DeepProbLog:
Neural Probabilistic Logic Programming

Robin Manhaeve
KU Leuven
robin.manhaeve@cs.kuleuven.be

Sebastijan Dumančić
KU Leuven
sebastijan.dumancic@cs.kuleuven.be

Angelika Kimmig
Cardiff University
KimmigA@cardiff.ac.uk

Thomas Demeester*
Ghent University - imec
thomas.demeester@ugent.be

Luc De Raedt*
KU Leuven
luc.deraedt@cs.kuleuven.be

Abstract

We introduce DeepProbLog, a probabilistic logic programming language that incorporates deep learning by means of neural predicates. We show how existing inference and learning techniques can be adapted for the new language. Our experiments demonstrate that DeepProbLog supports (i) both symbolic and sub-symbolic representations and inference, (ii) program induction, (iii) probabilistic (logic) programming, and (iv) (deep) learning from examples. To the best of our knowledge, this work is the first to propose a framework where general-purpose neural networks and expressive probabilistic-logical modeling and reasoning are integrated in a way that exploits the full expressiveness and strengths of both worlds and can be trained end-to-end based on examples.

1 Introduction

The integration of low-level perception with high-level reasoning is one of the oldest, and yet most current open challenges in the field of artificial intelligence. Today, low-level perception is typically handled by neural networks and deep learning, whereas high-level reasoning is typically addressed using logical and probabilistic representations and inference. While it is clear that there have been breakthrougths in deep learning, there has also been a lot of progress in the area of high-level reasoning. Indeed, today there exist approaches that tightly integrate logical and probabilistic reasoning with statistical learning; cf. the areas of statistical relational artificial intelligence [De Raedt et al., 2016, Getoor and Taskar, 2007] and probabilistic logic programming [De Raedt and Kimmig, 2015]. Recently, a number of researchers have revisited and modernized older ideas originating from the field of neural-symbolic integration [Garcez et al., 2012], searching for ways to combine the best of both worlds [Bošnjak et al., 2017, Rocktäschel and Riedel, 2017, Cohen et al., 2018, Santoro et al., 2017], for example, by designing neural architectures representing differentiable counterparts of symbolic operations in classical reasoning tools. Yet, joining the full flexibility of high-level probabilistic reasoning with the representational power of deep neural networks is still an open problem. This paper tackles this challenge from a different perspective. Instead of integrating reasoning capabilities into a complex neural network architecture, we proceed the other way round. We start

* joint last authors

32nd Conference on Neural Information Processing Systems (NIPS 2018), Montréal, Canada.
from an existing probabilistic logic programming language, ProbLog [De Raedt et al., 2007], and extend it with the capability to process neural predicates. The idea is simple: in a probabilistic logic, atomic expressions of the form \( q(t_1, ..., t_n) \) (aka tuples in a relational database) have a probability \( p \).

Consequently, the output of neural network components can be encapsulated in the form of “neural” predicates as long as the output of the neural network on an atomic expression can be interpreted as a probability. This simple idea is appealing as it allows us to retain all the essential components of the ProbLog language: the semantics, the inference mechanism, as well as the implementation. The main challenge is in training the model based on examples. The input data consists of feature vectors at the input of the neural network components (e.g., images) together with other probabilistic facts and clauses in the logic program, whereas targets are only given at the output side of the probabilistic reasoner. However, the algebraic extension of ProbLog (based on semirings) [Kimmig et al., 2011] already supports automatic differentiation. As a result, we can back-propagate the gradient from the loss at the output through the neural predicates into the neural networks, which allows training the whole model through gradient-descent based optimization. We call the new language DeepProbLog.

Before going into further detail, the following example illustrates the possibilities of this approach (also see Section 6). Consider the predicate \( \text{addition}(X, Y, Z) \), where \( X \) and \( Y \) are images of digits and \( Z \) is the natural number corresponding to the sum of these digits. After training, DeepProbLog allows us to make a probabilistic estimate on the validity of, e.g., the example \( \text{addition}(3, 2, 8) \).

