# Neuro-Symbolic Inductive Logic Programming with Logical Neural Networks

**Prithviraj Sen, Breno W. S. R. de Carvalho, Ryan Riegel, Alexander Gray**

IBM Research

**Abstract**

Recent work on neuro-symbolic inductive logic programming has led to promising approaches that can learn explanatory rules from noisy, real-world data. While some proposals approximate logical operators with differentiable operators from fuzzy or real-valued logic that are parameter-free thus diminishing their capacity to fit the data, other approaches are only loosely based on logic making it difficult to interpret the learned "rules". In this paper, we propose learning rules with the recently proposed logical neural networks (LNN). Compared to others, LNNs offer a strong connection to classical Boolean logic thus allowing for precise interpretation of learned rules while harboring parameters that can be trained with gradient-based optimization to effectively fit the data. We extend LNNs to induce rules in first-order logic. Our experiments on standard benchmarking tasks confirm that LNN rules are highly interpretable and can achieve comparable or higher accuracy due to their flexible parameterization.

## 1 Introduction

Inductive logic programming (ILP) (Muggleton 1996) has been of long-standing interest where the goal is to learn logical rules from labeled data. Since rules are explicitly symbolic, they provide certain advantages over black-box models. For instance, learned rules can be inspected, understood and verified forming a convenient means of storing learned knowledge. Consequently, a number of approaches have been proposed to address ILP including, but not limited to, statistical relational learning (Getoor and Taskar 2007) and more recently, neuro-symbolic methods.

Since logical operators such as conjunction and disjunction are not differentiable, one issue that most neuro-symbolic ILP techniques have to address is how to learn rules using gradient-based optimization. A popular solution is to employ extensions from fuzzy or real-valued logic that are either differentiable or have subgradients available. For instance, NeuralLP (Yang, Yang, and Cohen 2017) substitutes logical conjunction with product $t$-norm ($x \wedge y \equiv xy$) and logic tensor networks (Donadello, Serafini, and d'Avila Garcez 2017) with Łukasiewicz $t$-norm ($x \wedge y \equiv \max(0, x + y - 1)$). In an interesting experiment, Evans and Grefenstette (2018) show that among the various options, product $t$-norm seems to lead to the best ILP result. This indicates that the learning approach, rather than the user, should be in charge of substituting logical connectives besides learning the rules themselves. Neural logic machine (NLM) (Dong et al. 2019) achieves this but at the cost of interpretability. More precisely, it models propositional formulae (consisting of conjunctions, disjunctions and/or negations) with multi-layer perceptrons (MLP). Once trained, it may not be possible to interpret NLM as rules since there exists no standard translation from MLP to logic. What is needed is an extension of classical logic with ties strong enough to be amenable to interpretation and can learn not only rules but also the logical connectives using gradient-based optimization.

In this paper, we propose ILP with the recently proposed logical neural networks (LNN) (Riegel et al. 2020). Instead of forcing the user to choose a function that mimics a logical connective, LNNs employ constraints to ensure that neurons behave like conjunctions or disjunctions. By decoupling neuron activation from the mechanism to ensure that it behaves like a logical connective, LNNs offer tremendous flexibility in how to parameterize neurons thus ensuring that they fit the data better while maintaining close connections with classical Boolean logic which, in turn, facilitates principled interpretation. We propose first-order extensions of LNNs that can tackle ILP. Since vanilla backpropagation is insufficient for constraint optimization, we propose flexible learning algorithms capable of handling a variety of (linear) inequality and equality constraints. We experiment with diverse benchmarks for ILP including gridworld and knowledge base completion (KBC) that call for learning of different kinds of rules and show how our approach can tackle both effectively. In fact, our KBC results represents a $4$-$16\%$ relative improvement (in terms of mean reciprocal rank) upon the current best rule-based KBC results on popular KBC benchmarks. Additionally, we show that joint learning of rules and logical connectives leads to LNN rules that are easier to interpret vs. other neuro-symbolic ILP approaches.

## 2 Related Work

$\partial$ILP (Evans and Grefenstette 2018) is another neuro-symbolic ILP technique whose main parameter is a tensor with one cell per candidate logic program. Since the number of candidates is exponential in both the number of available predicates and the number of constituent rules

in the program, ∂ILP's complexity is exponential making it impractical for anything but the smallest learning task. To reign in the complexity, ∂ILP asks the user to specify the ILP task using a *template* consisting of two rules each containing a maximum of two predicates. In reality, most neuro-symbolic ILP approaches ask the user to specify a template. NeuralLP's (Yang, Yang, and Cohen 2017) template is meant for link prediction in incomplete knowledge bases (sometimes called open path or chain rule) which only includes binary predicates and is of the form $T(X_1, X_n) \leftarrow R_1(X_1, X_2) \wedge R_2(X_2, X_3) \wedge \dots R_{n-1}(X_{n-1}, X_n)$ positing that the head predicate $T(X_1, X_n)$ can be modeled as a path from $X_1$ to $X_n$. NLM (Dong et al. 2019) is restricted to learning rules where the head and body predicates each contain the same set of variables. For instance, to model the rule learned by NeuralLP, NLM would first need to add arguments to $R_1$ so that the new predicate contains all variables including $X_3, \dots X_n$. In contrast, our approach can use more flexible templates that can express programs beyond two rules, allows the use of $n$-ary predicates ($n$ possibly $> 2$), and allows the head to have fewer variables than the body thus going beyond all of the above mentioned approaches.

Neural theorem provers (NTP) (Rocktäschel and Riedel 2017) generalize the notion of unification by embedding logical constants into a high-dimensional latent space. NTPs can achieve ILP by learning the embedding for the unknown predicate which forms part of the rule, and subsequently comparing with embeddings of known predicates. NTPs can have difficulty scaling to real-world tasks since the decision to unify two constants is no longer Boolean-valued leading to an explosion of proof paths that need to be explored. To improve scalability, recent proposals either greedily choose (GNTP (Minervini et al. 2020a)) or learn to choose (CTP (Minervini et al. 2020b)) most promising proof paths. We compare against CTP in Section 5.

Lifted relational neural networks (LRNN) (Sourek et al. 2017) model conjunctions using (generalized) sigmoid but fix certain parameters to ensure that it behaves similar to Łukasiewicz $t$-norm. This limits how well LRNN can model the data, which is contrary to our goal as stated in the previous section. While other combinations of logic and neural networks exist, e.g. logic tensor networks (Donadello, Serafini, and d'Avila Garcez 2017), RelNN (Kazemi and Poole 2018), DeepProbLog (Manhaeve et al. 2018), to the best of our knowledge, none of these learn rules to address ILP.

