Learning the Parameters of Deep Probabilistic Logic Programs

Arnaud Nguembang Fadja[1], Fabrizio Riguzzi[2], Evelina Lamma[1]

[1] Dipartimento di Ingegneria – University of Ferrara
Via Saragat 1, I-44122, Ferrara, Italy
[2] Dipartimento di Matematica e Informatica – University of Ferrara
Via Saragat 1, I-44122, Ferrara, Italy
[arnaud.nguembafadja,fabrizio.riguzzi,evelina.lamma]@unife.it

Abstract. Probabilistic logic programming (PLP) is a powerful tool for reasoning in relational domains with uncertainty. However, both inference and learning from examples are computationally expensive tasks. We consider a restriction of PLP called hierarchical PLP whose clauses and predicates are hierarchically organized forming a deep neural network or arithmetic circuit. Inference in this language is much cheaper than for general PLP languages. In this work we present an algorithm called Deep Parameter learning for HIerarchical probabilistic Logic programs (DPHIL) that learns hierarchical PLP parameters using gradient descent and back-propagation.

Keywords: Probabilistic Logic Programming, Distribution Semantics, Deep Neural Networks, Arithmetic Circuits.

CEUR-WS.org/Vol-2219/paper2.pdf

1 Introduction

Probabilistic logic programming (PLP) under the distribution semantics [8] has been very useful in machine learning. However, inference is expensive so machine learning algorithms may turn out to be slow. We consider a restriction of the language of Logic Programs with Annotated Disjunctions (LPADs) called *hierarchical PLP* (HPLP), see [6] in which clauses and predicates are hierarchically organized. Any HPLP can be translated into an arithmetic circuit (AC) or a deep neural network. Inference is performed by evaluating the AC and parameter learning by applying gradient descent and back-propagation algorithms.

The paper is organized as follows: we describe HPLP in Section 2 and present parameter learning in Section 3. Related work is discussed in Section 4 and Section 5 concludes the paper.

2 Hierarchical PLP

Programs in LPADs allow alternatives in the head of clauses. They are set of clauses, C_i, of the form $h_{i1} : \pi_{i1}; \ldots; h_{in_i} : \pi_{in_i} :- b_{i1}, \ldots, b_{im_i}$ where

$h_{i1}, \ldots, h_{in_i}$ are logical atoms, $b_{i1}, \ldots, b_{im_i}$ are logical literals and $\pi_{i1}, \ldots, \pi_{in_i}$ are real numbers in the interval $[0, 1]$ that sum up to 1. Clauses where $\sum_{k=1}^{n_i} \pi_{ik} < 1$ implicitly contain an extra atom *null* that does not appear in the body of any clause and whose annotation is $1 - \sum_{k=1}^{n_i} \pi_{ik}$.

An HPLP is a restricted LPAD where clauses are of the form $C_{\boldsymbol{p}i} = a_{\boldsymbol{p}} : \pi_{\boldsymbol{p}i} :- \phi_{\boldsymbol{p}i}, b_{\boldsymbol{p}i1}, \ldots, b_{\boldsymbol{p}im_{\boldsymbol{p}}}$. $a_{\boldsymbol{p}}$, an atom possibly about *target predicate* (the atom we aim at predicting), is the single head atom annotated with probability $\pi_{\boldsymbol{p}i}$. The body of the clause has 2 parts: a conjunction of *input predicates*, $\phi_{\boldsymbol{p}i}$, in the sense that their definition is given as input and is certain, and a conjunction of literals for hidden predicates, $b_{\boldsymbol{p}ik}$, that are defined by clauses of the program. Hidden predicates are disjoint from input and target predicates. Moreover, the program is hierarchically organized so that it can be divided into layers forming an Arithmetic circuit or a deep neural netwoks, see [6].

Let us consider the following example in the UW-CSE domain where the objective is to predict the "advised by" relation (advisedby/2 is the target predicate)

$$
\begin{aligned}
C_1 &= advisedby(A, B) : 0.3 :- \\
&\quad student(A), professor(B), project(C, A), project(C, B), \\
&\quad r_{11}(A, B, C). \\
C_2 &= advisedby(A, B) : 0.6 :- \\
&\quad student(A), professor(B), ta(C, A), taughtby(C, B). \\
C_{111} &= r_{11}(A, B, C) : 0.2 :- \\
&\quad publication(D, A, C), publication(D, B, C).
\end{aligned}
$$

where $project(C, A)$ means that C is a project with participant A, $ta(C, A)$ means that A is a teaching assistant (TA) for course C and $taughtby(C, B)$ means that course C is taught by B. $publication(A, B, C)$ means that A is a publication with author B produced in project C. $student/1$, $professor/1$, $project/2$, $ta/2$, $taughtby/2$ and $publication/3$ are input predicates and $r_{11}/3$ is a hidden predicate.