While such a predicate can be directly learned by a standard neural classifier, such a method would have a hard time taking into account background knowledge such as the definition of the addition of two natural numbers. In DeepProbLog such knowledge can easily be encoded in rules such as \( \text{addition}(I_X, I_Y, N_Z) := \text{digit}(I_X, N_X), \text{digit}(I_Y, N_Y), N_Z := N_X + N_Y \) (with \( \text{is} \) the standard operator of logic programming to evaluate arithmetic expressions). All that needs to be learned in this case is the neural predicate \( \text{digit} \) which maps an image of a digit \( I_D \) to the corresponding natural number \( N_D \). The learned network can then be reused for arbitrary tasks involving digits. Our experiments show that this leads not only to new capabilities but also to significant performance improvements. An important advantage of this approach compared to standard image classification settings is that it can be extended to multi-digit numbers without additional training. We note that the single digit classifier (i.e., the neural predicate) is not explicitly trained by itself: its output can be considered a latent representation, as we only use training data with pairwise sums of digits.

To summarize, we introduce DeepProbLog which has a unique set of features: (i) it is a programming language that supports neural networks and machine learning, and it has a well-defined semantics (as an extension of Prolog, it is Turing equivalent); (ii) it integrates logical reasoning with neural networks; so both symbolic and subsymbolic representations and inference; (iii) it integrates probabilistic modeling, programming and reasoning with neural networks (as DeepProbLog extends the probabilistic programming language ProbLog, which can be regarded as a very expressive directed graphical modeling language [De Raedt et al., 2016]); (iv) it can be used to learn a wide range of probabilistic logical neural models from examples, including inductive programming. The code is available at https://bitbucket.org/problog/deepproblog.

## 2 Logic programming concepts

We briefly summarize basic logic programming concepts. Atoms are expressions of the form \( q(t_1, ..., t_n) \) where \( q \) is a predicate (of arity \( n \)) and the \( t_i \) are terms. A literal is an atom or the negation \( \neg q(t_1, ..., t_n) \) of an atom. A term \( t \) is either a constant \( c \), a variable \( V \), or a structured term of the form \( f(u_1, ..., u_k) \) where \( f \) is a functor. We will follow the Prolog convention and let constants start with a lower case character and variables with an upper case. A substitution \( \theta = \{ V_i = t_1, ..., V_n = t_n \} \) is an assignment of terms \( t_i \) to variables \( V_i \). When applying a substitution \( \theta \) to an expression \( e \) we simultaneously replace all occurrences of \( V_i \) by \( t_i \) and denote the resulting expression as \( e\theta \). Expressions that do not contain any variables are called ground. A rule is an expression of the form \( h := b_1, ..., b_n \) where \( h \) is an atom and the \( b_i \) are literals. The meaning of such a rule is that \( h \) holds whenever the conjunction of the \( b_i \) holds. Thus \( := \) represents logical implication (\( \leftarrow \)), and the comma (\( , \)) represents conjunction (\( \land \)). Rules with an empty body \( n = 0 \) are called facts.
4 DeepProbLog Inference

This section explains how a DeepProbLog model is used for a given query at prediction time.

**ProbLog Inference**  As inference in DeepProbLog closely follows that in ProbLog, we now summarize ProbLog inference using the burglary example explained before. For full details, we refer to Fierens et al. [2015]. The program describing the example is explained in Section 3, Equation (1). We can query the program for the probabilities of given query atoms, say, the single query \( \text{calls}(\text{mary}) \). ProbLog inference proceeds in four steps. (i) The first step grounds the logic program with respect to the query, that is, it generates all ground instances of clauses in the program the query depends on. The grounded program for query \( \text{calls}(\text{mary}) \) is shown in Figure 1a. (ii) The second step rewrites the ground logic program into a formula in propositional logic that defines the truth value of the query in terms of the truth values of probabilistic facts. In our example, the resulting formula is \( \text{calls}(\text{mary}) \leftrightarrow \text{hears}_\text{alarm}(\text{mary}) \land (\text{burglary} \lor \text{earthquake}) \). (iii) The third step compiles the logic formula into a Sentential Decision Diagram (SDD, Darwiche [2011]), a form that allows for efficient evaluation of the query, using knowledge compilation technology [Darwiche and Marquis, 2002]. The SDD for our example is shown in Figure 1b, where rounded grey rectangles depict variables corresponding to probabilistic facts, and the rounded red rectangle denotes the query atom defined by the formula. The white rectangles correspond to logical operators applied to their children. (iv) The fourth and final step evaluates the SDD bottom-up to calculate the success probability of the given query, starting with the probability labels of the leaves as given by the program and performing addition in every or-node and multiplication in every and-node. The intermediate results are shown next to the nodes in Figure 1b, ignoring the blue numbers for now.

