Deep Learning for Probabilistic Logic Programming

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

[1] Dipartimento di Ingegneria – University of Ferrara
[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. Due to its expressiveness and intuitiveness, Probabilistic logic programming (PLP) is a useful tool for reasoning in relational domains with uncertainty. However, both inference and learning are expensive tasks. In this paper we present various approaches for speeding up learning. We first consider a restriction of PLP called Liftable PLP (LPLP) in which clauses in the program share the same predicate (the target). Then we extend this restriction in Hierarchical PLP (HPLP) where predicates and clauses are hierarchically organized and can be translated into Deep Neural Networks or Arithmetic Circuits. For LPLP, we propose two parameter learning algorithms, Expectation Maximization (EM) and Limited memory BFGS (LBFGS), and a discriminative structure learning. We also propose and implement an algorithm, called Parameter learning for HIerarchical probabilistic Logic program (PHIL) [3] that learns the parameter of HPLP using EM and gradient method.

Keywords: Probabilistic Logic Programming, Hierarchical PLP, Liftable PLP, Deep Neural Networks, Arithmetic Circuits.

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. In Logic Programs with Annotated Disjunctions (LPADs) [9], programs allow alternatives in the head of clauses. Clauses, C_i, are 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. In order to speed up the inference and the learning in PLP, we consider two restrictions of LPADs: in the first, called Liftable PLP (LPLP), clauses in the program share the same predicate (the target) and inference is performed by reasoning on a whole populations of individuals rather than considering each individual separately [7]. In the second, called Hierarchical PLP (HPLP), see [3] clauses and

[3] The code and the datasets are available at `https://github.com/ArnaudFadja/phil`.

predicates are hierarchically organized and can be translated into an arithmetic circuit (AC) or a deep neural network. Inference in this case is done by evaluating the Network or the Arithmetic circuit. Parameter learning in both approaches can be done by applying gradient method or Expectation Maximization (EM) algorithms.

The paper is organized as follows: Sections 2 and 3 present Liftable PLP and Hierarchical PLP. Section 4 presents parameter learning of LPLP and HPLP and structure learning for LPLP. Finally Section 5 concludes and presents future work.

2 Liftable PLP

In order to improve inference in LPADs, we restrict the language of LPADs by allowing only clauses of the form $C_i = h_i : \Pi_i :- b_{i1}, \ldots, b_{iu_i}$ in the program where all the clauses share the same predicate for the single atom in the head, let us call this predicate r/a with a the arity. The literals in the body have predicates other than r/a and are defined by facts and rules that are certain, i.e., they have a single atom in the head with probability 1. The predicate r/a is called *target* and the others *input predicates*. A program containing n probabilistic clauses of the form above is called liftable PLP. The probability of a query q being true can be compute at liftable level with $P(q) = 1 - \prod_{i=1}^{n}(1 - \Pi_i)^{m_i}$ as described [4] where n is the number of clauses and m_i the number of ground instances of each clause. Here is an example of LPLP where the objective is to predict the target predicate "advised by"

$$
\begin{aligned}
&advisedby(A, B) : 0.3 :- \\
&\quad student(A), professor(B), project(C, A), project(C, B). \\
&advisedby(A, B) : 0.6 :- \\
&\quad student(A), professor(B), ta(C, A), taughtby(C, B).
\end{aligned}
$$

3 Hierarchical PLP

HPLP extends LPLP by adding more layers of rules. In fact, a program in HPLP contains a set of rules that define the target predicate r using a number of input and *hidden predicates*. Hidden predicates are disjoint from input and target predicates. Each rule in the program has a single head atom annotated with a probability. The program is hierarchically defined so that it can be divided into layers. Each layer contains a set of hidden predicates that are defined in terms of predicates of the layer immediately below or in terms of input predicates.

A generic clauses C is of the form

$$C = p(\boldsymbol{X}) : \pi :- \phi(\boldsymbol{X}, \boldsymbol{Y}), b_1(\boldsymbol{X}, \boldsymbol{Y}), \ldots, b_m(\boldsymbol{X}, \boldsymbol{Y}) \tag{1}$$

where $\phi(\boldsymbol{X}, \boldsymbol{Y})$ is a conjunction of literals for the input predicates using variables $\boldsymbol{X}, \boldsymbol{Y}$. The literals $b_i(\boldsymbol{X}, \boldsymbol{Y})$ for $i = 1, \ldots, m$ are built on a hidden predicate.

