate logic templates. Secondly, some ILP models still lack scalability. For instance, some ILP models may have a danger of overfitting when data is limited [61], then struggling to induce accurate logic programs from a small number of training examples. Furthermore, some ILP models may run out of memory when dealing with a large number of training examples [14,43], and then the model cannot learn logic programs on larger relational facts such as knowledge graphs.

In this paper, we propose a fully differentiable ILP method called differentiable first-order rule learner (DFORL), which generates first-order Horn clauses from input relational facts. The specific contributions of the paper are described as follows. Firstly, the proposed sampling method selects a subset of input relational facts to learn logic programs. With the sampling method, DFORL can learn from both smaller and larger relational facts. Secondly, a novel propositionalization method transfers symbolic relational facts into neural network-readable vector data and makes the neural networks learn logic programs in a supervised fashion. Thirdly, we extend the neural network proposed in the differentiable inductive logic programming model [21] to learn the first-order logic programs. The parameters in a well-trained neural network in DFORL have interpretability. Additionally, each parameter represents the possibility of a first-order atom appearing in a rule. We extend the neural network to support auxiliary predicate invention [54] during the training process. Additionally, by summarizing the regular patterns of common first-order rules such as chain rule [41,28] and reverse format rule [8], we design two meta-rules. We then design syntactic loss functions to learn the rules satisfying the proposed meta-rules based on the correlations of trainable parameters in the neural network and the atoms in symbolic rules. Lastly, we propose a rule extraction method to produce symbolic rules with precision values based on the interpretability of the well-trained weights in the neural network of DFORL.

Our DFORL system learns logic programs without strong language biases such as logic templates as input instead of simple declarations, i.e., the number of variables in logic programs. For some logic template-based ILP models, the occurrence order of variables in the body of a rule is pre-defined, and the model only needs to find the suitable predicates in each rule template [45]. The performances of these ILP models are significantly dependent on the quality and appropriateness of the logic templates used. However, with the number of variables as input, our system automatically learns the predicates and the variable occurrence orders in the bodies of the rules. Besides, our system is fast and scalable. For running time, the matrix representations of logic programs are regarded as the trainable parameters in the neural network; hence, the model supports learning from both small and large datasets in a CPU computation device within an acceptable running time. For scalability, the sampling method makes DFORL learn not only from smaller ILP datasets but also from larger knowledge graphs. Furthermore, employing neural networks to learn rules enhances the robustness of DFORL, enabling effective solving of noisy data, including mislabeled and probabilistic data. Then, we demonstrate that the proposed model outperforms baselines on most datasets, including small ILP datasets and large knowledge graphs.

DFORL also showcases a remarkable level of interpretability by allowing translation between neural network parameters and human-readable symbolic rules through the matrix representations of logic programs. Hence, the generated symbolic rules or prior background knowledge in the form of symbolic rules can be used to facilitate the training process through the interpretability of the neural network in DFORL.

This paper has greatly extended the contents in [20] by considering the sampling method and optimizing the propositionalization method for improving the scalability of DFORL on larger knowledge graphs. We tested the running time of DFORL to prove its efficiency. Then, we compared DFORL with the baselines such as CILP++ [18], D-LFIT [21], TransE [3], RotatE [55], DRUM [46], and RNNLogic [44] to prove its accuracy and relation prediction ability on knowledge graphs. We also proved the advantages of DFORL with both analytical discussions about the complexity and experimental results on more datasets such as UW-CSE [9], Alzheimers-amine [30], WN18 [3], WN18RR [11], and FB15KSelected [58].

The rest of the paper is organized as follows: In Section 2, we present an overview of related concepts. In Section 3, we describe the details of DFORL, including the sampling method, the propositionalization method, the adopted neural network, and the rule extraction method. In Section 4, we test and compare the performance of DFORL. In Section 5, we present a discussion of connections with the existing literature in the ILP research. In Section 6, we conclude the paper and describe further research plans.

## 2. Preliminaries

In this section, we revisit key concepts associated with logic programs, ILP, and the differentiable immediate consequence operator of a logic program.

### 2.1. Logic programs

A *(definite) logic program* $P$ is a finite set of *rules*. A rule is also called a clause, which is structured in the following form [36]:

$$\alpha \leftarrow \alpha_1 \wedge \alpha_2 \wedge \cdots \wedge \alpha_n, \tag{1}$$

where $\alpha$ and $\alpha_i$'s are atoms ($n \geq 0$). For any rule $r$, we designate the atom $\alpha$ as the *head (atom)* of $r$, denoted by head($r$). Moreover, the atoms $\alpha_i$'s are referred to as the *body atoms* of $r$. The conjunction $\alpha_1 \wedge \alpha_2 \wedge \cdots \wedge \alpha_n$ (also represented as $\alpha_1, \alpha_2, \ldots, \alpha_n$) constitutes the *body* of $r$. The set of all body atoms of a rule $r$ is denoted as $body(r)$. If the body of a rule $\alpha \leftarrow$ is empty ($n = 0$), we abbreviate the rule as $\alpha$ and we call the rule a unit clause. A ground unit clause is called a *(relational) fact* [6].

A group of rules sharing a common head atom $\alpha$ is referred to as a *same-head logic program*, represented in the following format:

$$\begin{aligned} &\alpha \leftarrow \beta_1, \\ &\alpha \leftarrow \beta_2, \\ &\quad \vdots \\ &\alpha \leftarrow \beta_m. \end{aligned}$$

This collection can be equivalently presented in a more concise rule format:

$$\alpha \leftarrow \beta_1 \vee \beta_2 \vee \cdots \vee \beta_m,$$

where each $\beta_i$ represents the body of the $i$-th rule, and $\beta_1 \vee \beta_2 \vee \cdots \vee \beta_m$ signifies a disjunction of conjunctions of atoms, i.e., a disjunction normal form formula.

In first-order logic programs, each atom $\alpha$ takes the form $p(t_1, t_2, \ldots, t_n)$, where $p$ denotes an $n$-ary predicate, and $t_1, t_2, \ldots, t_n$ are terms. A term is defined as a variable, a constant, or a function applied to terms.[1] In addition, all variables in first-order rules are universally quantified [36]. In the paper, we use uppercase letters for variables and lowercase letters for constants.

When an atom has no variable, the atom is a *ground atom* [36]. On the contrary, when an atom only has variables, the atom is an *unground atom* [13]. In first-order logic, a *substitution* $\theta$ is a finite set of the form $\{v_1/t_1, \ldots, v_n/t_n\}$, where each $v_i$ is a variable, each $t_i$ is a term distinct from $v_i$ and the variables $v_1, \ldots, v_n$ are distinct [36]. A substitution $\theta$ is called a ground substitution if the $t_i$ are all ground terms.[2] The notation $\alpha\theta$ indicates the ground atom where the atom $\alpha$ is grounded from the substitution $\theta$. Similarly, the notation $r\theta$ indicates the ground rule where the rule $r$ is grounded from the substitution $\theta$. In addition, the set of ground instances of all rules in a logic program $P$ is denoted as $ground(P)$.

In the paper, an atom with a unary predicate $p$ can be simulated with an atom with a binary predicate $p(X, X)$. In some ILP systems to learn first-order rules, *meta-rules* [41,8] are introduced, which are higher-order clauses [6]. Muggleton et al. [41] instantiate predicate variables to predicate symbols within meta-rules for learning first-order logic programs. Meta-rules specify the syntactic formats of logic programs, which include the occurrence orders of variables in the body of a rule or the number of atoms in a rule. Hence, meta-rules can be used as language bias for ILP models to induce logic programs. Examples of meta-rules include the *reverse* rule in the form (2) [8], the *precon* rule in the form (3) [5], the *postcon* rule in the form (4) [5], and the *chain* rule in the form (5) [41].

$$p_t(X, Y) \leftarrow p_i(Y, X). \tag{2}$$

$$p_t(X, Y) \leftarrow p_i(X, Y), p_j(X). \tag{3}$$

$$p_t(X, Y) \leftarrow p_i(X, Y), p_j(Y). \tag{4}$$

$$p_t(X, Y) \leftarrow p_i(X, Z), p_j(Z, Y). \tag{5}$$

In the above meta-rules, the notations $p_t$, $p_i$, and $p_j$ are predicate variables representing predicates symbols. Besides, the predicate variables are existentially quantified and the variables in terms are universally quantified in meta-rules [8].

We use *variable depth* to denote the count of variables that appear in the body of a rule but are not present in the head atom. For example, the variable depths are 0 and 1 in the rules instantiated by the meta-rules (2) and (5), respectively.

The *Herbrand base* $B$ for a first-order logic program $P$ is the set of all ground atoms whose predicate symbols occur in $P$ [36]. An *interpretation* $I$ is a subset of $B$ with the true ground atoms [36]. Given an interpretation $I$, the *immediate consequence operator* $T_P : 2^B \rightarrow 2^B$ of a definite logic program $P$ is defined as: $T_P(I) = \{head(r) \mid r \in ground(P), body(r) \subseteq I\}$ [36].

[1] For simplicity reasons, we impose that the only allowed terms are constants and variables, while function symbols are disallowed. However, the model can be easily generalized. In addition, the maximum arity of a predicate is two.

[2] In the paper, we consider each substitution to be a replacement from a variable to a constant.

## 2.2. Inductive logic programming

ILP has been studied to induce logic programs from training examples and background knowledge [7]. The learning setting of ILP include learning from entailments [14,28], interpretations [10], proofs [42], transitions [27,21,43], and answer sets [34]. In the paper, we consider the learning setting of learning from entailments and learning from interpretation transitions [27]. A *target predicate* is a concept of what we want to learn, which indicates the predicate in the head of a learned logic program [14,6]. Additionally, we define a *target atom* as having the target predicate $p_t$ and a predetermined order of variables, such as $p_t(X, Y)$, serving as the head atom in a learned logic program.

In the setting of learning from entailments, let $\mathcal{B}$ be a set of true ground atoms called *background facts*.[3] Let $\mathcal{P}$ be a set of true ground atoms with the target predicate called *positive examples* and let $\mathcal{N}$ be a set of false ground atoms with the target predicate called *negative examples*. A *solution* (or *hypothesis*) $P$ of an ILP task is:

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

In the process of learning first-order logic programs, propositionalization methods [33] are employed to transform relational data into attribute-valued data. This enables the utilization of neural networks to process the transformed attribute-valued data and subsequently learn the logic programs that describe the facts with target predicates. In the work of França et al. [18], the propositionalization method called bottom clause propositionalization transfers facts to unground atoms with Boolean values. These unground atoms are called *first-order features* or features for short[4] [17].

Besides, Inoue et al. [27] proposed an ILP learning setting called learning from interpretation transitions: Given a set of pairs of interpretations $(I, J)$ where $J = T_P(I)$ serves as positive examples, the goal is to induce a logic program $P$ that realizes the given transition relations. In D-LFIT [21], a neural network is employed to learn propositional logic programs from interpretation transition. In addition, by adapting the bottom clause propositionalization method, they implemented learning first-order logic programs. However, the salability of the bottom clause propositionalization limits the performance of D-LFIT on larger knowledge graphs.

In the paper, we use neural networks to learn from relational facts such as knowledge graphs and ILP datasets. Hence, we will transfer the learning setting from learning from entailments to learning from interpretation transition through a novel propositionalization method. Besides, predicate invention in ILP [54] is the problem of discovering new concepts, properties or relations in structured data [32]. A solution could potentially possess a simpler structure or improved readability, achieved by describing the target predicate using invented *auxiliary predicates* [6,41].

## 2.3. Differentiable immediate consequence operator of logic program

Differentiable ILP frameworks [45,14] improve the robustness of ILP models by adapting the trainable parameters, differentiable operations, gradient-based optimization, and loss functions. In differentiable logic programming process, differentiable operations enable the model to derive continuous values reflecting the confidence of conclusions instead of absolute logical reasoning [7]. In the differentiable ILP models, gradient-based optimization methods and loss functions guide the model to find the optimized parameters, and the well-trained parameters are finally interpreted to construct logic programs.

D-LFIT [21] induced rules through a matrix embedding method for logic programs and a differentiable version of the immediate consequence operator for logic programming. By learning from interpretation transition, the neural network is constrained to learn the optimal logic program matrix representations. In this subsection, we present the matrix representations of logic programs proposed in [21], which is an adaptation of [48] for differentiable settings, and the differentiable immediate consequence operator of a logic program.

Let $P$ be a same-head logic program with a head atom $\alpha_h$ and $m$ different rules. Assuming that the total number of body atoms in all rules of $P$ is $n$. Then $P$ is represented by a *same-head matrix* $\mathbf{M}_P \in [0,1]^{m\times n}$. Each element $a_{kj}$ in $\mathbf{M}_P$ is defined as follows [21]:

1. $a_{kj_i} = l_i$, where $l_i \in (0,1)$ and $\sum_{s=1}^{p} l_s = 1$ $(1 \le i \le p,\ 1 < p,\ 1 \le j_i \le n,\ 1 \le k \le m)$, if the $k$-th rule is $\alpha_h \leftarrow \alpha_{j_1} \wedge \alpha_{j_2} \wedge \cdots \wedge \alpha_{j_p}$;
2. $a_{kj} = 1$, if the $k$-th rule is $\alpha_h \leftarrow \alpha_j$;
3. $a_{kj} = 0$, otherwise.

We use $\mathbf{M}[k, \cdot]$ to denote the $k$-th row in the matrix $\mathbf{M}$. In fact, a same-head matrix is interpretable to a symbolic logic program. Each row in the same-head matrix $\mathbf{M}_P$ corresponds to a rule in $P$, and each non-zero value in a row of $\mathbf{M}_P$ corresponds to a body atom in the corresponding rule in $P$. Different from [48], the matrix representations of same-head logic programs are not unique. To implement the algebraic immediate consequence operator with matrix represents of logic programs and enable the interpretability from the matrix represents of logic programs to symbolic logic programs, we can freely choose the value corresponding to each atom in a row as long as their sum is one.

[3] In certain ILP systems, background facts are expressed as clauses rather than individual atoms. However, aligning with the ILP setting outlined in the work of Evans and Grefenstette [14], we define background facts as ground atoms.

[4] For our model, we only consider unground atoms as first-order features. In [17], first-order features also include constants in atoms.

We use the following example to illustrate one acceptable same-head matrix corresponding to a given same-head logic program based on the matrix representation method:

**Example 1.** Let a same-head program $P$ have three rules. The set of all body atoms of the rules in $P$ is $\{p(X,Z), p(Y,X), p(Y,Z), p(Z,X), p(Z,Y)\}$. The logic program $P$ is defined as follows:

$$p(X,Y) \leftarrow p(Y,Z) \wedge p(Z,X).$$

$$p(X,Y) \leftarrow p(X,Z) \wedge p(Z,Y).$$

$$p(X,Y) \leftarrow p(Y,X).$$

Then, according to the matrix representation method of a logic program, one of the acceptable same-head matrix $\mathbf{M}_P$ of the logic program $P$ is presented as follows:

$$\mathbf{M}_P = \begin{array}{ccccc} p(X,Z) & p(Y,X) & p(Y,Z) & p(Z,X) & p(Z,Y) \end{array} \\ \left[\begin{array}{ccccc} 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} \\ 0 & 1 & 0 & 0 & 0 \end{array}\right].$$

An *interpretation vector* describes the Boolean values of the atoms grounded by first-order features under a substitution. Assume a first-order feature set is $\{\alpha_1, \alpha_2, \dots, \alpha_n\}$, then we use the interpretation vector $\mathbf{v} = [a_1, a_2, \dots, a_n]^T$ to represent the Boolean values of these first-order features under a substitution. If the Boolean value of the first-order feature $\alpha_k$ is true under a substitution, then $a_k = 1$; otherwise, $a_k = 0$. We use $\mathbf{v}[k]$ to denote the $k$-th element in the vector $\mathbf{v}$. We use the following example to illustrate the interpretation vector.

**Example 2.** Consider the same-head logic program $P$ defined in Example 1, and $P$ includes the following first-order features $\{p(X,Z), p(Y,X), p(Y,Z), p(Z,X), p(Z,Y)\}$. Then, the interpretation vector $[0,0,1,1,0]^T$ represent the first-order features $p(Y,Z)$ and $p(Z,X)$ are true under a substitution.