## 3 Generalized Propositional Logic with Logical Neural Networks

Logical neural networks (LNN) (Riegel et al. 2020) allow the use of almost any parameterized function as a logical connective. We illustrate how LNNs generalize conjunction ($\wedge$). Let 0 denote `false` and 1 denote `true`. Let $a, b \in \{0, 1\}$ and $x, y \in [0, 1]$ denote Boolean-valued and continuous-valued variables, respectively. While Boolean logic defines the output of $\wedge$ when $x, y$ attain the extremities of their permissible domains (shown in Figure 1 (a)), to fully define real-valued logic's $\wedge$ we need to also extend its definition to $x, y \in (0, 1)$. Intuitively, the characteristic *shape* of $\wedge$ is to produce a 1) *low* output when *either input* is *low*, and 2) *high* output when *both inputs* are *high*. A simple way to capture *low* vs. *high* is via a user-defined hyperparameter $\alpha \in (\frac{1}{2}, 1]$: $x \in [0, 1-\alpha]$ constitutes *low* and $x \in [\alpha, 1]$ constitutes *high*. Figure 1 (b) expresses $\wedge$ for real-valued logic in terms of $\alpha$.

| $a$ | $b$ | $a \wedge b$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

(a)

| $x$ | $y$ | $x \wedge y$ |
|---|---|---|
| $[0, 1-\alpha]$ | $[0, 1-\alpha]$ | $[0, 1-\alpha]$ |
| $[0, 1-\alpha]$ | $(1-\alpha, 1]$ | $[0, 1-\alpha]$ |
| $(1-\alpha, 1]$ | $[0, 1-\alpha]$ | $[0, 1-\alpha]$ |
| $[\alpha, 1]$ | $[\alpha, 1]$ | $[\alpha, 1]$ |

(b)

Figure 1: (a) Truth table for $\wedge$ in Boolean logic, and (b) Shape of $\wedge$ extended to real-valued logic.

LNNs propose constraints to enforce the shape of $\wedge$. Let $f : [0, 1] \times [0, 1] \to [0, 1]$ denote a monotonically increasing function (in both inputs). In other words, $f(x, y') \geq f(x, y)$ $\forall y' \geq y$ and $f(x', y) \geq f(x, y)$ $\forall x' \geq x$. In accordance with figure 1 (b), LNNs enforce the following constraints:

$$
\begin{aligned}
f(x,y) &\leq 1-\alpha, \quad \forall\, x, y \in [0, 1-\alpha] \\
f(x,y) &\leq 1-\alpha, \quad \forall\, x \in [0, 1-\alpha],\ \forall y \in (1-\alpha, 1] \\
f(x,y) &\leq 1-\alpha, \quad \forall\, x \in (1-\alpha, 1],\ \forall y \in [0, 1-\alpha] \\
f(x,y) &\geq \alpha, \quad \forall\, x, y \in [\alpha, 1]
\end{aligned}
$$

Since $f$ is monotonically increasing, we can move all constraints to their corresponding extremities and eliminate the first constraint since it is redundant given the second and third.

$$f(1-\alpha, 1) \leq 1-\alpha,\ f(1, 1-\alpha) \leq 1-\alpha,\ f(\alpha, \alpha) \geq \alpha$$

Further simplifications may be obtained for a specific choice of $f$. For inspiration, we look towards *triangular norm* ($t$-norm) (Esteva and Godo 2001) which is defined as a symmetric, associative and non-decreasing function $T : [0, 1]^2 \to [0, 1]$ satisfying boundary condition $T(1, x) = x,\ \forall x \in [0, 1]$. Popular $t$-norms include *product* $t$-norm, $xy$, and *Łukasiewicz* $t$-norm, $\max(0, x + y - 1)$. We extend the latter to define LNN-$\wedge$ (other $t$-norms may also be extended similarly):

$$
\text{LNN-}\wedge(x, y; \beta, w_1, w_2) = \begin{cases} 0 & \text{if } \beta - w_1(1-x) - w_2(1-y) < 0 \\ 1 & \text{if } \beta - w_1(1-x) - w_2(1-y) > 1 \\ \beta - w_1(1-x) - w_2(1-y) & \text{otherwise} \end{cases}
$$

where $\beta, w_1, w_2$ denote learnable parameters subject to the following constraints translated from above[1]:

$$
\begin{aligned}
\text{LNN-}\wedge(1-\alpha, 1; \beta, w_1, w_2) &= \beta - w_1\alpha \leq 1-\alpha \\
\text{LNN-}\wedge(1, 1-\alpha; \beta, w_1, w_2) &= \beta - w_2\alpha \leq 1-\alpha \\
\text{LNN-}\wedge(\alpha, \alpha; \beta, w_1, w_2) &= \beta - (w_1 + w_2)(1-\alpha) \geq \alpha
\end{aligned}
$$

To ensure that LNN-$\wedge$ is monotonically increasing, we also need to enforce non-negativity of $w_1, w_2$. It is easy to extend

[1]We remove the upper and lower clamps since they do not apply in the active region of the constraints.

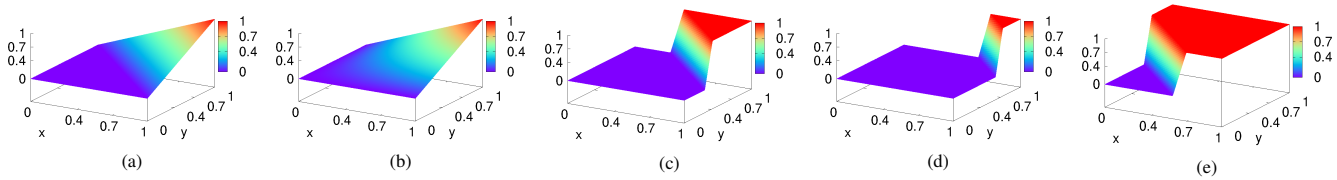



Figure 2: (a) Lukasiewicz $t$-norm vs. (b) Product $t$-norm vs. LNN-$\wedge$ with (c) $\alpha = 0.7$, (d) $\alpha = 0.9$. (e) LNN-$\vee$ ($\alpha = 0.7$).

LNN-$\wedge$ to an $n$-ary conjunction ($n \geq 2$):

$$\begin{aligned} \text{LNN-}\wedge(\mathbf{x}; \beta, \mathbf{w}) \equiv \quad & \texttt{relu1}(\beta - \mathbf{w}^\top(1 - \mathbf{x})) \\ \text{subject to:} \quad & \mathbf{w} \geq 0,\ \beta - \alpha\mathbf{w} \leq (1-\alpha)\mathbf{1} \\ & \beta - (1-\alpha)\mathbf{1}^\top\mathbf{w} \geq \alpha \end{aligned} \tag{1}$$

where $\texttt{relu1}(x)$ denotes $\max(0, \min(1, x))$ (Krizhevsky 2010) and, $\mathbf{x}$, $\mathbf{w}$ and $\mathbf{1}$ denote vectors of continuous-valued inputs, weights, and 1s, respectively.

