DeepStochLog: Neural Stochastic Logic Programming

Thomas Winters,[*1] **Giuseppe Marra,**[*1] **Robin Manhaeve,**[1] **Luc De Raedt**[1,2]

[1]Department of Computer Science; Leuven.AI, KU Leuven, Belgium,
[2]AASS, Örebro University, Sweden
{thomas.winters, giuseppe.marra, robin.manhaeve, luc.deraedt}@kuleuven.be

Abstract

Recent advances in neural-symbolic learning, such as Deep-ProbLog, extend probabilistic logic programs with neural predicates. Like graphical models, these probabilistic logic programs define a probability distribution over possible worlds, for which inference is computationally hard. We propose Deep-StochLog, an alternative neural-symbolic framework based on stochastic definite clause grammars, a kind of stochastic logic program. More specifically, we introduce neural grammar rules into stochastic definite clause grammars to create a framework that can be trained end-to-end. We show that inference and learning in neural stochastic logic programming scale much better than for neural probabilistic logic programs. Furthermore, the experimental evaluation shows that DeepStochLog achieves state-of-the-art results on challenging neural-symbolic learning tasks.

Introduction

The integration of neural and symbolic learning methods is high on the research agenda. It is popular to use (variants of) logic programs to represent the available symbolic knowledge and use Prolog-like mechanisms to generate computation structures that can then be differentiated (Manhaeve et al. 2018; Dai et al. 2019; Rocktäschel and Riedel 2017; Yang, Ishay, and Lee 2020; Cohen, Yang, and Mazaitis 2020; Sourek et al. 2018; Si et al. 2019). Several of these approaches also incorporate probability into these neural logic programming models, cf. (De Raedt et al. 2020). Most notably, one distinguishes probabilistic from stochastic logic programs (PLPs vs SLPs). The more popular PLPs (De Raedt and Kimmig 2015) are based on a possible worlds semantics (i.e. the distribution semantics (Sato 1995)), which extends probabilistic graphical models, while the SLPs (Muggleton 1996, 2000; Cussens 2001) are based on stochastic grammars. The difference can also be described as a random graph vs a random walk model. So far, the emphasis in neural-symbolic computation has been on the PLP approach, especially (Yang, Ishay, and Lee 2020; Manhaeve et al. 2018; Sourek et al. 2018; Tsamoura and Michael 2020), with only Tensorlog (Cohen, Yang, and Mazaitis 2020) adopting the SLP semantics in an efficient but restricted Datalog or database setting.

---

[*]These authors contributed equally.

To fill this gap, we introduce DeepStochLog, a neural stochastic logic programming approach. It incorporates ideas from DeepProbLog, such as the neural predicate. The neural predicate encapsulates neural networks to cope with sub-symbolic data such as images. Without loss of generality, we base DeepStochLog on stochastic definite clause grammars (SDCGs) as this notation is not only easier to introduce and use, but also results in a sequence-based model. SDCGs are a kind of probabilistic unification-based grammar formalism (Have 2009). However, SDCGs and SLPs are very closely related. SDCGs can be directly translated and executed as SLPs, and all the concepts we introduce for SDCGs can directly apply to SLPs as well. More specifically, the key contributions of this paper are: 1) the introduction of the neural stochastic logic programming framework DeepStochLog; 2) the introduction of inference and learning algorithms (through gradient descent) for DeepStochLog programs; and 3) experimental results that show that DeepStochLog obtains state-of-the-art results on a number of challenging tasks for neural-symbolic computation and that it is also several orders of magnitude faster than alternative approaches based on PLPs.

Stochastic DCGs

A context-free grammar (CFG) G is a 4-tuple (V, Σ, S, R), with V the set of non-terminals, Σ the set of terminals, $S \in V$ the starting symbol and R a set of rewrite rules of the form $N \to S_1, ..., S_k$ where N is a non-terminal, the S_i are either terminals or non-terminals. A probabilistic context-free grammar (PCFG) extends a CFG by adding probabilities to the rules R, i.e., the rules take the form $p_i :: N \to S_1, ..., S_k$, where p_i is a probability. Furthermore, the probabilities of rules with the same non-terminal N on the left-hand side must sum to 1. We use list notation for sequences of terminals such as $[cat]$ and $[the, cat]$. Whereas CFGs only define whether a sequence can be parsed, PCFGs define a probability distribution over possible parses. This allows for the most likely parse to be identified. An example PCFG is shown at the top of Example 1.

Definite clause grammars (DCGs) (Pereira and Warren 1980) are a well-known logic programming-based extension of CFGs. DCGs are unification base and can represent context-sensitive grammars. They differ from CFGs in that logical atoms are used instead of the non-terminals. An *atom* $a(t_1, ..., t_n)$ consists of a predicate a of arity n followed by

n terms t_i. Terms are either constants, logical variables or structured terms of the form $f(t_1, ..., t_k)$ with f a functor and t_j terms. The production rules are called definite clause grammar rules because they can be directly translated to a set of definite clauses (i.e., Horn clauses with exactly one positive literal) and can be executed as a Prolog program using SLD-resolution. The right hand side of DCG rules are also allowed to contain queries to Prolog predicates q_i between curly brackets $\{q_1(t_{1,1}, ..., t_{1,m_1}), ..., q_n(t_{n,1}, ..., t_{n,m_n})\}$ to impose further constraints and perform additional computations during the inference process. These are to be considered atoms as well. Substitutions $\{V_1 = t_1, ..., V_n = t_k\}$ are sets of variable/term pairs. Applying a substitution to an atom a yields the atom $a\theta$ where all variables V_i have been replaced by their corresponding terms t_i. θ is a unifier of an atom s and an atom u if and only if $s\theta = u\theta$. For more information, see standard textbooks on logic programming such as (Flach 1994; Sterling and Shapiro 1994).

Stochastic definite clause grammars (SDCGs) extend DCGs by associating probabilities to the rules, just like how PCFGs extend CFGs (Have 2009). As PCFGs, SDCGs require that the sum of the probabilities for the rules defining a single non-terminal predicate equals 1. SDCGs also correspond directly to stochastic logic programs (SLP) (Cussens 2001; Muggleton 1996, 2000) which are well-known in the probabilistic (logic) programming community (De Raedt and Kimmig 2015). An example SDCG is shown in Example 1 at the bottom.