To build the algebraic immediate consequence operator, we describe a threshold function $\varphi$ [49]. Given a vector $\mathbf{v} = [a_1, a_2, \dots, a_n]^T \in \mathbb{R}^n$, the threshold function applied on the vector $\varphi(\mathbf{v}) = [a'_1, a'_2, \dots, a'_n]^T$, where $a'_i = 1$ $(1 \le i \le n)$ if $a_i \ge 1$; otherwise, $a'_i = 0$. Let $P$ be a same-head logic program with $m$ different rules and let interpretation vector $\mathbf{v}_i$ describe the Boolean values of the atoms in $P$ under a substitution $\theta$. The Boolean value of the head atom under the substitution $\theta$ can be computed based on the algebraic immediate consequence operator $D_P$ [48]:

$$D_P(\mathbf{v}_i) = \bigvee_{k=1}^{m} \varphi(\mathbf{M}_P[k,\cdot]\mathbf{v}_i), \tag{6}$$

where interpretation vector $D_P(\mathbf{v}_i)$ indicates the Boolean value of the head atom grounded by the substitution $\theta$. The product $\mathbf{M}_P[k,\cdot]\mathbf{v}_i$ represents the sum of selected elements in the $k$-th row of $\mathbf{M}_P$, where the selected elements correspond to the true ground atoms in the $k$-th rule under the substitution $\theta$. Consequently, the threshold function $\varphi$ activates the truth value of the head atom when all body atoms of the $k$-th rule are true under the substitution $\theta$. We use the following example to illustrate the algebraic immediate consequence operator:

**Example 3.** Consider the same-head logic program $P$ which matrix representation $\mathbf{M}_P$ is defined in Example 1. An interpretation vector $\mathbf{v}$ is the same as the interpretation vector in Example 2. Based on the algebraic immediate consequence operator $D_P$ of the logic program $P$, we can obtain that:

$$\mathbf{M}_P\mathbf{v} = \begin{bmatrix} 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix} [0,0,1,1,0]^T = [1,0,0]^T. \tag{7}$$

Then, the interpretation vector $D_P(\mathbf{v}) = [1]$. Hence, when the atoms grounded from the features $p(Y,Z)$ and $p(Z,X)$ and a substitution $\theta$ are true in an interpretation $I$, then the head atom grounded from the feature $p(X,Y)$ and the substitution $\theta$ in the interpretation $T_P(I)$ is true.

In Eq. (6), the non-differentiable operations include the disjunction operator and the threshold function $\varphi$. To learn logic programs robustly, D-LFIT also introduces a differentiable version of the immediate consequence operator and associates it with a feedforward neural network. The learning processes are described as follows:

Firstly, a matrix denoted as $\tilde{\mathbf{M}}_P$ with a determined size and randomly initialized values in $[0,1]$ is regarded as the trainable parameters in the neural network, and the matrix $\tilde{\mathbf{M}}_P$ represents a logic program.

Secondly, pairs of interpretation vectors $(\mathbf{v}_i, \mathbf{v}_o)$ generated from the bottom clause propositionalization [18] method are given as input data to train the neural network, where each pair of $\mathbf{v}_i$ and $\mathbf{v}_o$ indicate the Boolean values of all body atoms and the target atom under a substitution, respectively. Let $P$ be a solution, then we have $\mathbf{v}_o = D_P(\mathbf{v}_i)$.

Thirdly, the neural network in which forward computation imitates the algebraic immediate consequence operator is proposed in D-LFIT. An adapted multilayer perception serves the architecture of the neural network in D-LFIT, and the forward computation of the neural network in D-LFIT is described as follows:

$$\tilde{\mathbf{v}}_o = \mathrm{FDL}(\phi(\tilde{\mathbf{M}}_P \mathbf{v}_i - \mathbf{1})), \tag{8}$$

where the activation function $\phi$ and the function $\mathrm{FDL} : [0,1]^n \to \mathbb{R}$, called a fuzzy disjunction layer, are defined as follows:

$$\phi(x) = \frac{1}{1+e^{-\gamma x}}, \tag{9}$$

$$\mathrm{FDL}(\mathbf{x}) = \widetilde{\bigvee}_{i=1}^{n} x_i = 1 - \prod_{i=1}^{n}(1 - x_i). \tag{10}$$

The activation function $\phi$ replaces the threshold function $\varphi$, and the hyperparameter $\gamma$ in the function $\phi$ controls the slope similarity between the differentiable activation function $\phi$ and the threshold function $\varphi$. Consequently, because the derivative value of the differentiable activation function $\phi$ reaches the maximum when $x = 0$, to keep the similarity between the threshold function $\varphi$ and differentiable activation function $\phi$, a bias is set to $-1$ in the neural network. In addition, the fuzzy disjunction [25] replaces the disjunction operator, which is differentiable to compute the gradients for neural networks. To avoid a zero gradient when applying the fuzzy disjunction layer in the neural network [14], the product t-norm is chosen as the semantics for the fuzzy disjunction layer.

Finally, through setting the loss function between predicted interpretation vector $\tilde{\mathbf{v}}_o$ indicating the predicted Boolean values for head atoms and label interpretation $\mathbf{v}_o$, the parameters in the neural network $\tilde{\mathbf{M}}_P$ are trained to fit the matrix representation $\mathbf{M}_P$ of a logic program $P$. Through the mapping relationship from the program matrix representations to symbolic logic programs, the learned solution $P$ can be extracted from the trainable parameters $\tilde{\mathbf{M}}_P$ in the neural network.

## 3. Methods

In this section, we propose a new method to learn logic programs from ILP datasets and knowledge graphs. To learn logic programs from entailments, Evans and Grefenstette [14] defined several ILP datasets. Each ILP dataset includes the positive examples $\mathcal{P}$, negative examples $\mathcal{N}$, and background facts $\mathcal{B}$. The atoms in ILP datasets are complete, which include the predicates among any pairs of constants in the data. Hence, a ground atom with the target predicate belongs to positive examples or negative examples. For ILP datasets, we use *fact set* denoted as $F$ to represent the facts appearing in both positive examples and background facts. A knowledge graph [2,37,16] is also a fact set denoted as $F$, and each fact $r(o_i, o_j)$ indicates the predicate (relation) $r$ between subject constant (entity) $o_i$ and object constant $o_j$ [59]. Compared with ILP datasets, most knowledge graphs are incomplete; hence some predicates between the constants in a knowledge graph $F$ may be missing. Hence, for knowledge graphs, based on the input facts $F$, the model learns a solution $P$ to derive positive examples in $F$. The learned solution $P$ may derive ground atoms with the target predicate not occurring in $F$, and then these ground atoms can be regarded as the predicted facts in the knowledge graph $F$.

DFORL learns a same-head logic program as the solution in each training process. The computations in a training process of DFORL include a sampling method, a propositionalization method, and neural network computations. In the section, we elaborate on the above computation processes and the architecture of the neural network in DFORL. In addition, we describe the definition of the precision of a rule in logic programs and the method to extract symbolic same-head logic programs in the section.

### *3.1. Data preprocessing*

In this subsection, we present two algorithms including a sampling method and a propositionalization method to transfer the input fact data $F$ into attribute-valued data $T$. In the paper, we use the symbol $T$ to denote the data generated from the propositionalization method, which are neural network-readable datasets and are used to train the neural network in DFORL.

#### *3.1.1. Sampling method*

When learning from knowledge graphs, the number of facts is typically very large. However, when considering a solution $P$, the size of the intersection of the facts in a knowledge graph and the ground instances of atoms appearing in the bodies of rules in $P$ is much smaller than the number of the facts in the knowledge graph. To delineate this subset of a knowledge graph, we introduce *essential facts* as follows:

**Definition 1** *(Essential facts)*. Given a knowledge graph $F$, let the set of all constants in $F$ be denoted by $C$. Suppose a solution $P$ has $d$ variables in its body. Then, the set of essential facts for $P$ is denoted as $E = \{\alpha \mid \alpha \in \mathrm{body}(P)\theta \cap F, \theta \in \underbrace{C \times \cdots \times C}_{d}\}$.

**Table 1**
Ground atoms in the knowledge graph $F$ represented by tuples in a multigraph $G_F$.

| Edge in $G_F$ | Represented ground atom(s) in $F$ |
|---|---|
| $(o_i, p_k, o_j)$ or $(o_j, p_k, o_i)$ | $p_k(o_i, o_j)$, if $p_k(o_i, o_j) \in F$<br>$p_k(o_j, o_i)$, if $p_k(o_j, o_i) \in F$<br>$p_k(o_i, o_j)$, $p_k(o_j, o_i)$, if $p_k(o_i, o_j), p_k(o_j, o_i) \in F$ |

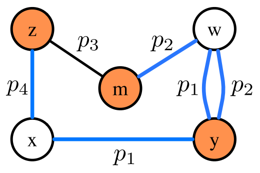


**Fig. 1.** All 2-hop traversed ground atoms from the constant $y$. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

In DFORL, to learn a solution $P$ from a knowledge graph $F$, we use the essential facts denoted as $F'$ selected from $F$ as the input that is enough. Hence, to improve the scalability of DFORL and to decrease the learning time, we design a sampling algorithm to select essential facts from knowledge graphs.

According to Definition 1, the number of variables in a rule determines the number of essential facts. Hence, the sampling algorithm receives the number of variables in the solution as input to generate the essential facts. During the sampling process, we regard a knowledge graph as an undirected multigraph.[5] Each constant in a knowledge graph is regarded as a node in the multigraph, and each predicate in a knowledge graph is regarded as a labeled edge. Specifically, if there exists a fact $p_k(o_i, o_j)$ or $p_k(o_j, o_i)$ in the knowledge graph $F$, then there exists an edge between the constants $o_i$ and $o_j$ labeled with the predicate $p_k$ denoted as $(o_i, p_k, o_j)$ or $(o_j, p_k, o_i)$ in the corresponding multigraph $G_F$. Hence, each edge in a multigraph $G_F$ maps one or two ground atoms in the knowledge graph $F$. In addition, if an atom with unary predicate $p_k(o_i)$ occurs in a knowledge graph $F$, then there exists a loop on the constant $o_i$ labeled with the predicate $p_k$ in the multigraph $G_F$. We use the symbol $(o_i, p_k, o_j)_F$ to represent the ground atoms in a knowledge graph $F$ represented by the edge $(o_i, p_j, o_j)$ in the multigraph $G_F$. The mapping relationship from an edge in a multigraph $G_F$ to the atoms in the knowledge graph $F$ is presented in Table 1.

The sampling algorithm selects the essential facts by traversing from nodes in the multigraph $G_F$ corresponding to a knowledge graph $F$. Now, we define an $n$-hop traverse as follows:

**Definition 2** *(N-hop traverse).* Given a knowledge graph $F$, let $G_F$ be the corresponding multigraph for $F$. A 0-hop traverse does not encompass any ground atoms. When $n \geq 1$, assuming $(o_{i_x}, p_{k_y}, o_{i_z}) \in G_F$ for $x, z \in [0, n]$ and $y \in [1, m]$, an $n$-hop traverse starting from the constant $o_{i_0}$ can be represented as the set $F_t = \{(o_{i_0}, p_{k_1}, o_{i_1})_F, (o_{i_1}, p_{k_2}, o_{i_2})_F, \dots, (o_{i_{n-1}}, p_{k_m}, o_{i_n})_F\}$, where $F_t$ is a subset of the knowledge graph $F$ and called traversed atom set. In an $n$-hop traverse, any two elements in the traversed atom set are different. Each ground atom in $F_t$ is called a traversed atom, and the constant $o_{i_n}$ is regarded as the target constant after the $n$-hop traverse.

We use the following example to illustrate all 2-hop traverses in a knowledge graph:

**Example 4.** Assume a knowledge graph $F = \{p_1(w, y),\ p_1(x, y),\ p_2(y, w),\ p_2(w, m),\ p_3(z, m),\ p_4(z, x)\}$. The undirected multigraph $G_F$ corresponding to $F$ can be represented by Fig. 1. Then all 2-hop traverses from the constant $y$ are $\{\{(y, p_1, w)_F, (w, p_2, y)_F\}, \{(y, p_1, w)_F, (w, p_2, m)_F\}, \{(y, p_2, w)_F, (w, p_2, m)_F\}, \{(y, p_1, x)_F, (x, p_4, z)_F\}\}$. Correspondingly, the traversed atoms in $F$ are $\{\{p_1(w, y),\ p_2(y, w)\},\ \{p_1(w, y),\ p_2(w, m)\},\ \{p_2(y, w),\ p_2(w, m)\},\ \{p_1(x, y),\ p_4(z, x)\}\}$. The edges in blue color represent the traversed atoms and the nodes in orange color represent the target constant after all 2-hop traverses.

In the sampling method, the input parameter variable depth $d$ is required to be greater than or equal to zero. To expedite the discovery of rules, we introduce a reduction ratio denoted by $\delta$, representing the proportion of positive examples used as input to the sampling method relative to the total number of positive examples. In general, utilizing distinct positive examples as input for the sampling method can yield varying rules learned by DFORL. Consequently, we can iteratively apply the sampling method and subsequently run the model to generate additional rules. Based on each input positive example, the sampling method first traverses from one constant $o_i$ in a positive example $p_t(o_i, o_j)$ with $n_1$-hop, then the sampling method traverses from another constant $o_j$ in the positive example $p_t(o_i, o_j)$ with $n_2$-hop. The sum of the hops satisfies $n_1 + n_2 = d + 1$, and we ask $|n_1 - n_2| \leq 1$. Then, these having the common target constants traversed from the two constants in a positive example are selected into essential facts. We present the sampling method in Algorithm 1.

[5] A graph in which multiple edges may connect the same pair of vertices is called a multigraph. A multigraph in the paper may include loops.

**Algorithm 1** The sampling method in DFORL.

```
Input: A knowledge graph $F$, a variable depth $d$, and a ratio $\delta$ indicating the number of considered positive examples to the number of all positive examples.
Output: A set $F'$ with essential facts.
1: Set an empty set $F'$ to store the essential facts.
2: Randomly choose $\delta \cdot |\mathcal{P}|$ positive examples as the considered positive examples and let $\mathcal{P}'$ be the set of considered positive examples.
3: for each positive example $p_t(o_a, o_b)$ in $\mathcal{P}'$ do
4:     Add the positive example $p_t(o_a, o_b)$ to the reduced set $F'$.
5:     for each $(s,t) \in \{(a,b),(b,a)\}$ do
6:         Initialize an empty set $O$. Start all traverses from the constant $o_s$ with $\lceil \frac{d}{2} \rceil$-hop, then add all target constants to $O$.
7:         If $d$ is odd, then start all traverses $T'$ from the constant $o_t$ with $\lceil \frac{d}{2} \rceil$-hop; If $d$ is even, then start all traverses $T'$ from the constant $o_t$ with $(\frac{d}{2}+1)$-hop. For each traverse in $T'$, let the target constant in each traverse be $o'$. When the constant $o' \in O$, then add the traversed ground atoms in this traverse process to $F'$.
8:     end for
9: end for
10: return $F'$
```

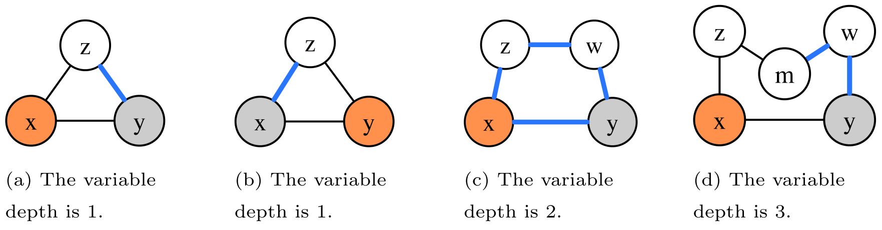

z
x
y
(a) The variable depth is 1.
z
x
y
(b) The variable depth is 1.
z
w
x
y
(c) The variable depth is 2.
z
m
w
x
y
(d) The variable depth is 3.

**Fig. 2.** Four examples of the sampling method with different variable depths as the input. When the orange constant is the start traverse constant $o_s$ in line 6 of Algorithm 1, and the gray constant is the start traverse constant $o_t$ in line 7 of Algorithm 1, then the traversed facts represented by the blue edges and the connected constants are added into the set $F'$ in line 7 of Algorithm 1.

From the viewpoint of logic programming, let $P$ be the learned solution from the essential fact set $F'$ from the sampling algorithm, then we have $T_P(F') \subseteq F'$. The computational complexity of the sampling method is $O(\delta \cdot |\mathcal{P}| \cdot \Delta(G)^{\lfloor \frac{d}{2} \rfloor + 1})$, where $d$ is the variable depth, $G$ is the undirected graph constructed from the training facts, and $\Delta(G)$ is the maximum degree of a graph $G$.

We present four examples in Fig. 2 to show the results of the sampling method. In the examples, each multigraph represents a knowledge graph, and we omit predicate labels for all edges. In summary, the sampling method operates by sampling facts within knowledge graphs at the fact level. Particularly, we employ essential facts $F'$ generated from the sampling method as the input for the propositionalization method especially when dealing with large datasets.[6]

### *3.1.2. Propositionalization method*

In DFORL, we use neural networks to learn the logic programs. However, the neural networks only process the attribute-value data, whereas the examples in input facts $F$ are relational data. Hence, we propose a propositionalization method to transfer facts $F$ to neural network-readable data $T$.

In propositionalization methods [18], each first-order feature is assigned a Boolean value. A Boolean value of a first-order feature represents the Boolean value of the ground atom instantiated by the first-order feature and a substitution. In the paper, we use an interpretation vector to represent the Boolean values for first-order features under a substitution.

In the paper, we generate a same-head logic program in a training process, where the same-head logic program is headed by the target atoms. For a solution, we specify that all possible *body features* are generated by combining all predicates and permutations of variables except the target atom. Hence, the terms in the body of a rule generated by DFORL only be variables. Specifically, when the target atom also occurs in the body of a rule, then the logic formula represented by this rule is a tautology. Hence, we do not allow the target atom to occur in the body of a rule.

