Deep Probabilistic Logic Programming

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

[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.nguembangfadja,fabrizio.riguzzi,evelina.lamma]@unife.it

Abstract. Probabilistic logic programming under the distribution semantics has been very useful in machine learning. However, inference is expensive so machine learning algorithms may turn out to be slow. In this paper we consider a restriction of the language called *hierarchical PLP* in which clauses and predicates are hierarchically organized. In this case the language becomes truth-functional and inference reduces to the evaluation of formulas in the product fuzzy logic. Programs in this language can also be seen as arithmetic circuits or deep neural networks and inference can be reperformed quickly when the parameters change. Learning can then be performed by EM or backpropagation.

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

1 Introduction

Probabilistic Logic Programming (PLP) under the distribution semantics [26] has found successful applications in machine learning [23,17,12], see in particular the systems SLIPCOVER [8] and LEMUR [11].

These systems need to perform inference repeatedly in order to evaluate clauses and theories. However, inference is expensive so learning may take much time. In this paper we consider a restriction of the language of Logic Programs with Annotated Disjunctions [31] called *hierarchical PLP* in which clauses and predicates are hierarchically organized. In this case the language becomes truth-functional and equivalent to the product fuzzy logic. Inference then is much cheaper as a simple dynamic programming algorithm similar to PRISM [26] is sufficient and knowledge compilation as performed for example by ProbLog2 [12] and cplint [18,24,25] is not necessary

Programs in this language can also be seen as arithmetic circuits or deep neural networks. From a network we can quickly recompute the probability of the root node if the clause parameters change, as the structure remains the same. Learning can then be performed by EM or backpropagation.

The paper is organized as follows: we first introduce general PLP in Section 2, then we present hierarchical PLP in Section 3. Section 4 discusses inference. Connections with related word are drawn in 5 and Section 6 concludes the paper.

2 Probabilistic Logic Programming under the Distribution Semantics

We consider PLP languages under the distribution semantics [26] that have been shown expressive enough to represent a wide variety of domains [2,21,1]. A program in a language adopting the distribution semantics defines a probability distribution over normal logic programs called *instances* or *worlds*. Each normal program is assumed to have a total well-founded model [30]. Then, the distribution is extended to queries and the probability of a query is obtained by marginalizing the joint distribution of the query and the programs.

A PLP language under the distribution semantics with a general syntax is *Logic Programs with Annotated Disjunctions* (LPADs) [31]. We present here the semantics of LPADs for the case of no function symbols, if function symbols are allowed see [20].

In LPADs, heads of clauses are disjunctions in which each atom is annotated with a probability. Let us consider an LPAD T with n clauses: $T = \{C_1, \ldots, C_n\}$. Each clause C_i takes the form: $h_{i1} : \Pi_{i1}; \ldots; h_{iv_i} : \Pi_{iv_i} :- b_{i1}, \ldots, b_{iu_i}$, where $h_{i1}, \ldots, h_{iv_i}$ are logical atoms, $b_{i1}, \ldots, b_{iu_i}$ are logical literals and $\Pi_{i1}, \ldots, \Pi_{iv_i}$ are real numbers in the interval $[0, 1]$ that sum to 1. $b_{i1}, \ldots, b_{iu_i}$ is indicated with $body(C_i)$. Note that if $v_i = 1$ the clause corresponds to a non-disjunctive clause. We also allow clauses where $\sum_{k=1}^{v_i} \Pi_{ik} < 1$: in this case the head of the annotated disjunctive clause implicitly contains an extra atom *null* that does not appear in the body of any clause and whose annotation is $1 - \sum_{k=1}^{v_i} \Pi_{ik}$. We denote by $ground(T)$ the grounding of an LPAD T.

Each grounding $C_i\theta_j$ of a clause C_i corresponds to a random variable X_{ij} with values $\{1, \ldots, v_i\}$ where v_i is the number of head atoms of C_i. The random variables X_{ij} are independent of each other. An *atomic choice* [15] is a triple (C_i, θ_j, k) where $C_i \in T$, θ_j is a substitution that grounds C_i and $k \in \{1, \ldots, v_i\}$ identifies one of the head atoms. In practice (C_i, θ_j, k) corresponds to an assignment $X_{ij} = k$.