Example 1 (PCFG and SDCG). *A PCFG:*

$$0.5 :: E \rightarrow N$$
$$0.5 :: E \rightarrow E, [+], N$$
$$0.1 :: N \rightarrow [0] \quad ... \quad 0.1 :: N \rightarrow [9]$$

and a similar SDCG that constrains the result of the expression.

$$0.5 :: e(N) \rightarrow n(N)$$
$$0.5 :: e(N) \rightarrow e(N1), [+], n(N2), \{N \text{ is } N1 + N2\}$$
$$0.1 :: n(0) \rightarrow [0] \quad ... \quad 0.1 :: n(9) \rightarrow [9]$$

The inference task in (S)DCGs consists of deriving a sequence of terminals from a goal (which often captures the starting symbol of the grammar). SLD-derivations are used for this. More formally, in an SLD-derivation for a DCG, a *goal* $g_1, ..., g_n$ is a sequence where each g_i is either a logical atom (a non-terminal) or a list containing terminals or logical variables. An SLD-derivation is shown in Example 2 and uses several resolution steps. Applying *resolution* to a goal $g_1, ..., g_n$ and a definite clause grammar rule $n \rightarrow t_1, ..., t_k$ yields the goal $g_1\theta, ..., g_{i-1}\theta, t_1\theta, ..., t_k\theta, g_{i+1}\theta, ..., g_n\theta$ provided that g_i is the leftmost atom in the goal (so $g_1, ..., g_{i-1}$ are terminal symbols), θ is the unifier of g_i and n, i.e., $g_i\theta = n\theta$.[1] We will write $g_1, ..., g_n \vdash t_1\theta, ..., t_k\theta, s_2\theta, ..., s_n\theta$. A derivation $d(G)$ is then the repeated application $G \vdash G_1\theta_1 \vdash$

---

[1] When a Prolog query q is the first non-terminal to occur in a goal during the derivation process, the query is executed in Prolog possibly yielding an answer substitution θ such that $q\theta$ is true. For instance, in Example 1, there is the Prolog query N `is` $N1 + N2$

$G_2\theta_1\theta_2 \vdash ... \vdash G_n\theta_1\theta_2...\theta_n$ of such resolution steps onto a goal G. We will write $G \vdash^* G_n$. Successful derivations of a goal end in a sequence T that consists only of terminal symbols, see Example 2 for an example. We will write that $d(G) = T$ and also say that $derives(G\theta, T)$ is true, with $\theta = \theta_1\theta_2...\theta_n$ the answer substitution. A successful derivation corresponds to a proof. The set of all possible proofs can be depicted using SLD-trees, see Figure 1a.

The probability $P(d(G))$ of a derivation $d(G)$ is the product of the probabilities $\prod p_i^{m_i}$ of the rules i used in the derivation with m_i the number of times the rule i was used. An important difference between the probability of a parse in a PCFG and a derivation in an SDCG is that there can be a loss of probability mass in the latter. Derivations can fail in case there are non-terminals in the goal that do not unify with the heads of any of the rules. This is due to unification and is different from (P)CFGs, where non-terminals can always be resolved using rules for that non-terminal. There also exist non-terminating derivations. The probability $P(derives(G\theta, T))$ of answer substitutions θ and terminal sequences T for a goal G is defined as $P(derives(G\theta, T)) = \sum_{d_i(G\theta)=T} P(d_i(G\theta))$, i.e. the sum of the probabilities of all derivations for G that result in the terminal sequence T and answer substitution θ. Notice that the distribution is always relative to the goal G and sequence T. The goal can consist of one or more atoms, but in general for parsing, this will typically be the starting symbol or atom of the grammar. Notice that if there are failing derivations, the total probability mass assigned to all sequences of terminals for a goal G may be strictly less than 1. This is discussed at length by (Cussens 2001). It is possible to obtain normalized probabilities by calculating the normalization constant, but this is computationally expensive. We avoid this normalization in the present paper because in practice, the goal is often to find the most likely derivation $d_{max}(G, T) = \arg\max_{d(G)=T} P(d(G))$, non-normalized probabilities usually suffice.

Example 2 (Derivations). *Consider the following successful derivation using the SDCG in Example 1 for the goal $G = [e(X)]$, the answer substitution $\theta = \{X/2\}$ and the terminal sequence $T = [2, +, 0]$.*

$$e(X) \vdash e(N1), [+], n(N2), \{X \text{ is } N1 + N2\}$$
$$\vdash n(N1), [+], n(N2), \{X \text{ is } N1 + N2\}$$
$$\vdash [2, +], n(N2), \{X \text{ is } 2 + N2\}$$
$$\vdash [2, +, 0], \{2 \text{ is } 2 + 0\}$$
$$\vdash [2, +, 0]$$

with $\theta_1 = \{\}$, $\theta_2 = \{\}$, $\theta_3 = \{N1/2\}$ *and* $\theta_4 = \{X/2, N2/0\}$. *Moreover,* $p = 0.5 \times 0.5 \times 0.1 \times 0.1$

DeepStochLog

DeepStochLog integrates neural networks and SDCGs by introducing neural definite clause grammars (NDCG).

---

which computes N as the sum of $N1$ and $N2$. In this paper, we assume that when such a query is called there is at most one answer substitution that is true. If there were more such substitutions, we would have to introduce a probability for such substitutions in the SDCG case, which unnecessarily complicates the semantics.

More formally, DeepStochLog allows for specifying an SDCG that additionally supports *neural definite clause grammar rules*, or neural rules for short. These are statements of the form:

$$nn(m, \mathcal{I}, \mathcal{O}, \mathcal{D}) :: nt \rightarrow g_1, ..., g_n$$

where nt is an atom, $g_1, ..., g_n$ is a goal, and the $\mathcal{I} = [I_1, ..., I_m]$ and $\mathcal{O} = [O_1, ..., O_L]$ are lists of variables occurring in $g_1, ..., g_n$ and nt. The nn declaration states that m is a neural network that takes the variables $I_1, ..., I_m$ as input and produces a probability distribution over the output variables for $O_1, ..., O_L$ as output. It thus maps an input substitution σ for the variables $I_1, ..., I_m$ to a set of output substitutions θ_j with probability p_j. $\mathcal{D} = [D_1, ..., D_L]$ is a list of unary predicates, where D_i defines the domain of the output variable O_i. The neural rule serves as a template. For every input substitution σ, the template $(nt \rightarrow g_1, ..., g_n)\sigma$ defines the set of instantiated stochastic definite clause grammar rules $p_j :: (nt \rightarrow g_1, ..., g_n)\sigma\theta_j$.

Example 3 (Neural definite clause grammar rules). *Consider the SDCG in Example 1. We can substitute the $n(X) \rightarrow [X]$ rules with the following neural rule*

$$nn(mnist, [Mnist], [N], [digit]) :: n(N) \rightarrow [Mnist].$$

Here, the neural network called $mnist$ takes as input an MNIST image (Lecun et al. 1998) and returns a probability for all digits between 0 and 9, indicating its confidence for each digit. The predicate $digit$ is defined as the facts $digit(0)$ up to $digit(9)$. Given the neural network and the input substitution $\sigma = \{Mnist = \boxed{0}\}$ (which could be obtained through unification with the terminal sequence), the neural network could generate the output substitutions $\theta_0 = \{N = 0\}; ... ; \theta_9 = \{N = 9\}$; with probabilities 0.87; ... ;0.05. Thus, the neural rule with the input substitution $\sigma = \{Mnist = \boxed{0}\}$ denotes the following set of grammar rules: $0.87 :: n(0) \rightarrow \boxed{0}$; ...; $0.05 :: n(9) \rightarrow \boxed{0}$

The neural rules are reminiscent of the neural predicates in DeepProbLog (Manhaeve et al. 2018), which also encapsulate a neural network that outputs a distribution over a number of alternatives. It is worth analyzing how a neural rule behaves w.r.t the neural inputs (e.g. images). In fact, a neural rule defines a probability distribution over the values of the output variables *given* the neural inputs, whose distribution is not modeled in the program. This *conditional* setting is akin to *conditional* PCFGs (Riezler et al. 2002; Sutton and McCallum 2006) and it is a common modeling strategy in discriminative parsing. DeepStochLog could also be used to define generative grammars on subsymbolic inputs (e.g. images) if provided with neural models that can provide a joint probability distribution of both outputs and images. This is also discussed in (Manhaeve et al. 2021) but will not be further analyzed in the current paper.