Inoue et al. [27] proposed an ILP learning strategy that learning rules from interpretation transition: Given pairs of interpretations $(I, J)$, the model learns a logic program $P$ which satisfies $J = T_P(I)$, where the interpretation $I$ represents the Boolean values for all body features, and the interpretation $J$ represents the Boolean values for head atoms. Through the vector representation of an interpretation $I$ as the input for a neural network and the vector representation of an interpretation $J$ as the label output of the neural network, Gao et al. [21] train the neural network to learn logic program $P$ from pairs of input and output interpretation vectors. In the paper, we also consider generating pairs of interpretation vectors through the proposed propositionalization method from input facts $F$, and then we use neural networks to learn the solution from pairs of interpretation vectors.

The main idea of the proposed propositionalization method can be described in three phases. The first phase is called the preparation process. To generate pairs of interpretations, we need to determine the Boolean values for all features in the propositionalization method. In a rule, each variable may bind a set of constants. Hence, we will check the sets of constants bound by the variables in the target atom first. Then, based on these constant sets bound by variables, we can obtain all substitutions.

[6] We provide a declaration when applying the sampling method in DFORL.

The second phase is called the generation process. Under each substitution, we can generate an *input interpretation vector* denoted as $\mathbf{v}_i$ to include the Boolean values of all body features. Specifically, let $\alpha_k$ be the $k$-th body feature, then we can determine the Boolean value of $\alpha_k\theta$ based on a substitution $\theta$ and input facts $F$. For ILP datasets, the input facts $F$ include complete ground atoms, where all possible predicates between constants are included. Hence, if $\alpha_k\theta \in F$, then $\mathbf{v}_i[k] = 1$; otherwise, the ground atom $\alpha_k\theta$ is false, and we set $\mathbf{v}_i[k] = 0$. For a knowledge graph, we also set $\mathbf{v}_i[k] = 1$ when $\alpha_k\theta \in F$. However, some ground atoms may be missing in a knowledge graph, where some predicates between constants may not be included. The Boolean values of the missing ground atoms are uncertain, yet within most knowledge graphs, there is a high likelihood that these missing ground atoms are false. For the sake of the robustness of neural networks, we can set $\mathbf{v}_i[k] = 0$ when $\alpha_k\theta \notin F$. Similarly, we can generate an *output interpretation vector* $\mathbf{v}_o$ to include the Boolean value of the target atom $\alpha_t$. Specifically, if the $\alpha_t\theta \in F$, then $\mathbf{v}_o = \mathbf{1}$; otherwise, $\mathbf{v}_o = \mathbf{0}$.

Based on the substitutions, we obtain pairs of interpretation vectors $(\mathbf{v}_i, \mathbf{v}_o)$. In each substitution, when the target atom is true, we let the conjunction of the true body features be the body of a rule $r_\perp$, and let the target atom be the head of the rule $r_\perp$, then the rule $r_\perp$ can be called a bottom rule [40], the logically most-specific clause that explains the target predicate. Let a logic program $P_\perp$ include all the bottom rules generated by all substitutions, and then the logic program $P_\perp$ can be regarded as a boundary in the solution search space during ILP learning. We have $\mathbf{v}_o = D_{P_\perp}(\mathbf{v}_i)$, where $(\mathbf{v}_i, \mathbf{v}_o)$ is any pair of interpretation vectors generated from the substitutions. Let $r$ be a rule in a simpler solution $P$, then we have $body(r) \subseteq body(r_\perp)$, where $r_\perp$ is a bottom rule. Then, we have $\mathbf{v}_o = D_P(\mathbf{v}_i)$, for any pairs of interpretation vectors $(\mathbf{v}_i, \mathbf{v}_o)$ generated from all substitutions. Hence, we can design a neural network described in Eq. (8) to find a solution $P$ given pairs of interpretation vectors $(\mathbf{v}_i, \mathbf{v}_o)$ as the training data. After the training, the forward computation of the neural network can be regarded as the immediate consequence operator of the solution $P$. By interpreting the neural network in DFORL, we can induce a simpler logic program $P$ from the well-trained neural network which satisfies $\mathbf{v}_o = D_P(\mathbf{v}_i)$. Hence, the extracted rules from DFORL are regarded as solutions to ILP problems.

Based on two specific situations, we design the third phase in the propositionalization method called the examination process. On the one hand, we delete invalid instances with the input interpretation vector $\mathbf{v}_i = \mathbf{0}$ but the output interpretation vector $\mathbf{v}_o = \mathbf{1}$, which indicates that all body features are false, but the target atom is true. This situation contradicts any possible solutions but happens under some substitutions. For example, assume a target atom is $grandparent(X, Y)$ and body features are generated by combining the predicates $father$, $mother$, and $grandparent$ and two variable permutations from all variables $X$, $Y$, and $Z$ such as $father(X, Z)$ and $mother(Z, Y)$. Let the variables $X$ and $Y$ be substituted by the persons $Alice$ and $Bob$ who have the $grandparent$ relationship in a family and let the variable $Z$ be substituted by the person $Carol$ from another family. Then, the target atom $grandparent(X, Y)$ is true but all body features are false under this substitution. Besides, invalid instances also increase the loss during the training phase, and we discard invalid instances in the examination process.

On the other hand, the Boolean values for some body features are always false under all substitutions in some tasks, and these body features called invalid features are impossible to appear in the body of a solution. For example, if a target atom $born\_in(X, Y)$ indicates a people $X$ is born in a city $Y$, then the body feature $born\_in(Y, X)$ is always false when the domains X and Y include all people and cities in a dataset, respectively. Besides, the elements in the input interpretation vectors corresponding to invalid features will not contribute any information to adjust the corresponding weights when training the neural network in DFORL. Hence, we delete invalid features and the corresponding elements in all input interpretation vectors in the examination process. After deleting invalid features in the examination process of the propositionalization method, the rest unground atoms may occur in the body of a solution called *valid features*. Under a substitution, the Boolean value of each valid feature is represented by each element in the $\mathbf{v}_i$. Then, we denote the number of valid features as $n_f$. Hence, the size of each input interpretation vector is $|\mathbf{v}_i| = n_f$.

Now, we describe the input arguments of the propositionalization method as follows: Firstly, the set with all variables in a solution denoted as $V$ is an input argument of the propositionalization method, which determines the variable depth $d$, the size of the search space for a solution, and the number of body features in the solution.

Secondly, the target atom and all possible body features are the input for the propositionalization method. Let the number of binary predicates be $n_b$, let the number of unary predicates be $n_u$, and let the set of all possible body features occurring in the solution be $P_F$. We have $|P_F| = n_b \cdot \mathrm{P}(|V|, 2) + n_u \cdot |V| - 1$, where $\mathrm{P}(|V|, 2)$ is the number of permutations of two items from the variable set $V$.

Thirdly, we define a partial substitution set $S$. The substitutions in a partial substitution set are randomly chosen from all substitutions. A parameter $\xi$ represents the number of partial substitutions to the number of all substitutions, which is also an input for the propositionalization method. When learning logic programs with large variable depths, the running time of the propositionalization method may be longer than the requirements. For the sake of neural networks, we can generate a part of interpretation vectors generated from partial substitutions and let the neural networks find the solution. Compared with the sampling method, the partial substitutions can be regarded as another step of sampling at the substitution level. In general, a smaller ratio $\xi$ will make the learning process more difficult.[7] Now, we present the propositionalization method in Algorithm 2.

After applying the propositionalization method, the ground atoms in input facts $F$ are transformed into a neural network-readable dataset $T$, which consists of pairs of interpretation vectors $(\mathbf{v}_i, \mathbf{v}_o)$. Moreover, the complexity of the propositionalization method is $O(|S| \cdot |P_F|)$. We use Example 5 under the *predecessor* relation to illustrate the proposed propositionalization method.

**Example 5.** In the Predecessor (pre) task, assume the constant set is $E = \{0, 1, 2\}$. The training background facts $\mathcal{B}$ and the training positive examples $\mathcal{P}$ are $\{succ(0, 1), succ(1,2)\}$ and $\{pre(1, 0), pre(2, 1)\}$, respectively. Assume that the variable set is $V = \{X, Y\}$.

[7] We consider $\xi$ as a parameter in the whole learning process, and we let $\xi = 1$ by default without a special statement.

**Algorithm 2** The propositionalization method in DFORL.

**Input**: A variable set $V = \{X, Y, V_1, V_2, \ldots, V_d\}$; target atom $\alpha_t$, e.g., $p_t(X, Y)$ with binary predicate or $p_t(X)$ with unary predicate; body features set $P_F$; the ratio of partial substitutions to all substitutions $\xi$; training positive examples $\mathcal{P}$; and the training fact set $F$.
**Output**: A dataset $T$ for training neural networks.

```
   (Preparation process)
1: Let the domains X, Y, $\mathrm{V}_1$, $\mathrm{V}_2$, …, $\mathrm{V}_d$ include the constants represented by the variables $X$, $Y$, $V_1$, $V_2$, …, $V_d$, respectively. Initiate $T$ as an empty set.
2: If the target predicate $p_t$ is binary, then for each positive example $p_t(o_1, o_2) \in \mathcal{P}$, add $o_1$ and $o_2$ to the sets X and Y, respectively. Besides, add all constants in $F$ to the sets $\mathrm{V}_1, \mathrm{V}_2, \ldots, \mathrm{V}_d$. If $p_t$ is unary, for each positive example $p_t(o_1) \in \mathcal{P}$, add $o_1$ to the set X. Besides, add all constants in $F$ to the sets Y, $\mathrm{V}_1, \mathrm{V}_2, \ldots, \mathrm{V}_d$.
3: Let $U$ indicate the complete substitution set $\mathrm{X} \times \mathrm{Y} \times \mathrm{V}_1 \times \mathrm{V}_2 \times \cdots \times \mathrm{V}_d$, and randomly choose $\xi \cdot |U|$ substitutions from the set $U$ to a partial substitution set $S$.
   (Generation process)
4: for each $\theta_k = \{X/x_k, Y/y_k, V_1/v_1^k, \ldots, V_d/v_d^k\} \in S$ do
5:     To add interpretation vectors $(\mathbf{v}_i^k, \mathbf{v}_o^k)$ to $T$, begin by initializing $\mathbf{v}_i^k = \mathbf{0}$ and $\mathbf{v}_o^k = [0]$. Under the substitution $\theta_k$, for each $\alpha_j \in P_F$, $\mathbf{v}_i^k[j] = 1$ if $\alpha_j\theta \in F$; Then $\mathbf{v}_o^k = [1]$ if $\alpha_t\theta \in \mathcal{P}$. Then, add the pair of interpretation vectors $(\mathbf{v}_i^k, \mathbf{v}_o^k)$ to $T$.
6: end for
   (Examination process)
7: For each instance in $T$, delete the instance $(\mathbf{v}_i^k, \mathbf{v}_o^k)$ iff $\mathbf{v}_i^k = \mathbf{0}$ and $\mathbf{v}_o^k = \mathbf{1}$.
8: Discard the $m$-th body feature and the corresponding values in all input interpretation vectors iff for all $k \in [1, |T|]$, $\mathbf{v}_i^k[m] = 0$ ($m \in [1, |P_F|]$) holds.
9: return $T$
```

The target atom is $pre(X, Y)$, and all possible body features are $P_F = \{succ(X, Y), succ(Y, X), pre(Y, X)\}$. In the preparation process, according to all positive examples, the domains X={1,2} and Y={0,1}. Then, the set with all substitutions is $\{\{X/1, Y/0\}, \{X/1, Y/1\}, \{X/2, Y/0\}, \{X/2, Y/1\}\}$, and the element $\{X/1, Y/0\}$ indicates the variables $X$ and $Y$ are substituted by the constants 1 and 0, respectively. In the generation process, according to all substitutions, we can determine the truth value for each body feature based on the training positive examples and background facts. Hence, the neural network-readable dataset is $T = \{([0, 1, 0], [1]), ([0, 0, 0], [0]), ([0, 0, 0], [0]), ([0, 1, 0], [1])\}$. Each element in $T$ represents a pair of training instance $(\mathbf{v}_i, \mathbf{v}_o)$. The input interpretation vectors represent input data for neural networks, and the output interpretation vectors represent labels for neural networks. During the examination process, we check that the values of the first and the third elements in each input interpretation vector are always zero. Therefore, the first element and the third element in all input interpretation vectors are deleted. In addition, the first body feature $succ(X, Y)$ and the third body feature $pre(Y, X)$ in $P_F$ are discarded. As a result, only the valid body predicate $succ(Y, X)$ is kept, and the neural network-readable data $T = \{([1], [1]), ([0], [0])\}$. During the training process, only one weight corresponding to the feature $succ(Y, X)$ is trained. Hence, we can extract one rule: $pre(X, Y) \leftarrow succ(Y, X)$.

## 3.2. The neural network in DFORL

In DFORL, we design a neural network to learn logic programs from the neural network-readable data $T$ generated from the propositionalization method. In this subsection, we describe the architecture of the proposed neural network and loss functions in DFORL.

In each pair of interpretation vectors of the neural network-readable data $T$, each element in an input vector $\mathbf{v}_i$ represents the Boolean value of the valid feature under a substitution $\theta$, and the value of the output interpretation vector $\mathbf{v}_o$ indicates the Boolean value of the target atom under the substitution $\theta$. In addition, the body atoms in a solution must be the subset of the valid features. Let the solution be $P$, then for each pair of interpretation vectors denoted as $\mathbf{v}_o$ and $\mathbf{v}_i$, we have $\mathbf{v}_o = D_P(\mathbf{v}_i)$, where the function $D_P$ is the algebraic immediate consequence operator of the solution $P$ defined in Eq. (6).

In this subsection, we will present how to train a neural network with the pairs of interpretation vectors $(\mathbf{v}_i, \mathbf{v}_o)$ generated from the propositionalization method, where $\mathbf{v}_i$ as the input of the neural network and $\mathbf{v}_o$ as the label output of the neural network. The neural network in DFORL is adapted based on the neural network in D-LFIT [21] defined in Eq. (8). In D-LFIT, a trainable matrix $\mathbf{M}_P^S$ called *same-head tensor* is regarded as the weights in the neural network. Through applying the differentiable activation function $\phi$ defined in Eq. (9) and the fuzzy disjunction layer defined in Eq. (10), the forward computing of the neural network emulates the algebraic immediate consequence operator of the solution. In D-LFIT, through applying the loss functions which we call as *semantics loss function* in the paper, the weights $\mathbf{M}_P^S$ meet algebraic immediate consequence operator $\mathbf{v}_o = D_P(\mathbf{v}_i)$. Hence, the weights $\mathbf{M}_P^S$ in the well-trained neural network can be interpreted as a matrix representation for a solution $P$. Specifically, the semantics loss functions in D-LFIT include *inference loss* and *sum loss* functions.

In the paper, we add more parameters, loss functions, and operations to improve the accuracy when learning first-order logic programs. On the one hand, we also use a same-head tensor as the matrix representation for a solution. On the other hand, we introduce additional parameters in the neural network referred to as *auxiliary tensor*, along with merge operations to facilitate auxiliary predicate invention. The introduced auxiliary tensor improves the accuracy of DFORL within the specified running time. After the merge operation on the auxiliary tensor, we obtain an *average tensor*. Then, we concatenate the same-head tensor and average tensor to form a *program tensor* denoted as $\mathbf{M}_P$, which is interpretable to a solution after training.

In addition, we ask DFORL to learn the rules that are instantiated based on proposed meta-rules. Because the occurrence orders of variables in rules instantiated from the pre-defined meta-rules have obvious patterns, we define two syntactic constraints that any solution should follow. Correspondingly, we propose the *syntactic loss functions*, which include *range-restricted loss*, *connectedness loss*, and two *similarity loss* functions. The range-restricted loss and connectedness loss serve as mechanisms to satisfy the two defined syntactic constraints within the neural network. Additionally, the similarity loss is designed to increase the dissimilarity between different rules, promoting rule diversity and improving the learning process.

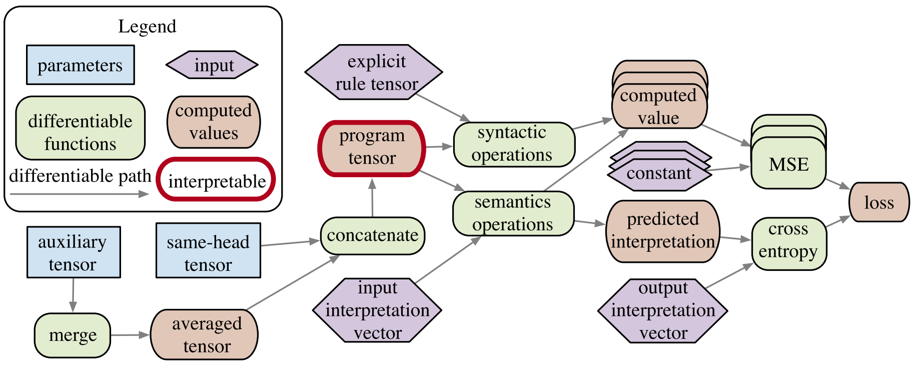


**Fig. 3.** The architecture of DFORL. The stacked elements indicate the multiple values or functions. MSE indicates mean square error loss function. Syntactic and semantics operations represent the operations in the syntactic loss and semantic loss functions, respectively.