The probability $p = advisedby(harry, ben)$ depends on the number of joint courses and projects and on the number of joint publications from projects. The clause for the hidden predicate $r_{11}/3$ computes an aggregation over publications of the same project and the clause of the level above aggregates over projects. The corresponding arithmetic circuit is shown in Fig 1 .

In order to perform inference with such a program, we can generate its grounding. Each ground probabilistic clause is associated with a random variable whose probability of being true is given by the parameter of the clause and that is independent of all the other clause random variables. Given a ground instance of $C_{\boldsymbol{p}i}$, the probability that the body is true is computed by multiplying the probability of being true of each individual atom in positive literals and one minus the probability of being true of each individual atom in negative literals. Therefore the probability of the body of $C_{\boldsymbol{p}i}$ is $P(b_{\boldsymbol{p}i1}, \ldots, b_{\boldsymbol{p}im_{\boldsymbol{p}}}) = \prod_{i=k}^{m_{\boldsymbol{p}}} P(b_{\boldsymbol{p}ik})$ and $P(b_{\boldsymbol{p}ik}) = 1 - P(a_{\boldsymbol{p}ik})$ if $b_{\boldsymbol{p}ik} = \neg a_{\boldsymbol{p}ik}$. If $b_{\boldsymbol{p}ik} = a$ is a literal for an input predicate, $P(a) = 1$ if it belongs to the example interpretation and $P(a) = 0$ otherwise. To compute the probability, $P(a_{\boldsymbol{p}})$, of an atom $a_{\boldsymbol{p}}$ of a given hidden

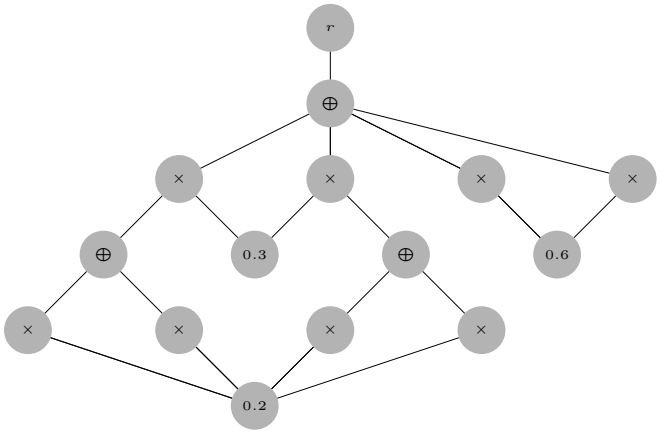

Fig. 1: Arithmetic circuit.

.

literal $b_{\boldsymbol{p}}$, we need to take into account the contribution of every ground clause for the atom $a_{\boldsymbol{p}}$. For a single clause, $P(a_{\boldsymbol{p}}) = \pi_{\boldsymbol{p}1} \cdot P(body(C_{\boldsymbol{p}1}))$. For two clauses, we can compute the contribution of each clause as above and then combine them with the formula $P(a_{\boldsymbol{p}i}) = 1 - (1 - \pi_{\boldsymbol{p}1} \cdot P(body(C_{\boldsymbol{p}1})) \cdot (1 - \pi_{\boldsymbol{p}2} \cdot P(body(C_{\boldsymbol{p}2})))$. We defined the operator $\oplus$ that combines two probabilities as follows $p \oplus q = 1 - (1 - p) \cdot (1 - q)$. This operator is commutative and associative. For many clauses, we can compute sequences of applications as $\bigoplus_i p_i = 1 - \prod_i (1 - p_i)$. AC are built by performing inference using tabling and answer subsumption using PITA(IND,IND) [7] as described in [6]. However, writing HPLP can be unintuitive because of the constraints imposed on it and because of uncleared meaning of hidden predicates. We plan in our future work to design an algorithm for learning both the parameters and the structure of HPLP.

3 Parameter learning

We propose a back-propagation algorithm for learning HPLP parameters. Given an HPLP T with parameters Π, an interpretation I defining input predicates and a set of positive and negative examples $E = \{e_1, \ldots, e_M, \textbf{not } e_{M+1}, \ldots, \textbf{not } e_N\}$ where each e_i is a ground atom for the target predicate r, we want to find the values of Π that minimize the sum of *cross entropy errors*, $err_i = -y_i \log(p_i) - (1 - y_i) \log(1 - p_i)$, for all the examples. The algorithm, called *DPHIL* 1, starts by building a set of ground ACs sharing parameters Π, line 2. Then weights ($W[i]$), gradients ($G[i]$) and moments (adam's parameters, see [3]) are initialized, lines 2–6 . At each iteration, three actions are performed on each AC: the *Forward* pass computes the output $v(n)$ of each node n in the AC, the *Backward* pass computes the derivative of the error $d(n)$ 1, that is the gradient, with respect to

each parameter

$$d(n) = \begin{cases} d(pa_n)\frac{v(pa_n)}{v(n)} & \text{if n is a } \bigoplus node, \\ d(pa_n)\frac{1-v(pa_n)}{1-v(n)} & \text{if n is a } \times \text{ node} \\ \sum_{pa_n} d(pa_n).v(pa_n).(1-\pi_i) & \text{if n}=\sigma(W_i) \\ -d(pa_n) & pa_n = not(n) \end{cases} \quad (1)$$

where pa_n is the parent of node n and $\pi_i = \sigma(W_i) = \frac{1}{1+e^{-W_i}}$, lines 9–16. Parameters are *updated* using Adam the optimizer [3], line 17. These actions are repeatedly performed until a maximum number of steps is reached or until the difference between the log likelihood of the current and the previous iteration drops below a threshold or the difference is below a fraction of the current log likelihood. Finally the theory is updated with the learned parameters, line 19.