Using empty production rules can be a technique for more flexible modeling. This also more clearly shows the link between SLPs and DCGs. This is illustrated in Example 4.

Example 4 (Empty productions). *We show a variant of the single digit MNIST Addition problem using empty produc-*

tions.

$$nn(mnist, [X], [Y], [digit]) :: number(X, Y) \rightarrow [].$$
$$addition(X, Y, N) \rightarrow number(X, N1),$$
$$number(Y, N2), \{N \text{ is } N1 + N2\}.$$

This grammar will always produce the empty sequence, instead of producing a sequence of MNIST digit images that must be summed. Instead, the single digit numbers are given as arguments X and Y to the goal addition predicate, which are then passed to the number predicate, in turn giving them as input to the neural network. Through the Prolog unification mechanism and the probabilistic modeling, we can thus express complex stochastic logic programs that include calls to neural networks.

Inference in DeepStochLog

The goal of the inference is to compute the probability $P(derives(G, T))$ for a given goal G and (possibly unknown) sequence T. This is divided into two steps: logical inference and probabilistic inference.

Logical inference Given a DeepStochLog program, a goal G and a (potentially unknown) sequence T, logical inference uses resolution to answer $derives(G, T)$[2]. This corresponds to finding all the possible derivations for G that result in a terminal sequence T. The resolution process is then turned into a compact AND-OR circuit, which represents all possible derivations and will be the input for the probabilistic inference. The logical inference procedure is illustrated in Figure 1. The SLD resolution tree for the given goal is at the top and its corresponding AND-OR circuit below. The translation to the AND-OR circuit is straightforward. It has exactly the same structure as the SLD-tree. For every resolution step with a rule $p_i :: r_i$, an AND node is added. Furthermore, for a normal SDCG rule, the corresponding probability p_i is added, and for a neural grammar rule, there is a call to the neural network that returns the probability p_m. Whenever there are two (or more) branches in the SLD tree for a goal, an OR node is added. Notice that all the leaves are either probabilities given as parameters or the result of a neural call.

During SLD resolution, many identical intermediate goals may be proved multiple times which results in an explosion of inference time. To avoid proving the same goals, we use SLG resolution (Chen and Warren 1996), which plays a similar role as the dynamic programming CYK algorithm for CFGs (Kasami 1966). Tabling using SLG resolution is a standard logic programming technique that memoizes the answers of predicates by tabling the evaluations. This technique is incorporated in Prolog implementations such as XSB, SWI-Prolog and Prism (Sato and Kameya 1997). The important difference with SLD resolution is that the results are not a single derivation tree, but rather a forest, where certain parts are re-used for multiple derivations thanks to tabled evaluations. The effect of tabling is carried over to the creation of the AND-OR tree. Each time a derivation is reused from the

---

[2]This is sometimes called *phrase* or *sentence* in actual Prolog implementations and it requires an automatic syntactical translation of a DCG into Prolog. We show an example in the Appendix.
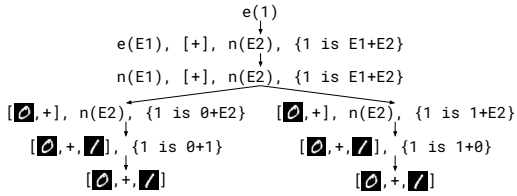
(a) The SLD tree for $derives(e(1), [\blacksquare, +, \blacksquare])$. Failing branches are omitted.
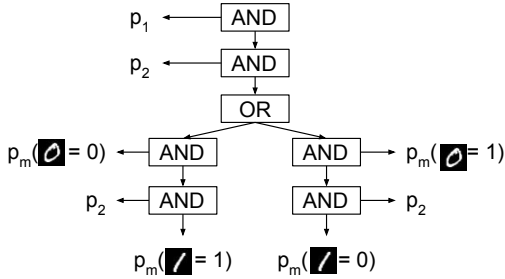
(b) AND-OR circuit for $derives(e(1), [\blacksquare, +, \blacksquare])$

Figure 1: The different steps of inference on an example grammar.

table, its corresponding node is returned and linked to the new derivation. Thus, also the AND-OR circuit turns into a forest. During training, DeepStochLog caches the AND-OR forests and is able to re-evaluate them with different parameters and inputs. This prevents unnecessary computation and makes training more tractable.

Probabilistic inference. With probabilistic inference, we refer to the task of calculating the probability:

$$P(derives(G,T)) = \sum_{d(G\theta)=T} P(d(G\theta))$$
$$= \sum_{d(G\theta)=T} \prod_{r_i \in d(G\theta)} p_i^{m_i},$$

i.e. the sum of the probabilities of all derivations for a given G that result in a given terminal sequence T and answer substitution θ. Thanks to SLG resolution and tabling, the shared sub-structure of many derivations is explicit in the AND-OR circuit obtained from the logical inference. This dissipates the need for a specialized algorithm, like the *inside* algorithm used in the probabilistic extension of CYK. Computing the probability $P(derives(G\Theta,T))$ is just a bottom-up evaluation of the AND-OR circuit where AND-nodes are substituted by multiplications and OR-nodes by summations, i.e. compiling the logical circuit to an arithmetic circuit using the $(+, \times)$ semiring (Kimmig, Van den Broeck, and De Raedt 2011). Analogously, the most probable derivation for the goal G is found with the $(\max, \times)$ semiring.

Learning in DeepStochLog

Let us consider a dataset of triples $\mathcal{D} = \{(G_i\theta_i, T_i, t_i)\}$, where G_i is a goal, θ_i a substitution for G_i, T_i a sequence

of terminals and t_i a target probability. Let us also consider a DeepStochLog program parameterized by the vector p of rule probabilities.

Learning in DeepStochLog is defined as the following optimization problem, with $\mathcal{L}$ being any differentiable loss function:

$$\min_p \sum_{(G_i\theta_i, T_i, t_i) \in \mathcal{D}} \mathcal{L}\bigg(P(derives(G_i\theta_i, T_i)), t_i\bigg) \quad (1)$$

Computing the probability in terms of an arithmetic circuit has an important advantage. In fact, the corresponding computational graph is differentiable and the derivatives of the loss function $\mathcal{L}$ w.r.t. the probabilities p can be carried out automatically using out-of-the-box differentiation frameworks. Moreover, when the probabilities p are computed by a neural network as for a neural grammar rule, the gradients can be seamlessly backpropagated to the network to train its internal parameters. We solve the learning problem using standard gradient descent techniques from deep learning, e.g. Adam (Kingma and Ba 2015).