Note that, Figure 1 (b) does not enforce constraints on $1 - \alpha < x, y < \alpha$. Essentially, $\alpha$ acts as a tunable knob that controls the size of this unconstrained region where we can learn LNN operators without impedance which is an arguably better approach than choosing a parameter-less $t$-norm that would arbitrarily interpolate from Boolean-logic's $\wedge$ to real-valued logic's $\wedge$. Figure 2 illustrates how Łukasiewicz (Figure 2 (a)) and product (Figure 2 (b)) $t$-norms differ from LNN-$\wedge$ learned by fitting to the four rows in Figure 1 (a)'s truth-table. Even pictorially, LNN-$\wedge$ looks distinctly conjunction-like, i.e., when either $x$ or $y$ is low it produces a value close to 0 while rising quickly to 1 when both $x, y$ are high. When $\alpha = 0.9$ (Figure 2 (d)), the region devoid of constraints is larger than at $\alpha = 0.7$ (Figure 2 (c)) ($\because [0.1, 0.9] \supset [0.3, 0.7]$), so the curve can rise later to provide a better fit . In contrast, Łukasiewicz $t$-norm remains at 0 until the $x + y = 1$ line, post which it increases linearly. Product $t$-norm is similar, adding a slight, upward curve.

Other propositional logic operators include negation ($\neg$) and disjunction ($\vee$). LNN negation is given by $1 - \mathbf{x}$ and LNN disjunction, LNN-$\vee$, is defined in terms of LNN-$\wedge$:

$$\text{LNN-}\vee(\mathbf{x}; \beta, \mathbf{w}) = 1 - \text{LNN-}\wedge(1 - \mathbf{x}; \beta, \mathbf{w})$$

where constraints defined in Equation 1 apply. Figure 2 (e) pictorially depicts LNN-$\vee$ (with $\alpha = 0.7$). In contrast to Figure 2 (c), it clearly shows how maximum output is achieved for smaller values of $x, y$, as a disjunction operator should.

## 4 Learning First-Order LNNs

Following previous work (Yang, Yang, and Cohen 2017; Evans and Grefenstette 2018; Dong et al. 2019), we also utilize program *templates* expressed in higher-order logic to be fed by the user to guide the learning in the right direction. Our definition of a program template draws inspiration from meta-interpretive learning (Muggleton et al. 2014). In contrast to previous work on neuro-symbolic AI however, our definition of a program template is more general and includes as special cases the templates utilized by Evans and Grefenstette (considers only rules whose body contains up to 2 predicates), Yang, Yang, and Cohen (considers only binary predicates) and Dong et al. (considers only rules whose head includes all variables contained in the body). After introducing our logic program template, we then describe how to combine it with data to construct a neural network that may then be trained to learn the logic program of interest.

Let $\text{pred}(X_1, \ldots X_n)$ denote an $n$-ary *predicate* which returns `true` (1) or `false` (0) for every possible joint assignment of $X_1, \ldots X_n$ to constants in the knowledge base. The main construct in first-order logic (FOL) is a rule or *clause*:

$$h \leftarrow b_1 \wedge b_2 \wedge \ldots b_m$$

where $b_1, \ldots b_m$ denote predicates in its *body* and $h$ denotes the *head* predicate. *If* the conjunction of all predicates in the body is true *then* the head is also true. The head $h$ in a clause may contain fewer logical variables than the body $b_1, \ldots b_m$ which means that there must exist an assignment to the missing variables in the head for it to hold. More precisely, if $\mathbf{B} = b_1, \ldots b_m$ denotes the body and is defined over logical variables $\mathbf{Y}$ then $h(\mathbf{X})$, such that $\mathbf{X} \subseteq \mathbf{Y}$, is `true` if $\exists \mathbf{Y} \setminus \mathbf{X} : \mathbf{B}(\mathbf{Y})$ is `true`. Assignments that lead to the predicate being true are also called *facts*.

Figure 3 (a) introduces a toy knowledge base (KB) which will serve as a running example. The KB contains three binary predicates, each containing their respective facts along with a unique identifier for easy reference. Thus, $A(X, Y)$'s facts are $A(1, 2)$ (denoted $a_1$) and $A(1, 5)$ (denoted $a_2$). Figure 3 (b) shows a template for learning $R(X, Z)$ with three crucial pieces of information: 1) the first predicate in its body $P$ is binary and while we do not know the identity of this predicate we know its domain $\text{Dom}(P)$ is $\{A, B\}$, 2) to keep the example simple, the second predicate in the body $Q$ has a singleton domain ($\{C\}$), and lastly 3) the second argument in the first predicate should be equal to the first argument in the second predicate indicated by repeated use of $Y$. Figure 3 (b) (bottom) expresses the same template as a tree where $P(X, Y)$ and $Q(Y, Z)$ are annotated with their respective domains forming children of $R(X, Z)$ whose label $\wedge$ indicates that the predicates in its body are to be conjuncted together.

Figure 3 (c) shows a more complex template that disjuncts $R(X, Z)$ with $O(X, Z)$, such that $\text{Dom}(O) = \{A, B\}$, to produce $S(X, Z)$. Figure 3 (d) shows *generated* facts (with unknown truth-values) that can be possibly produced when this template is combined with the KB from Figure 3 (a). $P, Q$ and $O$ contain the union of all facts included in the predicates in their respective domains. Since $p_1$ and $q_1$ are the only two facts that agree on the value of $Y$, $PQ$, the predicate produced by the body of $R$, contains only one generated fact. $R$ is obtained by dropping $Y$, which is then unioned with $O$ to produce $S$. By comparing generated facts in $S$ with labels in

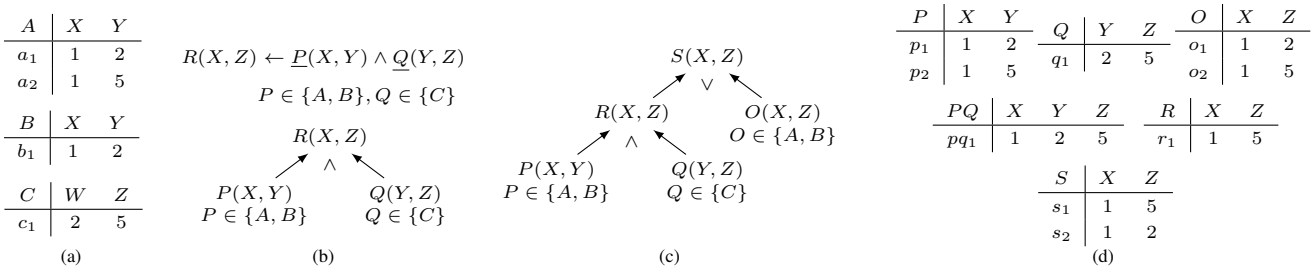



Figure 3: (a) A toy KB. (b) An example rule template (top) and its tree form (bottom). (c) A more complex program template. (d) Generated facts for our running example.

the training data, it is possible to learn a full program which in this case constitutes learning: 1) which predicates to replace $P, Q$ and $O$ with, and 2) the logical connectives, LNN-$\wedge$ and LNN-$\vee$, used to model $R$ and $S$ with, respectively. We next state a more formal problem definition.