A *selection* σ is a set of atomic choices that, for each clause $C_i\theta_j$ in $ground(T)$, contains an atomic choice (C_i, θ_j, k). Let us indicate with S_T the set of all selections. A selection σ identifies a normal logic program l_σ defined as $l_\sigma = \{(h_{ik} :- body(C_i))\theta_j | (C_i, \theta_j, k) \in \sigma\}$. l_σ is called an *instance*, *possible world* or simply *world* of T. Since the random variables associated to ground clauses are independent, we can assign a probability to instances: $P(l_\sigma) = \prod_{(C_i,\theta_j,k)\in\sigma} \Pi_{ik}$.

We consider only *sound* LPADs where, for each selection σ in S_T, the well-founded model of the program l_σ chosen by σ is two-valued. We write $l_\sigma \models q$ to mean that the query q is true in the well-founded model of the program l_σ. Since the well-founded model of each world is two-valued, q can only be true or false in l_σ.

We denote the set of all instances by L_T. Let $P(L_T)$ be the distribution over instances. The probability of a query q given an instance l is $P(q|l) = 1$ if $l \models q$

and 0 otherwise. The probability of a query q is given by

$$P(q) = \sum_{l \in L_T} P(q, l) = \sum_{l \in L_T} P(q|l)P(l) = \sum_{l \in L_T : l \models q} P(l) \quad (1)$$

Example 1. Let us consider the UW-CSE domain [14] where the objective is to predict the "advised by" relation between students and professors. In this case a program for $advisedby/2$ may be

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

where $project(C, A)$ means that C is a project with participant A, $ta(C, A)$ means that A is a teaching assistant for course C and $taughtby(C, B)$ means that course C is taught by B. The probability that a student is advised by a professor depends on the number of joint projects and the number of courses the professor teaches where the student is a TA, the higher these numbers the higher the probability.

Suppose we want to compute the probability of $q = advisedby(harry, ben)$ where $harry$ is a student, ben is a professor, they have one joint project and ben teaches one course where $harry$ is a TA. Then the first clause has one grounding with head q where the body is true and the second clause has one groundings with head q where the body. The worlds where q is true are those that contain at least one of these ground clauses independently of the presence of other groundings so $P(advisedby(harry, ben)) = 0.3 \cdot 0.6 + 0.3 \cdot 0.4 + 0.7 \cdot 0.6 = 0.72$.

3 Hierarchical PLP

Suppose we want to compute the probability of atoms for a predicate r using a probabilistic logic program under the distribution semantics [26]. In particular, we want to compute the probability of a ground atom $r(\boldsymbol{t})$, where $\boldsymbol{t}$ is a vector of term. $r(\boldsymbol{t})$ can be an example in a learning problem and r a *target predicate*. We want to compute the probability of $r(\boldsymbol{t})$, starting from a set of ground atoms (an interpretation) that represent what is true about the example encoded by $r(\boldsymbol{t})$. These ground atoms are build over a set of predicates that we call *input predicates*, in the sense that their definition is given as input and is certain.

We consider a specific form of an LPADs defining r in terms of the input predicates. The program contains a set of rules that define 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. Moreover, 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. The partition of predicates into layers induces a partition of

clauses into layers, with the clauses of layer i defining the predicates of layer i. The rules in layer 1 define r.

This is an extreme form of program stratification: it is stronger than acyclicity [3] because it is not imposed on the atom dependency graph but on the predicate dependency graph, and is also stronger than stratification [9] that, while applied to the predicate dependency graph, allows clauses with positive literals built on predicates in the same layer. As such, it also prevents inductive definitions and recursion in general, thus making the language not Turing-complete.