One interesting case is when the loss function $\mathcal{L}$ is the negative log-likelihood, as it brings DeepStochLog into the standard learning scenario for probabilistic grammars, where the optimization problem is usually carried out in the expectation-maximization (EM) framework. Here, given an *inside* algorithm that computes the probability of a given input, a correspondent *outside* algorithm is designed to extract the expected counts of the various grammar rules from data (E-step) and then the counts are used to update the probabilities (M-step). Most of the developed inside-outside algorithms are tailored to a specific formalism. However, the gradient descent approach of DeepStochLog on the negative log-likelihood is actually equivalent to the EM approach but it does not require the explicit definition of the corresponding outside algorithm. In fact, the gradients obtained by the backward pass through the AND-OR circuit have been shown to actually compute the outside probabilities (E-step), while the gradient descent step is used to update the parameters (M-step) (Salakhutdinov, Roweis, and Ghahramani 2003; Berg-Kirkpatrick et al. 2010; Eisner 2016).

Code and Data

We released DeepStochLog as an installable Python package and published all code and data used evaluation tasks on https://github.com/ML-KULeuven/deepstochlog.

Evaluation

Research Questions

The goal of our experiments is to answer the following questions:

Q1 Does DeepStochLog reach state-of-the-art predictive performance on neural-symbolic tasks?

Q2 How does the inference time of DeepStochLog compare to other neural-symbolic frameworks and what is the role of tabling?

Q3 Can DeepStochLog handle larger-scale tasks?

Q4 Can DeepStochLog go beyond grammars and encode more general programs?

Tasks

We used several tasks to evaluate DeepStochLog. Complete details are specified in the appendix and the code repository. We used similar or the same architectures and hyperparameters for the neural network models as used in DeepProbLog.

T1: MNIST Addition. In the MNIST Addition task (Manhaeve et al. 2018), the model is given two sequences of length N of MNIST images containing handwritten images, each representing an N-digit number. The task is to predict the sum of these numbers. The training data only contains the two image sequences and the sum of the corresponding numbers, thus not providing the digit labels of the individual images. For example, a top level query for this task is $derives(addition(5), [\text{🖼}, +, \text{🖼}]))$. A neural rule is defined for recognizing single digits. These are then used in the rules that sum the numbers represented by these images. The datasets for each digit length use all 60K images of MNIST images exactly once.

T2: Handwritten Formulas. In the Handwritten Formulas (HWF) task, the goal is to solve mathematical expressions, where both digits and operators (addition, subtraction, multiplication and division) are images of handwritten characters. Like T1, the data of T2 only contains the outcome of the expression and the sequence of images. For example, a top level query for this task is $derives(expression(-2.5), [\text{🖼}, \text{🖼}, \text{🖼}, \text{🖼}, \text{🖼}])$. One neural rule recognizes digits, while another recognizes operators. A different rule then performs the mathematical operations to predict the outcome. For this task, we use the Handwritten Formula (HWF) dataset, introduced in (Li et al. 2020). The dataset contains 10000 expressions of lengths 1, 3, 5 and 7. Unlike the original paper, we do not consider a curriculum learning setting here, and split the dataset into 4 separate parts by length.

T3: Well-formed Parentheses. We introduce the Well-formed Parentheses task, where the model is asked to recognize image sequences that represent well-formed parentheses. Well-formed parentheses is a classic context-free grammar language where $\Sigma = \{(,)\}$, and $R = \{s \rightarrow ()|(s)|ss\}$, i.e. all open brackets are closed in the right order. As images, we use the zeros from MNIST as "(" and ones as ")", and generate 1000 well-formed parenthesis sequences without labels as training data. The goal is to predict the most probable parse of the sequence.

T4: Context-Sensitive Grammar. Since DCGs support context-sensitive grammars, we created a dataset of 2000 image sequences representing the canonical context-sensitive grammar $a^n b^n c^n$. Since each sequence of length $3n$ only has one valid parse, we increased the difficulty by allowing permutations such as $b^n a^n c^n$. We also generated 2000 negative examples, i.e. random sequences of the form $a^k b^l c^m$, and permutations like $c^k a^l b^m$ such that k, l, m are all larger than 1, sum to a multiple of 3 and are not all the same number. The goal of the task is to predict whether the input image sequence belongs to the first grammar or the second.

T5: Semi-supervised classification in citation networks. Given a set of scientific papers represented as bag-of-words and their citation network, the goal is to assign the correct class to a large test set of documents by having access only to the true labels of a small training set. The intuition is that one can infer the class of a paper not only by the features of the document but also by the class of its neighbors. This task is interesting from a neural-symbolic perspective because one must be able to use both the features of the documents and the symbolic network. Two well-known datasets for this task are the Cora (2708 nodes and 5429 edges) and Citeseer (3327 nodes and 9228 edges) (Sen et al. 2008).

T6: Word Algebra Problems. In this task, a natural language text describes a word algebra problem (e.g, *"Mark has 6 apples. He eats 2 and divides the remaining among his 2 friends. How many apples did each friend get?"*). This dataset of this task contains 300 training instances and was introduced in (Roy and Roth 2015). Each text contains 3 numbers, and all numbers have to be used exactly once in a formula containing addition subtraction, multiplication and division. The task is to predict the right numerical answer to the expression implied by textual description.

Results

For experiments **T1**, **T2** and **T3**, we report the mean parse accuracy. For experiments **T4**, **T5** and **T6**, we report the mean classification accuracy. For all experiments, we report the standard deviation over 5 runs. We report "timeout" if a single of these 5 runs took more than 1 hour to execute.

Q1: Performance of DeepStochLog We first investigate whether DeepStochLog achieves state-of-the-art results compared to similar neural-symbolic frameworks. Table 1 shows the result for the MNIST addition task (**T1**), for training and testing on lengths 1 to 4. It shows that DeepStochLog performs similarly to DeepProbLog (Manhaeve et al. 2018) and NeurASP (Yang, Ishay, and Lee 2020), but scales to larger sequences. Table 2 shows the result on the HWF task (**T2**). DeepStochLog performs similar to NGS and DeepProbLog for expressions of length 1 and 3. Starting from expression length 5, it becomes infeasible to train DeepProbLog. NGS (Li et al. 2020) can still be trained, but for expression length 7, some runs fail to converge. DeepStochLog, however, performs well for all expression lengths. Table 3 shows the accuracy on task **T3**. Here we can see that both DeepStochLog and DeepProbLog achieve high accuracy, but DeepStochLog reaches a slightly higher accuracy for a longer length. For task **T6**, DeepStochLog and DeepProbLog achieve a similar accuracy of 94.8 ± 1.1 and 94.2 ± 1.4 respectively. We also compare to $\delta 4$ (Bošnjak, Rocktäschel, and Riedel 2017), but the authors only report the maximum accuracy reached. For all three frameworks, the max accuracy reached is 96.0. To conclude, DeepStochLog is able to achieve similar or better performance compared to other state-of-the-art neural-symbolic frameworks.

