DeepStochLog: Neural Stochastic Logic Programming*

Thomas Winters[1,3], Giuseppe Marra[1,3], Robin Manhaeve[1] and Luc De Raedt[1,2]

[1]KU Leuven, Dept. of Computer Science; Leuven.AI, B-3000 Leuven, Belgium
[2]AASS, Örebro University, Sweden
[3]The authors contributed equally to this paper.

Abstract

Recent advances in neural symbolic learning, such as DeepProbLog, 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 DeepStochLog, an alternative neural symbolic framework based on stochastic definite clause grammars, a type of stochastic logic program, which defines a probability distribution over possible derivations. 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.

Keywords

Neural Symbolic Learning, Stochastic Logic Programming, Definite Clause Grammar, Probabilistic Logic

1. 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 [1, 2, 3, 4, 5, 6, 7]. Several of these approaches also incorporate probability into these neural logic programming models, cf. [8]. There are several ways to incorporate probability into logic programs [9]. Most notably, one distinguishes probabilistic from stochastic logic programs (PLPs vs SLPs). The more popular PLPs are based on a possible worlds semantics (the so-called distribution semantics), which extends probabilistic graphical models, while the SLPs 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 neurosymbolic computation has been on the PLP approach, especially [4, 1, 6, 10], with only Tensorlog [5] adopting the SLP semantics in an efficient but restricted Datalog setting.

NESY'20/21 @ IJCLR: 15Th International Workshop on Neural-Symbolic Learning Reasoning
thomas.winters@kuleuven.be (T. Winters); giuseppe.marra@kuleuven.be (G. Marra); robin.manhaeve@kuleuven.be (R. Manhaeve); luc.deraedt@kuleuven.be (L. De Raedt)
© 2021 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

* This is a shortened version of the ArXiv paper: https://arxiv.org/abs/2106.12574

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, for instance, with subsymbolic 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 [11]. 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.

2. 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 $N \rightarrow S_1, ..., S_k$ 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 $p_i :: N \rightarrow S_1, ..., S_k$, where p_i is a probability. Furthermore, the sum of the probabilities of rules with the same non-terminal N on the left-hand side equals 1. We use list notation for sequences of terminals such as $[cat]$ and $[the, cat]$. Whereas CFGs 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, representing single digit additions, is shown on the left of Example 1.

Definite clause grammars (DCGs) are a well-known logic programming-based extension of CFGs [12]. DCGs can represent context-sensitive languages and are unification-based. 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. DCGs use substitutions $\{V_1 = t_1, ..., V_n = t_k\}$, which are sets of variable/term pairs, to unify atoms with heads from the rules. 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 [13, 14].

Stochastic definite clause grammars (SDCGs) extend DCGs by associating probabilities to the rules, just like how PCFGs extend CFGs [11]. Like 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) [15, 16, 17] which are well-known in the probabilistic (logic) programming community [9]. An example SDCG is shown in Example 1 on the right.

Example 1. A PCFG (left) and a similar SDCG(right) that constrains the result of the expression.

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

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) through SLD-resolution, as we show in Example 2. 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 and 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., $g_i\theta = n\theta$. We write this as $g_1, ..., g_n \vdash t_1\theta, ..., t_k\theta, s_2\theta, ..., s_n\theta$. A derivation $d(S)$ is then the repeated application $G \vdash G_1\theta_1 \vdash G_2\theta_1\theta_2 \vdash ... \vdash G_n\theta_1\theta_2...\theta_n$ 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 $d(S) = T$ and also say that $derives(S\theta, T)$ 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. This in turn lets us define the probability $P_G(derives(G, T))$ of a terminal sequence T relative to the goal G as $\sum_{d_i(G\theta)=T} P_G(d_i(G\theta))$, i.e. the sum of the probabilities of all derivations $G\theta$ for G that result in the terminal sequence T with answer substitution θ. 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 whenever a derivation fails. Derivations can fail when 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. Non-terminating derivations can also lead to loss of probability mass.

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 $T = [2, +, 0]$.