A high-level architecture of DFORL is presented in Fig. 3. The inputs in purple include pairs of interpretation vectors generated from the propositionalization method, some constant values, and the well-trained rule embeddings from the previous training processes. DFORL is a fully differentiable model, and the differentiable functions are shown in green. The trainable parameters are shown in blue, and the loss is used to update these parameters by gradient descent algorithm. After a training process, the interpretable program tensor in the red box can be mapped to symbolic logic programs directly.

### 3.2.1. *Semantics loss functions*

In this subsection, we provide a detailed explanation of the parameters in the neural network, the operations on the parameters, and the logic program semantics loss functions in DFORL. The semantics loss functions include the inference loss and sum loss functions. We applied these loss functions to the neural network based on the differentiable immediate consequence operator defined in Eq. (8) and the same-head matrix representations of logic programs.

First, according to the matrix representations of logic programs defined in Section 2.3, we set a same-head tensor $\mathbf{M}_P^S \in [0,1]^{m_1 \times n_f}$ to encode rules in the logic program $P$ headed by the target atom. Because of the number of rows in a same-head matrix representing the number of rules in a logic program, the hyperparameter $m_1$ in the same-head tensor indicates the number of learned rules encoded by $\mathbf{M}_P^S$ through a training process, and $n_f$ is a constant indicating the number of valid features.

Now, we define a setting for auxiliary predicate [41] invention: Consider a rule $r$ as the solution, and let an auxiliary predicate be the head of another rule $r'$, where the set with predicates in the body of $r'$ is a subset of the set with predicates in the body of $r$. In this case, the target predicate can also be described using these auxiliary predicates. Besides, an auxiliary predicate characterizes a relation that might not be present in the input relational dataset. For example, let a logic program $P$ with *grandparent* as the target predicate:

$$\begin{aligned}
&grandparent(X,Y) \leftarrow mother(X,Z), father(Z,Y).\\
&grandparent(X,Y) \leftarrow mother(X,Z), mother(Z,Y).\\
&grandparent(X,Y) \leftarrow father(X,Z), father(Z,Y).\\
&grandparent(X,Y) \leftarrow father(X,Z), mother(Z,Y).
\end{aligned} \tag{11}$$

Then, the logic program $P$ can be described by the auxiliary predicate *parent* in the rule (12), and the auxiliary predicate *parent* serves as the target predicate of the rules (13) and (14).

$$grandparent(X,Y) \leftarrow parent(X,Z), parent(Z,Y). \tag{12}$$

$$parent(X,Y) \leftarrow mother(X,Y). \tag{13}$$

$$parent(X,Y) \leftarrow father(X,Y). \tag{14}$$

We implement this auxiliary predicate invention setting in this subsection. We set a trainable matrix $\mathbf{M}_P^A \in [0,1]^{m_2 \times n_a \times n_f}$ called auxiliary tensor to encode the rules with auxiliary predicates. For the construction of the auxiliary tensor, we stack $n_a$ matrices with the size of $m_2 \times n_f$. We use the merge operation to implement the auxiliary predicate invention as follows:

$$\mathbf{M}_P^{A'} = \frac{1}{n_a}\sum_{i=1}^{n_a} \mathbf{M}_P^A[\cdot, i, \cdot],$$

where, $\mathbf{M}_P^A[\cdot, i, \cdot]$ denotes the $i$-th sub-matrix with the size of $m_2 \times n_f$ in the three-dimensional tensor $\mathbf{M}_P^A$. The merge operation performs on the sub-matrix with the size of $m_2 \times n_f$ in the auxiliary tensor $\mathbf{M}_P^A$ and produces an averaged tensor $\mathbf{M}_P^{A'} \in [0,1]^{m_2 \times n_f}$.

Now, we analyze why the auxiliary tensor and the merge operation can implement auxiliary predicate invention. If the $k$-th row in the averaged tensor $\mathbf{M}_P^{A'}$ with the size of $m_2 \times n_f$ encodes a rule headed by the target atom, then each row of the sub-matrix $\mathbf{M}_P^A[k,\cdot,\cdot]$ $(1 \le k \le m_2)$ with the size of $n_a \times n_f$ in the auxiliary tensor $\mathbf{M}_P^A$ encodes a part of or the complete body atoms of the rule. That is, a long rule $r_l$ encoded by the row $\mathbf{M}_P^{A'}[k,\cdot]$ in the averaged tensor is divided to several shorter rules $r_{s_i}$ $(1 \le i \le n_a)$ encoded by rows in the submatrix $\mathbf{M}_P^A[k,\cdot,\cdot]$ of the auxiliary tensor, and $body(r_{s_i}) \subseteq body(r_l)$ holds. Besides, for some long rules encoded by rows in averaged tensor, the heads of the shorter rules encoded by rows in submatrix $\mathbf{M}_P^A[k,\cdot,\cdot]$ may be interpreted to auxiliary predicates. Hence, the hyperparameter $m_2$ indicates the number of rules headed by the target atom, and the hyperparameter $n_a$ indicates the number of rules headed by auxiliary predicates. From the viewpoint of parameters, auxiliary tensor $\mathbf{M}_P^A$ has more trainable parameters than the same-head tensor $\mathbf{M}_P^S$. Hence, with more trainable parameters, DFORL can find a solution accurately.

We use the following example to illustrate the auxiliary predicate invention:

**Example 6.** We consider the following logic program $P$ describing the target predicate *grandparent* ($g$) with the body predicates *mother* ($m$) and *father* ($f$):

$$g(X,Y) \leftarrow m(X,Z), f(Z,Y).$$

Assume the valid body feature set is equal to the set with all body features $P_F = \{m(X,Y), m(X,Z), m(Y,X), m(Y,Z), m(Z,X), m(Z,Y), f(X,Y), f(X,Z), f(Y,X), f(Y,Z), f(Z,X), f(Z,Y), g(X,Z), g(Y,X), g(Y,Z), g(Z,X), g(Z,Y)\}$ and the number of valid features is $n_f = 17$. Then, consider a same-head matrix $\mathbf{M}_P$ in $[0,1]^{1\times 17}$ corresponding to the logic program $P$ as follows:

$$\mathbf{M}_P = \begin{array}{c} \\ \end{array}\begin{array}{ccccc} m(X,Y) & m(X,Z) & \dots & f(Z,Y) & \dots & g(Z,Y) \\ \end{array}$$
$$\mathbf{M}_P = \left[\begin{array}{cccccc} 0 & 0.5 & \dots & 0.5 & \dots & 0 \end{array}\right].$$

If we do not consider the same-head tensor $\mathbf{M}_P^S$, then a possible auxiliary tensor $\mathbf{M}_P^A \in [0,1]^{1\times 2\times 17}$ is:

$$\mathbf{M}_P^A[1,\cdot,\cdot] = \begin{array}{cccccc} m(X,Y) & m(X,Z) & \dots & f(Z,Y) & \dots & g(Z,Y) \\ \end{array}$$
$$\mathbf{M}_P^A[1,\cdot,\cdot] = \left[\begin{array}{cccccc} 0 & 1 & \dots & 0 & \dots & 0 \\ 0 & 0 & \dots & 1 & \dots & 0 \end{array}\right].$$

The logic programs encoded in the auxiliary tensor $\mathbf{M}_P^A$ are interpreted as:

$$p_1 \leftarrow m(X,Z). \tag{15}$$

$$p_2 \leftarrow f(Z,Y). \tag{16}$$

The rules (15) and (16) are regarded as rules headed by the auxiliary atom $p_1$ and $p_2$, which can be interpreted as *parent*$(X,Z)$ and *parent*$(Z,Y)$, respectively.

Then, we design a concatenate function to form the same-head tensor and averaged tensor into the program tensor $\mathbf{M}_P \in [0,1]^{m\times n_f}$ where $m = m_1 + m_2$ as follows:

$$\mathbf{M}_P = \text{concat}\left(\mathbf{M}_P^S, \mathbf{M}_P^{A'}\right),$$

where tensors in the arguments of the concatenate function are joined along the vertical dimension. The concatenate function applies gradient information equally on the same-head tensor and auxiliary tensor. Hence, the concatenate function makes the same-head tensor and auxiliary tensor encode the rules simultaneously through a training process.

To learn the optimal program tensor $\mathbf{M}_P$ encoding the logic program from pairs of interpretation vectors, we set the inference loss in the neural network. The inference loss enforces the forward computation of the neural network to imitate the immediate consequence operator of a same-head logic program $P$:

$$\bar{\mathbf{v}}_o = \text{FDL}\left(\phi(\mathbf{M}_P\mathbf{v}_i - \mathbf{1})\right),\ L_I = H(\bar{\mathbf{v}}_o, \mathbf{v}_o),$$

where the activation function $\phi$ is defined in Eq. (9). The function FDL is regarded as a fuzzy disjunction layer in the neural network. The function $H$ denotes the binary cross-entropy function. When a program tensor is constrained by the inference loss, the averaged tensor and same-head tensor represent two parts of rules headed by the target atom.[8]

The merge and concatenate functions are illustrated in Fig. 4. The merge function reduces a dimension from an auxiliary tensor to an averaged tensor. A same-head tensor and the average tensor are concatenated together to a program tensor $\mathbf{M}_P$. In Fig. 4, the inference loss in the yellow box computes over the program tensor. The program tensor is computed from the auxiliary tensor, the averaged tensor, and the same-head tensor. Hence the inference loss also affects the values of the auxiliary tensor, the averaged tensor, and the same-head tensor. Besides, we explicitly present the inference loss in Fig. 5. The operations related to the inference

[8] We adjust the values of hyperparameters $\gamma$ in the function $\phi$ described in Eq. (9) and the learning rate in different tasks to avoid the zero or exploding gradients.

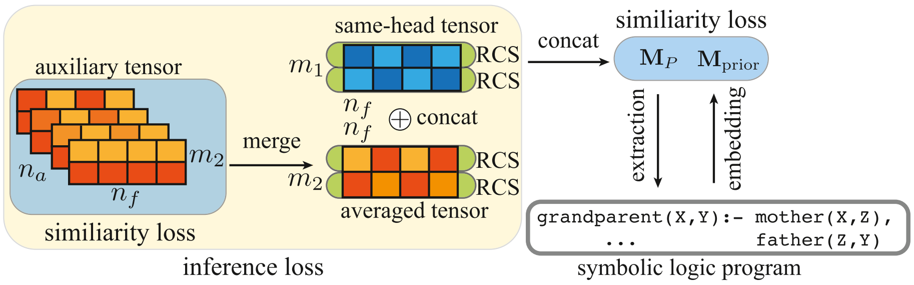


**Fig. 4.** The merge and concatenate functions operate on an auxiliary tensor, a same-head tensor, and an averaged tensor with the parameters $n_a = 4$, $m_1 = m_2 = 2$, and $n_f = 4$. Each colored box is titled with a specific loss function. Specifically, the blue box represents the similarity loss, which enforces constraints on the auxiliary tensor, program tensor $\mathbf{M}_P$, and explicit rule tensor $\mathbf{M}_{\text{prior}}$. The range-restricted loss, connectedness loss, and sum loss (collectively denoted as RCS) enforce constraints on both the same-head tensor and averaged tensor. Additionally, the inference loss is applied to the program tensor, which is obtained by concatenating the same-head tensor and the average tensor.

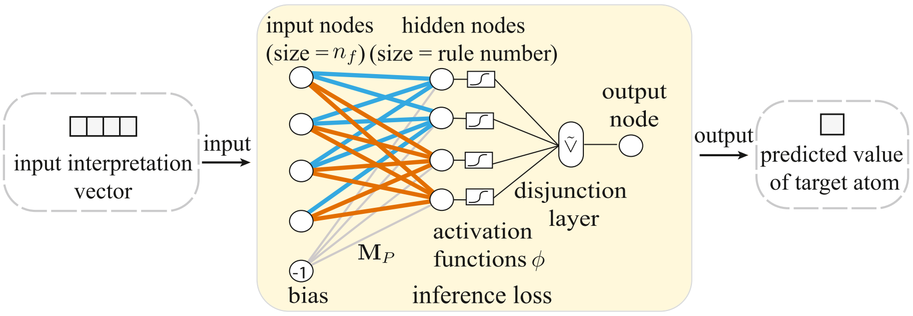


**Fig. 5.** The feedforward computation in inference loss under an instance with $m_1 = m_2 = 2$, and the number of valid body features $n_f = 4$.

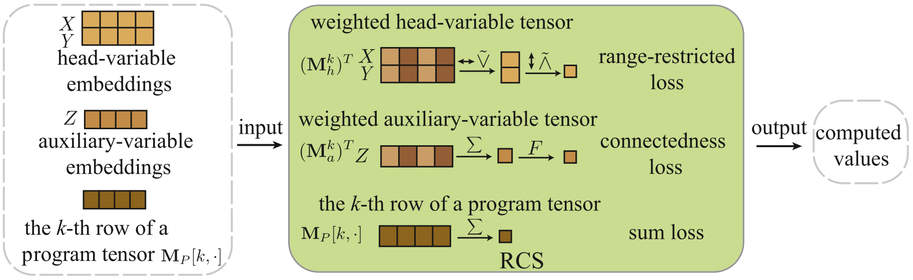

$X$
$Y$
head-variable embeddings
$Z$
auxiliary-variable embeddings
the $k$-th row of a program tensor $\mathbf{M}_P[k,\cdot]$
input
weighted head-variable tensor
$(\mathbf{M}_h^k)^T$ $X$ $Y$
$\tilde{\vee}$
$\tilde{\wedge}$
range-restricted loss
weighted auxiliary-variable tensor
$(\mathbf{M}_a^k)^T Z$
$\sum$
$F$
connectedness loss
the $k$-th row of a program tensor
$\mathbf{M}_P[k,\cdot]$
$\sum$
sum loss
RCS
output
computed values

**Fig. 6.** The operations associated with the range-restricted loss, connectedness loss, and sum loss under an instance with the number of valid body features $n_f = 4$. In addition, the variables $X$, $Y$, and $Z$ appear in a logic program but only the variables $X$ and $Y$ appear in the target atom.

loss can be represented by a feedforward neural network with an extra fuzzy disjunction layer in the yellow box. The weights in blue correspond to the same-head tensor, and the weights in orange correspond to the averaged tensor. The number of hidden nodes $m_1 + m_2$ is equal to the number of extracted rules in a training process, and the number of input nodes is equal to the number of valid body features.

To make the well-trained logic program tensor be regarded as a same-head matrix and be interpreted to a symbolic logic program, according to the same-head matrix representation method of a logic program $P$ in Section 2.3, we use the sum loss function $L_S$ such that the sum of each row in the program tensor is equal to one:

$$L_S = \sum_{k=1}^{m} \text{MSE}\left(\sum_{i}^{n_f} \mathbf{M}_P[k,i], 1\right),$$

where MSE is the mean square error loss function.

In Fig. 4, we use RCS to indicate the range-restricted loss, connectedness loss, and sum loss because all these loss functions compute on each row of the program tensor. In addition, we use Fig. 6 to explicitly present the operations associated with the sum loss based on the $k$-th row in a program tensor $\mathbf{M}_P$.

### 3.2.2. Syntactic loss functions

Now, we describe the logic program syntactic loss functions, including the range-restricted loss, connectedness loss, and two similarity loss functions. The range-restricted loss function and connectedness loss function formulate the syntactic conditions, and the syntactic conditions enforce that the generated rules meet the logic syntactic formats instantiated from pre-defined meta-rules. The two similarity loss functions ensure the diversity of rules represented by auxiliary tensor and implement the curriculum learning strategy [15].

We define the variables present in the target atom as *head variables*, and we denote the set containing these head variables as $V_h$, referred to as the head-variable set. Similarly, we define the variables present not in the target atom but in the logic program as *auxiliary variables*, and we denote the set containing auxiliary variables as $V_a$, referred to as the auxiliary-variable set.

In the paper, we consider the rules with a unary predicate in the head to be instantiated from the meta-rule (17) and consider the rules with a binary predicate in the head to be instantiated from the meta-rule (18) as follows:

$$p_t(Z_0) \leftarrow p_1(Z_0, Z_1), p_2(Z_1, Z_2), \dots, p_n(Z_{n-1}, Z_n), \tag{17}$$

$$p_t(Z_0, Z_n) \leftarrow p_1(Z_0, Z_1), p_2(Z_1, Z_2), \dots, p_n(Z_{n-1}, Z_n), \tag{18}$$

where the positions of two variables in each body atom of the meta-rules can be exchanged, the variable $p_t$ indicates the target predicate, each variable $p_i$ can represent any predicates within an input fact set $F$, and each variable $Z_i$ can be replaced with head variables or body variables. If a predicate variable $p_i$ represents the target predicate in an instantiation, then the instantiated rule is recursive. Because the positions of two variables in a body atom can be exchanged, a rule with the form $p_t(X, Y) \leftarrow p_i(Z, X), p_j(Z, Y)$, where the head predicate is an Euclidean relation, is also accepted. We refer to this type of rule as a tree-like format rule. By analyzing the occurrences and positions of variables in the bodies of the rules instantiated by the above meta-rules, we design two syntactic conditions outlined as follows:

1. Range-restricted condition: All head variables of each rule must be present in the body of that rule.
2. Connectedness condition: In the body of each rule, each auxiliary variable must occur more than once.