A generic clauses C is of the form

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

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

A generic program defining r is thus:

$$\begin{aligned}
C_1 = r(\boldsymbol{X}) : \pi_1 &:- \phi_1, b_{11}, \ldots, b_{1m_1} \\
&\ldots \\
C_n = r(\boldsymbol{X}) : \pi_n &:- \phi_n, b_{n1}, \ldots, b_{nm_n} \\
C_{111} = r_{11}(\boldsymbol{X}) : \pi_{111} &:- \phi_{111}, b_{1111}, \ldots, b_{111m_{111}} \\
&\ldots \\
C_{11n_{11}} = r_{11}(\boldsymbol{X}) : \pi_{11n_{11}} &:- \phi_{11n_{11}}, b_{11n_{11}1}, \ldots, b_{11n_{11}m_{11n_{11}}} \\
&\ldots \\
C_{n11} = r_{n1}(\boldsymbol{X}) : \pi_{n11} &:- \phi_{n11}, b_{n111}, \ldots, b_{n11m_{n11}} \\
&\ldots \\
C_{n1n_{n1}} = r_{n1}(\boldsymbol{X}) : \pi_{n1n_{n1}} &:- \phi_{n1n_{n1}}, b_{n1n_{n1}1}, \ldots, b_{n1n_{n1}m_{n1n_{n1}}} \\
&\ldots
\end{aligned}$$

where we omitted the variables except for rule heads. Such a program can be represented with the tree of Figure 1 that contains a node for each literal of hidden predicates and each clause. The tree is divided into levels with levels containing literal nodes alternating with levels with clause nodes.

Each clause node is indicate with $C_{\boldsymbol{p}}$ where $\boldsymbol{p}$ is a sequence of integers encoding the path from the root to the node. For example C_{124} indicates the clause obtained by going to the first child from the left of the root (C_1), then to the second child of C_1 (b_{12}) then to the fourth child of b_{12}. Each literal node is indicated similarly with $b_{\boldsymbol{p}}$ where $\boldsymbol{p}$ is a sequence of integers encoding the path from

the root. Therefore, nodes have a subscript that is formed by as many integers as the number of levels from the root. The predicate of literal $b_{\boldsymbol{p}}$ is $r_{\boldsymbol{p}}$ which is different for every value of $\boldsymbol{p}$.

The clauses in lower layers of the tree include a larger number of variables with respect to upper layers: the head $r_{\boldsymbol{p}ij}(\boldsymbol{X}, \boldsymbol{Y})$ of clause $C_{\boldsymbol{p}ijk}$ contains more variables than the head $r_{\boldsymbol{p}}(\boldsymbol{X})$ of clause $C_{\boldsymbol{p}i}$ that calls it.

The constraints imposed on the program require different program literals to have a unique predicate with respect to the literals in the body of every other clause.

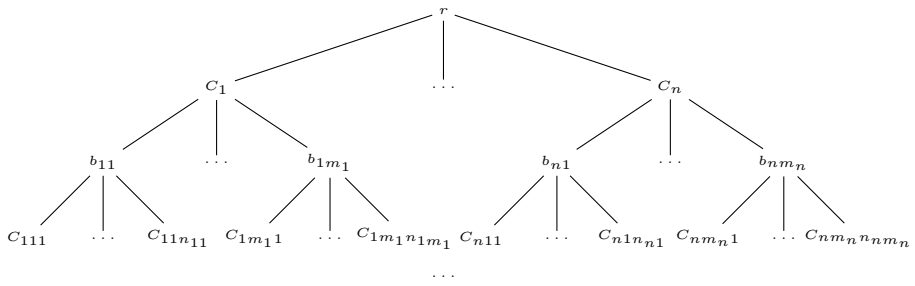

Fig. 1: Probabilistic program tree.

Example 2. Let us consider a modified version of the program of Example 1:

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

where $publication(A,B,C)$ means that A is a publication with author B produced in project C and $student/1, professor/1, project/2, ta/2, taughtby/2$ and $publication/3$ are input predicates.

In this case, the probability of $q = advisedby(harry, ben)$ depends not only on the number of joint courses and projects but also on the number of joint publications from projects. The clause for $r_{11}(A,B,C)$ computes an aggregation over publications of a projects and the clause level above aggregates over projects. Such a program can be represented with the tree of Figure 2.