Algorithm 1 Function DPHIL.

```
1: function DHPIL(Theory, ϵ, δ, MaxIter, β1, β2, η, ê, Strategy)
2:     Examples ← BuildACs(Theory)                          ▷ Build the set of ACs
3:     for i ← 1 → |Theory| do      ▷ Initialize weights,gradient and moments vector
4:         W[i] ← random(Min, Max)                          ▷ initially W[i] ∈ [Min, Max].
5:         G[i] ← 0, Moment0[i] ← 0, Moment1[i] ← 0
6:     end for
7:     Iter ← 1, LL ← 0
8:     repeat
9:         LL0 ← LL
10:        LL ← 0
11:        Batch ← NextBatch(Examples)        ▷ Select the batch according to the
   strategy
12:        for all node ∈ Batch do
13:            P ← Forward(node)
14:            Backward(G, −1/P, node)
15:            LL ← LL + log P
16:        end for
17:        UpdateWeightsAdam(W, G, Moment0, Moment1, β1, β2, η, ê, Iter)
18:    until LL − LL0 < ϵ ∨ LL − LL0 < −LL.δ ∨ Iter > MaxIter
19:    FinalTheory ← UpdateTheory(Theory, W)
20:    return FinalTheory
21: end function
```

4 Related Work

HPLP is related to [4,2,5] where the probability of a query is computed by combining the contribution of different rules and grounding of rules with noisy-Or or Mean combining rules. In First-Order Probabilistic Logic (FOPL), [4]

and Bayesian Logic Programs (BLP), [2], each ground atoms is considered as a random variable and rules have a single atom in the head and only positive literals in the body. Each rule is associated with a conditional probability table defining the dependence of the head variable from the body ones. Similarly to HPLP, FOPL and BLP allow multiple layer of rules. Differently from FOPL and HPLP, BLP allows different combining rules. Like FOLP, BLP and HPLP, First-Order Conditional Influence Language (FOCIL) [5], uses probabilistic rules for compactly encoding probabilistic dependencies. The probability of a query is computed using two combining rules: the contributions of different groundings of the same rule with the same random variable in the head are combined by taking the *Mean*, and the contributions of different rules are combined either with a weighted mean or with a *noisy-OR* combining rule. FOPL, BLP and FOCIL implement parameter learning using gradient descent, as DPHIL, or using the Expectation Maximization algorithm. By suitably restricting the form of programs, we were able to provide simpler formulas for the computation of the gradient with respect to [4,2,5].

5 Conclusion

We have presented the algorithm DPHIL that performs parameter learning of HPLP. DPHIL converts an HPLP into an AC and performs gradient descent and back-propagation. In the future we plan to experiments with DPHIL and investigate the Expectation Maximization algorithm as an alternative parameter learning method. We also plan to design an algorithm for learning both the structure and the parameters of HPLPs, taking inspiration from SLIPCOVER [1].

References

1. Bellodi, E., Riguzzi, F.: Structure learning of probabilistic logic programs by searching the clause space. Theor. Pract. Log. Prog. 15(2), 169–212 (2015)
2. Kersting, K., De Raedt, L.: Basic principles of learning bayesian logic programs. In: Institute for Computer Science, University of Freiburg. Citeseer (2002)
3. Kingma, D., Ba, J.: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
4. Koller, D., Pfeffer, A.: Learning probabilities for noisy first-order rules. In: IJCAI. pp. 1316–1323 (1997)
5. Natarajan, S., Tadepalli, P., Kunapuli, G., Shavlik, J.: Learning parameters for relational probabilistic models with noisy-or combining rule. In: Machine Learning and Applications, 2009. ICMLA'09. International Conference on. pp. 141–146. IEEE (2009)
6. Nguembang Fadja, A., Lamma, E., Riguzzi, F.: Deep probabilistic logic programming. CEUR-WS, vol. 1916, pp. 3–14. Sun SITE Central Europe (2017)
7. Riguzzi, F.: Speeding up inference for probabilistic logic programs. Comput. J. 57(3), 347–363 (2014)
8. Sato, T.: A statistical learning method for logic programs with distribution semantics. In: Sterling, L. (ed.) ICLP 1995. pp. 715–729. MIT Press (1995)