Now, we describe the range-restricted loss function and the connectedness loss function to implement the two syntactic conditions: Firstly, we map each valid feature $\alpha_i$ with a *head-variable embedding* denoted as $h(\alpha_i)$ and an *auxiliary-variable embedding* denoted as $a(\alpha_i)$ to indicate the occurrences of all variables in each feature. Then, based on the occurrence possibility of each valid feature in each rule of a logic program $P$ reflected by the corresponding values in each row of the program tensor $\mathbf{M}_P$, we can enforce constraints on the occurrence possibility of all variables in each rule, thereby satisfying the two syntactic conditions.

Let the arity of the predicate in the target atom be $t$ in a logic program $P$, then the variable depth is $|V| - t$, where $V$ denotes the set of all variables in $P$. For each valid feature $\alpha_i$, we define its head-variable embedding as $h(\alpha) \in \{0,1\}^t$, and if the $i$-th head variable in $V_h$ appears in a valid feature $\alpha_i$, then the $i$-th element in the head-variable embedding of the feature $\alpha_i$ is $h(\alpha)[i] = 1$; otherwise, $h(\alpha)[i] = 0$. Similarly, for each valid feature $\alpha_i$, we define its auxiliary-variable embedding as $a(\alpha) \in \{0,1\}^{|V|-t}$, and if the $i$-th auxiliary variable in $V_a$ appears in a valid feature $\alpha_i$, then the $i$-th element in the auxiliary-variable embedding of the feature $\alpha_i$ is $a(\alpha)[i] = 1$; otherwise, $a(\alpha)[i] = 0$.

Next, we define the *weighted head-variable tensor* for the $k$-th rule in a solution $P$, denoted as $\mathbf{M}_h^k$. This tensor is constructed by concatenating all head-variable embeddings of valid body features in the solution $P$, along with the occurrence possibilities of these features in the $k$-th row of the program tensor $\mathbf{M}_P$:

$$\mathbf{M}_h^k = \text{concat}\left(\mathbf{M}_P[k,1]h(\alpha_1), \mathbf{M}_P[k,2]h(\alpha_2), \dots, \mathbf{M}_P[k,n_f]h(\alpha_{n_f})\right),$$

where $\alpha_i$ is the $i$-th valid feature.

Each row in the program tensor $\mathbf{M}_P$ represents a rule, and each element in a row of $\mathbf{M}_P$ corresponds to the occurrence possibility of a feature in the rule. Hence, the weighted head-variable tensor $\mathbf{M}_h^k \in [0,1]^{n_f \times t}$ can also be regarded as the concatenation of each weighted head-variable embedding of the feature. The weights are determined by the $k$-th row of the logic tensor $\mathbf{M}_P$.

Each column in the weighted head-variable tensor $\mathbf{M}_h^k$ corresponds to a head variable. In addition, each column of $\mathbf{M}_h^k$ encapsulates all occurrence possibilities of the head variable in all valid body features of the $k$-th rule within a logic program. To calculate the likelihood of all head variables appearing simultaneously in the $k$-th rule, we utilize fuzzy conjunction operators $\tilde{\wedge}$ and fuzzy disjunction operators $\tilde{\vee}$. Similarly, to avoid a zero gradient when applying the fuzzy disjunction and fuzzy conjunction operations in the neural networks [14], the product t-norm is chosen as the semantics for the fuzzy logic. The fuzzy conjunction in product t-norm semantics is defined as $\widetilde{\bigwedge}_{i=1}^{n} x_i = \prod_{i=1}^{n} x_i$, and the fuzzy disjunction in product t-norm is defined in Eq. (10). We define the range-restricted loss function $L_R$ as follows:

$$h_k = \widetilde{\bigwedge}_{j=1}^{t} \left(\mathbf{M}_h^k[1,j] \,\tilde{\vee}\, \mathbf{M}_h^k[2,j] \,\tilde{\vee}\, \dots \,\tilde{\vee}\, \mathbf{M}_h^k[n_f,j]\right), \; L_R = \sum_{k=1}^{m} \text{MSE}\left(h_k, 1\right).$$

The fuzzy disjunction operator applied to a column of the weighted head-variable tensor $\mathbf{M}_h^k$ computes the occurrence possibility of the corresponding head variable in the $k$-th rule based on all valid body features. Furthermore, the fuzzy conjunction operator calculates the possibility that all head variables in $V_h$ occur simultaneously in the $k$-th rule.

To implement the connectedness loss function, we use the following equation to concatenate all weighted auxiliary-variable embeddings of all valid body features to *weighted auxiliary-variable tensor* $\mathbf{M}_a^k \in [0,1]^{n_f \times (|V|-t)}$:

$$\mathbf{M}_a^k = \text{concat}\left(\mathbf{M}_P[k,1]a(\alpha_1), \mathbf{M}_P[k,2]a(\alpha_2), \ldots, \mathbf{M}_P[k,n_f]a(\alpha_{n_f})\right).$$

Likewise, each column in the weighted auxiliary-variable tensor $\mathbf{M}_a^k$ corresponds to an auxiliary variable in $V_a$. Besides, each column of $\mathbf{M}_a^k$ encapsulates all occurrence possibilities of the corresponding auxiliary variable in all valid features of the $k$-th rule.

Next, we aggregate the possibility for each auxiliary variable across all valid features in the $k$-th rule into the vector $\mathbf{w}_a^k = \sum_{i=1}^{|V|-t} \mathbf{M}_a^k[\cdot, i]$. Each element in $\mathbf{w}_a^k \in [0,1]^{|V|-t}$ represents the probability of occurrence for the corresponding auxiliary variable in the $k$-th rule. Subsequently, we define a measurement function $F(x): \mathbb{R} \to [0,a]$ to limit the number of occurrences of each variable in $V_a$. To constrain the occurrences of auxiliary variables, we define the connectedness loss function $L_O$ as follows:

$$F(x) = a \cdot e^{b-c(x-d)^2},\ L_O = \sum_{k=1}^{m} F\left(\mathbf{w}_a^k\right),$$

where $a$, $b$, $c$, and $d$ are hyperparameters.[9] As $x$ approaches $d$, the value of $F(x)$ increases; conversely, as $x$ deviates from $d$, $F(x)$ tends towards 0. Therefore, when minimizing the connectedness loss, proper tuning of hyperparameters allows us to prevent each variable in $V_a$ from occurring only once in a rule.

We use an Example 7 in Appendix A to illustrate the complete computational process from a same-head tensor and an auxiliary tensor to obtain the range-restricted and connectedness loss. In addition, Fig. 6 explicitly presents the operations related to the range-restricted and connectedness loss functions. The inputs for the range-restricted and connectedness loss functions include the head-variable embeddings and auxiliary-variable embeddings of four valid body features. Each row of the transposed head-variable and auxiliary-variable embedding corresponds to a variable. Take the $k$-th row of a program tensor as input, the weighted head-variable tensor of the rule in the figure includes the occurrence possibility of the variables $X$ and $Y$ in the four valid body features of the rule. After the fuzzy disjunction operation on all valid body features for each variable, we obtain the occurrence possibilities of the variables $X$ and $Y$ in the rule, respectively. Then, the fuzzy conjunction operation obtains the possibility that variables $X$ and $Y$ occur in the rule simultaneously. The weighted auxiliary-variable tensor includes the occurrence possibility of the variable $Z$ in four valid body features. The summation function calculates the overall occurrence probability of the variable $Z$ among the four valid body features. The measurement function $F$ restricts the occurrence count of the variable $Z$ within the rule.

Then, we describe the similarity loss functions and consider the cosine similarity, $\cos(\mathbf{u},\mathbf{v}) = \frac{\mathbf{u}\cdot\mathbf{v}}{|\mathbf{u}||\mathbf{v}|}$, denoting the similarity between two vectors. When the two vectors have greater dissimilarity, their cosine similarity value is close to -1; otherwise, the cosine similarity value is close to 1. Since the parameter $n_a$ in an auxiliary tensor $\mathbf{M}_P^A$ indicates the number of rules headed by auxiliary predicates and each row in the matrix $\mathbf{M}_P^A[k,\cdot,\cdot] \in [0,1]^{n_a \times n_f}$ encodes a rule headed by an auxiliary predicate, we reduce the cosine similarity values between all pairs of rows in the matrix $\mathbf{M}_P^A[k,\cdot,\cdot]$ for generating more possible formats of rules headed by the auxiliary predicates. Then, we calculate the similarity loss function $L_D$ as follows:

$$L_D = \sum_{k=1}^{m_2} \sum_{(i_1,i_2)\in\binom{[1,n_a]}{2}} \text{MSE}\left(\cos\left(\mathbf{M}_P^A[k,i_1,\cdot], \mathbf{M}_P^A[k,i_2,\cdot]\right), -1\right),$$

where $\binom{[1,n_a]}{2}$ is the set of all 2-combinations of the integer set $[1,n_a]$.

For applying the curriculum learning: The system should consolidate what it learns in one episode, storing it as background knowledge, and reusing it in subsequent episodes [15]. We devise a strategy to implement the curriculum learning that DFORL uses explicit rules extracted in every few epochs as prior knowledge and uses the prior knowledge to learn more diverse rules compared with the prior knowledge in the following epochs. Specifically, we reduce the cosine similarity between the rows in the trainable program tensor $\mathbf{M}_P$ and an *explicit rule tensor* $\mathbf{M}_{\text{prior}} \in [0,1]^{m_p \times n_f}$. The explicit rule tensor is the concatenated tensor from the rows corresponding to the explicit rules in the trained program tensor $\mathbf{M}_P$. Hence, the parameter $m_p$ is the number of explicit rules extracted from DFORL, and the similarity loss function $L_C$ is defined as follows:

$$L_C = \sum_{(k_1,k_2)\in[1,m]\times[1,m_p]} \text{MSE}\left(\cos\left(\mathbf{M}_P[k_1,\cdot], \mathbf{M}_{\text{prior}}[k_2,\cdot]\right), -1\right).$$

We illustrate the associated tensors with the two similarity loss functions in blue boxes of Fig. 4: A similarity loss function computes over an auxiliary tensor, and another similarity loss function computes on a program tensor and an explicit rule tensor $\mathbf{M}_{\text{prior}}$. The symbolic logic program is interpreted directly from the program tensor $\mathbf{M}_P$, and a well-trained program embedding represented by the explicit rule tensor $\mathbf{M}_{\text{prior}}$ can be used as a prior knowledge input to DFORL to generate more diverse rules.

In summary, the final loss is the weighted sum of the losses in $\mathbf{L}$=[$L_I$, $L_S$, $L_R$, $L_O$, $L_D$, $L_C$], i.e., loss = $\Theta \cdot \mathbf{L}$, where $\Theta$ is a hyperparameter vector. In the learning process, we use the Adam optimization algorithm [31] to minimize the final loss.

[9] We set the hyperparameters $a$, $b$, and $d$ to 1, and set $c$ to 10 in the experiments.

### 3.3. Rule extraction

In this subsection, we describe the method of extracting the symbolic same-head logic program $P$ from a trained program tensor and the definition of the precision of a rule. In a trained program tensor $\mathbf{M}_P$, the element at the $k$-th row and $j$-th column represents the possibility of $j$-th valid body feature in the $k$-th rule $r_k$ in $P$. We use multiple thresholds called *rule filters* denoted by $\tau_f$'s on $\mathbf{M}_P$ to extract rules. Let rule filters range from 0 to 1 with step 0.05 and let $\mathsf{T}$ be the set with all rule filters. For each filter $\tau_f$, we begin by employing the conjunction operator to link all valid body features with values surpassing the rule filter $\tau_f$. Subsequently, we can extract a rule $r$ where the body of the rule $r$ is the formed conjunction, and the head of the rule $r$ is the target atom. We iteratively apply each $\tau_f$ on the well-trained program tensor $\mathbf{M}_P$. Then, we have a rule set $\tilde{R}$ with no more than $m \times |\mathsf{T}|$ rules after applying the rule filters on the $\mathbf{M}_P$, where $m$ is the number of rows in the program tensor $\mathbf{M}_P$.

Then, we describe the definition of the precision of a rule. Based on the input facts, let $n_r$ be the number of the substitutions that satisfy both the body and the head atom of a rule, and let $n_b$ be the number of the substitutions that satisfy only the body of a rule. Then, we regard the ratio $\frac{n_r}{n_b}$ as the precision of the rule $r$. A *correct rule* has a precision value of 1, and a rule is incorrect with a precision value of 0. In addition, if a precision value floats within the interval $(0,1)$, then the rule may also be correct due to the incompleteness of the training facts. Therefore, we set another threshold called *soundness filter* $\tau_s$. Rules with precision values no lower than $\tau_s$ are regarded as *explicit rules* and are added to $P$ from the set $\tilde{R}$. Again, to use the curriculum learning strategy, we concatenate the rows in a well-trained program tensor $\mathbf{M}_P$ corresponding to the explicit rules into $\mathbf{M}_{\text{prior}}$ after a few training epochs. The explicit rule tensor $\mathbf{M}_{\text{prior}}$ is considered as the prior knowledge to boost the training process in the following training epochs. When all training epochs are finished, the explicit rules are stored in $P$. We also use Example 7 in Appendix A to illustrate the rule extraction method from a given program tensor $\mathbf{M}_P$.

## 4. Experimental evaluation

In this section, we present several experimental results on classic ILP datasets and knowledge graphs to prove the qualities of rules and the ability of DFORL to make link predictions with generated rules. In DFORL, the variable depth in each task does not exceed two. We set the hyperparameters $m_1$ in the same-head tensor, $m_2$, and $n_a$ in the auxiliary tensor to four, four, and two, respectively. Besides, the hyperparameter $\gamma$ in the activation function and the learning rate in most tasks are set to 10 and 0.001, respectively. We reduce the values of $\gamma$ and the learning rate if the gradients explode. We use 10% data generated after the propositionalization process as the validation set, and we use the rest of the data to train the neural network in DFORL. When comparing with baselines, the same training facts and test facts are used for all models, and the values of hyperparameters for baselines are chosen according to their settings. All experiments were executed over the 24 GB memory and an 8-core Intel i7-6700 CPU.

We present some selected generated symbolic logic programs from classical ILP tasks and knowledge graphs in Appendix B and the complete solutions for all tasks are present in Supplementary. Each rule is presented with the value of precision $\frac{n_r}{n_b}$, the number of substitutions satisfying both the body and head of the rule $n_r$ in the training facts, and the number of substitutions satisfying only the body of the rule $n_b$ in the training facts, i.e., $(\frac{n_r}{n_b}, n_r, n_b)$. The precision of a logic program is the average of the precisions of all the rules in the logic program. Additionally, the accuracy of the logic program $P$ is determined by the ratio of testing positive examples generated from learned solutions based on all facts in a dataset and the logic program $P$ to all testing positive examples. This accuracy metric also represents the recall value of $P$ within the testing dataset.

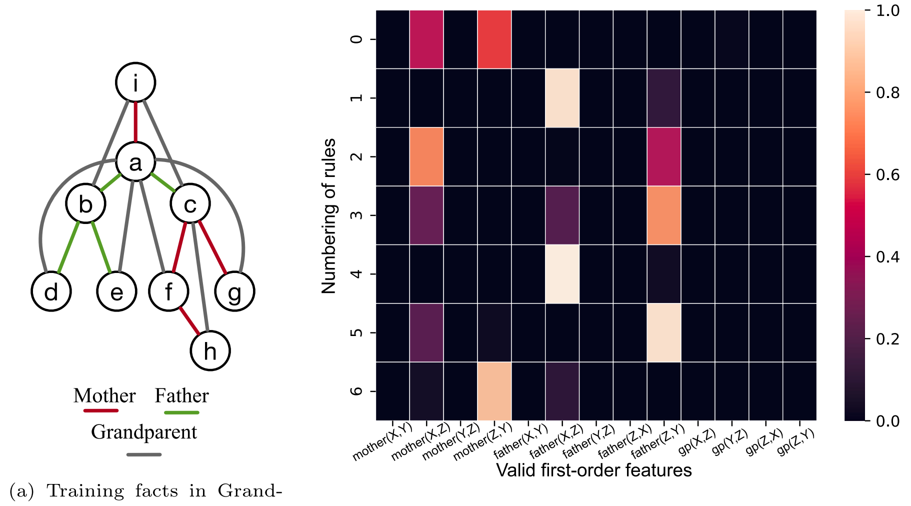
### 4.1. Learning from ILP datasets

In this subsection, we present the results of DFORL on 22 tasks proposed by [14], which include 20 smaller ILP datasets and two tasks called Husband and Uncle tasks on an incomplete large family relation knowledge graph [60]. The family relation knowledge graph contains 3010 constants and 12 relations. The complete descriptions for the classical ILP datasets are presented in Appendix B and Supplementary. We present the performance of DFORL and make comparisons with NTP$\lambda$ [45], $\partial$ILP [14], and NeuralLP [61] as the symbolic rules are also generated in the implementations of these models.

For the Husband and Uncle tasks, we use the sampling algorithm and set the reduced ratio $\delta = 1$. Let the soundness filter be $\tau_s = 0.3$ because of the incompleteness of these two tasks. The accuracies of generated logic programs on the Husband and Uncle tasks are 95.90% and 99.50%, respectively. In addition, the precisions of logic programs from Husband and Uncle tasks are 66.50% and 59.00%, respectively. For the rest of the ILP datasets, we set the soundness filter $\tau_s = 1$. As a result, the precision and the accuracy of each generated logic program in the rest of the ILP tasks are both 100%. The maximum running times of DFORL on the above classic ILP datasets are smaller than five minutes in our setting. Hence, DFORL learns rules in a fast manner.