$\boldsymbol{Y}$ is a possibly empty vector of variables. They are existentially quantified with scope the body. Only literals for input predicates can introduce new variables into the clause and all literals for hidden predicates must use the whole set of variables $\boldsymbol{X}, \boldsymbol{Y}$. Moreover, we require that the predicate of each $b_i(\boldsymbol{X}, \boldsymbol{Y})$ does not appear elsewhere in the body of C or in the body of any other clause. We call hierarchical PLP the language that admits only programs of this form. An example of HPLP is shown in the following program:

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

where $r_{11}/3$ is a hidden predicate.

The grounding of an HPLP can be generated and translated into Arithmetic Circuits (ACs) sharing parameters. In the AC we define 2 operators: the operator $\times$ that computes the joint probability of its arguments (literals in the body of a clause) and the operator $\oplus$ that computes the probability of the disjunction of n independent random variables associated with individual clauses.

4 Parameter and Structure Learning

The parameter learning algorithm can be expressed as follows: Given a LPLP or an HPLP T with parameters Π, an interpretation I defining input predicates and a set of positive and negative examples $E = \{e_1, \ldots, e_M, \text{not } e_{M+1}, \ldots, \text{not } e_N\}$ where each e_i is a ground atom for the target predicate r, find the values of Π that maximize the (log) likelihood (LL):

$$\arg\max_{\Pi} \sum_{i=1}^{M} \log P(e_i) + \sum_{i=M+1}^{N} \log(1 - P(e_i)) \tag{2}$$

or 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 two formulations are equivalent). We propose an EM and a gradient method algorithm for each language.

EM finds the maximun likelihood estimates of parameters in models with hidden variables by alternating between an Expectation and a Maximization step. Gradient method instead computes the partial derivatives of the (log) likelihood w.r.t each parameter in order to move the parameters in the direction that minimize the (log) likelihood. Let X_{ij} be the random variable associated with a grounding $C_i\theta_j$ of clause C_i. To perform EM we need to compute the distribution of the hidden variables given the observed ones, that are $P(X_{ij} = 1|e)$ and $P(X_{ij} = 1|\neg e)$. For a single example e, the Expectation step computes $\mathbf{E}[c_{i0}|e]$ and $\mathbf{E}[c_{i1}|e]$ for all rules C_i where c_{ix} is the number of times a variable X_{ij} takes

value x for $x \in \{0,1\}$ and for all $j \in g(i)$ i.e

$$\mathbf{E}[c_{ix}|e] = \sum_{j \in g(i)} P(X_{ij} = x|e)$$

where $g(i) = \{j|\theta_j \text{ is a substitution grounding } C_i\}$. These values are aggregated over all examples obtaining $E[c_{i0}] = \sum_{e \in E} \sum_{j \in g(i)} P(X_{ij} = 0|e)$ and $E[c_{i1}] = \sum_{e \in E} \sum_{j \in g(i)} P(X_{ij} = 1|e)$.

Then the Maximization computes

$$\pi_i = \frac{\mathbf{E}[c_{i1}]}{\mathbf{E}[c_{i0}] + \mathbf{E}[c_{i1}]} \tag{3}$$

For LPLP, see [4], we have:

$$\begin{aligned} P(X_{ij} = 1|e) &= \frac{\Pi_i}{1 - \prod_{i=1}^{n} (1 - \Pi_i)^{m_i}} \\ P(X_{ij} = 1|\neg e) &= 0 \end{aligned}$$

where m_i is the number of instanciation of C_i whose head is e and Π_i the parameter associate with C_i.

In HPLP, $P(X_{ij} = 1|e)$ is computed by performing two steps over the factor graph associated with the AC as described in [5]

We also propose a gradient method algorithm for LPLP and HPLP.

In LPLP, we present a gradient-based method using LBFGS ([6]), see [4].

