SPLog: Sum-Product Logic

Extended Abstract

ARSENY SKRYAGIN, KARL STELZNER, ALEJANDRO MOLINA, FABRIZIO VENTOLA, and KRISTIAN KERSTING, TU Darmstadt, Germany

1 DEEP PROBABILISTIC PROGRAMMING

In the past three years, many of deep probabilistic programming languages (DPPL) have been proposed, e.g., Edward [Tran et al. 2017] and Pyro [Bingham et al. 2018], among others. The main focus of these works was to leverage the expressive power of deep neural networks within probabilistic programming systems. In particular, DeepProblog [Manhaeve et al. 2018] targets similar goals in the relational setting, by allowing probabilistic predicates to be specified as (conditional) distributions defined via deep neural networks. The resulting systems allow relational probabilistic models with deep components to be trained end-to-end. However, in this setting, deep models are used purely as *conditional* density estimators. This limits the types of inferences that are possible in the resulting system: Lacking any model for the inputs of the neural network, missing values can not be inferred or sampled.

A possible remedy to these issues may be to use deep generative models such as variational autoencoders [Kingma and Welling 2014] or normalizing flows [Rezende and Mohamed 2015]. While these models specify joint probability distributions, inference within them is highly intractable. If the goal of the overall system is to answer a variety of probabilistic queries, the cost of computing them within these models can become prohibitive. To overcome these obstacles we propose a novel deep relational probabilistic PL — "SPLog: Sum-Product Logic", the main components of which are Sum-Product Networks (SPNs) and ProbLog. SPNs [Poon and Domingos 2011] are a type of deep generative model with the crucial property that all conditional and marginal queries may be answered exactly in linear time. In this extended abstract, we show how to leverage this feature in the context of Problog, i.e., how to include SPNs as components within Problog programs, and how to perform joint training and inference.

2 SUM-PRODUCT LOGIC (PROGRAMMING)

ProbLog. The ProbLog language [Kimmig et al. 2007] allows its users to write probabilistic logic programs, in which some logical facts are annotated with probabilities. In particular one can use the annotated disjunctions (AD) to assign logical rules a probability:

$$p :: a(\vec{x}) :- b_1, \ldots, b_m.$$

This example statement indicates if the atoms $b_1, \ldots, b_m$ are true, the predicate $a(x)$ is implied with probability p. This function opened the doors for further possibilities, beyond just specifying constant probabilities. One approach is to delegate the choice of probability to an external model which assigns them based on the predicate's arguments $\vec{x}$. This is the approach taken in Hybrid ProbLog [Gutmann et al. 2010]: here, primitive continuous distributions such as Gaussians are used to specify distributions $p(\vec{X})$, enabling the definition of flexible predicates such as:

$$p(\vec{X} = \vec{x}) :: a(\vec{x}) :- b_1, \ldots, b_m.$$

Authors' address: Arseny Skryagin, arseny.skryagin@cs.tu-darmstadt.de; Karl Stelzner, stelzner@cs.tu-darmstadt.de; Alejandro Molina, molina@cs.tu-darmstadt.de; Fabrizio Ventola, ventola@cs.tu-darmstadt.de; Kristian Kersting, kersting@cs.tu-darmstadt.de, TU Darmstadt, Artificial Intelligence and Machine Learning Lab, Computer Science Department, Germany.

To employ more flexible distributions, DeepProbLog [Manhaeve et al. 2018] uses deep neural networks to provide conditional probabilities for predicates. In this case, predicates of the form:

$$p(\vec{Y} = \vec{y} | \vec{X} = \vec{x}) :: a(\vec{x}, \vec{y}) :- b_1, \ldots, b_m.$$

are specified, where $p(\vec{Y} = \vec{y} | \vec{X} = \vec{x})$ is the probability the neural network assigns the output $\vec{y}$ when given the input $\vec{x}$. By implementing automatic differentiation in ProbLog, the network may be trained jointly with the ProbLog model.

However, not all deep neural networks are safe to encode calibrated distributions with complex dependency structures nor do they encode joint distributions over inputs and outputs. To elevate this problem, we now show how, analogously, ProbLog programs may refer to SPNs as external probabilistic models. This will give rise to the SPLog[1] framework.

SPLog. An SPLog program is a ProbLog program that is extended with a set of ground sum-product annotated disjunctions (spADs) of the form:

$$spn(m_a, \vec{Q}, \vec{E} = \vec{e}) :: a(\vec{e}, \vec{q_1}); \ldots ; a(\vec{e}, \vec{q_n}) :- b_1, \ldots, b_m$$

where the b_i are atoms, $\vec{E} = \vec{e}$ is a vector of random variables representing *evidence* as the input of the SPN for predicate a, $\vec{Q}$ is the set of random variables being *queried*, and $q_1, \ldots, q_n$ are vectors of its realisations. m_a is the identifier of a SPN model which specifies a probability distribution over the set of variables $\vec{X}$, where $\vec{Q} \subseteq \vec{X}, \vec{E} \subseteq \vec{X}, \vec{E} \cap \vec{Q} = \emptyset$. We use the notation $spn(m_a, \vec{Q}, \vec{E} = \vec{e})$ to indicate that this set of logical rules is true with the probabilities $p_{m_a}(\vec{Q} = \vec{q_i} | \vec{E} = \vec{e})$, which the SPN assigns to the query realisations given the evidence. It is clear that SPLog directly inherits its semantics, and to large extent also its inference, from (Deep)ProbLog.