We use the *grandparent* predicate as an example to show the results. The background facts include the facts with *mother* and *father* predicates, the positive examples atoms include the facts with *grandparent* predicate, and the negative examples are all ground atoms involving the predicate *grandparent* except those in the positive set. The training facts are presented in Fig. 7a. The learned program tensor $\mathbf{M}_P$ is presented in Fig. 7b.[10] Based on the well-trained program tensor $\mathbf{M}_P$, the extracted logic program $P$ is the same as the logic program (11), which obtains 100% accuracy and precision.

[10] The *grandparent* predicate is denoted by *gp* in the figure.



(a) Training facts in Grandparent genealogy dataset.

(b) The learned program tensor $\mathbf{M}_P$.

**Fig. 7.** The training facts and the learned program tensor $\mathbf{M}_P$ in the Grandparent task.

**Table 2**
The results about accuracy on ILP datasets. The symbols ✓, $*$, and $-$ indicate that the accuracy of the generated logic program is equal to 100%, less than 100%, and equal to 0%, respectively.

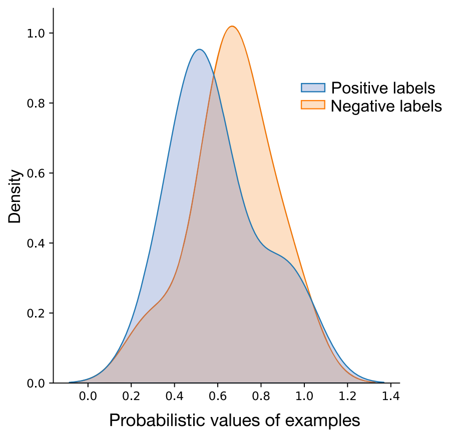
| Domain | Task | ∂ILP | NeuralLP | DFORL |
|---|---|---|---|---|
| Arithmetic | Predecessor | ✓ | ✓ | ✓ |
| | Odd | ✓ | – | ✓ |
| | Even / Succ2 (10) | ✓ | – | ✓ |
| | Even / Succ2 (20) | – | – | ✓ |
| | Lessthan | ✓ | ✓ | ✓ |
| | Fizz | ✓ | – | ✓ |
| | Buzz | ✓ | – | ✓ |
| Lists | Member | ✓ | * | ✓ |
| | Length | ✓ | – | ✓ |
| Family Tree | Son | ✓ | * | ✓ |
| | Grandparent | ✓ | ✓ | ✓ |
| | Husband | * | – | * |
| | Uncle | * | – | * |
| | Relatedness | ✓ | * | ✓ |
| | Father | ✓ | – | ✓ |
| Graphs | Directed Edge | ✓ | * | ✓ |
| | Adjacent to Red | ✓ | – | ✓ |
| | Two Children | ✓ | – | ✓ |
| | Graph Coloring (6) | ✓ | – | ✓ |
| | Graph Coloring (10) | – | – | ✓ |
| | Connectedness | ✓ | * | ✓ |
| | Cyclic | ✓ | – | ✓ |

Based on the comparison of accuracy in Table 2, DFORL completes all 22 benchmarks and outperforms NeuralLP in 19 of all benchmarks. Since ∂ILP is a memory-expensive solution to ILP [14], ∂ILP cannot generate logic programs in the case of a large number of constants, e.g., Even (20) task, Graph Coloring (10) task, and the considered knowledge graph datasets in the paper. However, DFORL is scalable because of the small-size trainable weights and the effective differentiable immediate consequence operator of logic programs. Besides, ∂ILP requires a logical template, which is a more specific prior knowledge than the syntactic conditions considered in DFORL.

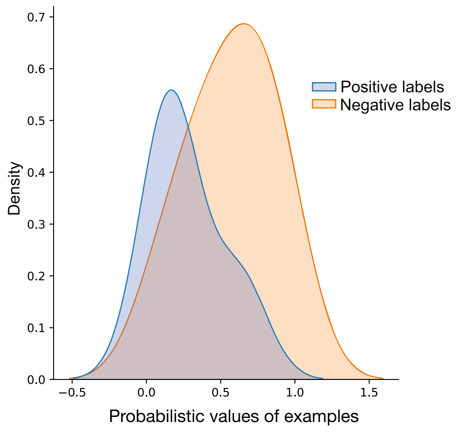
## *4.2. Learning from noisy datasets*

In this subsection, we test the robustness of DFORL and consider two cases: (1) learning from probabilistic facts; (2) learning from mislabeled facts. In addition, we set the soundness filter $\tau_s$ to 1 for obtaining rules with precision values of 100%.

For the first case, each ground atom in the training fact set $F$ and training negative example set $\mathcal{N}$ has a probability, where the training fact set includes training background facts and training positive examples. Let $\epsilon \sim N(0, \sigma^2)$, then the probabilities of facts in $F$ are $p_i^+ = max(min(1-\epsilon, 1), 0)$, and the probabilities of ground atoms in $\mathcal{N}$ are $p_i^- = min(max(\epsilon, 0), 1)$. The standard deviation $\sigma$



(a) Lessthan task.



(b) Connectedness task.

**Fig. 8.** Distribution of examples in the Lessthan task when $\sigma = 3$ and in the Connectedness task when $\sigma = 2$.

**Table 3**
The results on noisy datasets. The notations Con and DE represent Connectedness and Directed Edge tasks, respectively.

| | Lessthan | Predecessor | Member | Son | Con | DE |
|---|---|---|---|---|---|---|
| $\sigma$ | 3 | 3 | 3 | 3 | 2 | 3 |
| $\mu$ | 0.95 | 0.95 | 0.90 | 0.95 | 0.95 | 0.95 |

ranges from 0.5 to 3 with step 0.5. We present the distribution of the positive and negative examples on the tasks Lessthan with $\sigma = 3$ and Connectedness with $\sigma = 2$ in Figs. 8a and 8b, respectively. From the two distributions, we derive that the values of examples are sufficiently noisy in the corresponding settings. For the second case, both positive and negative training examples are mislabeled with the mutation rate $\mu$ ranging from 0.05 to 1 with step 0.05. In Table 3, we show the maximum standard deviation $\sigma$ and mutation rate $\mu$ when the accuracy values of the generated logic programs are 100%. From the results, we conclude that DFORL can handle noisy data very well, and this is because the neural network in DFORL can search for the optimal parameters in a robust manner. In addition, when learning from mislabeling data and $\mu = 1$, the semantics of the head predicate in Lessthan task is *largethan*, and the generated symbolic logic program from DFORL is:

$$lessthan(X,Y) \leftarrow succ(Y,X).$$
$$lessthan(X,Y) \leftarrow lessthan(X,Z) \wedge lessthan(Z,Y).$$

## *4.3. Learning from knowledge graphs*

In this subsection, we present the qualities of logic programs generated from DFORL on knowledge graphs. The descriptions of all knowledge graphs in this subsection are presented in Table 4. For each knowledge graph, the training data and testing data are split based on the configurations outlined in the original work. We first present the comparison of accuracy between DFORL and neural network-based rule learners including NTP$\lambda$ and NeuralLP on the Countries dataset [4], Unified Medical Language System (UMLS) [32], Nations [32], and Kinship datasets [32].

Then, we present the comparison of accuracy between DFORL and bottom clause propositionalization-based methods including CILP++ and D-LFIT on UW-CSE [9] and Alzheimers-amine datasets [30]. Bottom clause propositionalization also uses the most specific clause to be a solution for each positive example in input facts $F$ [18]. CILP++ and D-LFIT both regard body features in bottom clauses as the input node in neural networks, and a more general solution is extracted from the neural networks after the training process.

In addition, to prove that DFORL can not only interpret the data from the generated rules but also make link predictions based on learned rules. In link prediction tasks, we need to infer the likelihood of a predicate existing between two constants in a knowledge graph [55]. We compare the mean reciprocal rank (MRR) [61] and HITS@m [3] of DFORL on link prediction tasks with the embedding-based models including TransE and RotatE, and the symbolic rule learners including NeuralLP, DRUM, and RNNLogic on the datasets WN18 and WN18RR from WordNet [16,37], and FB15KSelected from Freebase [2]. Embedding-based models usually embed constants and relations in a dataset into uninterpretable vector space. By minimizing the space distance, the embedding-based models encode the logical relations inside and make link prediction tasks. When calculating the accuracy, MRR, and HITS@m on the test facts, we use all facts in the task and the generated logic programs to make derivations. When computing MRR and HITS@m, we

**Table 4**
Dataset statistics of Countries, Nations, UMLS, Kinship, UW-CSE, Alzheimers-amine, WN18, WN18RR, and FB15KSelected.

| Dataset | #Constant | #Relation | #Training Fact | #Test Fact |
|---|---|---|---|---|
| Countries-S1/S2/S3 | 252 | 2 | 1,110/1,063/979 | 24 |
| Nations | 14 | 56 | 2,364 | 201 |
| UMLS | 135 | 49 | 5,598 | 661 |
| Kinship | 104 | 26 | 9,612 | 1,074 |
| UW-CSE | 1,209 | 15 | 2,675 | 113 |
| Alzheimers-amine | 147 | 32 | 980 | 343 |
| WN18 | 40,943 | 18 | 146,442 | 5,000 |
| WN18RR | 40,943 | 11 | 89,869 | 3,134 |
| FB15KSelected | 14,541 | 237 | 289,650 | 20,466 |

**Table 5**
The results of MRR, HITS@m (in %), and accuracy (in %) on Countries, Nations, UMLS, and Kinship. The results in bold indicate the best result. The notation − indicates the accuracy is 0%. The ACC@$h$ represents the accuracy of the logic program with the head predicate $h$. Besides, *blo*, *int*, *neg*, and *intw* denote the relations of *blockpositionindex*, *intergovorgs3*, *negativecomm*, and *interacts_with*, respectively.

| Dataset | Metrics | NTP$\lambda$ | NeuralLP | DFORL |
|---|---|---|---|---|
| Countries | ACC@S1 | **100.00** | **100.00** | **100.00** |
| | ACC@S2 | **100.00** | **100.00** | **100.00** |
| | ACC@S3 | **100.00** | − | **100.00** |
| Nations | MRR | 0.418 | 0.565 | **0.789** |
| | HITS@1 | 41.79 | 52.49 | **73.88** |
| | HITS@3 | 41.79 | 60.95 | **84.58** |
| | HITS@10 | 41.79 | 61.19 | **85.07** |
| | ACC@*blo* | **100.00** | 50.00 | **100.00** |
| | ACC@*int* | **84.62** | **84.62** | **84.62** |
| | ACC@*neg* | 37.50 | **75.00** | **75.00** |
| UMLS | MRR | 0.301 | 0.667 | **0.750** |
| | HITS@1 | 29.95 | 61.27 | **71.41** |
| | HITS@3 | 30.11 | 72.31 | **78.82** |
| | HITS@10 | 30.11 | 72.31 | **78.97** |
| | ACC@*isa* | 65.96 | 63.83 | **91.48** |
| | ACC@*intw* | 83.67 | 86.67 | **100.00** |
| Kinship | MRR | 0.544 | 0.401 | **0.876** |
| | HITS@1 | 52.51 | 40.13 | **83.15** |
| | HITS@3 | 56.33 | 40.13 | **92.27** |
| | HITS@10 | 56.33 | 40.13 | **92.36** |
| | ACC@*term15* | 91.21 | 83.52 | **95.86** |
| | ACC@*term16* | 92.37 | 88.55 | **96.95** |

need to generate corrupted atoms $\mathcal{N}' = \{p(o_1', o_2) \mid o_1' \in E\} \cup \{p(o_1, o_2') \mid o_2' \in E\} - F$ according to each test fact $p(o_1, o_2)$ and compute the confidence ranks of the test facts among the corrupted atoms and test atoms, where $E$ is the constant set and $F$ is all facts in a task. The confidence of the inferred atom is equal to the precision of the rule that derives it, and the confidence of an atom is 0 if no rule can derive it based on all facts.

For the Countries dataset, training facts are differently selected from all facts into S1, S2, and S3 sub-datasets, and the learning difficulty is increasing from S1 to S3 because the related relations are missing corresponding to the test cities [45]. The soundness filter is $\tau_s = 0.3$ because of the data incompleteness in these four datasets during extracting rules. For the S3 sub-datasets, the minimal variable depth is 2 in the solution. Hence, we set $\xi = 1\%$ in the propositionalization method to get the solution for a shorter running time. The precisions of logic programs from DFORL over the sub-datasets S1, S2, and S3 of the Countries dataset are 71.25%, 81.86%, and 42.33%, respectively. In addition, the average precisions of all logic programs headed by all predicates in Nations, UMLS, and Kinship are 80.13%, 76.54%, and 57.49%, respectively. The comparison results of accuracy, MRR and HITS@m based on the rules generated from DFORL, NTP$\lambda$, and NeuralLP are presented in Table 5. Based on the comparison results of MRR and HITS@m, we conclude that logic programs generated from DFORL have better performance on link prediction tasks than NTP$\lambda$ and NeuralLP in general.

We present the total running time of the models NTP$\lambda$, NeuralLP, and DFORL on Countries, Nations, UMLS, and Kinship for generating all logic programs with all predicates appearing in these datasets in Table 6. In addition, the average running times of DFORL to generate a logic program with a target atom on Nations, UMLS, and Kinship are 0.1, 0.1, and 16.0 minutes, respectively. Although DFORL is not as fast as NeuralLP, DFORL promises accuracy within an acceptable time limit.

When comparing with the bottom clause propositionalization-based models including CILP++ and D-LFIT on UW-CSE and Alzheimers-amine datasets, the soundness filter $\tau_s$ is set to 0.2. The sampling algorithm is used on the UW-CSE dataset and the

**Table 6**
The running time in minutes of NTP$\lambda$, NeuralLP, and DFORL when generating all logic programs on Countries, Nations, UMLS, and Kinship. The results in bold indicate the best result.

| Dataset | NTP$\lambda$ | NeuralLP | DFORL |
|---|---|---|---|
| Countries-S1 | 30.1 | **0.1** | 11 |
| Countries-S2 | 43.2 | **0.1** | 8 |
| Countries-S3 | 252.4 | **0.1** | 88.4 |
| Nations | 600.4 | **0.4** | 0.6 |
| UMLS | 150.4 | **0.6** | 5.2 |
| Kinship | 320 | **2.3** | 417 |

**Table 7**
The accuracy results on the UW-CSE and Alzheimer-amine datasets. The results of baselines are taken from [18,21].

| Model | CILP++ | D-LFIT | DFORL |
|---|---|---|---|
| UW-CSE | **81.98** | 79.44 | 46.90 |
| Alzheimers-amine | 78.70 | 67.75 | **99.13** |

**Table 8**
The precision (in %) and accuracy (in %) of the generated logic programs from NeuralLP and DFORL on WN18 and WN18RR.

| Dataset | Metrics | NeuralLP | DFORL |
|---|---|---|---|
| WN18 | Precision | 0.08 | **97.89** |
| | Accuracy | 0.30 | **99.56** |
| WN18RR | Precision | 1.66 | **26.27** |
| | Accuracy | 1.53 | **40.91** |

**Table 9**
The MRR and HITS@m (abbreviated as H@m, in %) results on the WN18 and WN18RR datasets. The results of baselines are taken from [55,61,46,44].

| Model | WN18 | | | | WN18RR | | | |
|---|---|---|---|---|---|---|---|---|
| | MRR | H@1 | H@3 | H@10 | MRR | H@1 | H@3 | H@10 |
| TransE | 0.495 | 11.30 | 88.80 | 94.30 | 0.226 | - | - | 50.1 |
| RotatE | 0.949 | 94.40 | 95.20 | 95.90 | 0.476 | 42.80 | 49.20 | **57.10** |
| NeuralLP | 0.940 | - | - | 94.49 | 0.435 | 37.10 | 43.40 | 56.60 |
| DRUM | - | - | - | 95.21 | 0.382 | 36.90 | 38.30 | 41.00 |
| RNNLogic | - | - | - | - | **0.483** | **44.60** | **49.70** | 55.80 |
| DFORL | **0.995** | **99.56** | **99.56** | **99.56** | 0.404 | 40.00 | 40.76 | 40.91 |

**Table 10**
The MRR and HITS@m (in %) results on the FB15KSelected dataset. The results of baselines are taken from [55,61,46,44].

| Model | FB15KSelected | | | |
|---|---|---|---|---|
| | MRR | HITS@1 | HITS@3 | HITS@10 |
| TransE | 0.294 | - | - | 46.50 |
| RotatE | 0.338 | 24.10 | 37.50 | 53.30 |
| NeuralLP | 0.240 | - | - | 36.20 |
| DRUM | 0.238 | 17.40 | 26.10 | 36.40 |
| RNNLogic | 0.344 | **25.20** | 38.00 | 53.00 |
| DFORL | **0.370** | 23.54 | **47.41** | **62.78** |

reduced ratio $\delta$ is set to 0.3. The precisions of logic programs generated by DFORL on UW-CSE and Alzheimer-amine datasets are 71.27% and 57.02%, respectively. The accuracy of the test facts is present in Table 7. In the UW-CSE dataset, many constants only appear under one predicate, and the perfect solution to describe the UW-CSE dataset should support constants as terms. However, the terms in the generated logic programs from DFORL only include variables. Hence, the generated logic programs from DFORL have lower accuracy compared with the bottom clause propositionalization-based baselines on the UW-CSE dataset. However, the generated rules presented in Appendix B on the UW-CSE dataset are still reliable with high precision values. In the Alzheimers-amine dataset, we obtain the best accuracy compared with the baselines.