Let $\mathcal{T} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ denote a tree-structured *program template* where $\mathcal{V}$ denotes the set of nodes, $\mathcal{E}$ denotes the set of edges and $\mathcal{L}$ denotes a mapping from $\mathcal{V}$ to node labels. $\mathcal{L}$ maps $\mathcal{T}$'s leaves to the corresponding domain of predicates in the KB. In the worst case, the domain can be the subset of predicates in the KB that agrees with the arity of the leaf. $\mathcal{L}$ maps internal nodes to a logical operator $\{\wedge, \vee, \neg\}$. The ILP task can then be stated as, given $\mathcal{T}$, a knowledge base KB, and truth-values corresponding to generated facts in the root of $\mathcal{T}$, to learn all logical connectives involved along with selecting predicates for each leaf in $\mathcal{T}$.

In the remainder of this section, we describe how to achieve the above ILP task given ground truth values for generated facts belonging to the root of the template. Let $\psi(v)$ denote the truth value associated with (generated) fact $v$. Our strategy is to build a neural network that connects the truth values of generated facts from the root of the template to other (generated) facts in its lineage right down to the facts in the base KB whose truth values are defined to be 1. The whole neural network can then be trained end-to-end using backpropagation. Let $\mathbf{V}(\mathbf{X})$ denote any node in $\mathcal{T}$ whose predicate is defined over variables $\mathbf{X}$ and whose children in $\mathcal{T}$ is denoted by $\mathcal{N}(\mathbf{V})$. Also, let $\mathbf{V}(\mathbf{x})$ denote a fact obtained by the substitution $\mathbf{X} = \mathbf{x}$ and $\mathcal{F}(\mathbf{V})$ denote all facts of $\mathbf{V}$.

## 4.1 Combining Base Facts

Let $\mathbf{V}(\mathbf{X})$ denote a leaf in $\mathcal{T}$ with domain $\mathcal{L}(\mathbf{V})$ then $\mathcal{F}(\mathbf{V})$ is given by $\bigcup_{\mathbf{P} \in \mathcal{L}(\mathbf{V})} \mathcal{F}(\mathbf{P})$. Computing $\psi(\mathbf{V}(\mathbf{X} = \mathbf{x}))$ corresponding to $\mathbf{X} = \mathbf{x}$ requires truth values of all facts corresponding to the same substitution. We provide two options that associate parameters with predicates in $\mathcal{L}(\mathbf{V})$: 1) *attention* (Yang, Yang, and Cohen 2017) and 2) our proposed LNN-`pred` operator:

$$1) \quad \psi(\mathbf{V}(\mathbf{x})) = \sum_{\mathbf{P} \in \mathcal{L}(\mathbf{V})} w_{\mathbf{P}} \psi(\mathbf{P}(\mathbf{x}))$$

$$2) \quad \psi(\mathbf{V}(\mathbf{x})) = 1 - \texttt{relu1}\left(\beta - \sum_{\mathbf{P} \in \mathcal{L}(\mathbf{V})} w_{\mathbf{P}} \psi(\mathbf{P}(\mathbf{x}))\right)$$

where $\beta$, $w_{\mathbf{P}}$ denote learnable parameters. One issue with attention is that it may lack sparsity assigning a majority of predicates in $\mathcal{L}(\mathbf{V})$ non-zero weights thus hampering interpretability. To address this, we propose an alternate parameterization, LNN-`pred`, that is a simpler version of LNN-$\vee$ and lacks all constraints except for non-negativity of $w_{\mathbf{P}}$ (since we do not require disjunctive semantics). As an example, the lower left corner of Figure 4 shows how to compute $\psi(p_1)$ from $\psi(a_1), \psi(b_1)$ since $a_1, b_1$ form the lineage for $p_1$ (Figure 3). Here, $\beta_1, \mathbf{w}_1$ denote LNN-`pred`'s parameters.

## 4.2 Combining Facts with Conjunction

We handle conjunctions in two steps: 1) first construct the result of the body of the clause, and then 2) construct the head, dropping variables if needed. Let $\mathbf{V}(\mathbf{Y})$ in $\mathcal{T}$ be such that $\mathcal{L}(\mathbf{V}) = \wedge$ and $\mathcal{N}(\mathbf{V})$ denote its children. Also, let $\mathbf{I}(\mathbf{X})$ denote the intermediate predicate produced from the body of $\mathbf{V}(\mathbf{Y})$ potentially containing additional variables such that $\mathbf{X} \supseteq \mathbf{Y}$. We use LNN-$\wedge$ to compute $\psi$ of $\mathbf{I}(\mathbf{x})$:

$$\psi(\mathbf{I}(\mathbf{x})) = \texttt{relu1}\left(\beta - \sum_{\mathbf{P} \in \mathcal{N}(\mathbf{V})} w_{\mathbf{P}}(1 - \psi(\mathbf{P}(\mathbf{x}_{\text{var}(\mathbf{P})})))\right)$$

where $\text{var}(\mathbf{P})$ denotes predicate $\mathbf{P}$'s arguments and $\mathbf{x}_{\text{var}}$ denotes the substitution restricted to variables in var. When $\mathbf{X} \supset \mathbf{Y}$, multiple facts from $\mathcal{F}(\mathbf{I})$ may combine to produce a fact in $\mathbf{V}(\mathbf{Y})$ and we use `maxout` for this:

$$\psi(\mathbf{V}(\mathbf{y})) = \texttt{maxout}(\{\psi(\mathbf{I}(\mathbf{x})) \mid \mathbf{x}_{\mathbf{Y}} = \mathbf{y}, \forall \mathbf{I}(\mathbf{x}) \in \mathcal{F}(\mathbf{I})\})$$

where $\texttt{maxout}(\{x_1, \ldots\})$ (Goodfellow et al. 2013) returns the maximum of the set. Figure 4 shows how $\psi(pq_1)$ is computed from $\psi(p_1), \psi(q_1)$ where $\beta_2, \mathbf{w}_2$ denotes LNN-$\wedge$'s parameters. Since $pq_1$ is the only intermediate fact leading to $r_1$, we do not need `maxout` in this case. However, if that was not the case, Figure 4 shows where `maxout` would appear.