$e(X) \vdash e(N1), [+], n(N2), \{X \text{ is } N1 + N2\}$	$\theta_1 = \{\}$	$p = 0.5$
$\vdash n(N1), [+], n(N2), \{X \text{ is } N1 + N2\}$	$\theta_2 = \{\}$	$\times 0.5$
$\vdash [2, +], n(N2), \{X \text{ is } 2 + N2\}$	$\theta_3 = \{N1/2\}$	$\times 0.1$
$\vdash [2, +, 0], 2 \text{ is } 2 + 0$	$\theta_4 = \{X/2, N2/0\}$	$\times 0.1$

3. DeepStochLog

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

$$nn(m, [I_1, ..., I_m], [O_1, ..., O_L], [D_1, ..., D_L]) :: nt \rightarrow g_1, ..., g_n$$

where nt is an atom, $g_1, ..., g_n$ is a goal, and the $I_1 ... I_m$ and $O_1, ..., O_L$ are variables occurring in $g_1, ..., g_n$ and nt. The D_i are unary predicates defining the domain of the output variables O_i. The nn declaration states that m is a neural network that takes the variables $I_1, \ldots, I_m$ as input and outputs a probability distribution over output variables $O_1, ..., O_L$ (i.e. a probability distribution over the cross product of the domains specified by D_i). 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. 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

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

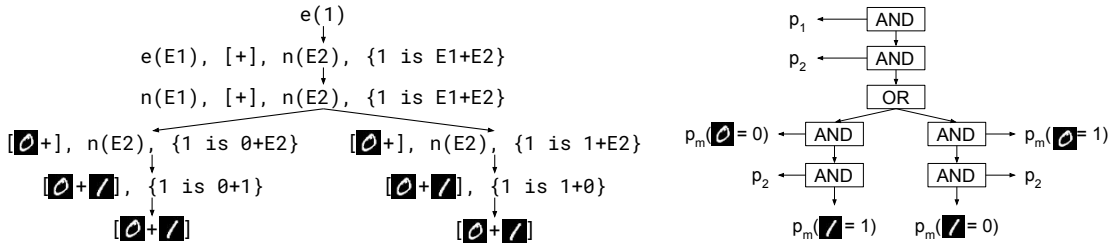
Here, the neural network called *mnist* takes as input the *Mnist* image and returns a probability for every possible number between 0 and 9, indicating how likely it is that every number is for the given MNIST image [18]. The predicate *digit* is defined as $digit(0), digit(1), ..., digit(9)$. Given the neural network and the input substitution $\sigma = \{\texttt{Mnist} = \boxed{0}\}$ (which could be obtained through unification with the terminal sequence), the neural network could generate the output substitutions $\theta_0 = \{N = 0\}; \ldots ; \theta_9 = \{N = 9\}$; with probabilities 0.87; ... ;0.05. Thus, the neural rule with the input substitution $\sigma = \{\texttt{Mnist} = \boxed{0}\}$ denotes the following set of grammar rules: $0.87 :: n(0) \rightarrow [\boxed{0}]; \quad \ldots ; \quad 0.05 :: n(9) \rightarrow [\boxed{0}]$

The neural rules are reminiscent of the neural predicates in DeepProbLog [1], which also encapsulate a neural network that outputs a distribution over a number of alternatives.

4. Inference and Learning in DeepStochLog

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

Given a DeepStochLog program, a goal G and a (potentially unknown) sequence T, logical inference uses resolution to answer $derives(G, T)$. 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



(a) The SLD tree for *derives*($e(1)$, [+]). Failing branches are omitted. Notice that only the left-hand branch derives the correct parse of the images.

(b) AND-OR circuit for *derives*($e(1)$, [+])

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

for the probabilistic inference. The logical inference procedure is illustrated in Figure 1, where the SLD resolution tree for the given goal is on the left and its corresponding AND-OR circuit on the right. 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 resultion [19], which plays a similar role as the dynamic programming CYK algorithm for CFGs [20].

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 [21]. Analogously, the most probable derivation for the goal G is found with the $(\max, \times)$ semiring.

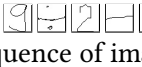
Learning Learning in DeepStochLog is achieved by optimizing the parameters of the neural networks and the parameters of the logic programs itself. Let us consider a dataset of triples $\mathcal{D} = \{(G_i\theta_i, T_i, t_i)\}$, where G_i is a goal, θ_i a substitution for G_i, T_i a sequence of terminals 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 $\min_p \sum_{(G_i\theta_i, T_i, t_i) \in \mathcal{D}} \mathcal{L}\Big(P_G(derives(G_i\theta_i, T_i); p), t_i\Big)$ with $\mathcal{L}$ being any differentiable loss function. Representing the dynamic programming computation for the *inside* probability in terms of an arithmetic circuit has an important advantage allows to obtain a differentiable computational graph. 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.