The approach to jointly train the parameters of probabilistic facts and sum-product networks in the SPLog program is the following. Similar to [Manhaeve et al. 2018], we use the *learning from entailment* setting. I.e. for the given SPLog program with parameters $\mathcal{X}$ and a set $\mathcal{Q}$ of pairs (q, p) where q a query and p its desired success probability we compute for a loss function L:

$$\arg\min_{x} \frac{1}{|\mathcal{Q}|} \sum_{(q,p) \in \mathcal{Q}} L\left(P_{\mathcal{X}=\vec{x}}(q), p\right)$$

Given an SPLog program, its SPN models and a query are used as a training example. Specifically, one follows the following steps:

(1) ground the program for the given query
(2) get the current parameters of spADs from the external model
(3) use the ProbLog mechanisms to compute the loss and its gradient
(4) use these to update the parameters of the SPNs and those of the probabilistic program

Specifically, to compute the gradient of the SPN parameters with respect to the loss function (step 3), we need to apply automatic differentiation both to the ProbLog program and to the SPNs. For the former, the algebraic extension of ProbLog [Kimmig et al. 2011] is used. It associates to each probabilistic fact a value from an arbitrary commutative semiring. aProbLog uses a labeling function that explicitly associates values from the chosen semiring with both facts and their negations combining these using semiring addition $\oplus$ and multiplication $\otimes$ on the SDD. For SPLog we are using gradient semiring, whose elements are tuples of the form $(p, \frac{\partial p}{\partial x})$, where p is a probability and $\frac{\partial p}{\partial x}$ is the partial derivative of the probability p with respect to a parameter x. We write $t(p_i) :: f_i$ for the learnable probability of a probabilistic fact. This implements forward mode automatic differentiation within ProbLog, and it results in the partial derivatives of the loss with respect to

[1]Code will be available on Github upon acceptance.

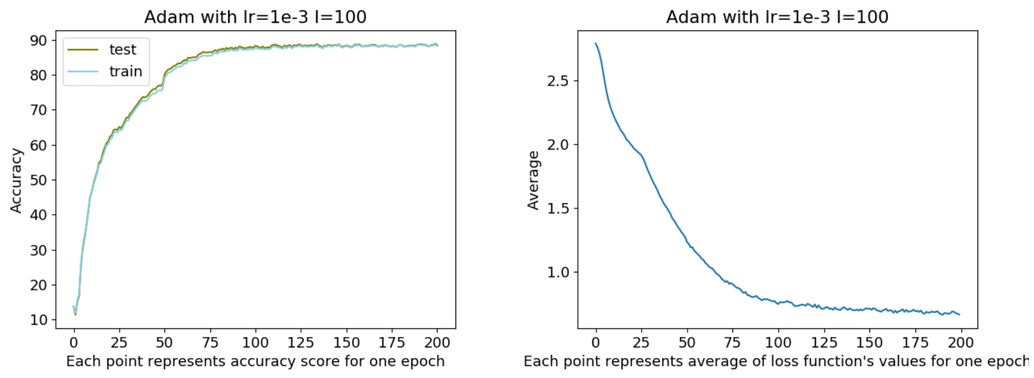

Fig. 1. Accuracy (left) and loss (right) learning curves of the SPLog for the MNIST addition task.

the SPN probabilities $\partial L / \partial p_{m_a}(\vec{Q} = \vec{q}_i | \vec{E} = \vec{e})$. For all technical details we refer to [Manhaeve et al. 2018]. In order to compute the gradients within the SPN, we can use backward mode automatic differentiation as implemented in standard deep learning frameworks. This provides the partial derivatives of SPN outputs with respect to its parameters θ, i.e., $\partial p_{m_a}(\vec{Q} = \vec{q}_i | \vec{E} = \vec{e}) / \partial\theta$. By multiplying the two sets of gradients, we obtain the partial derivatives of the loss with respect to the SPN parameters $\partial L / \partial\theta$, which we use for gradient descent.

3 EMPIRICAL ILLUSTRATION: MNIST ADDITION WITH LOGICALLY COUPLED AES

For our baseline, we replicate the MNIST Addition experiment from [Manhaeve et al. 2018]. We implemented the experiments by using PyTorch based on the SPFlow library [Molina et al. 2019] and the code from [Manhaeve et al. 2018]. The idea behind the experiments is to perform the addition of two digits represented by two randomly chosen images from the MNIST data set. One query of the program contains the indices of the two MNIST images as the inputs and the resulting sum as the label. The corresponding SPLog program is:

```
spn(mnist_spn,[X],Y,[0,1,2,3,4,5,6,7,8,9]) :: digit(X,Y).
addition(X,Y,Z) :- digit(X,X2), digit(Y,Y2), Z is X2+Y2.
```