## 4.3 Combining Facts with Disjunction

Given $\mathbf{V}(\mathbf{X})$ in $\mathcal{T}$ such that $\mathcal{L}(\mathbf{V}) = \vee$, $\psi(\mathbf{V}(\mathbf{x}))$ can be computed using LNN-$\vee$:

$$\psi(\mathbf{V}(\mathbf{x})) = 1 - \texttt{relu1}\left(\beta - \sum_{\mathbf{P} \in \mathcal{N}(\mathbf{V})} w_{\mathbf{P}} \psi(\mathbf{P}(\mathbf{x}))\right)$$

In Figure 4 shows how $\psi(s_1)$ is computed from $\psi(r_1)$ and $\psi(o_2)$ where $\beta_r, \mathbf{w}_4$ denotes LNN-$\vee$'s parameters.

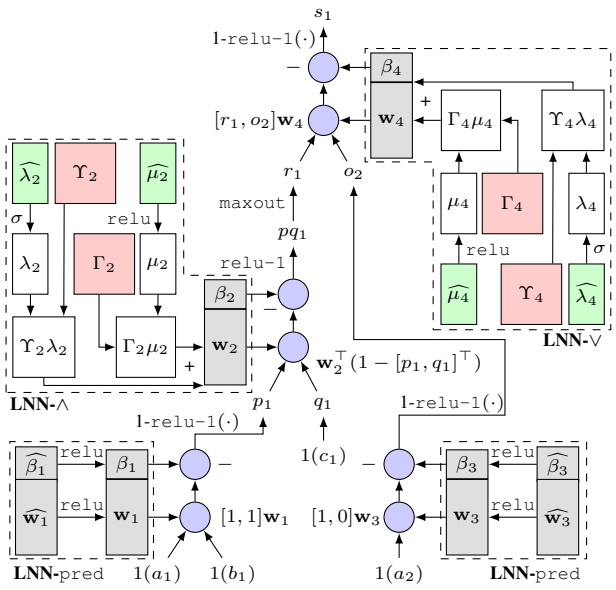



Figure 4: Neural network constructed for $s_1 \in \mathcal{F}(S)$. $\sigma$ denotes `softmax`.

### 4.4 Other Operations and Extensions

Implementing negation is more involved since it requires that we consider assignments that *may* lead to facts in $\mathbf{V}$ but are not present in its child predicate $\mathbf{P}$. Given a universe of all possible assignments $\mathcal{U}$, we express $\psi(\mathbf{V}(\mathbf{x}))$ as:

$$\psi(\mathbf{V}(\mathbf{x})) = 1 - \psi(\mathbf{P}(\mathbf{x})),\ \forall \mathbf{x} \in \mathcal{U}$$

where $\psi(\mathbf{P}(\mathbf{x}))$ is defined as 0 if $\mathbf{P}(\mathbf{x}) \notin \mathcal{F}(\mathbf{P})$.

Note that, any LNN operator introduced for $\mathbf{V}(\mathbf{X})$ is shared across all $\mathbf{V}(\mathbf{x}) \in \mathcal{F}(\mathbf{V})$ since the result of ILP should be agnostic of individual facts. Thus, even though the (sub)network constructed for $\mathbf{V}(\mathbf{x})$ may differ from $\mathbf{V}(\mathbf{x}')$'s, e.g., $s_2$'s neural network (not shown) is simpler than $s_1$'s, gradient updates flow to the same LNN parameters. For simplicity, we only discussed templates comprising a tree of Horn clauses but the ideas presented here can easily extend to DAG-structured templates and going beyond equality conditions in the body of a clause, e.g., $R(X, Z) \leftarrow P(X, Y) \wedge Q(Y', Z) \wedge Y > Y'$.

### 4.5 Learning Constrained Activations

So far, we have shown how to construct a neural network from a KB and template comprising parameters of LNN operators but we have not addressed how to enforce constraints on said parameters. More precisely, $\beta_i, \mathbf{w}_i,\ \forall i = 1, \ldots 4$ in Figure 4 need to satisfy the respective constraints associated with the LNN operators they form parameters for (as described in Section 3). Since backpropagation does not handle constraints, we propose to apply a recently proposed approach to "fold" in a system of linear constraints as layers into the neural network (Frerix, Nießner, and Cremers 2020). We note that Riegel et al. (2020) also devise a training algorithm for learning LNN operators but this is tightly connected to a specific kind of LNN operator called tailored activation. For the small systems of inequality constraints introduced by LNN-$\wedge$, LNN-$\vee$ and LNN-`pred`, the approach presented here conveniently allows learning all (constrained) parameters of LNNs using vanilla backpropagation alone.

Frerix, Nießner, and Cremers (2020) recently showed how to handle a system of linear inequality constraints of the form $\boldsymbol{A}\mathbf{z} \leq \mathbf{b}$ where $\boldsymbol{A}$ denotes a matrix containing coefficients in the constraints, $\mathbf{z}$ denotes the parameters (in our case, some concatenation of $\beta$ and $\mathbf{w}$), and $\mathbf{b}$ denotes the constants in the constraints. We begin with the Minkowski-Weyl theorem:

**Theorem 1.** *A set $\mathcal{C} = \{\mathbf{z} \mid \boldsymbol{A}\mathbf{z} \leq \mathbf{b}\}$ is a convex polyhedron if and only if:*

$$\mathcal{C} = \left\{\Upsilon\mu + \Gamma\lambda \,\middle|\, \mu, \lambda \geq \mathbf{0}, \mathbf{1}^\top\lambda = 1\right\}$$

*where $\Upsilon$ and $\Gamma$ contain a finite number of rays and vertices, respectively.*

which states that there exists a translation from $\boldsymbol{A}, \mathbf{b}$ to $\Upsilon, \Gamma$ obtained via the *double-description* method (Motzkin et al. 1953). Assuming we can generate non-negative vectors $\mu, \lambda$ and additionally ensure that $\lambda$ sums to 1, then one can access a point $\mathbf{z} = [\beta, \mathbf{w}^\top]^\top$ from the feasible set $\mathcal{C}$ by computing $\Upsilon\mu + \Gamma\lambda$. Sampling vectors $\mu, \lambda$ can be achieved, for instance, by using `relu` (Nair and Hinton 2010) and `softmax`:

$$\mu = \max(0, \widehat{\mu}) \qquad \lambda = \frac{\exp(\widehat{\lambda})}{Z},\ Z = \mathbf{1}^\top \exp(\widehat{\lambda})$$

$$[\beta, \mathbf{w}^\top]^\top = \Upsilon\mu + \Gamma\lambda$$

Additionally, these operations can be easily included into any neural network as additional layers. For instance, Figure 4 contains in dashed boxes the above set of layers needed to generate $\mathbf{w}_i, \beta_i,\ \forall i = 1, \ldots 4$. The resulting neural network is self-contained, and can be trained by vanilla backpropagation end-to-end thus ensuring that the learned LNN parameters satisfy their respective constraints.