5. Evaluation

5.1. Tasks

We specify the tasks used in this paper. Complete details are specified in Appendix B.

T1: MNIST Addition. In the MNIST Addition task [1], the model is given two sequences of length N of MNIST images containing handwritten images [18], each representing an N-digit number. The task is to predict the sum of these numbers ( +  = 56). 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.

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 . Like T1, the data of T2 only contains the outcome of the expression and the sequence of images. For this task, we use the Handwritten Formula (HWF) dataset, introduced in [22], where we 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 bracket 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 recognize whether the input sequence belongs to the first grammar or the second.

5.2. Results

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?

For all experiments and metrics, we report the mean accuracy (or the mean most likely parse accuracy where applicable, i.e. **T1**, **T2** and **T3**) and 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

Table 1
The test accuracy (%) on the MNIST addition (**T1**).

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

Table 2
The accuracy (%) on the HWF dataset (**T2**).

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

Table 3
The parse accuracy (%) on the well-formed parentheses dataset (**T3**) and on the $a^n b^n c^n$ dataset (**T4**)

	T3			**T4**		
Max. Length	10	14	18	12	15	18
DeepProbLog	100.0 ± 0.0	99.4 ± 0.5	99.2 ± 0.8	99.8 ± 0.3	timeout	timeout
DeepStochLog	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	99.4 ± 0.5	99.2 ± 0.4	98.8 ± 0.2

that DeepStochLog performs similarly to the DeepProbLog [1] and NeurASP [4] frameworks 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 [22] 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.

Q2: DeepStochLog scalability We now investigate whether DeepStochLog is more scalable than similarframeworks. First, we observe that in the tasks **T1**, **T2** and **T4**, DeepStochLog scales to settings that are infeasible for the competitors. Next, in Table 4, we compare the execution times for task **T1**. We selected 100 queries from the training data and we computed the average time required from the system to 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. The advantage of DeepStochLog over the competitors is due to 1) the use of SLG resolution and 2) a random walk semantics which is computationally cheaper than the possible world semantics exploited by DeepProbLog and NeurASP.

Table 4
Inference times in milliseconds for DeepStochLog, DeepProbLog and NeurASP on task **T1** for variable number lengths.

Numbers Length	1	2	3	4
DeepStochLog	1.3 ± 0.9	2.3 ± 0.4	4.0 ± 0.4	5.7 ± 1.8
DeepProbLog	13.5 ± 3.0	36.0 ± 0.5	199.7 ± 14.0	timeout
NeurASP	9.2 ± 1.4	85.7 ± 22.6	158.2 ± 47.7	timeout

6. Related Works

DeepStochLog is a neural logic programming language in the spirit of DeepProbLog [1], NeurASP [4], the neural theorem prover [3] and lifted relational neural networks (LRNNs) [6]. 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 [5], 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.

DeepStochLog can also be viewed as a neural-based grammar, similarly to Neural Grammars [23] and NGS [22]. Neural Grammars show an effective strategy to learn PCFGs using neural reparameterizations of PFCG. They can 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 do not consider subsymbolic inputs (as in all our tasks T1-T4). 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 evidence automatically through any NDCG.

Many systems in the neural symbolic community [24, 25, 6, 26] obtain differentiable logics by relaxing logical programs or theories using fuzzy logic and t-norms. They allow for more scalable systems as compared to probabilistic logic based on the possible world semantics. 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.

7. Conclusions

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

References

[1] R. Manhaeve, S. Dumancic, A. Kimmig, T. Demeester, L. De Raedt, Deepproblog: Neural probabilistic logic programming, Advances in Neural Information Processing Systems 31 (2018) 3749–3759.

[2] W.-Z. Dai, Q. Xu, Y. Yu, Z.-H. Zhou, Bridging machine learning and logical reasoning by abductive learning, in: H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, R. Garnett (Eds.), Advances in Neural Information Processing Systems, volume 32, Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper/2019/file/9c19a2aa1d84e04b0bd4bc888792bd1e-Paper.pdf.

[3] T. Rocktäschel, S. Riedel, End-to-end differentiable proving, in: Advances in Neural Information Processing Systems, volume 30, 2017, pp. 3788–3800.