On the WN18, WN18RR, and FB15KSelected datasets with large numbers of constants, the sampling algorithm is used and the reduced ratio $\delta$ is set to 0.01. The soundness filter is set as $\tau_s = 0.6$ in WN18 dataset, and $\tau_s = 0.01$ in WN18RR and FB15KSelected datasets. The comparison results on the precisions and accuracies of the generated solutions between DFORL and NeuralLP are presented in Table 8, and we conclude that the logic programs generated by DFORL have both higher precision and accuracy results on WN18 and WN18RR datasets than NeuralLP. NeuralLP fails in rule extraction when the number of relations and constants in a knowledge graph is quite large. In DFORL, the number of trainable parameters is related to the number of valid body features. Hence, DFORL can learn solutions from datasets with a large number of relations or constants. Under the experiment setting, the average running times of DFORL to generate a logic program with a predicate on WN18, WN18RR, and FB15KSelected are 4.3, 4.1, and 5.2 minutes, respectively. Now, we present the MRR and HITS@m results with the link prediction models in Tables 9 and 10. According to the comparison, we find DFORL makes the best MRR and HITS@m values on the WN18 datasets, and the values of MRR and HITS@m of DFORL on the WN18RR dataset are comparable with other models. Besides, DFORL outperforms other baseline models on the FB15KSelected dataset except for the metric of HITS@1. Then, DFORL promises interpretability compared with the embedding-based models TransE and RotatE. On the FB15KSelected dataset, we present two logic programs in Appendix B with accuracies of 100% and 90.16% under the relations *category_of* and *place_lived_location*, respectively. We can interpret the data and enumerate some knowledge from the logic program headed by *place_lived_location* as follows: Firstly, from the second rule with the highest precision, we know that the city where a person lives may depend on the place of birth of that person. Then, the city in which an artist lives is also related to the artist's place of origin. Furthermore, we infer that when a person lives in a city that is contained in a country, then that person lives in that country.

In conclusion, DFORL still gets high-quality rules when the knowledge graphs are super large. Besides, learning symbolic logic programs with DFORL on large knowledge graphs is still fast and computationally cheap.

## 5. Related work

Neuro-symbolic models have been focused on over the past decades. For learning propositional logic programs, d'Avila Garcez et al. [22] and Lehmann et al. [35] designed algorithms to extract propositional logic programs from neural networks. In contrast, compared with first-order logic programs, propositional logic programs have less ability to describe relational facts. Phua and Inoue [43], Gentet et al. [24], and Gao et al. [21] make LFIT differentiable, which learns symbolic proposition rules from pairs of interpretations proposed by Inoue et al. [27].

For learning first-order logic programs, D-LFIT can also incorporate bottom clause propositionalization [18] to learn first-order logic programs from relational facts. However, the scalability of D-LFIT severely relies on the performance of the bottom clause propositionalization algorithm. Compared with D-LFIT, we proposed a novel propositionalization method to incorporate the proposed neural network in DFORL to learn from relational facts directly. Besides, the neural network in DFORL supports predicate invention and syntactic conditions, and the quality of rules is analyzed through precision values to measure the performance of DFORL. There are other related works learning first-order logic programs with neural networks, including [14,15,45,50,53]. In these models, the explicit logic programs are learned based on the given templates. In [45,50], the logic templates include the order of variables and the number of predicates in the body of the rule. The models need to fill in the proper predicates in the templates. In [14,15,53], the models need to learn the weights of the given rule templates or select the best rules from the templates. In contrast, we construct logic programs without any explicit templates but follow more flexible syntactic conditions. In most works, such as [61,44,46], the generated rules are strictly followed by forward-chained format. In DFORL, the syntactic conditions implement that a generated rule can be in the forward-chained format, reverse format, or other formats such as tree-like format. Thus, DFORL can not only find rules without strong language bias but also learn rules in diverse structures. Besides, Shindo et al. [52] used a refinement clause generator to replace the logic templates. The inference process in their method is similar to $\partial$ILP [14], which involves several differentiable functions. Similar to $\partial$ILP, the trainable weights are assigned to all possible rules. In contrast, we use matrices to represent logic programs, and the trainable weights are assigned to body features in rules. The differentiable immediate consequence operator of logic programs implemented by the neural network in DFORL can perform the deduction process quickly and precisely. Hence, DFORL has more scalability than [52] and $\partial$ILP.

From the perspective of the representations of logic programs, Qu et al. [44] regarded logic programs as latent variables. They developed an EM-based algorithm for extracting logic programs through a rule generator and a reasoning predictor. Besides, Kaur et al. [29] regarded a rule as a lifted random walk. In contrast to them, DFORL uses small matrix embeddings to encode logic programs. Hence, with the help of the differentiable immediate consequence operator of logic programs, neural networks can be adopted in DFORL to search matrix representations of logic programs quickly and robustly. When adopting neural networks to get logic programs, Yang et al. [61] and Sadeghian et al. [46] used matrices to represent knowledge graphs and differentiable operations on the training facts to represent deductive reasoning. The confidence of each atom in a rule is represented by the parameter inside the differentiable operations. Besides, these confidence values are extracted with an auxiliary algorithm. In contrast, we give a more straightforward representation of logic programs through matrices, which is more interpretable. Besides, we depend on the differentiable immediate consequence operator of logic programs to perform the deduction process. Therefore, the trainable parameters are much smaller and there is no complex algorithm in DFORL to transfer subsymbolic matrices into symbolic logic programs. Moreover, the learning process is also faster and computationally cheaper. CILP++ proposed by França et al. [18] uses the bottom clauses propositionalization method and three-layer neural networks to imitate the immediate consequence operator of logic programs. In contrast, CILP++ cannot be interpreted into symbolic logic programs directly. Then, França et al. [19] proposed a first-order rule extraction method from the well-trained CILP++ based on the decision tree and information gain-based heuristic method. The neural network in CILP++ is regarded as an oracle, and the data examples are used during the rule extraction process. In DFORL, the learned rules are extracted from the weights of the neural network without any data examples. Besides, we do not rely on any heuristic methods but extract first-order rules over the connection between logic programs and their same-head embeddings. Hence, the neural network in DFORL is more interpretable.

Excepting these induction tasks, d'Avila Garcez and Zaverucha [23] and Serafini and d'Avila Garcez [51] considered the deduction tasks with neural networks based on the pre-defined logic programs. Aspis et al. [1] and Takemura and Inoue [56] regarded logic programs as matrices based on matrix representation of logic programs [47,48], and differentiably computed the stable and supported models of propositional logic programs in continuous vector spaces.

In addition, Teru et al. [57] and Hohenecker used graph neural networks to perform the reasoning tasks on graphs and the relation prediction tasks. Bordes et al. [3], Sun et al. [55], and Hohenecker and Lukasiewicz [26] perform prediction tasks by measuring the similarity of embeddings associated with constants within the knowledge graphs. In contrast, these graph neural network-based methods and embedding-based methods do not generate explicit logic programs when performing relation prediction tasks.

## 6. Conclusion

In this paper, we proposed a flexible differentiable first-order rule learner (DFORL), which learns first-order logic programs from relational facts without logic templates. Through the novel sampling method, DFORL is scalable to learn logic programs from large knowledge graph datasets. Then, the proposed propositionalization method in DFORL translates relational facts to neural network-readable data, then we can use neural networks to learn rules robustly. By applying the proposed logic program semantics and syntactic loss functions, DFORL can learn the target logic programs with a few trainable parameters in neural networks. We extract symbolic logic programs from the trainable matrices directly and quickly. In addition, we can also apply prior knowledge as constraints to implement curriculum learning. Experimental results indicate that DFORL can learn first-order logic programs with both high accuracy and precision from several standard inductive logic programming tasks, probabilistic relational facts, and knowledge graphs. Hence, DFORL can learn rules from relational facts in a flexible, precise, robust, scalable, and computationally cheap manner.

In the future, we will extend DFORL to explainable deep neural networks to generate logic programs with larger variable depths. In addition, we will support DFORL to learn logic programs with larger arities of atoms by improving the propositionalization method and extending the structures of neural networks. We also plan to explore more efficient models to support the constant terms and the terms with functions. Furthermore, we will support negations in the bodies of rules and rule induction tasks over non-symbolic inputs.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Data availability

Code can be found here: https://github.com/gaokun12/DFORL

classic ILP datasets, Countries dataset, Unified Medical Language System (UMLS), Nations, Kinship, UW-CSE, Alzheimers-amine, WN18, WN18RR, and FB15KSelected. (Reference Data) (Github)

## Acknowledgements

We express our gratitude to the Editor, Associate Editor, and anonymous Reviewers for their valuable feedback, which has contributed significantly to enhancing the clarity and presentation of our paper.

This work was supported by the National Key R&D Program of China under Grant 2021YFF1201102 and the National Natural Science Foundation of China under Grants 61972005, 62172016, and 61932001. This work has also been supported by the NII international internship program, JSPS KAKENHI Grant Number JP21H04905 and JST CREST Grant Number JPMJCR22D3, Japan.

## Appendix A. A complete computational process in DFORL

In this section, we use Example 7 to illustrate the merge operation, concatenate operation, head-variable embedding, auxiliary-variable embedding, and the differentiable operations related to the head-variable and auxiliary-variable embeddings. In this example, we also instance two rule filters and a soundness filter to extract a symbolic logic program $P$ from a program tensor $\mathbf{M}_P$.

**Example 7.** Let target atom be $p(X, Y)$, variable set be $\{X, Y, Z\}$, and valid body feature set $P_v$ be $\{p(X,Z), p(Y,X), p(Y,Z), p(Z,X), p(Z,Y)\}$. Thus, we have $V_h = \{X,Y\}$ and $V_a = \{Z\}$. Assume that an auxiliary tensor $\mathbf{M}_P^A \in [0,1]^{1\times 2\times 5}$ and a same-head tensor $\mathbf{M}_P^S \in [0,1]^{1\times 5}$ are:

$$\mathbf{M}_P^A[1,\cdot,\cdot] = \begin{bmatrix} 0.01 & 0.90 & 0 & 0.04 & 0 \\ 0.05 & 0.80 & 0.20 & 0 & 0 \end{bmatrix},$$
$$\mathbf{M}_P^S = \begin{bmatrix} 0 & 0.95 & 0 & 0.03 & 0.02 \end{bmatrix}.$$

Then, the averaged tensor $\mathbf{M}_P^{A'} \in [0,1]^{1\times 5}$ and the program tensor $\mathbf{M}_P \in [0,1]^{2\times 5}$ are presented as follows:

$$\mathbf{M}_P^{A'} = \begin{bmatrix} 0.03 & 0.85 & 0.10 & 0.02 & 0 \end{bmatrix},$$
$$\mathbf{M}_P = \begin{bmatrix} 0 & 0.95 & 0 & 0.03 & 0.02 \\ 0.03 & 0.85 & 0.10 & 0.02 & 0 \end{bmatrix}.$$

In addition, the head-variable embeddings of all the body features are: $h(P_v) = \{[1,0], [1,1], [0,1], [1,0], [0,1]\}$, and the auxiliary-variable embeddings for all the body features are: $a(P_v)= \{[1],[0],[1],[1],[1]\}$. Then, the weighted head-variable tensors $\mathbf{M}_h^1$ and $\mathbf{M}_h^2$ are:

$$\mathbf{M}_h^1 = \begin{bmatrix} 0 & 0.95 & 0 & 0.03 & 0 \\ 0 & 0.95 & 0 & 0 & 0.02 \end{bmatrix}^T,$$
$$\mathbf{M}_h^2 = \begin{bmatrix} 0.03 & 0.85 & 0 & 0.02 & 0 \\ 0 & 0.85 & 0.10 & 0 & 0 \end{bmatrix}^T.$$

Hence, we get the possibility of the variables $X$ and $Y$ occur simultaneously in the two rules are: $\mathrm{h}_1 = (1 - 0.05 \times 0.97) \times (1 - 0.05 \times 0.98) = 0.90$ and $\mathrm{h}_2 = (1 - 0.97 \times 0.15 \times 0.98) \times (1 - 0.15 \times 0.90) = 0.74$. In addition, the weighted auxiliary-variable tensors $\mathbf{M}_a^1$ and $\mathbf{M}_a^2$ are:

$$\mathbf{M}_a^1 = \begin{bmatrix} 0 & 0 & 0 & 0.03 & 0.02 \end{bmatrix}^T,$$
$$\mathbf{M}_a^2 = \begin{bmatrix} 0.03 & 0 & 0.10 & 0.02 & 0 \end{bmatrix}^T.$$

When the hyperparameters $a = b = d = 1$, and $c = 10$, then the connectedness loss function $L_O = (e^{1-10\times(0.05-1)^2}) + (e^{1-10\times(0.15-1)^2}) = (2.31 \times 10^{-3})$.

Based on the program tensor $\mathbf{M}_P$, we can extract the following rules when the rule filters $\tau_f = 0.02$:

$$p(X,Y) \leftarrow p(Y,X), p(Z,X). \tag{A.1}$$

$$p(X,Y) \leftarrow p(Y,X), p(X,Z), p(Y,Z). \tag{A.2}$$

Furthermore, when the rule filter $\tau_f = 0.5$, we obtain the following rule:

$$p(X,Y) \leftarrow p(Y,X). \tag{A.3}$$

Then, assume that the precision values for the rules (A.1), (A.2), and (A.3) are 0.4, 0.04, and 0.8, respectively. When the soundness filter is set as $\tau_s = 0.6$, then the output solution $P$ of DFORL is:

$$p(X,Y) \leftarrow p(Y,X).$$

## Appendix B. Learned logic programs

In this section, we present several selected generated logic programs from DFORL. For the classical inductive logic programming tasks, we also present the problem descriptions in the training data. In addition, the complete generated logic programs can be checked in Supplementary.

1. **The Odd task:** The constant set is:

   $$E = \{0, 1, \dots, 8\}$$

   The background fact set is:

   $$\mathcal{B} = \{succ(X, X+1) \mid X \in E\} \cup zero(0)$$

   The positive example set is:

   $$\mathcal{P} = \{odd(X) \mid X \bmod 2 = 1, X \in E \cup \{9\}\}$$

   The negative example set is:

   $$\mathcal{N} = \{odd(X) \mid X \bmod 2 = 0, X \in E \cup \{9\}\}$$

For the testing data, we use positive and negative examples of the *odd* predicate on numbers greater than 9. The result is shown as follows:

1) $\mathrm{odd(X) \leftarrow zero(Y), succ(Y, X).}$
2) $\mathrm{odd(X) \leftarrow succ(Z, X), succ(Z, Y), odd(Y).}$
3) $\mathrm{odd(X) \leftarrow succ(X, Z), succ(Z, Y), odd(Y).}$

Because DFORL can learn tree-like format rules, rule (2) is also correct and it reflects the fact that the number represented by the variables $X$ is equal to the number represented by the variable $Y$.

2. **The Lessthan (*lt*) task:** The constant set is:

   $$E = \{0, 1, \dots, 9\}$$

   The background fact set is:

   $$\mathcal{B} = \{succ(X, X+1) \mid X \in E - \{9\}\} \cup zero(0)$$

   The positive example set is:

   $$\mathcal{P} = \{lt(X, Y) \mid X < Y, X \in E, Y \in E\}$$

   The negative example set is:

   $$\mathcal{N} = \{lt(X, Y) \mid X \geq Y, X \in E, Y \in E\}$$

   For the testing data, we use positive and negative examples of the *lt* predicate on numbers greater than 9. The result is shown as follows:

   1) lt(X, Y) ← succ(X, Y).
   2) lt(X, Y) ← lt(X, Z), lt(Z, Y).