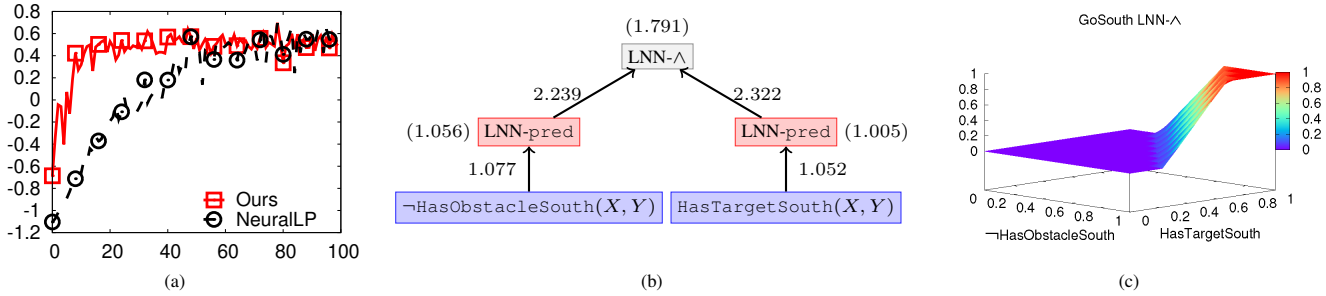
## 5 Experiments

Our experiments compare ILP with LNN against other neuro-symbolic ILP approaches on standard benchmarks. We evaluate rules in terms of application-specific goodness metrics and interpretability.

### 5.1 Gridworld

The goal in Gridworld is to learn rules that can help an agent move across an $N \times N$ regular grid. Some of the cells on the grid are deemed obstacles that the agent cannot step onto, and the agent's goal is to arrive at the cell which has been deemed the destination.

**Predicates, Template and Rewards**: We include two kinds of base predicates to describe the grid 1) `HasObstacle-`*dir*$(X, Y)$ is `true` if the cell next to $X, Y$ in direction *dir* contains an obstacle, 2) `HasTarget-`*dir*$(X, Y)$ is `true` if the destination lies in direction *dir* from cell $X, Y$. There are four directions, North, South, East, and West, and including their negated counterparts, $\neg$`HasObstacle-`*dir*$(X, Y)$ and $\neg$`HasTarget-`*dir*$(X, Y)$, brings the total number of base predicates to 8. The template is $S(X, Y) \leftarrow P(X, Y) \wedge$




(a) (b) (c)

Figure 5: Gridworld: (a) Avg. Rewards vs. Training Grids. (b) LNN Rule and (c) LNN-∧ for GoSouth(X,Y).

$Q(X, Y)$ where $P$'s domain includes all `HasObstacle`-*dir* predicates and their negated counterparts, and $Q$'s domain includes all `HasTarget`-*dir* predicates and their negated counterparts. We set $\alpha = 0.8$ and use a simple reward mechanism: +1 for moving towards the destination, -1 for moving away, and -2 for stepping on an obstacle. The learning objective is to maximize rewards on randomly generated $5 \times 5$ grids with 3 obstacles sampled uniformly at random. We test the learned rules on randomly generated $5 \times 5$ grids with 12 obstacles.

**Results**: We compare our approach based on LNN-`pred` and LNN-∧, against NeuralLP which uses attention and product $t$-norm. Figure 5 (a) shows mean rewards averaged across all cells of 50 test grids produced by the learned rules on the y-axis vs. number of training grids observed on the x-axis. NeuralLP requires a lot more grids before it can learn the desired rules whereas we learn almost perfect rules after observing as few as 20 training grids. Essentially, in comparison to product $t$-norm, due to the extra learnable parameters in LNN-∧, we can learn with fewer learning iterations. Figure 5 (b) shows the weights for the LNN rule for `GoSouth`$(X, Y)$ on the edges and biases in parenthesis. This rule allows the agent to go South if 1) target is in that direction and, 2) there is no obstacle to the immediate South of the current cell. Despite the leaves of the template containing 8 predicates each in their respective domains, the learned LNN-`pred`s are quite sparse and thus highly interpretable. The left LNN-`pred` assigns 1 non-zero weight to `¬HasObstacleSouth` out of all `HasObstacle`-*dir* predicates and their negated counterparts, and the right LNN-`pred` assigns 1 non-zero weight to `HasTargetSouth` out of all `HasTarget`-*dir* predicates and their negated counterparts. This may be due to the use of `relu1` in LNN-`pred` whose sparsity-inducing properties have been noted before (Krizhevsky 2010). In Figure 5 (c), we also plot `GoSouth`$(X, Y)$'s learned LNN-∧.

## 5.2 Knowledge Base Completion

Knowledge base completion (KBC) is a standard benchmark for ILP. We experiment with publicly available KBC datasets Kinship, UMLS (Kok and Domingos 2007), WN18RR (Dettmers et al. 2018), and FB15K-237 (Toutanova and Chen 2015) [2] (see Table 1 for statistics). We compare against a host of rule-based KBC approaches: NeuralLP (Yang, Yang, and Cohen 2017), DRUM (Sadeghian et al. 2019), CTP (Minervini et al. 2020b) which is an improvement on neural theorem provers (Rocktäschel and Riedel 2017), and the recently proposed RNNLogic[3] (Qu et al. 2021).

| Name | Vertices | Predicates | Facts | Queries |
|---|---|---|---|---|
| UMLS | 135 | 49 | 5216 | 661 |
| Kinship | 104 | 26 | 10686 | 1074 |
| WN18RR | 40945 | 11 | 86835 | 3134 |
| FB15K-237 | 14505 | 237 | 272115 | 20466 |

Table 1: Statistics of KBC datasets.