---

![Figure 1: Inference in ProbLog.](image)

(a) The ground program.

(b) SDD for query \( \text{calls}(\text{mary}) \).

---

**DeepProbLog Inference**  Inference in DeepProbLog works exactly as described above, except that a forward pass on the neural network components is performed every time we encounter a neural predicate during grounding. When this occurs, the required inputs (e.g., images) are fed into the neural network, after which the resulting scores of their softmax output layer are used as the probabilities of the ground AD.

5 Learning in DeepProbLog

We now introduce our approach to jointly train the parameters of probabilistic facts and neural networks in DeepProbLog programs. We use the learning from entailment setting [De Raedt et al., 2016], that is, given a DeepProbLog program with parameters \( \mathcal{X} \), a set \( Q \) of pairs \((q, p)\) with \( q \) a query and \( p \) its desired success probability, and a loss function \( L \), compute:

\[
\arg\min_{\mathcal{X}} \frac{1}{|Q|} \sum_{(q, p) \in Q} L(P_{\mathcal{X}=x}(q), p)
\]

In contrast to the earlier approach for ProbLog parameter learning in this setting by Gutmann et al. [2008], we use gradient descent rather than EM, as this allows for seamless integration with neural network training. An overview of the approach is shown in Figure 2a. Given a DeepProbLog program, its neural network models, and a query used as training example, we first ground the program with respect to the query, getting the current parameters of nADs from the external models, then use the ProbLog machinery to compute the loss and its gradient, and finally use these to update the parameters in the neural networks and the probabilistic program.
More specifically, to compute the gradient with respect to the probabilistic logic program part, we rely on Algebraic ProbLog (aProbLog, [Kimmig et al., 2011]), a generalization of the ProbLog language and inference to arbitrary commutative semirings, including the gradient semiring [Eisner, 2002]. In the following, we provide the necessary background on aProbLog, discuss how to use it to compute gradients with respect to ProbLog parameters and extend the approach to DeepProbLog.

**aProbLog and the gradient semiring** ProbLog annotates each probabilistic fact $f$ with the probability that $f$ is true, which implicitly also defines the probability that $\neg f$ is false, and thus its negation $\neg f$ is true. It then uses the probability semiring with regular addition and multiplication as operators to compute the probability of a query on the SDD constructed for this query, cf. Figure 1b. This idea is generalized in aProbLog to compute such values based on arbitrary commutative semirings. Instead of probability labels on facts, aProbLog uses a labeling function that explicitly associates values from the chosen semiring with both facts and their negations, and combines these using semiring addition $\oplus$ and multiplication $\otimes$ on the SDD. We use the gradient semiring, whose elements are tuples $(p, \frac{\partial p}{\partial x})$, where $p$ is a probability (as in ProbLog), and $\frac{\partial p}{\partial x}$ is the partial derivative of that probability with respect to a parameter $x$, that is, the probability $p_x$ of a probabilistic fact with learnable probability, written as $t(p_x) :: f_i$. This is easily extended to a vector of parameters $\vec{x} = [x_1, \ldots, x_N]^T$, the concatenation of all $N$ parameters in the ground program. Semiring addition $\oplus$, multiplication $\otimes$ and the neutral elements with respect to these operations are defined as follows:

\[
(a_1, a_2) \oplus (b_1, b_2) = (a_1 + b_1, a_2 + b_2) \quad (2) \\
(a_1, a_2) \otimes (b_1, b_2) = (a_1 b_1, a_2 b_2) \quad (3) \\
op\text{e} \oplus = (0, 0) \quad (4) \\
\text{e} \otimes = (1, 0) \quad (5)
\]

Note that the first element of the tuple mimics ProbLog’s probability computation, whereas the second simply computes gradients of these probabilities using derivative rules.