Writing programs in hierarchical PLP may be unintuitive for humans because of the need of satisfying the constraints and because the hidden predicates may not have a clear meaning. However, the idea is that the structure of the program

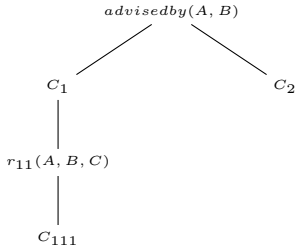

Fig. 2: Probabilistic program tree for Example 2.

is learned by means of a specialized algorithm, with hidden predicates generated by a form of predicate invention. The learning algorithm should search only the space of programs satisfying the constraints, to ensure that the final program is a hierarchical PLP.

4 Inference

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.

Ground atoms are Boolean random variables as well whose probability of being true can be computed by performing inference on the program. Given a ground clause $C_{\boldsymbol{p}i} = a_{\boldsymbol{p}} : \pi_{\boldsymbol{p}i} :- b_{\boldsymbol{p}i1}, \ldots, b_{\boldsymbol{p}im_{\boldsymbol{p}}}$. where $\boldsymbol{p}$ is a path, we can compute the probability that the body is true 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. In fact, different literals in the body depend on disjoint sets of clause random variables because of the structure of the program, so the random variables of different literals are independent as well. 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 $b_{\boldsymbol{p}ik} = \neg a_{\boldsymbol{p}ik}$. Note that if a is a literal for an input predicate, then $P(a) = 1$ if a belongs to the example interpretation and $P(a) = 0$ otherwise.

Now let us determine how to compute the probability of atoms for hidden predicates. Given an atom $a_{\boldsymbol{p}}$ of a literal $b_{\boldsymbol{p}}$, to compute $P(a_{\boldsymbol{p}})$ we need to take into account the contribution of every ground clause for the predicate of $a_{\boldsymbol{p}}$. Suppose these clauses are $\{C_{\boldsymbol{p}1}, \ldots, C_{\boldsymbol{p}o_{\boldsymbol{p}}}\}$. If we have a single clause $C_{\boldsymbol{p}1} = a_{\boldsymbol{p}} : \pi_{\boldsymbol{p}1} :- b_{\boldsymbol{p}11}, \ldots, b_{\boldsymbol{p}1m_{\boldsymbol{p}1}}.$, then $P(a_{\boldsymbol{p}}) = \pi_{\boldsymbol{p}1} \cdot P(body(C_{\boldsymbol{p}1}))$. If we have two clauses, then we can compute the contribution of each clause as above and then combine them with the formula $P(a_{\boldsymbol{p}i}) = 1 - (1 - \pi_{\boldsymbol{p}1} \cdot P(body(C_{\boldsymbol{p}1})) \cdot (1 - \pi_{\boldsymbol{p}2} \cdot P(body(C_{\boldsymbol{p}2})))$ because the contributions of the two clauses depend on a disjoint set of variables and so they are independent. In fact, the probability of a disjunction of two independent random variables is $P(a \vee b) = 1 - (1 - P(a)) \cdot$

$(1 - P(b)) = P(a) + P(b) - P(a) \cdot P(b)$. We can define the operator $\oplus$ that combines two probabilities as follows $p \oplus q = 1 - (1-p) \cdot (1-q)$. This operator is commutative and associative and we can compute sequences of applications as

$$\bigoplus_i p_i = 1 - \prod_i (1 - p_i)$$

The operators $\times$ and $\oplus$ so defined are respectively the t-norm and t-conorm of the product fuzzy logic [13]. They are called *product t-norm* and *probabilistic sum* respectively. For programs of the type above, their interpretation according to the distribution semantics is thus equivalent to that of a fuzzy logic. Therefore the probability of a query can be computed in a truth-functional way, by combining probability/fuzzy values in conjunctions and disjunctions using the t-norm and t-conorm respectively, without considering how the values have been generated: the probability of a conjunction or disjunction of literals depends only on the probability of the two literals. This is in marked contrast with general probabilistic logic programming where knowing the probability of two literals is not enough to compute the probability of their conjunction or disjunction, as they may depend on common random variables.

Therefore, if the probabilistic program of Figure 1 is ground, the probability of the example atom can be computed with the arithmetic circuit [10] of Figure 3, where nodes are labeled with the operation they perform and edges from $\oplus$ nodes are labeled with a probabilistic parameter that must be multiplied by the child output before combining it with $\oplus$.