Q2: DeepStochLog scalability We now investigate whether DeepStochLog is more scalable than similar neural-symbolic frameworks. First, we observe that in the tasks **T1**, **T2**, **T4** and **T5**, DeepStochLog scales to settings or datasets that are infeasible for the competitors. Next, in Table 7, we compare the execution times for inference in task **T1**. We randomly selected 100 queries from the training data and we computed the average time required from the system to

	Number of digits per number (N)			
	1	2	3	4
NA	97.3 ± 0.3	93.9 ± 0.7	timeout	timeout
DPL	97.2 ± 0.5	95.2 ± 1.7	timeout	timeout
DSL	97.9 ± 0.1	96.4 ± 0.1	94.5 ± 1.1	92.7 ± 0.6

Table 1: The test accuracy (%) on the MNIST addition (**T1**) for NeurASP (NA), DeepProbLog (DPL) and Deep-StochLog(DSL).

	Expression length			
	1	3	5	7
NGS	90.2 ± 1.6	85.7 ± 1.0	91.7 ± 1.3	20.4 ± 37.2
DPL	90.8 ± 1.3	85.6 ± 1.1	timeout	timeout
DSL	90.8 ± 1.0	86.3 ± 1.9	92.1 ± 1.4	94.8 ± 0.9

Table 2: The accuracy (%) on the HWF dataset (**T2**) for Neuro-Symbolic Grammars (NGS), DeepProbLog (DPL) and DeepStochLog (DSL).

	Maximum expression length		
	10	14	18
DPL	100.0 ± 0	99.4 ± 0.5	99.2 ± 0.8
DSL	100.0 ± 0	100.0 ± 0	100.0 ± 0

Table 3: The parse accuracy (%) on the well-formed parentheses dataset (**T3**) for DeepProbLog (DPL) and DeepStochLog (DSL).

	Expression length		
	3-12	3-15	3-18
DPL	99.8 ± 0.3	timeout	timeout
DSL	99.4 ± 0.5	99.2 ± 0.4	98.8 ± 0.2

Table 4: The accuracy (%) on the $a^n b^n c^n$ dataset (**T4**) for DeepProbLog (DPL) and DeepStochLog (DSL).

compute the probability of the query. We repeated the experiment for increasing number lengths. DeepStochLog shows a huge gap over the competitors, especially for large numbers. This is due to the fact that DeepStochLog is based on a random walk semantics which is computationally cheaper than the possible world semantics exploited by DeepProbLog and NeurASP.

Another speedup in DeepStochLog is due to it being natively implemented on top of SLG resolution and tabling, which plays a fundamental role in compactly representing derivations and SLD-trees. We analyzed the impact of tabling in Table 6, where we show the comparison between SLD and SLG resolution in DeepStochLog. In particular, we compared the resolution time required to find all the possible answers for expressions of variable lengths (on task **T2**).

Q3: Larger scale relational datasets The complexity of many of the previous experiments comes from the large number of derivations for a single goal, while the number of subsymbolic inputs (e.g. images) in a single relational example was quite limited. Here, we focus on task **T5**, i.e. semi-supervised classification in citation networks, where the complexity mainly comes from the large number of elements of the unique relational example, i.e. the citation network. This task is usually out of the scope of (neural) PLP approaches due to the fact that there is a unique large relational example and the possible world semantics is prohibitive in this scenario. We compare against the following baselines: label propagation (LP) (Zhu, Ghahramani, and Lafferty 2003), semi-supervised embedding (SemiEmb) (Weston et al. 2012), manifold regularization (ManiReg) (Belkin, Niyogi, and Sindhwani 2006), skip-gram based graph embeddings (DeepWalk) (Perozzi, Al-Rfou, and Skiena 2014), ICA (Lu and Getoor 2003) and GCN (Kipf and Welling 2017). All these baselines are specific to the semi-supervised classification task, while DeepStochLog is a much more general framework. We finally tried to compare with DeepProbLog,

which, however, does not scale to the size of this problem due to the different probabilistic semantics. Results are reported in Table 5. DeepStochLog compares similarly or favorably to most of the other methods, even though it is the only one that has not been developed for the specific task. However, it still underperforms w.r.t. ICA and GCN. But these methods use extra knowledge as input to the classifier in the form of precomputed or learned features of the neighbors of a document, which is very useful for this task but not considered in the DeepStochLog experiment. Adding or learning relational features for input to the neural modules is, however, an interesting future direction.

Q4: General programs in DeepStochLog Even though DeepStochLog naturally represents grammars for parsing sequences, NDCGs with Prolog goals are a powerful formalism to express more complex relational problems and programs. Actually, both task **T5** and **T6** have been solved with programs that depart from the pure grammar formalism and are more like general logic programs. We provide the complete models in the appendix. The main ingredients are (neural) empty production rules, sometimes referred to as non-consuming or ϵ-production rules. They allow to take probabilistic decisions, including also neural networks, without consuming any element of the sequence, as shown in Example 4. This also shows that DeepStochLog has the full power of stochastic logic programs.

Related Works

DeepStochLog is an expressive neural-symbolic framework whose distinguishing features are: 1) it is based on the expressive stochastic logic programming paradigm, which can express some probabilistic programs (as in T5-T6) as well as probabilistic unification based grammars (T1-T4); 2) it can work with both symbolic and subsymbolic data such as images (as shown in T1-T4); and 3) its inference and learning mechanism is based on SLG-resolution that naturally supports tabling, a form of dynamic programming (see Q2).

There are several strands of related research. First, Deep-

Method	Citeseer	Cora
ManiReg	60.1	59.5
SemiEmb	59.6	59.0
LP	45.3	68.0
DeepWalk	43.2	67.2
ICA	69.1	75.1
GCN	70.3	81.5
DeepProbLog	timeout	timeout
DeepStochLog	65.0	69.4

Table 5: Accuracy (%) of the classification on the test nodes on task **T5**

Lengths	# Answers	No Tabling	Tabling
1	10	0.067	0.060
3	95	0.081	0.096
5	1066	3.78	0.95
7	10386	30.42	10.95
9	68298	1494.23	132.26
11	416517	timeout	1996.09

Table 6: Parsing time in seconds (**T2**). Comparison of the DeepStochLog with and without tabling (SLD vs SLG resolution).

StochLog is a neural logic programming language in the spirit of DeepProbLog (Manhaeve et al. 2018), NeurASP (Yang, Ishay, and Lee 2020), the neural theorem prover (Rocktäschel and Riedel 2017) and lifted relational neural networks (LRNNs) (Sourek et al. 2018). The first two systems are based on a probabilistic possible world semantics, while DeepStochLog is based on stochastic grammars, which – as we have shown – scales much better (in part also due to the use of tabling). The latter two approaches focus on Datalog (which cannot deal with function symbols) and use the logic to construct the neural network in a kind of knowledge based model construction approach. Furthermore, they are neither probabilistic nor do they deal with subsymbolic inputs such as images. Another related system is Tensorlog (Cohen, Yang, and Mazaitis 2020), which is based on stochastic logic programming. While sharing their roots in SLPs, it is less expressive than DeepStochLog, as it considers only Datalog and predicates of arity 2. While Tensorlog's implementation is fast thanks to being in terms of tensors, it has only been applied to symbolic data.