**Task Description and Template**: A popular abstraction of the KBC task is to complete edges or triples missing from the knowledge graph (KG). More precisely, given a query $\langle h, r, ? \rangle$, where $h$ denotes a source vertex and $r$ denotes a relation from the KG, most KBC approaches provide a ranked list of destination vertices. Most of the aforementioned rule-based KBC approaches exclusively focus on learning chain FOL rules as discussed in Section 2. There are at least two prevalent approaches to learning chain rules for KBC. The first approach (Yang, Yang, and Cohen 2017; Sadeghian et al. 2019) represents each predicate in the body as a mixture of relations present in the KG, and subsequently combines these via a conjunction operator. Figure 3 (c)'s template captures this where the subtree rooted at $R$ defines chain rules of length 2 predicates and $O$ captures length 1 thus enabling learning of chain rules capturing multi-hop paths of length up to 2. The only change we need to make to this template is to include all relations in the KG into the domains of the leaves. It is also easy to extend the template to learn longer chain rules. A different approach, pioneered by MINERVA (Das et al. 2018) and RNNLogic (Qu et al. 2021), is to define the chain rule as a mixture over all possible paths that can exist in the KG. This latter approach leads to more effective KBC and thus we report results by expressing it in our LNN framework as follows: 1) Express each possible multi-hop path as a *base* relation, and 2) Use one LNN-pred operator to express a mixture over all multi-hop paths.

**Metrics and Methodology**: We learn a chain rule per relation present in the KG. Following previous work (Yang, Yang,

[2] All available at github.com/shehzaadzd/MINERVA

[3] We compare with rules-only RNNLogic ("w/o embd."), since using embeddings is out of scope of this work.

| | | NeuralLP | DRUM | CTP | RNNLogic | Ours |
|---|---|---|---|---|---|---|
| Kinship | Hits@10 | 89.1 | 86.1 | 93.9 | 91.1 | **98.4** |
| | Hits@3 | 63.0 | 48.2 | 79.7 | 72.9 | **89.3** |
| | MRR | 48.8 | 40.0 | 70.3 | 64.5 | **81.9** |
| UMLS | Hits@10 | 93.0 | 97.9 | 97.0 | 91.1 | **99.4** |
| | Hits@3 | 75.4 | 91.2 | 91.0 | 82.1 | **98.3** |
| | MRR | 55.3 | 61.8 | 80.1 | 71.0 | **90.0** |
| WN18RR | Hits@10 | 50.2 | 52.1 | – | 53.1* | **55.5** |
| | Hits@3 | 43.3 | 44.6 | – | 47.5* | **49.7** |
| | MRR | 33.7 | 34.8 | – | 45.5* | **47.3** |
| FB15K-237 | Hits@10 | 32.8 | 33.1 | – | 44.5* | **47.0** |
| | Hits@3 | 19.8 | 20.0 | – | 31.5* | **34.2** |
| | MRR | 17.4 | 17.5 | – | 28.8* | **30.7** |

Table 2: KBC Results: Bold font denotes best in a row. CTP does not scale to WN18RR, FB15K-237. * indicates results copied from original paper.

and Cohen 2017), we also add inverse relations to the KG which switches the source and destination vertices. We also include inverse triples into our test set. We compute *filtered* ranks for destinations (Bordes et al. 2013), which removes all true triples ranked above, and compute the following metrics based on Sun et al. (2020)'s suggestions. Let $n$ denote the number of destinations that have a score strictly greater than destination $t$'s and let the number of destinations assigned the same score as $t$'s be denoted by $m$ (including $t$), then we compute $t$'s mean reciprocal rank (MRR) and Hits@K as:

$$\text{MRR} = \frac{1}{m}\sum_{r=n+1}^{n+m}\frac{1}{r}, \quad \text{Hits@K} = \frac{1}{m}\sum_{r=n+1}^{n+m}\delta(r \leq K)$$

where $\delta()$ denotes the Dirac delta function. For each method, we report averages across all test set triples. We learn chain rules containing up to 3 predicates for Kinship and UMLS, 4 for FB15K-237, and 5 for WN18RR in the body of the rule. We provide additional details including the training algorithm used and hyperparameter tuning in Appendix A.

**Results**: Table 2 reports results for all methods. While CTP improves upon the efficiency of neural theorem provers (Rocktäschel and Riedel 2017), it still does not scale beyond Kinship and UMLS (indicated by $-$). Also, we copy previously published results for RNNLogic on WN18RR and FB15K-237[4] (indicated by $^*$). On the smaller datasets, CTP is the best baseline but our results are significantly better producing $16.5\%$ and $12.4\%$ relative improvements in MRR on Kinship and UMLS, respectively. On the larger datasets, RNNLogic is the current state-of-the-art within rule-based KBC and we outperform it producing $4\%$ and $6.6\%$ relative improvements in MRR on WN18RR and FB15K-237, respectively. Despite both learning a mixture over relation sequences appearing on KG paths, one reason for our relative success could be that RNNLogic uses an inexact training algorithm (Qu et al. 2021), relying on expectation-maximization, ELBO bound, whereas we employ no such approximations.

**Learned Rules for FB15K-237**: Table 3 presents a few rules learned from FB15K-237. Rule 1 in Table 3 infers the language a person speaks by exploiting knowledge of the language spoken in her/his country of nationality. In terms of multi-hop path, this looks like: $P(\text{person}) \xrightarrow{\text{nationality}} N(\text{nation}) \xleftarrow{\text{spoken\_in}} L(\text{language})$. Similarly, Rule 2 uses the film_country relation instead of nationality to infer the language used in a film. Besides spoken_in, FB15K-237 contains other relations that can be utilized to infer language such as the official_language spoken in a country. Rule 3 uses this relation to infer the language spoken in a TV program by first exploiting knowledge of its country of origin. Rules 5, 6 and 7 are longer rules containing 3 relations each in their body. Rule 5 infers a TV program's country by first exploiting knowledge of one of its actor's birth place and then determining which country the birth place belongs to. Rule 6 is similar but uses a film crew member's marriage location instead to infer the region where the film was released. Lastly, Rule 7 infers the marriage location of a celebrity by exploiting knowledge of where their friends got married.

1. $\text{person\_language}(P, L) \leftarrow \text{nationality}(P, N) \wedge \text{spoken\_in}(L, N)$
2. $\text{film\_language}(F, L) \leftarrow \text{film\_country}(F, C) \wedge \text{spoken\_in}(L, C)$
3. $\text{tv\_program\_language}(P, L) \leftarrow \text{country\_of\_tv\_program}(P, N) \wedge \text{official\_language}(N, L)$
4. $\text{burial\_place}(P, L) \leftarrow \text{nationality}(P, N) \wedge \text{located\_in}(L, N)$
5. $\text{tv\_program\_country}(P, N) \leftarrow \text{tv\_program\_actor}(P, A) \wedge \text{born\_in}(A, L) \wedge \text{located\_in}(L, N)$
6. $\text{film\_release\_region}(F, R) \leftarrow \text{film\_crew}(F, P) \wedge \text{marriage\_location}(P, L) \wedge \text{located\_in}(L, R)$
7. $\text{marriage\_location}(P, L) \leftarrow \text{celebrity\_friends}(P, F) \wedge \text{marriage\_location}(F, L') \wedge \text{location\_adjoins}(L', L)$

Table 3: Learned rules from FB15K-237.