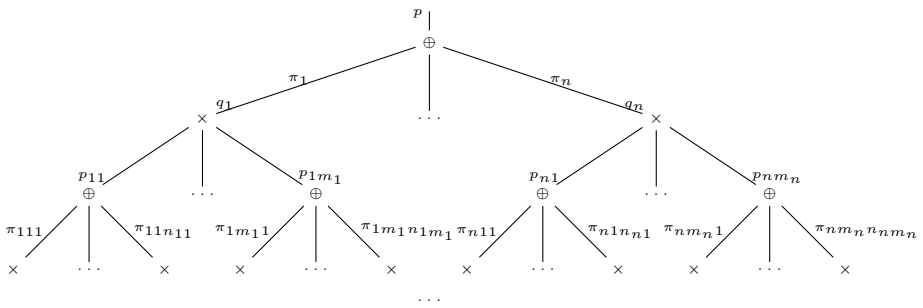

Fig. 3: Arithmetic circuit/neural net.

The arithmetic circuit can also be interpreted as a deep neural network where nodes can have different activation functions: nodes labeled with $\times$ compute the product of their inputs, nodes labeled with $\oplus$ compute a weighted probabilistic sum of this form:

$$\pi_1 \cdot q_1 \oplus \ldots \oplus \pi_n \cdot q_n$$

where q_i is the output of the ith child and π_i is the weight of the edge to the ith child.

The output of nodes is also indicated in Figure 3, using letter p for $\oplus$ nodes and letter q for $\times$ nodes, subscripted with the path from the root to the node. The network built in this way provides the value p of the probability of the example. Moreover, if we update the parameters π of clauses, we can quickly recompute p in time linear in the number of ground clauses.

When the program is not ground, we can build its grounding obtaining a circuit/neural network of the type of Figure 3, where however some of the parameters can be the same for different edges. So in this case the circuit will exhibit parameter sharing.

Example 3. Consider the program of Example 2 and suppose $harry$ and ben have two joint courses $c1$ and $c2$, two joint projects $pr1$ and $pr2$, two joint publications $p1$ and $p2$ from project $pr1$ and two joint publications $p3$ and $p4$ from project $pr2$. The resulting ground program is

$$
\begin{aligned}
G_1 &= advisedby(harry, ben) : 0.3 :- \\
&\quad student(harry), professor(ben), project(pr1, harry), \\
&\quad project(pr1, ben), r_{11}(harry, ben, pr1). \\
G_2 &= advisedby(harry, ben) : 0.3 :- \\
&\quad student(harry), professor(ben), project(pr2, harry), \\
&\quad project(pr2, ben), r_{11}(harry, ben, pr2). \\
G_3 &= advisedby(harry, ben) : 0.6 :- \\
&\quad student(harry), professor(ben), ta(c1, harry), taughtby(c1, ben). \\
G_4 &= advisedby(harry, ben) : 0.6 :- \\
&\quad student(harry), professor(ben), ta(c2, harry), taughtby(c2, ben). \\
G_{111} &= r_{11}(harry, ben, pr1) : 0.2 :- \\
&\quad publication(p1, harry, pr1), publication(p1, ben, pr1). \\
G_{112} &= r_{11}(harry, ben, pr1) : 0.2 :- \\
&\quad publication(p2, harry, pr1), publication(p2, ben, pr1). \\
G_{211} &= r_{11}(harry, ben, pr2) : 0.2 :- \\
&\quad publication(p3, harry, pr2), publication(p3, ben, pr2). \\
G_{212} &= r_{11}(harry, ben, pr2) : 0.2 :- \\
&\quad publication(p4, harry, pr2), publication(p4, ben, pr2).
\end{aligned}
$$

The program tree is shown in Figure 4. The corresponding arithmetic circuit is shown in Figure 5 together with the values computed by the nodes.

The network can be built by performing inference using tabling and answer subsumption using PITA(IND,IND) [19]: a program transformation is applied that adds an extra argument to each subgoal of the program and of the query. The extra argument is used to store the probability of answers to the subgoal: when a subgoal returns, the extra argument will be instantiated to the probability of the ground atom that corresponds to the subgoal without the extra argument. In programs of hierarchical PLP, when a subgoal returns the original arguments are guaranteed to be instantiated.