Second, DeepStochLog can be viewed as a neural-based grammar, similarly to Neural Grammars (Dyer et al. 2016) and NGS (Li et al. 2020). Neural Grammars have been introduced in the natural language community as an effective strategy to learn PCFGs. They are neural parameterizations of PFCG and it is possible to learn the structure of the grammar by enumerating a set of candidate rules and using neural networks to learn their probabilities. Differently from DeepStochLog, they are restricted to context-free grammars. Furthermore, Neural Grammars (Dyer et al. 2016) do not consider subsymbolic inputs (as in all our tasks T1-T6). Different

	Number length			
	1	2	3	4
NA	9.2 ± 1.4	85.7 ± 22.6	158.2 ± 47.7	timeout
DPL	13.5 ± 3.0	36.0 ± 0.5	199.7 ± 14.0	timeout
DSL	1.3 ± 0.9	2.3 ± 0.4	4.0 ± 0.4	5.7 ± 1.8

Table 7: Inference times in milliseconds for NeurASP (NA), DeepProbLog (DPL) and DeepStochLog (DSL) on task **T1** for variable number lengths.

from the probabilistic interface of DeepStochLog, NGS uses backsearch, a greedy search that defines the backward feedback from the grammar to the neural nets. While this makes NGS very scalable, the backsearch must be defined *per-program*, while DeepStochLog backpropagates the backward feedback automatically through any NDCG. (Mukherjee et al. 2021) integrates attribute grammars with neural networks. While this is also an expressive grammatical framework, the two approaches and their applications are quite different and it has not been applied to subsymbolic data.

Third, many systems in the neural-symbolic community (Donadello, Serafini, and d'Avila Garcez 2017; Diligenti, Gori, and Sacca 2017; Sourek et al. 2018) obtain differentiable logics by relaxing logical programs or theories using fuzzy logic and t-norms. While the shift in semantics from probabilistic to fuzzy logic has known issues (van Krieken, Acar, and van Harmelen 2020), fuzzy logic allows for more scalable systems as compared to probabilistic logic based on the possible world semantics. But by exploiting the stochastic grammars, DeepStochLog shows the same benefits as fuzzy logic in terms of computational complexity (i.e. no disjoint-sum problem required) by resorting to an alternative probabilistic semantics.

Conclusions

We have introduced a novel and very expressive neural-symbolic model based on stochastic logic programming, that allows to integrate symbolic knowledge with sub-symbolic representations, that scales well, and gives state-of-the-art results on a wide range neural-symbolic computation tasks.

There are several limitations of DeepStochLog that we want to explore in further research. First, DeepStochLog does not yet learn the structure of the rules, while the neural theorem prover (Rocktäschel and Riedel 2017), DiffLog (Si et al. 2019) and the neural grammars (Dyer et al. 2016) can all enumerate rules and then identify the most relevant ones. Second, DeepStochLog's inference could be further optimised by parallelization of the circuit using ideas from TensorLog (Cohen, Yang, and Mazaitis 2020). Third, SLPs and hence, DeepStochLog, may lose probability mass due to failing derivations. This can be addressed by normalizing and computing the partition function (Cussens 2001). It would be interesting to approximate the partition function and also to further speed up the inference by sampling or by searching for the k-best derivations. Finally, it would be interesting to explore the use of DeepStochLog as a generative model to generate sequences.

Appendix

Data and Licenses   For tasks T1, T3 and T4, we used the MNIST dataset from (Lecun et al. 1998) to generate new datasets. The MNIST dataset itself was released under the Creative Commons Attribution-Share Alike 3.0 license. We distribute our new datasets built on top of MNIST and the corresponding generating code under the Apache License 2.0 on https://github.com/ML-KULeuven/deepstochlog/releases/tag/0.0.1. The Handwritten Formula Recognition (HWF) dataset (used in T2) originates from (Li et al. 2020). The Cora and Citeseer datasets (T5) are form Sen et al. (2008). The dataset of the Word Algebra Problem (T6) originates from (Roy and Roth 2015).

Computational Details   Inference time experiments are all executed on a MacBookPro 13 2020 (2.3 GHz Quad-Core Intel Core i7 and 16 GB 3733 MHz LPDDR4).

Task Details

In this section we provide additional experimental details for all tasks and show DeepStochLog programs for selected tasks. The full implementation of these tasks and data are also available on the DeepStochLog repository: https://github.com/ML-KULeuven/deepstochlog/tree/main/examples.

MNIST Digit Addition   For the MNIST addition problem, we trained the model for 25 epochs using the Adam optimizer with a learning rate of 0.001 and used 32 training terms in each batch for each digit length. It is modeled in Listing 1 for single-digit numbers and Listing 2 for any number of digits.

Handwritten Mathematical Expressions   For the Handwritten Mathematical Expression problem, we trained the model for 20 epochs using the Adam optimizer with a learning rate of 0.003 and a batch size of 2. We used two separate, similar neural networks for recognising the numbers and the operators. The encoder of the neural networks has a convolutional layer with 1 input channel, 6 output channels, kernel size 3, stride 1, padding 1, a ReLu, max pooling with kernel size 2, a convolutional layer with 6 inputs, 16 outputs, kernel size 3, stride 1, padding 1, a ReLu, a max pooling with kernel size 2, and a 2d dropout layer with $p = 0.4$. They then use two fully connected layers, one from 1936 to 128 with a ReLu, and one linear to the number of classes (10 and 4 respectively).

Well-Formed Parentheses   We ran the well-formed parenthesis problem for 1 epoch using the Adam optimizer with a learning rate of 0.001 and a batch size of 4. This problem is modelled in Listing 3.

Listing 1: DeepStochLog program for single-digit numbers

```
1  digit(Y) :- member(Y
       ,[0,1,2,3,4,5,6,7,8,9]).
2  nn(number, [X],[Y],[digit]) ::
3        number(Y) --> [X].
4  addition(N) --> number(N1), number(N2),
5        {N is N1+N2}.
```

Listing 2: DeepStochLog program for L-long digits numbers

```
1  digit(Y) :- member(Y
       ,[0,1,2,3,4,5,6,7,8,9]).
2  nn(number, [X],[Y],[digit]) ::
3        number(Y) --> [X].
4  addition(N) --> number(N1), number(N2),
5        {N is N1+N2}.
6  0.5::multi_addition(N,1)-->addition(N).
7  0.5::multi_addition(N,L)-->{L>1, L2 is L
       -1}, addition(N1), multi_addition(N2,
       L2), {N is N1*(10**L2) + N2}.
```

Listing 3: DeepStochLog program for well-formed parentheses

```
1  bracket_d(Y) :- member(Y,["(",")"]).
2  s_switch_d(Y) :- member(Y,[0,1,2]).
3
4  nn(bracket_nn,[X], [Y], [bracket_d]) ::
5        bracket(Y) --> [X].
6  nn(s_nn,[],[Y],[s_switch_d])::
7        s --> s_switch(Y).
8  0.33 :: s_switch(0) --> s, s.
9  0.33 :: s_switch(1) -->
10        bracket("("), s, bracket(")").
11 0.33 :: s_switch(2) -->
12        bracket("("), bracket(")").
```

Context-Sensitive Grammar   The model was trained on the canonical context-sensitive grammar $a^n b^n c^n$ problem for 1 epoch using the Adam optimizer with a learning rate of 0.001 and batch size 4. It is modeled in Listing 4.