[4] Z. Yang, A. Ishay, J. Lee, Neurasp: Embracing neural networks into answer set programming, in: Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI, 2020, pp. 1755–1762.

[5] W. W. Cohen, F. Yang, K. R. Mazaitis, Tensorlog: Deep learning meets probabilistic databases, Journal of Artificial Intelligence Research 1 (2018) 1–15.

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

[7] X. Si, M. Raghothaman, K. Heo, M. Naik, Synthesizing datalog programs using numerical relaxation, in: S. Kraus (Ed.), Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019, ijcai.org, 2019, pp. 6117–6124.

[8] L. De Raedt, S. Dumančić, R. Manhaeve, G. Marra, From statistical relational to neuro-symbolic artificial intelligence, in: C. Bessiere (Ed.), Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20, International Joint Conferences on Artificial Intelligence Organization, 2020, pp. 4943–4950. URL: https://doi.org/10.24963/ijcai.2020/688. doi:10.24963/ijcai.2020/688, survey track.

[9] L. De Raedt, A. Kimmig, Probabilistic (logic) programming concepts, Machine Learning 100 (2015) 5–47.

[10] E. Tsamoura, L. Michael, Neural-symbolic integration: A compositional perspective, CoRR abs/2010.11926 (2020). URL: https://arxiv.org/abs/2010.11926. arXiv:2010.11926.

[11] C. T. Have, Stochastic definite clause grammars, in: Proceedings of the International Conference RANLP-2009, 2009, pp. 139–143.

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

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

[14] L. Sterling, E. Y. Shapiro, The art of Prolog: advanced programming techniques, MIT press, 1994.

[15] J. Cussens, Parameter estimation in stochastic logic programs, Machine Learning 44 (2001) 245–271.

[16] S. Muggleton, Stochastic logic programs, Advances in inductive logic programming 32 (1996) 254–264.
[17] S. Muggleton, Learning stochastic logic programs, Electron. Trans. Artif. Intell. 4 (2000) 141–153.
[18] Y. Lecun, L. Bottou, Y. Bengio, P. Haffner, Gradient-based learning applied to document recognition, in: Proceedings of the IEEE, 1998, pp. 2278–2324.
[19] W. Chen, D. S. Warren, Tabled evaluation with delaying for general logic programs, J. ACM 43 (1996) 20–74. URL: https://doi.org/10.1145/227595.227597. doi:10.1145/227595.227597.
[20] T. Kasami, An efficient recognition and syntax-analysis algorithm for context-free languages, Coordinated Science Laboratory Report no. R-257 (1966).
[21] A. Kimmig, G. Van den Broeck, L. De Raedt, An algebraic prolog for reasoning about possible worlds, in: Proceedings of the AAAI Conference on Artificial Intelligence, volume 25, 2011.
[22] Q. Li, S. Huang, Y. Hong, Y. Chen, Y. N. Wu, S.-C. Zhu, Closed loop neural-symbolic learning via integrating neural perception, grammar parsing, and symbolic reasoning, in: International Conference on Machine Learning, PMLR, 2020, pp. 5884–5894.
[23] C. Dyer, A. Kuncoro, M. Ballesteros, N. A. Smith, Recurrent neural network grammars, in: Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, 2016, pp. 199–209.
[24] I. Donadello, L. Serafini, A. S. d'Avila Garcez, Logic tensor networks for semantic image interpretation, in: C. Sierra (Ed.), Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017, ijcai.org, 2017, pp. 1596–1602. URL: https://doi.org/10.24963/ijcai.2017/221. doi:10.24963/ijcai.2017/221.
[25] M. Diligenti, M. Gori, C. Sacca, Semantic-based regularization for learning and inference, Artificial Intelligence 244 (2017) 143–165.
[26] E. van Krieken, E. Acar, F. van Harmelen, Analyzing differentiable fuzzy implications, in: D. Calvanese, E. Erdem, M. Thielscher (Eds.), Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning, KR 2020, Rhodes, Greece, September 12-18, 2020, 2020, pp. 893–903. URL: https://doi.org/10.24963/kr.2020/92. doi:10.24963/kr.2020/92.

A. Appendix

A.1. Data and Licenses

- For tasks T1, T3 and T4, we used the MNIST dataset from [18] to generate new datasets. The MNIST dataset itself was released under the Creative Commons Attribution-Share Alike 3.0 license.
- The Handwritten Formula Recognition (HWF) dataset (used in T2) originates from [22].