The program transformation also adds to the body of clauses suitable literals that combine the extra arguments of the subgoals: the probabilities of the

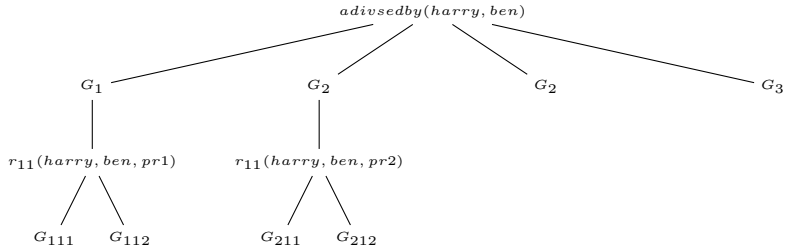

Fig. 4: Ground probabilistic program tree for Example 3.

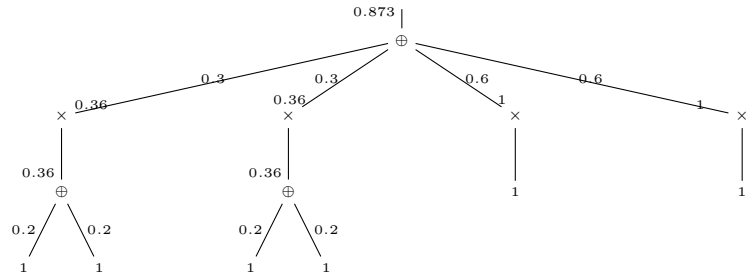

Fig. 5: Arithmetic circuit/neural net for Example 3.

answers for the subgoals in the body should be multiplied together to give the probability to be assigned to the extra argument of the head atom.

Since a subgoal may unify with the head of multiple groundings of multiple clauses, we need to combine the contributions of these groundings. This is achieved by means of tabling with answer subsumption. Tabling is a Logic Programming technique that reduces computation time and ensures termination for a large class of programs [28]. The idea of tabling is simple: keep a store of the subgoals encountered in a derivation together with answers to these subgoals. If one of the subgoals is encountered again, its answers are retrieved from the store rather than recomputing them. Besides saving time, tabling ensures termination for programs without function symbols under the Well-Founded Semantics [30].

Answer subsumption [28] is a tabling feature that, when a new answer for a tabled subgoal is found, combines old answers with the new one. In PITA(IND, IND) the combination operator is probabilistic sum. Computation by PITA(IND, IND) is thus equivalent to the evaluation of the program arithmetic circuit.

Parameter learning can be performed by EM or backpropagation. In this case inference has to be performed repeatedly on the same program with different values of the parameters. So we could use an algorithm similar to PITA(IND,IND) to build a representation of the arithmetic circuit, instead of just computing the probability. To do so it is enough to use the extra argument for storing a term

representing the circuit instead of the probability and changing the implementation of the predicates for combining the values of the extra arguments in the body and for combining the values from different clause groundings.

The results of inference would thus be a term representing the arithmetic circuit, that can be then used to perform regular inference several times with different parameters values.

Implementing EM would adapt the algorithm of [7,6] for hierarchical PLP.

5 Related Work

In [27] the authors discuss an approach for building deep neural networks using a template expressed as a set of weighted rules. Similarly to our approach, the resulting network has nodes representing ground atoms and nodes representing ground rules and the values of ground rule nodes are aggregated to compute the value of atom nodes. Differently from us, the contribution of different ground rules are aggregated in two steps, first the contributions of different groundings of the same rule sharing the same head and then the contributions of groundings for different rules, resulting in an extra level of nodes between the ground rule nodes and the atom nodes.

The proposal is parametric in the activation functions of ground rule nodes, extra level nodes and atom nodes. In particular, the authors introduce two families of activation functions that are inspired by Lukasiewicz fuzzy logic.

In this paper we show that by properly restricting the form of weighted rules and by suitably choosing the activation functions, we can build a neural network whose output if the probability of the example according to the distribution semantics.