3. **The Alzheimers-amine task:** The logic program in the Alzheimers-amine knowledge graph with the target atom $great\_ne(X, Y)$ is shown as follows:

   1) great_ne(X, Y) ← r_subst_1(X, Z), r_subst_1(Y, Z). #(0.32, 195, 607)
   2) great_ne(X, Y) ← great_ne(X, Z), great_ne(Z, Y). #(1.0, 2733, 2733)
   3) great_ne(X, Y) ← alk_groups(X, Z), alk_groups(Y, Z), ring_substitutions(Y, Z). #(0.38, 150, 391)
   4) great_ne(X, Y) ← great_ne(Z, X), great_ne(Z, Y). #(0.47, 2733, 5841)
   5) great_ne(X, Y) ← r_subst_2(X, Z), r_subst_2(Y, Z). #(0.35, 190, 540)
   6) great_ne(X, Y) ← x_subst(X, Z), ring_subst_2(Y, Z). #(0.47, 7, 15)

4. **The UW-CSE task:** The logic program in the UW-CSE knowledge graph with the target atom $advisedby(X, Y)$ is shown as follows:

1) advisedby(X, Y) ← sameperson(Y, Y), student(X), publication(Z, X), publication(Z, Y). #(0.5, 42, 84)
2) advisedby(X, Y) ← student(X), publication(Z, X), publication(Z, Y). #(0.5, 42, 84)
3) advisedby(X, Y) ← publication(Z, X), publication(Z, Y), professor(Y), sameperson(X, X), sameperson(Y, Y), student(X). #(1.0, 42, 42)
4) advisedby(X, Y) ← publication(Z, X), publication(Z, Y), sameperson(X, X), sameperson(Y, Y), student(X). #(0.48, 26, 54)
5) advisedby(X, Y) ← professor(Y), sameperson(X, X), taughtby(Z, Y), ta(Z, X). #(1.0, 9, 9)
6) advisedby(X, Y) ← publication(Z, X), publication(Z, Y), sameperson(Y, Y), student(X). #(0.52, 29, 56)
7) advisedby(X, Y) ← sameperson(X, X), sameperson(Y, Y), student(X), taughtby(Z, Y), ta(Z, X). #(1.0, 8, 8)
8) advisedby(X, Y) ← professor(Y), sameperson(X, X), sameperson(Y, Y), publication(Z, X), publication(Z, Y). #(0.51, 67, 132)
9) advisedby(X, Y) ← professor(Y), sameperson(X, X), sameperson(Y, Y), student(X), taughtby(Z, Y), ta(Z, X). #(1.0, 5, 5)
10) advisedby(X, Y) ← sameperson(X, X), sameperson(Y, Y), professor(Y), taughtby(Z, Y), ta(Z, X). #(1.0, 8, 8)
11) advisedby(X, Y) ← sameperson(Y, Y), publication(Z, X), publication(Z, Y). #(0.25, 54, 219)
12) advisedby(X, Y) ← publication(Z, X), publication(Z, Y), sameperson(Y, Y), professor(Y). #(0.5, 56, 112)
13) advisedby(X, Y) ← sameperson(X, X), taughtby(Z, Y), ta(Z, X). #(1.0, 8, 8)
14) advisedby(X, Y) ← professor(Y), student(X), taughtby(Z, Y), ta(Z, X). #(1.0, 7, 7)
15) advisedby(X, Y) ← sameperson(X, X), professor(Y), taughtby(Z, Y), ta(Z, X). #(1.0, 6, 6)
16) advisedby(X, Y) ← sameperson(Y, Y), publication(Z, X), publication(Z, Y). #(0.25, 23, 92)
17) advisedby(X, Y) ← sameperson(X, X), publication(Z, X), publication(Z, Y). #(0.25, 15, 60)
18) advisedby(X, Y) ← sameperson(X, X), sameperson(Y, Y), student(X), taughtby(Z, Y), ta(Z, X). #(1.0, 1, 1)

5. **The FB15KSelected-category_of task:** The logic program with *category_of* as the target predicate in the FB15KSelected dataset is presented as follows. The accuracy of this logic program is 100%, which means the logic program covers all 20 test examples under the *category_of* relation.

1) category_of(X, Y) ← instance_of_recurring_event(Z, Y), award_honor_ceremony(X, Z). # (0.91, 143, 158)
2) category_of(X, Y) ← category_of(Y, X). # (0.10, 18, 181)
3) category_of(X, Y) ← category_of(Z, X), category_of(Z, Y). # (0.69, 18, 26)

6. **The FB15KSelected-place_lived_location task:** The logic program with *place_lived_location* as the target predicate in the FB15KSelected dataset is presented as follows. The accuracy of this logic program is 90.16%, which means the logic program covers 275 among 305 test examples under the *place_lived_location* relation.

1) place_lived_location(X, Y) ← place_of_death(X, Y). #(0.09, 78, 834)
2) place_lived_location(X, Y) ← place_of_birth(X, Y). # (0.41, 634, 1562)
3) place_lived_location(X, Y) ← place_lived_location(Z, Y), popstra_dated_participant(Z, X). #(0.06, 119, 2126)
4) place_lived_location(X, Y) ← place_lived_location(X, Z), location_contains(Z, Y). #(0.01, 224, 83480)
5) place_lived_location(X, Y) ← bibs_location_country(Y, Z), person_nationality(X, Z), country_capital(Z, Y). #(0.04, 187, 4267)
6) place_lived_location(X, Y) ← people_marriage_location_of_ceremony(Z, Y), people_marriage_type_of_union(X, Z). #(0.01, 2137, 1632667)
7) place_lived_location(X, Y) ← _music_artist_origin(X, Y). #(0.26, 157, 609)
8) place_lived_location(X, Y) ← place_lived_location(Z, Y), popstra_friendship_participant(X, Z). #(0.05, 124, 2545)
9) place_lived_location(X, Y) ← place_lived_location(X, Z), location_contains(Y, Z). #(0.03, 210, 7838)
10) place_lived_location(X, Y) ← place_of_birth(Z, Y), award_nomination_award_nominee(X, Z). #(0.02, 184, 9641)
11) place_lived_location(X, Y) ← person_nationality(X, Z), location_contains(Z, Y). #(0.01, 2877, 3142123)

## Appendix C. Supplementary material

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.artint.2024.104108.

## References

[1] Y. Aspis, K. Broda, A. Russo, J. Lobo, Stable and supported semantics in continuous vector spaces, in: Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning (KR'20), Rhodes, Greece, vol. 17, 2020, pp. 59–68.
[2] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, J. Taylor, Freebase: a collaboratively created graph database for structuring human knowledge, in: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD'08), 2008, pp. 1247–1250.
[3] A. Bordes, N. Usunier, A. García-Durán, J. Weston, O. Yakhnenko, Translating embeddings for modeling multi-relational data, in: Advances in Neural Information Processing Systems (NeurIPS'13), 2013, pp. 2787–2795.
[4] G. Bouchard, S. Singh, T. Trouillon, On Approximate Reasoning Capabilities of Low-Rank Vector Spaces, AAAI Spring Symposium Series, 2015.
[5] A. Cropper, Forgetting to learn logic programs, in: Proceedings of the AAAI Conference on Artificial Intelligence (AAAI-20), vol. 34, 2020, pp. 3676–3683.
[6] A. Cropper, S. Dumančić, R. Evans, S.H. Muggleton, Inductive logic programming at 30, Mach. Learn. 111 (2022) 147–172.
[7] A. Cropper, S. Dumancic, S.H. Muggleton, Turning 30: new ideas in inductive logic programming, in: International Joint Conference on Artificial Intelligence (IJCAI'20), Yokohama, Japan, 2020, pp. 4833–4839.
[8] A. Cropper, S.H. Muggleton, Logical minimisation of meta-rules within meta-interpretive learning, in: Proceedings of the 25th International Conference on Inductive Logic Programming (ILP'15), Tokyo, Japan, 2015, pp. 62–75.
[9] J. Davis, E. Burnside, I. de Castro Dutra, D. Page, V.S. Costa, An integrated approach to learning Bayesian networks of rules, in: Machine Learning: ECML 2005, Berlin, Heidelberg, vol. 3720, 2005, pp. 84–95.
[10] L. De Raedt, L. Dehaspe, Clausal discovery, Mach. Learn. 26 (1997) 99–146.
[11] T. Dettmers, P. Minervini, P. Stenetorp, S. Riedel, Convolutional 2d knowledge graph embeddings, in: Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI'18), New Orleans, USA, vol. 32, 2018, pp. 1811–1818.
[12] H. Dong, J. Mao, T. Lin, C. Wang, L. Li, D. Zhou, Neural logic machines, in: International Conference on Learning Representations (ICLR'19), New Orleans, USA, 2019.
[13] R. Evans, M. Bosnjak, L. Buesing, K. Ellis, D.P. Reichert, P. Kohli, M.J. Sergot, Making sense of raw input, Artif. Intell. 299 (2021) 103521.
[14] R. Evans, E. Grefenstette, Learning explanatory rules from noisy data, J. Artif. Intell. Res. 61 (2018) 1–64.
[15] R. Evans, J. Hernández-Orallo, J. Welbl, P. Kohli, M.J. Sergot, Making sense of sensory input, Artif. Intell. 293 (2021) 103438.
[16] C. Fellbaum, WordNet: An Electronic Lexical Database, The MIT Press, 1998.
[17] P. Flach, N. Lachiche, 1BC: a first-order Bayesian classifier, in: Inductive Logic Programming (ILP'99), Berlin, Heidelberg, 1999, pp. 92–103.
[18] M.V.M. França, G. Zaverucha, A.S. d'Avila Garcez, Fast relational learning using bottom clause propositionalization with artificial neural networks, Mach. Learn. 94 (2014) 81–104.
[19] M.V.M. França, A.S. D'Avila Garcez, G. Zaverucha, Relational knowledge extraction from neural networks, in: Proceedings of the 2015th International Conference on Cognitive Computation: Integrating Neural and Symbolic Approaches (COCO'15), Aachen, DEU, vol. 1583, 2015, pp. 146–154.
[20] K. Gao, K. Inoue, Y. Cao, H. Wang, Learning first-order rules with differentiable logic program semantics, in: Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, (IJCAI'22), 2022, pp. 3008–3014.
[21] K. Gao, H. Wang, Y. Cao, K. Inoue, Learning from interpretation transition using differentiable logic programming semantics, Mach. Learn. 111 (2022) 123–145.
[22] A.S. d'Avila Garcez, K. Broda, D.M. Gabbay, Symbolic knowledge extraction from trained neural networks: a sound approach, Artif. Intell. 125 (2001) 155–207.
[23] A.S. d'Avila Garcez, G. Zaverucha, The connectionist inductive learning and logic programming system, Appl. Intell. 11 (1999) 59–77.
[24] E. Gentet, S. Tourret, K. Inoue, Learning from interpretation transition using feed-forward neural networks, in: International Conference on Inductive Logic Programming (ILP'16), 2016, pp. 27–33.
[25] P. Hájek, Metamathematics of Fuzzy Logic, Trends in Logic, vol. 4, Kluwer, 1998.
[26] P. Hohenecker, T. Lukasiewicz, Ontology reasoning with deep neural networks, J. Artif. Intell. Res. 68 (2020) 503–540.
[27] K. Inoue, T. Ribeiro, C. Sakama, Learning from interpretation transition, Mach. Learn. 94 (2014) 51–79.

[28] T. Kaminski, T. Eiter, K. Inoue, Exploiting answer set programming with external sources for meta-interpretive learning, Theory Pract. Log. Program. 18 (2018) 571–588.
[29] N. Kaur, G. Kunapuli, S. Joshi, K. Kersting, S. Natarajan, Neural networks for relational data, in: International Conference on Inductive Logic Programming (ILP'19), Plovdiv, Bulgaria, 2019, pp. 62–71.
[30] R.D. King, M.J. Sternberg, A. Srinivasan, Relating chemical activity to structure: an examination of ILP successes, New Gener. Comput. 13 (1995) 411–433.
[31] D.P. Kingma, J. Ba, Adam: a method for stochastic optimization, in: International Conference on Learning Representations (ICLR'15), San Diego, USA, 2015.
[32] S. Kok, P.M. Domingos, Statistical predicate invention, in: Proceedings of the 24th International Conference on Machine Learning (ICML'07), Corvalis, USA, 2007, pp. 433–440.
[33] S. Kramer, N. Lavrač, P. Flach, Propositionalization Approaches to Relational Data Mining, Springer, Berlin Heidelberg, 2001, pp. 262–291.
[34] M. Law, A. Russo, K. Broda, Inductive learning of answer set programs, in: Proceedings of the 14th European Conference on Logics in Artificial Intelligence (JELIA-14), Springer International Publishing, Cham, 2014, pp. 311–325.
[35] J. Lehmann, S. Bader, P. Hitzler, Extracting reduced logic programs from artificial neural networks, Appl. Intell. 32 (2010) 249–266.
[36] J.W. Lloyd, Foundations of Logic Programming, Springer-Verlag, Berlin, Heidelberg, 1984.
[37] G.A. Miller, Wordnet: a lexical database for English, Commun. ACM 38 (1995) 39–41.
[38] S. Muggleton, Inductive logic programming, New Gener. Comput. 8 (1991) 295–318.
[39] S. Muggleton, L. De Raedt, D. Poole, I. Bratko, P. Flach, K. Inoue, A. Srinivasan, ILP turns 20, Mach. Learn. 86 (2012) 3–23.
[40] S.H. Muggleton, Inverse entailment and progol, New Gener. Comput. 13 (1995) 245–286.
[41] S.H. Muggleton, D. Lin, A. Tamaddoni-Nezhad, Meta-interpretive learning of higher-order dyadic datalog: predicate invention revisited, Mach. Learn. 100 (2015) 49–73.
[42] A. Passerini, P. Frasconi, L.D. Raedt, Kernels on prolog proof trees: statistical learning in the ILP setting, J. Mach. Learn. Res. 7 (2006) 307–342.
[43] Y.J. Phua, K. Inoue, Learning logic programs from noisy state transition data, in: International Conference on Inductive Logic Programming (ILP'19), Plovdiv, Bulgaria, 2019, pp. 72–80.
[44] M. Qu, J. Chen, L.-P. Xhonneux, Y. Bengio, J. Tang, RNNLogic: learning logic rules for reasoning on knowledge graphs, in: International Conference on Learning Representations (ICLR'21), 2021.
[45] T. Rocktäschel, S. Riedel, End-to-end differentiable proving, in: Advances in Neural Information Processing Systems (NeurIPS'17), Long Beach, USA, 2017, pp. 3788–3800.
[46] A. Sadeghian, M. Armandpour, P. Ding, D.Z. Wang, DRUM: End-to-End Differentiable Rule Mining on Knowledge Graphs, vol. 32, Vancouver, Canada, 2019.
[47] C. Sakama, K. Inoue, T. Sato, Linear algebraic characterization of logic programs, in: International Conference on Knowledge Science, Engineering and Management (KSEM'17), Springer International Publishing, Melbourne, Australia, 2017, pp. 520–533.
[48] C. Sakama, K. Inoue, T. Sato, Logic programming in tensor spaces, Ann. Math. Artif. Intell. 89 (2021) 1133–1153.
[49] C. Sakama, H. Nguyen, T. Sato, K. Inoue, Partial evaluation of logic programs in vector spaces, in: Proceedings of the 11th Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP'18), Oxford, UK, 2018.
[50] P. Sen, B.W. de Carvalho, R. Riegel, A. Gray, Neuro-symbolic inductive logic programming with logical neural networks, in: Proceedings of the AAAI Conference on Artificial Intelligence (AAAI-22), vol. 36, 2022, pp. 8212–8219.
[51] L. Serafini, A.S. d'Avila Garcez, Logic tensor networks: deep learning and logical reasoning from data and knowledge, in: Workshop on Neural-Symbolic Learning and Reasoning at HLAI (NeSy'16), New York, USA, vol. 1768, 2016.
[52] H. Shindo, M. Nishino, A. Yamamoto, Differentiable inductive logic programming for structured examples, in: Thirty-Fifth AAAI Conference on Artificial Intelligence (AAAI'21), vol. 35, 2021, pp. 5034–5041.
[53] G. Sourek, V. Aschenbrenner, F. Zelezný, S. Schockaert, O. Kuzelka, Lifted relational neural networks: efficient learning of latent relational structures, J. Artif. Intell. Res. 62 (2018) 69–100.
[54] I. Stahl, Predicate invention in ILP—an overview, in: Proceedings of the 6th European Conference on Machine Learning (ECML'93), Springer, Vienna, Austria, 1993, pp. 311–322.
[55] Z. Sun, Z.-H. Deng, J.-Y. Nie, J. Tang, Rotate: knowledge graph embedding by relational rotation in complex space, in: International Conference on Learning Representations (ICLR'19), New Orleans, USA, 2019.
[56] A. Takemura, K. Inoue, Gradient-based supported model computation in vector spaces, in: Proceedings of the 16th International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR'22), 2022, pp. 336–349.
[57] K. Teru, E. Denis, W. Hamilton, Inductive relation prediction by subgraph reasoning, in: Proceedings of the 37th International Conference on Machine Learning, (ICML'20), 2020, pp. 9448–9457.
[58] K. Toutanova, D. Chen, Observed versus latent features for knowledge base and text inference, in: Proceedings of the 3rd Workshop on Continuous Vector Space Models and Their Compositionality, Beijing, China, 2015, pp. 57–66.
[59] R. Trivedi, B. Sisman, X.L. Dong, C. Faloutsos, J. Ma, H. Zha, LinkNBed: multi-graph representation learning with entity linkage, in: Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (ACL'18), Melbourne, Australia, 2018, pp. 252–262.
[60] W.Y. Wang, K. Mazaitis, W.W. Cohen, A soft version of predicate invention based on structured sparsity, in: Proceedings of the 24th International Conference on Artificial Intelligence (IJCAI'15), Buenos Aires, Argentina, 2015, pp. 3918–3924.
[61] F. Yang, Z. Yang, W.W. Cohen, Differentiable learning of logical rules for knowledge base reasoning, in: Advances in Neural Information Processing Systems (NeurIPS'17), Long Beach, USA, 2017, pp. 2319–2328.