Semi-Supervised Classification in Citation Networks   The model was trained on the Cora and Citeseer datasets using the Adam optimizer with a learning rate of 0.01. We trained for 100 epochs and selected the model corresponding to the epoch with maximum accuracy on the validation set.

The program simply states that a document X is of class Y (and produce the terminal Y) either if a neural network classifies it so or if it is cited by one or more documents of the same class. Even if strange, many of these citation networks are actually cyclic. Therefore, we limited the depth of the derivations to a maximum value to avoid having a cyclical program. We also experimented with a variant of the program in which the probabilities of the rules expanding the

Listing 4: DeepStochLog program for the $a^n b^n c^n$ problem

```
1  rep_d(Y):- member(Y, [a,b,c]).
2  0.5:: s(0) --> akblcm(K,L,M),
3        {K\=L; L\=M; M\=K}.
4  0.5:: s(1) --> akblcm(N,N,N).
5  akblcm(K,L,M) --> rep(K,A), rep(L,B),
       rep(M,C), {A\=B, B\=C, C\=A}.
6  0.5 :: rep(s(0), C) --> terminal(C).
7  0.5 :: rep(s(N), C) --> terminal(C),
8        rep(N,C).
9  nn(mnist, [X], [C], [rep_d]) ::
10        terminal(C) --> [X].
```

10097

Task	Network	Architecture
T1	number	MNISTConv, **Lin(120)**, **Lin(84)**, Lin(10)
T2	number	**Conv(6, 3)**, MaxPool(2), **Conv(16,3)**, MaxPool(2), Dropout(0.4), **Lin(128)**, Lin(10)
	operator	**Conv(6, 3)**, MaxPool(2), **Conv(16,3)**, MaxPool(2), Dropout(0.4), **Lin(128)**, Lin(4)
T3	bracket_nn	MNISTConv, **Lin(120)**, **Lin(84)**, Lin(2)
T4	mnist	MNISTConv, **Lin(120)**, **Lin(84)**, Lin(3)
T5	classifier-Citeseer	**Lin(50)**, Lin(6)
	classifier-Cora	**Lin(50)**, Lin(7)
T6	RNN	Embedding(256), BiGRU(512), Dropout(0.5)*
	nn_permute	Lin(6)
	nn_op1	Lin(4)
	nn_swap	Lin(2)
	nn_op2	Lin(4)

MNISTConv: Conv(6,5), **MP(2,2)**, Conv(16,5), **MP(2,2)***

AlexNetConv: **Conv(64, 11, 2,2)**, MP(3,2), **Conv(192, 5, 2)**, MP(3,2), **Conv(384, 3, 1)**, **Conv(256, 3, 1)**, **Conv(256, 3, 1)**, MP(3,2)*

* Does not end with a Softmax layer.

Table 8: Overview of the neural network architectures used in the experiments.

Listing 5: DeepStochLog program for semi-supervised citation classification

```
1   class(Y) :- member(Y,
2          [class0, class1, ..., class6]).
3   nn(classifier,[X],[Y],[class]) ::
4          doc_neural(X,Y) --> [].
5   % Ni is the number of cite(i,X).
6   1/Na :: cite(a,b) --> [].
7   1/Na :: cite(a,c) --> [].
8   1/Nb :: cite(b,d) --> [].
9   ...
10  0.5::doc(X,Y)-->doc_neural(X,Y).
11  0.5::doc(X,Y)-->cite(X, X1),doc(X1,Y).
12  s(X) --> doc(X,Y), [Y].
```

doc predicate are trained separately for each class Y.

Word Algebra Problem We trained the model on the Word Algebra Problem for 40 epochs using the Adam optimizer with a learning rate of 0.001 and a batch size of 32.

Neural Network Architectures Table 8 summarizes the neural network architectures used in the experiment. *Conv(o,k)* is a convolutional layer with o output channels and kernel size k. *Lin(n)* is a fully-connected layer of size n. *(Bi)GRU(h)* is a single-layer (bi-directional) GRU with a hidden size h. *MaxPool(k)* is a max-pooling layer with kernal size k, and *Dropout(p)* a dropout layer with probability p. A layer in bold means it is followed by a ReLU activation function. All neural networks end with a Softmax layer, unless otherwise specified.

Translation Example of Neural Definite Clause Grammar to Prolog

The program of Listing 6 is translated to Prolog as shown in Listing 7. Both the calls to nn and p are considered always true during logical inference. During evaluation of

Listing 6: DeepStochLog program for parsing sums and subtractions

```
1   dig(Y):-member(Y,[0,1,2,3,4,5,6,7,8,9]).
2   op(Y) :- member(Y, [+,-]).
3   nn(mnist,[I],[N],[dig]) :: n(N)-->[I].
4   nn(operator,[I],[N],[op]) :: o(N)-->[I].
5   0.33::e(N) --> n(N).
6   0.33::e(S) --> e(E1), o(+), n(E2),
7          {S is E1 + E2}.
8   0.33::e(S) --> e(E1), o(-), n(E2),
9          {S is E1 - E2}.
```

Listing 7: Translated addition and substraction program to Prolog

```
1   dig(Y):-member(Y,[0,1,2,3,4,5,6,7,8,9]).
2   op(Y) :- member(Y, [+,-]).
3   n(N, [I | X], [X]) :- nn(mnist,[I],[N]),
4                   dig(X).
5   n(N, [I | X], [X]) :- nn(mnist,[I],[O]),
6                   op(O).
7   e(N, A, B) :- n(N, A, B), p(0.33).
8   e(S, A,D) :- e(E1, A, B), o(+, B, C),
9     n(E2, C, D), S is E1 + E2, p(0.33).
10  e(S, A,D) :- e(E1, A, B), o(-, B, C),
11    n(E2, C, D), S is E1 - E2, p(0.33).
```

the arithmetic circuit, nn(mnist,[I],[O]) returns the probability of the output O when provided with the image I as input, while p(x) constantly returns the probability x. It is easy to see that any derivation of the Prolog program always end with either a nn or a p call, which constitute the leaves of the correspondent arithmetic circuit.

Acknowledgements

References

Belkin, M.; Niyogi, P.; and Sindhwani, V. 2006. Manifold regularization: A geometric framework for learning from labeled and unlabeled examples. *Journal of machine learning research*, 7(11).

Berg-Kirkpatrick, T.; Bouchard-Côté, A.; DeNero, J.; and Klein, D. 2010. Painless unsupervised learning with features. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, 582–590.

Bošnjak, M.; Rocktäschel, T.; and Riedel, S. 2017. Programming with a differentiable forth interpreter. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70, 547–556.

Chen, W.; and Warren, D. S. 1996. Tabled Evaluation with Delaying for General Logic Programs. *J. ACM*, 43(1): 20–74.

Cohen, W. W.; Yang, F.; and Mazaitis, K. R. 2020. TensorLog: A Probabilistic Database Implemented Using Deep-Learning Infrastructure. *Journal of Artificial Intelligence Research*, 67: 285–325.