Our proposal aims at combining deep learning with probabilistic programming as [29], where the authors propose the Turing-complete probabilistic programming language Edward. Programs in Edward define computational graphs and inference is performed by stochastic graph optimization using TensorFlow as the underlying engine. Hierarchical PLP is not Turing-complete as Edward but ensures fast inference by circuit evaluation. Moreover, it is based on logic so it handles well domains with multiple entities connected by relationships. Similarly to Edward, hierarchical PLP can be compiled to TensorFlow and investigating the advantages of this approach is an interesting direction for future work.

Hierarchical PLP is also related to Probabilistic Soft Logic (PSL) [4] which differs from Markov Logic because atom random variables are defined over continuous variables in the $[0, 1]$ unit interval and logic formulas are interpreted using Lukasiewicz fuzzy logic. We differ from PSL because PSL defines a joint probability distribution over fuzzy variables, while the random variables in hierarchical PLP are still Boolean and the fuzzy values are the probabilities that are combined with the product fuzzy logic. Moreover, the main inference problem in PSL is MAP, i.e., finding a most probable assignment, rather than MARG, i.e., computing the probability of a query, as in hierarchical PLP.

Hierarchical PLP is similar to sum-product networks [16]: the circuits can be seen as sum-product networks where children of sum nodes are not mutually exclusive but independent and each product node has a leaf child that is associated to a hidden random variable. The aim however is different: while sum-product networks represent a distribution over input data, the programs in hierarchical PLP describe only a distribution over the truth values of the query.

Inference in hierarchical PLP is in a way "lifted" [5,22]: the probability of the ground atoms can be computed knowing only the sizes of the populations of individuals that can instantiate the existentially quantified variables,

6 Conclusion

We have presented hierarchical PLP, a restriction of the language of LPADs that allows to perform inference quickly using a simple and cheap dynamic programming algorithm such as PITA(IND,IND). Such a language is particularly useful in machine learning where inference needs to be repeated many times.

Programs in this restricted language can also be seen as arithmetic circuits, similar to sum-product networks, and as neural networks.

The parameters can be trained by gradient descent, computing the gradients using a form of backpropagation. Since random variables associated to clauses are unobserved, an EM approach can also be tried for parameter learning. In the future, we plan to develop and test these parameter learning algorithms and to study the problem of structure learning.

References

1. Alberti, M., Bellodi, E., Cota, G., Riguzzi, F., Zese, R.: `cplint` on SWISH: Probabilistic logical inference with a web browser. Intell. Artif. 11(1) (2017)
2. Alberti, M., Cota, G., Riguzzi, F., Zese, R.: Probabilistic logical inference on the web. In: Adorni, G., Cagnoni, S., Gori, M., Maratea, M. (eds.) AI*IA 2016. LNCS, vol. 10037, pp. 351–363. Springer International Publishing (2016)
3. Apt, K.R., Bezem, M.: Acyclic programs. New Generat. Comput. 9(3/4), 335–364 (1991)
4. Bach, S.H., Broecheler, M., Huang, B., Getoor, L.: Hinge-loss markov random fields and probabilistic soft logic. arXiv preprint arXiv:1505.04406 [cs.LG] (2015)
5. Bellodi, E., Lamma, E., Riguzzi, F., Costa, V.S., Zese, R.: Lifted variable elimination for probabilistic logic programming. Theor. Pract. Log. Prog. 14(4-5), 681–695 (2014)
6. Bellodi, E., Riguzzi, F.: Experimentation of an expectation maximization algorithm for probabilistic logic programs. Intell. Artif. 8(1), 3–18 (2012)
7. Bellodi, E., Riguzzi, F.: Expectation Maximization over Binary Decision Diagrams for probabilistic logic programs. Intell. Data Anal. 17(2), 343–363 (2013)
8. Bellodi, E., Riguzzi, F.: Structure learning of probabilistic logic programs by searching the clause space. Theor. Pract. Log. Prog. 15(2), 169–212 (2015)
9. Chandra, A.K., Harel, D.: Horn clauses queries and generalizations. J. Logic Program. 2(1), 1–15 (1985)