**Gradient descent for ProbLog** To use the gradient semiring for gradient descent parameter learning in ProbLog, we first transform the ProbLog program into an aProbLog program by extending the label of each probabilistic fact $p :: f$ to include the probability $p$ as well as the gradient vector of $p$ with respect to the probabilities of all probabilistic facts in the program, i.e.,

\[
L(f) = (p, \vec{0}) \quad \text{for } p :: f \text{ with fixed } p \quad (6) \\
L(f_i) = (p_i, e_i) \quad \text{for } t(p_i) :: f_i \text{ with learnable } p_i \quad (7) \\
L(-f) = (1 - p, -\nabla p) \quad \text{with } L(f) = (p, \nabla p) \quad (8)
\]

where the vector $e_i$ has a 1 in the $i$th position and 0 in all others. For fixed probabilities, the gradient does not depend on any parameters and thus is 0. For the other cases, we use the semiring labels as introduced above. For instance, assume we want to learn the probabilities of $\text{earthquake}$
The learning curves of both models on T1 (Figure 3a) show the benefit of combined symbolic and subsymbolic reasoning: the DeepProbLog model uses the encoded knowledge to reach a higher F1 score than the CNN, and does so after a few thousand iterations, while the CNN converges much slower. We also tested an alternative for the neural network baseline. It evaluates convolutional layers with shared parameters on each image separately, instead of a single set of convolutional layers on the concatenation of both images. It converges quicker and achieves a higher final accuracy than the other baseline, but is still slower and less accurate than the DeepProbLog model. Figure 3b shows the learning curve for T2. DeepProbLog achieves a somewhat lower accuracy compared to the single digit problem due to the compounding effect of the error, but the model generalizes well. The CNN does not generalize to this variable-length problem setting.

Program Induction The second set of problems demonstrates that DeepProbLog can perform program induction. We follow the program sketch setting of differentiable Forth [Bošnjak et al., 2017], where holes in given programs need to be filled by neural networks trained on input-output examples for the entire program. As in their work, we consider three tasks: addition, sorting [Reed and de Freitas, 2016] and word algebra problems (WAPs) [Roy and Roth, 2015].

T3: forth.addition/4: where the input consists of two numbers and a carry, with the output being the sum of the numbers and the new carry. The program specifies the basic addition algorithm in which we go from right to left over all digits, calculating the sum of two digits and taking the carry over to the next pair. The hole in this program corresponds to calculating the resulting digit (result/4) and carry (carry/4), given two digits and the previous carry.

T4: sort/2: The input consists of a list of numbers, and the output is the sorted list. The program implements bubble sort, but leaves open what to do on each step in a bubble (i.e. whether to swap or not, swap/2).

T5: wap/2: The input to the WAPs consists of a natural language sentence describing a simple mathematical problem, and the output is the solution to this question. These WAPs always contain three numbers and are solved by chaining 4 steps: permuting the three numbers (permute/2), applying an operation on the first two numbers (addition, subtraction or product operation_1/2), potentially swapping the intermediate result and the last digit (swap/2), and performing a last operation (operation_2/2). The hole in the program is in deciding which of the alternatives should happen on each step.

DeepProbLog achieves 100% on the Forth addition (T3) and sorting (T4) problems (Table 1a). The sorting problem yields a more interesting comparison: differentiable Forth achieves a 100% accuracy with a training length of 2 and 3, but performs poorly on a training length of 4; DeepProbLog generalizes well to larger lengths. As shown in Table 1b, DeepProbLog runs faster and scales better with increasing training length, while differentiable Forth has issues due to computational complexity with larger lengths, as mentioned in the paper.