**Additional KBC Results**: Due to space constraints, in Appendix B we report results on the Countries dataset (Bouchard, Singh, and Trouillon 2015) for which ground truth rules are known. On Countries, our KBC accuracy is comparable to other approaches and the learned LNN rules form a close match with the ground truth rules specified in Nickel, Rosasco, and Poggio (2016).

## 6 Conclusion

Our experiments show that learning rules and logical connectives jointly is not only possible but leads to more accurate rules than other neuro-symbolic ILP approaches. Templates provide a flexible way to express a wide range of ILP tasks. The templates used for Gridworld and KBC are distinct, yet we outperformed baselines in both cases. LNN rules use weights sparingly and are eminently interpretable while LNN operators' constraint formulation ensures close ties to classical logic's precise semantics compared to other approaches (e.g., NLM). While our neural network requires grounding the KB, our approach is still scalable enough to tackle the larger KBC benchmarks whereas others are not (e.g., CTP). In terms of future work, we aim to combine the ideas presented here with embedding of predicates and constants in a high-dimensional latent space to hopefully further improve performance. We would also like to extend our approach to learn more general Prolog-style rules (e.g., recursion).

[4]Despite exchanging multiple emails with its authors, we were unable to run RNNLogic code on the larger KBC datasets.

## References


Bordes, A.; Usunier, N.; Garcia-Duran, A.; Weston, J.; and Yakhnenko, O. 2013. Translating embeddings for modeling multi-relational data. In *NeurIPS*.

Bouchard, G.; Singh, S.; and Trouillon, T. 2015. On Approximate Reasoning Capabilities of Low-Rank Vector Spaces. In *AAAI*.

Das, R.; Dhuliawala, S.; Zaheer, M.; Vilnis, L.; Durugkar, I.; Krishnamurthy, A.; Smola, A.; and McCallum, A. 2018. Go for a Walk and Arrive at the Answer: Reasoning Over Paths in Knowledge Bases using Reinforcement Learning. In *ICLR*.

Dettmers, T.; Minervini, P.; Stenetorp, P.; and Riedel, S. 2018. Convolutional 2d knowledge graph embeddings. In *Thirty-second AAAI conference on artificial intelligence*.

Donadello, I.; Serafini, L.; and d'Avila Garcez, A. S. 2017. Logic Tensor Networks for Semantic Image Interpretation. In *IJCAI*.

Dong, H.; Mao, J.; Lin, T.; Wang, C.; Li, L.; and Zhou, D. 2019. Neural Logic Machines. In *ICLR*.

Esteva, F.; and Godo, L. 2001. Monoidal t-norm based logic: Towards a logic for left-continuous t-norms. *Fuzzy Sets and Systems*.

Evans, R.; and Grefenstette, E. 2018. Learning Explanatory Rules from Noisy Data. *JAIR*.

Frerix, T.; Nießner, M.; and Cremers, D. 2020. Homogeneous Linear Inequality Constraints for Neural Network Activations. In *CVPR Workshops*.

Getoor, L.; and Taskar, B. 2007. *Introduction to Statistical Relational Learning (Adaptive Computation and Machine Learning)*. The MIT Press.

Goodfellow, I.; Warde-Farley, D.; Mirza, M.; Courville, A.; and Bengio, Y. 2013. Maxout Networks. In *ICML*.

Kazemi, S. M.; and Poole, D. 2018. RelNN: A Deep Neural Model for Relational Learning. In *AAAI*.

Kok, S.; and Domingos, P. 2007. Statistical predicate invention. In *Proceedings of the 24th international conference on Machine learning*, 433–440.

Krizhevsky, A. 2010. Convolutional deep belief networks on CIFAR-10. Unpublished Manuscript.

Manhaeve, R.; Dumancic, S.; Kimmig, A.; Demeester, T.; and Raedt, L. D. 2018. DeepProbLog: Neural Probabilistic Logic Programming. *CoRR*.

Minervini, P.; Bosnjak, M.; Rocktäschel, T.; Riedel, S.; and Grefenstette, E. 2020a. Differentiable reasoning on large knowledge bases and natural language. In *AAAI*.

Minervini, P.; Riedel, S.; Stenetorp, P.; Grefenstette, E.; and Rocktäschel, T. 2020b. Learning Reasoning Strategies in End-to-End Differentiable Proving. In *ICML*.

Motzkin, T. S.; Raiffa, H.; Thompson, G. L.; and Thrall, R. M. 1953. The double description method. *Contributions to the Theory of Games*.

Muggleton, S. 1996. Learning from positive data. In *Worshop on ILP*.

Muggleton, S. H.; Lin, D.; Pahlavi, N.; and Tamaddoni-Nezhad, A. 2014. Meta-interpretive Learning: Application to Grammatical Inference. *Machine Learning*.

Nair, V.; and Hinton, G. 2010. Rectified Linear Units Improve Restricted Boltzmann Machines. In *ICML*.

Nickel, M.; Rosasco, L.; and Poggio, T. 2016. Holographic Embeddings of Knowledge Graphs. In *AAAI*.

Qu, M.; Chen, J.; Xhonneux, L.-P.; Bengio, Y.; and Tang, J. 2021. {RNNL}ogic: Learning Logic Rules for Reasoning on Knowledge Graphs. In *ICLR*.

Riegel, R.; Gray, A.; Luus, F.; Khan, N.; Makondo, N.; Akhalwaya, I. Y.; Qian, H.; Fagin, R.; Barahona, F.; Sharma, U.; Ikbal, S.; Karanam, H.; Neelam, S.; Likhyani, A.; and Srivastava, S. 2020. Logical Neural Networks. *CoRR*.

Rocktäschel, T.; and Riedel, S. 2017. End-to-End Differentiable Proving. In *NeurIPS*.

Sadeghian, A.; Armandpour, M.; Ding, P.; and Wang, D. Z. 2019. DRUM: End-To-End Differentiable Rule Mining On Knowledge Graphs. In *NeurIPS*.

Sourek, G.; Svatos, M.; Zelezny, F.; Schockaert, S.; and Kuzelka, O. 2017. Stacked Structure Learning for Lifted Relational Neural Networks. In *International Conference on Inductive Logic Programming*.

Sun, Z.; Vashishth, S.; Sanyal, S.; Talukdar, P.; and Yang, Y. 2020. A Re-evaluation of Knowledge Graph Completion Methods. In *ACL*.

Toutanova, K.; and Chen, D. 2015. Observed versus latent features for knowledge base and text inference. In *Proceedings of the 3rd workshop on continuous vector space models and their compositionality*, 57–66.

Yang, F.; Yang, Z.; and Cohen, W. W. 2017. Differentiable Learning of Logical Rules for Knowledge Base Reasoning. In *NeurIPS*.