That is, we specify the spAD as:

$$spn(m_{digit}, \vec{X}, \vec{Y}) :: digit(\vec{x}, 0); \ldots ; digit(\vec{x}, 9)$$

where $\vec{X} = x_1, \ldots, x_k$ is a vector of ground terms representing the inputs of the SPN for predicate *digit* while $0, \ldots, 9$ are possible classes of the SPN. That is, we specify the spAD for an image of the digit 5 as follows:

$$spn(m_{digit}, \boxed{5}, [0, \ldots, 9]) :: digit(\boxed{5}, 0), \ldots, digit(\boxed{5}, 9)$$

Even though SPNs resolve the issues we are addressing, we did not intend to replace the DNNs. Quite the contrary, they play an important role in our framework. Using a simple non-linear auto-encoder based on feed-forward DNN we have run the experiment in the following fashion. The original images were replaced by the codes produced by the encoder network of the pre-trained auto-encoder. The results proved that the model converges well and those can bee seen in Figure 1. With the proposed framework we are also able to deal with partially observed data. This eases the adoption of our framework in many real scenarios where data is missing values.

4 CONCLUSION AND FUTURE WORK

Triggered by the re-emerging research area of Hybrid AI—the computational and mathematical modeling of complex AI systems—we sketched SPLog, a novel deep probabilistic PL (DPPL), which combines deep probabilistic models—SPNs—with neural probabilistic logic programming —DeepProbLog. This paves the way towards a unifying deep programming language for System 2 approaches [Bengio 2019].

SPlog provides several interesting avenues for future work. Because the root layer of the RAT-SPN in our MNIST example represents the posterior probability (i.e. $p(C|X)$) for each class given the image/code, one should investigate the joint distribution $p(C|X) * P(X) = P(C, X)$. This would also allow one to perform inference in any direction. Furthermore, end-to-end training with auto-encoders and more experiments as well as a comparison to logically, coupled variational AEs should be performed. Since "SPNs know what they don't know"[Peharz et al. 2018], rerunning our MNIST addition experiments in a transfer setting is also interesting. The model will be trained on MNIST and then tested on an appropriate part of, say, the SVHN data set. This would show how robust the system is and how well it can recognize unseen data.

ACKNOWLEDGMENTS

The authors would like to thank Steven Lang for very valuable discussions. AS and KK acknowledge support by the German Federal Ministry of Education and Research an the Hessian Ministry of Science and the Arts within the National Research Center for Applied Cybersecurity ATHENE. FV and KK acknowledge the support by the German Research Foundation (DFG) as part of the Research Training Group Adaptive Preparation of Information from Heterogeneous Sources (AIPHES) under grant No. GRK 1994/1. KK also acknowledges the support of the Rhine-Main Universities Network for "Deep Continuous-Discrete Machine Learning" (DeCoDeML).

REFERENCES

Yoshua Bengio. 2019. From System 1 Deep Learning to System 2 Deep Learning. Invited talk NeurIPS. https://www.youtube.com/watch?v=T3sxeTgT4qc

Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. 2018. Pyro: Deep Universal Probabilistic Programming. In *Journal of Machine Learning Research.*

Bernd Gutmann, Manfred Jaeger, and Luc De Raedt. 2010. Extending ProbLog with Continuous Distributions. https://link.springer.com/chapter/10.1007/978-3-642-21295-6_12. *Inductive Logic Programming* (2010).

Angelika Kimmig, Luc De Raed, and Hannu Toivonen. 2007. ProbLog: A probabilistic Prolog and its application in link discovery. In *IJCAI.* 2462–2467.

Angelika Kimmig, Guy Van den Broeck, and Luc De Raed. 2011. An Algebraic Prolog for Reasoning about Possible Worlds. In *AAAI.*

Diederik P. Kingma and Max Welling. 2014. Auto-Encoding Variational Bayes. In *ICLR.*

Robin Manhaeve, Sebastijan Dumančić, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. 2018. DeepProbLog: Neural Probabilistic Logic Programming. In *NeurIPS.* 3753–3763.

Alejandro Molina, Antonio Vergari, Karl Stelzner, Robert Peharz, Pranav Subramani, Nicola Di Mauro, Pascal Poupart, and Kristian Kersting. 2019. SPFlow: An Easy and Extensible Library for Deep Probabilistic Learning using Sum-Product Networks. arXiv:1901.03704.

Robert Peharz, Antonio Vergari, Karl Stelzner, Alejandro Molina, Martin Trapp, Kristian Kersting, and Zoubin Ghahramani. 2018. Probabilistic Deep Learning using Random Sum-Product Networks. In *UAI.*

Hoifung Poon and Pedro Domingos. 2011. Sum-Product Networks: A New Deep Architecture. In *UAI.* 337–346.

Danilo Jimenez Rezende and Shakir Mohamed. 2015. Variational Inference with Normalizing Flows. *Proceedings of the 32nd International Conference on Machine Learning, Lille, France, 2015. JMLR: W&CP volume 37.* (2015).

Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei. 2017. Deep Probabilistic Programming. In *ICLR.*