A.2. 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).

B. Task Details

In this section we show the full DeepStochLog programs and additional experimental details for all tasks.

B.1. 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. DeepStochLog program for single-digit numbers:

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

DeepStochLog program for L-long digits numbers:

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

B.2. 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 (see Table 5. 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). This problem is modeled as follows in DeepStochLog:

```
digit(Y) :- member(Y,[0,1,2,3,4,5,6,7,8,9]).
operator_d(Y) :- member(Y,[plus, minus, times, div]).
term_switch_d(Y) :- member(Y,[0, 1, 2]).
e_switch_d(Y) :- member(Y,[0, 1, 2]).
```

```
nn(number, [X],[Y],[digit]):: number(Y) --> [X].
nn(operator, [X],[Y],[operator_d]) :: operator(Y) --> [X].
1 :: factor(N) --> number(N).

nn(term, [], [Y], [term_switch_d]) :: term(N) --> term_switch(N,Y).
0.33 :: term_switch(N, 0) --> factor(N).
0.33 :: term_switch(N, 1) --> term(N1), operator(times), factor(N2),
                              {N is N1 * N2}.
0.33 :: term_switch(N, 2) --> term(N1), operator(div), factor(N2),
                              {N2>0, N is N1 / N2}.

nn(expression, [], [Y], [e_switch_d]) :: expression(N) --> e_switch(N,Y).
0.33 :: e_switch(N,0) --> term(N).
0.33 :: e_switch(N,1) --> expression(N1), operator(plus), term(N2),
                          {N is N1 + N2}.
0.33 :: e_switch(N,2) --> expression(N1), operator(minus), term(N2),
                          {N is N1 - N2}.
```

B.3. 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 as follows:

```
bracket_d(Y) :- member(Y,["(",")"]).
s_switch_d(Y) :- member(Y,[0,1,2]).

nn(bracket_nn,[X], [Y], [bracket_d])::bracket(Y) --> [X].
nn(s_nn,[X],[Y],[s_switch_d])::s --> s_switch(Y).
0.33::s_switch(0) --> s, s.
0.33::s_switch(1) --> bracket("("), s, bracket(")").
0.33::s_switch(2) --> bracket("("), bracket(")").
```

B.4. 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. This problem is modelled in DeepStochLog as follows:

```
rep_d(Y):- domain(Y, [a,b,c]).
0.5:: s(0) --> akblcm(K,L,M),{K\=L; L\=M; M\=K}.
0.5:: s(1) --> akblcm(N,N,N).
akblcm(K,L,M) --> rep(K,A), rep(L,B), rep(M,C),{A\=B, B\=C, C\=A}.
0.5 :: rep(s(0), C) --> terminal(C).
0.5 :: rep(s(N), C) --> terminal(C), rep(N,C).
```

```
nn(mnist, [X], [C], [rep_d]) :: terminal(C) --> [X].
```

B.5. Neural network architectures

Table 5 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.

Table 5
Overview of the neural network architectures used in the experiments.

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)

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

C. Translation Example of Neural Definite Clause Grammar to Prolog

Consider the following DeepStochLog program.

```
digit(Y) :- member(Y, [0,1,2,3,4,5,6,7,8,9]).
op(Y) :- member(Y, [+,-]).

nn(mnist,[I],[N],[digit]) :: n(N) --> [I].
nn(operator,[I],[N],[op]) :: o(N) --> [I].

0.33::e(N) --> n(N).
0.33::e(S) --> e(E1), o(+), n(E2), {S is E1 + E2}.
0.33::e(S) --> e(E1), o(-), n(E2), {S is E1 - E2}.
```

It is translated into:

```
digit(Y) :- member(Y, [0,1,2,3,4,5,6,7,8,9]).
op(Y) :- member(Y, [+,-]).

n(N, [I | X], [X]) :- nn(mnist,[I],[N]), digit(X).
```

```
n(N, [I | X], [X]) :- nn(mnist,[I],[O]), op(O).

e(N, A, B) :- n(N, A, B), p(0.33).
e(S, A,D) :- e(E1, A, B), o(+, B, C), n(E2, C, D), S is E1 + E2, p(0.33).
e(S, A,D) :- e(E1, A, B), o(-, B, C), n(E2, C, D), S is E1 - E2, p(0.33).
```

Both the calls to `nn` and `p` are considered always `true` during logical inference. During evaluation of 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.