For HPLP, the algorithm, called *DPHIL* [1], starts by building a set of ground ACs sharing parameters Π. 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 $d(n)$ 4, that is the gradient, with respect to each parameter

$$d(n) = \begin{cases} d(pa_n)\frac{v(pa_n)}{v(n)} & \text{if n is a } \bigoplus node, \\ d(pa_n)\frac{1-v(pa_n)}{1-v(n)} & \text{if n is a } \times \text{ node} \\ \sum_{pa_n} d(pa_n).v(pa_n).(1-\pi_i) & \text{if n}=\sigma(W_i) \\ -d(pa_n) & pa_n = not(n) \end{cases} \tag{4}$$

where pa_n is the parent of node n and $\pi_i = \sigma(W_i) = \frac{1}{1+e^{-W_i}}$. Parameters are *updated* using Adam the optimizer. These actions are repeatedly performed until a maximum number of steps is reached or until certain conditions are satisfied.

For LPLP we present a discriminative structure learning called LIFTCOVER [4] [4] which, from a set of positive and negative examples, a background knowledge and language bias (defining which predicates can appear in the head/body of a clause) finds a liftable PLP that maximize the LL of the examples. We solve this problem by first identifying good clauses guided by the LL. Clauses are found by

[4] The code and the datasets are available at `https://bitbucket.org/machinelearningunife/liftcover`.

a top-down beam search. The refinement operator adds a literal to the body of the current clause, the literal is taken from a *bottom clause* built as in Progol [2]. The set of clauses found in this phase is then considered as a single theory and parameter learning is performed on it. Then the clauses with a parameter below a user define threshold are discarded and the theory is returned. We experimented and compare these algorithms with state-of-the-art learning algorithms and we obtained similar and often better accuracy in a shorter time.

5 Conclusion and Future Work

In this paper we presented two restrictions of the language of Logic Program with Annotated Disjunction called Liftable and Hierarchical Probabilistic Logic Programming (LPLP and HPLP) that allow fast inference. We also presented an Expectation Maximization and a gradient method algorithm for learning the parameters of both languages. Finally we presented, LIFTCOVER, an algorithm for learning the structure of LPLP. In our future work, we plan to design an implement an algorithm for learning both the parameters and the structure of HPLP. We also plan to learn programs with continuous random variables for HPLP in order to deal with continuous data such as images.

References

1. Fadja, A.N., Riguzzi, F., Lamma, E.: Learning the parameters of deep probabilistic logic programs. In: Bellodi, E., Schrijvers, T. (eds.) Probabilistic Logic Programming (PLP 2018). CEUR Workshop Proceedings, vol. 2219, pp. 9–14. Sun SITE Central Europe, Aachen, Germany (2018)
2. Muggleton, S.: Inverse entailment and Progol. New Generat. Comput. 13, 245–286 (1995)
3. Nguembang Fadja, A., Lamma, E., Riguzzi, F.: Deep probabilistic logic programming. CEUR-WS, vol. 1916, pp. 3–14. Sun SITE Central Europe (2017)
4. Nguembang Fadja, A., Riguzzi, F.: Lifted discriminative learning of probabilistic logic programs. Machine Learning (Aug 2018), `https://doi.org/10.1007/s10994-018-5750-0`
5. Nguembang Fadja, A., Riguzzi, F., Lamma, E.: Expectation maximization in deep probabilistic logic programming. In: Ghidini, C., Magnini, B., Passerini, A. (eds.) Proceedings of the 17th Conference of the Italian Association for Artificial Intelligence (AI*IA2018), Trento, Italy, 20-23 November, 2018. Lecture Notes in Computer Science, Springer, Heidelberg, Germany (2018), `http://mcs.unife.it/~friguzzi/Papers/NguRigLam-AIXIA18.pdf`
6. Nocedal, J.: Updating quasi-newton matrices with limited storage. Math. Comput. 35(151), 773–782 (1980)
7. Poole, D.: First-order probabilistic inference. In: Gottlob, G., Walsh, T. (eds.) IJCAI-2003. pp. 985–991. Morgan Kaufmann Publishers (2003)
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)
9. Vennekens, J., Verbaeten, S., Bruynooghe, M.: Logic Programs With Annotated Disjunctions. In: ICLP-2004. LNCS, vol. 3132, pp. 431–445. Springer (2004)