Cussens, J. 2001. Parameter estimation in stochastic logic programs. *Machine Learning*, 44(3): 245–271.

Dai, W.-Z.; Xu, Q.; Yu, Y.; and Zhou, Z.-H. 2019. Bridging Machine Learning and Logical Reasoning by Abductive Learning. In Wallach, H.; Larochelle, H.; Beygelzimer, A.; d'Alché-Buc, F.; Fox, E.; and Garnett, R., eds., *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

De Raedt, L.; Dumančić, S.; Manhaeve, R.; and Marra, G. 2020. From Statistical Relational to Neuro-Symbolic Artificial Intelligence. In Bessiere, C., ed., *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, 4943–4950. International Joint Conferences on Artificial Intelligence Organization. Survey track.

De Raedt, L.; and Kimmig, A. 2015. Probabilistic (logic) programming concepts. *Machine Learning*, 100(1): 5–47.

Diligenti, M.; Gori, M.; and Sacca, C. 2017. Semantic-based regularization for learning and inference. *Artificial Intelligence*, 244: 143–165.

Donadello, I.; Serafini, L.; and d'Avila Garcez, A. 2017. Logic Tensor Networks for Semantic Image Interpretation. In Sierra, C., ed., *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, 1596–1602. ijcai.org.

Dyer, C.; Kuncoro, A.; Ballesteros, M.; and Smith, N. A. 2016. Recurrent Neural Network Grammars. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 199–209.

Eisner, J. 2016. Inside-outside and forward-backward algorithms are just backprop (tutorial paper). In *Proceedings of the Workshop on Structured Prediction for NLP*, 1–17.

Flach, P. A. 1994. *Simply Logical Intelligent Reasoning by Example*. John Wiley & Sons, Inc.

Have, C. T. 2009. Stochastic definite clause grammars. In *Proceedings of the International Conference RANLP-2009*, 139–143.

Kasami, T. 1966. An efficient recognition and syntax-analysis algorithm for context-free languages. *Coordinated Science Laboratory Report no. R-257*.

Kimmig, A.; Van den Broeck, G.; and De Raedt, L. 2011. An algebraic Prolog for reasoning about possible worlds. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 25.

Kingma, D. P.; and Ba, J. 2015. Adam: A Method for Stochastic Optimization. In *Proceedings of the 3rd International Conference on Learning Representations*.

Kipf, T. N.; and Welling, M. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*.

Lecun, Y.; Bottou, L.; Bengio, Y.; and Haffner, P. 1998. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, 2278–2324.

Li, Q.; Huang, S.; Hong, Y.; Chen, Y.; Wu, Y. N.; and Zhu, S.-C. 2020. Closed Loop Neural-Symbolic Learning via Integrating Neural Perception, Grammar Parsing, and Symbolic Reasoning. In *International Conference on Machine Learning*, 5884–5894. PMLR.

Lu, Q.; and Getoor, L. 2003. Link-based Classification. In Fawcett, T.; and Mishra, N., eds., *Machine Learning, Proceedings of the Twentieth International Conference (ICML 2003), August 21-24, 2003, Washington, DC, USA*, 496–503. AAAI Press.

Manhaeve, R.; Dumančić, S.; Kimmig, A.; Demeester, T.; and De Raedt, L. 2018. Deepproblog: Neural probabilistic logic programming. *Advances in Neural Information Processing Systems*, 31: 3749–3759.

Manhaeve, R.; Dumančić, S.; Kimmig, A.; Demeester, T.; and De Raedt, L. 2021. Neural probabilistic logic programming in DeepProbLog. *Artificial Intelligence*, 298: 103504.

Muggleton, S. 1996. Stochastic logic programs. *Advances in inductive logic programming*, 32: 254–264.

Muggleton, S. 2000. Learning stochastic logic programs. *Electronic Transactions on Artificial Intelligence*, 4(B): 141–153.

Mukherjee, R.; Chaudhari, D.; Amodio, M.; Reps, T.; Chaudhuri, S.; and Jermaine, C. 2021. Neural Attribute Grammars for Semantics-Guided Program Generation. arXiv:1705.09231.

Pereira, F. C.; and Warren, D. H. 1980. Definite clause grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks. *Artificial intelligence*, 13(3): 231–278.

Perozzi, B.; Al-Rfou, R.; and Skiena, S. 2014. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 701–710.

Riezler, S.; King, T. H.; Kaplan, R. M.; Crouch, R.; Maxwell III, J. T.; and Johnson, M. 2002. Parsing the Wall Street Journal using a Lexical-Functional Grammar and Discriminative Estimation Techniques. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, 271–278. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics.

Rocktäschel, T.; and Riedel, S. 2017. End-to-end Differentiable Proving. In *Advances in Neural Information Processing Systems*, volume 30, 3788–3800.

Roy, S.; and Roth, D. 2015. Solving General Arithmetic Word Problems. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, 1743–1752.

Salakhutdinov, R.; Roweis, S. T.; and Ghahramani, Z. 2003. Optimization with EM and expectation-conjugate-gradient. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, 672–679.

Sato, T. 1995. A Statistical Learning Method for Logic Programs with Distribution Semantics. In Sterling, L., ed., *Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming, Tokyo, Japan, June 13-16, 1995*, 715–729. MIT Press.

Sato, T.; and Kameya, Y. 1997. PRISM: a language for symbolic-statistical modeling. In *IJCAI*, volume 97, 1330–1339.

Sen, P.; Namata, G.; Bilgic, M.; Getoor, L.; Galligher, B.; and Eliassi-Rad, T. 2008. Collective classification in network data. *AI magazine*, 29(3): 93–93.

Si, X.; Raghothaman, M.; Heo, K.; and Naik, M. 2019. Synthesizing Datalog Programs using Numerical Relaxation. In Kraus, S., ed., *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, 6117–6124. ijcai.org.

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

Sterling, L.; and Shapiro, E. Y. 1994. *The art of Prolog: advanced programming techniques*. MIT press.

Sutton, C.; and McCallum, A. 2006. An introduction to conditional random fields for relational learning. *Introduction to statistical relational learning*, 2: 93–128.

Tsamoura, E.; and Michael, L. 2020. Neural-Symbolic Integration: A Compositional Perspective. *CoRR*, abs/2010.11926.

van Krieken, E.; Acar, E.; and van Harmelen, F. 2020. Analyzing Differentiable Fuzzy Implications. In Calvanese, D.; Erdem, E.; and Thielscher, M., eds., *Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning, KR 2020, Rhodes, Greece, September 12-18, 2020*, 893–903.

Weston, J.; Ratle, F.; Mobahi, H.; and Collobert, R. 2012. Deep learning via semi-supervised embedding. In *Neural networks: Tricks of the trade*, 639–655. Springer.

Yang, Z.; Ishay, A.; and Lee, J. 2020. Neurasp: Embracing neural networks into answer set programming. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI*, 1755–1762.

Zhu, X.; Ghahramani, Z.; and Lafferty, J. D. 2003. Semi-supervised learning using gaussian fields and harmonic functions. In *Proceedings of the 20th International conference on Machine learning (ICML-03)*, 912–919.