10. Darwiche, A.: A differential approach to inference in bayesian networks. J. ACM 50(3), 280–305 (2003)
11. Di Mauro, N., Bellodi, E., Riguzzi, F.: Bandit-based Monte-Carlo structure learning of probabilistic logic programs. Mach. Learn. 100(1), 127–156 (2015)
12. Fierens, D., Van den Broeck, G., Renkens, J., Shterionov, D.S., Gutmann, B., Thon, I., Janssens, G., De Raedt, L.: Inference and learning in probabilistic logic programs using weighted boolean formulas. Theor. Pract. Log. Prog. 15(3), 358–401 (2015)
13. Hájek, P.: Metamathematics of fuzzy logic, vol. 4. Springer (1998)
14. Kok, S., Domingos, P.: Learning the structure of Markov Logic Networks. In: ICML 2005. pp. 441–448. ACM (2005)
15. Poole, D.: The Independent Choice Logic for modelling multiple agents under uncertainty. Artif. Intell. 94, 7–56 (1997)
16. Poon, H., Domingos, P.M.: Sum-product networks: A new deep architecture. In: Cozman, F.G., Pfeffer, A. (eds.) UAI 2011. pp. 337–346. AUAI Press (2011)
17. Riguzzi, F.: ALLPAD: Approximate learning of logic programs with annotated disjunctions. Mach. Learn. 70(2-3), 207–223 (2008)
18. Riguzzi, F.: Extended semantics and inference for the Independent Choice Logic. Logic Journal of the IGPL 17(6), 589–629 (2009)
19. Riguzzi, F.: Speeding up inference for probabilistic logic programs. Comput. J. 57(3), 347–363 (2014)
20. Riguzzi, F.: The distribution semantics for normal programs with function symbols. Int. J. Approx. Reason. 77, 1 – 19 (October 2016)
21. Riguzzi, F., Bellodi, E., Lamma, E., Zese, R., Cota, G.: Probabilistic logic programming on the web. Softw.-Pract. Exper. 46(10), 1381–1396 (October 2016)
22. Riguzzi, F., Bellodi, E., Zese, R., Cota, G., Lamma, E.: A survey of lifted inference approaches for probabilistic logic programming under the distribution semantics. Int. J. Approx. Reason. 80, 313–333 (January 2017)
23. Riguzzi, F., Di Mauro, N.: Applying the information bottleneck to statistical relational learning. Mach. Learn. 86(1), 89–114 (2012)
24. Riguzzi, F., Swift, T.: The PITA system: Tabling and answer subsumption for reasoning under uncertainty. Theor. Pract. Log. Prog. 11(4–5), 433–449 (2011)
25. Riguzzi, F., Swift, T.: Well-definedness and efficient inference for probabilistic logic programming under the distribution semantics. Theor. Pract. Log. Prog. 13(Special Issue 02 - 25th Annual GULP Conference), 279–302 (2013)
26. Sato, T.: A statistical learning method for logic programs with distribution semantics. In: Sterling, L. (ed.) ICLP 1995. pp. 715–729. MIT Press, Cambridge, Massachusetts (1995)
27. Sourek, G., Aschenbrenner, V., Zelezný, F., Kuzelka, O.: Lifted relational neural networks. In: Besold, T.R., d'Avila Garcez, A.S., Marcus, G.F., Miikkulainen, R. (eds.) NIPS Workshop on Cognitive Computation 2015. CEUR Workshop Proceedings, vol. 1583. CEUR-WS.org (2016)
28. Swift, T., Warren, D.S.: XSB: Extending prolog with tabled logic programming. Theor. Pract. Log. Prog. 12(1-2), 157–187 (2012)
29. Tran, D., Hoffman, M.D., Saurous, R.A., Brevdo, E., Murphy, K., Blei, D.M.: Deep probabilistic programming. In: International Conference on Learning Representations (2017)
30. Van Gelder, A., Ross, K.A., Schlipf, J.S.: The well-founded semantics for general logic programs. J. ACM 38(3), 620–650 (1991)
31. Vennekens, J., Verbaeten, S., Bruynooghe, M.: Logic Programs With Annotated Disjunctions. In: ICLP 2004. LNCS, vol. 3132, pp. 431–445. Springer (2004)