On the WAPs (T5), DeepProbLog reaches an accuracy between 96% and 97%, similar to Bošnjak et al. [2017] (96%).

Probabilistic programming and deep learning The coin-ball problem is a standard example in the probabilistic programming community [De Raedt and Kimmig, 2014]. It describes a game in which we have a potentially biased coin and two urns. The first urn contains a mixture of red and blue balls, and the second urn a mixture of red, blue and green balls. To play the game, we toss the coin and take a ball out of each urn. We win if both balls have the same colour, or if the coin came up heads and we have at least one red ball. We want to learn the bias of the coin (the probability of heads), and the ratio of the coloured balls in each urn. We simultaneously train one neural network to classify an image of the coin as being heads or tails (coin/2), and a neural network to classify the colour of the ball as being either red, blue or green (colour/4). These are given as RGB triples. Task T6 is thus to learn the game/4 predicate, requiring a combination of subsymbolic reasoning, learning and probabilistic reasoning. The input consists of an image, two RGB pairs and the output is the outcome of the game. The coin-ball problem uses a very simple neural network component. Training on a set of 256 instances converges after 5 epochs, leading to 100% accuracy on the test set (64 instances). At this point, both networks correctly classify the colours and the coins, and the probabilistic parameters reflect the distributions in the training set.
Figure 3: MNIST Addition problems: displaying training loss (red for CNN, orange for DeepProbLog) and F1 score on the test set (green for CNN, blue for DeepProbLog).

Table 1: Results on the Differentiable Forth experiments

<table>
<thead>
<tr>
<th>Test Length</th>
<th>Sorting (T4): Training length</th>
<th>Addition (T3): training length</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td>6</td>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td>8</td>
<td>100.0</td>
<td>100.0</td>
</tr>
</tbody>
</table>

(a) Accuracy on the addition (T3) and sorting (T4) problems (results for δ4 reported by Boštanjak et al. [2017]).

(b) Time until 100% accurate on test length 8 for the sorting (T4) problem.

Implementation details In all experiments we optimize the cross-entropy loss between the predicted and desired query probabilities. The network used to classify MNIST images is a basic architecture based on the PyTorch tutorial. It consists of 2 convolutional layers with kernel size 5, and respectively 6 and 16 filters, each followed by a maxpool layer of size 2, stride 2. After this come 3 fully connected layers of sizes 120, 84 and 10 (19 for the CNN baseline). It has a total of 44k parameters. The last layer is followed by a softmax layer, all others are followed by a ReLu layer. The colour network consists of a single fully connected layer of size 3. For all experiments we use Adam [Kingma and Ba, 2015] optimization for the neural networks, and SGD for the logic parameters. The learning rate is 0.001 for the MNIST network, and 1 for the colour network. For robustness in optimization, we use a warm-up of the learning rate of the logic parameters for the coin-ball experiments, starting at 0.0001 and raising it linearly to 0.01 over four epochs. For the Forth experiments, the architecture of the neural networks and other hyper-parameters are as described in Boštanjak et al. [2017]. For the Coin-Urn experiment, we generate the RGB pairs by adding Gaussian noise (σ = 0.03) to the base colours in the HSV domain. The coins are MNIST images, where we use even numbers as heads, and odd for tails. For the implementation we integrated ProbLog2 [Dries et al., 2015] with PyTorch [Paszke et al., 2017]. We do not perform actual mini-batching, but instead use gradient accumulation. All programs are listed in the appendix.

7 Related Work

Most of the work on combining neural networks and logical reasoning comes from the neuro-symbolic reasoning literature [Garcez et al., 2012, Hammer and Hitzler, 2007]. These approaches typically focus on approximating logical reasoning with neural networks by encoding logical terms in Euclidean space. However, they neither support probabilistic reasoning nor perception, and are often limited to non-recursive and acyclic logic programs [Hölldobler et al., 1999]. DeepProbLog takes a different approach and integrates neural networks into a probabilistic logic framework, retaining the full power of both logical and probabilistic reasoning and deep learning.
References


