Tensorised Probabilistic Inference for Neural Probabilistic Logic Programming

Lennert De Smet[1] **Robin Manhaeve**[1] **Giuseppe Marra**[1] **Pedro Zuidberg Dos Martires**[2]

[1]Department of Computer Science, KU Leuven
[2]Department of Computer Science, Örebro University

Abstract

Neural Probabilistic Logic Programming (NPLP) languages have illustrated how to combine the neural paradigm with that of probabilistic logic programming. Together, they form a neural-symbolic framework integrating low-level perception with high-level reasoning. Such an integration has been shown to aid in the limited data regime and to facilitate better generalisation to out-of-distribution data. However, probabilistic logic inference does not allow for data-parallelisation because of the asymmetries arising in the proof trees during grounding. By lifting part of this inference procedure through the use of symbolic tensor operations, facilitating parallelisation, we achieve a measurable speed-up in learning and inference time. We implemented this tensor perspective in the NPLP language DeepProbLog and demonstrated the speed-up in a comparison to its regular implementation that utilises state-of-the-art probabilistic inference techniques.

1 INTRODUCTION

Probabilistic logic inference has recently entered the field of Neural-Symbolic AI (NeSy) by means of the Neural Probabilistic Logic Programming (NPLP) languages [10, 14] and other NeSy methods with probabilistic semantics [11]. These methods promise to facilitate sound probabilistic reasoning [13] on so-called sub-symbolic representations, which are representations of data that do not crisply define the concepts that they contain. For example, an MNIST [7] image of the digit 7 represents the abstract notion of the number 7, but requires processing to actually get to the number 7. While the last decade has shown that neural networks are very efficient in solving problems on such kind of sub-symbolic representations, they lack a form of high-level reasoning. Probabilistic logic, on the other hand, excels in tasks pertaining to such reasoning, but can only be applied to symbolic representations of data. Hence, NeSy tries to bridge these two complementary paradigms.

Probabilistic inference is often mapped onto the weighted model counting task or, more generally, onto the Algebraic Model Counting (AMC) task [6]. Since these tasks are generalisations of model counting, they are also $\#P$-complete and can thus be computationally expensive. There has fortunately been a lot of research into efficient algorithms for model counting, such as tackling AMC by using knowledge compilation [2]. This approach compiles, given a probabilistic query, a series of graphs called arithmetic circuits in which the algebraic model count can be calculated in polytime. However, the structure of these ACs is dependent on the query and hence hard to parallelise. While we will focus on alleviating the computational burden imposed by grounding in the context of NPLP by exploiting data-parallelism, our construction does tensorise part of the probabilistic logic inference via AMC to facilitate this data-parallelism. As AMC is used in other probabilistic logic frameworks [5, 4] as well, the proposed perspective might be of interest to the general statistical-relational AI [12] community.

2 PRELIMINARIES

In this paper, we focus on improving inference in the NPLP setting and we utilise DeepProbLog to illustrate our approach. DeepProbLog inference is composed of three steps: (1) grounding, (2) translation into a logical formula and (3) knowledge compilation. To illustrate each of these steps, we exploit the following example.

Example 2.1 (Addition)**.** Consider the DeepProbLog program

```
nn(classifier, [I], D, [0, ..., 9]) ::
    digit(I, D).

addition(I1, I2, Sum) :-
```

Accepted for the 5th Workshop on Tractable Probabilistic Modeling at UAI (TPM 2022).

```
    digit(I1, D1), digit(I2, D2),
    Sum is D1 + D2.
```

that encodes the task of classifying the sum `Sum` of two MNIST images `I1` and `I2`. The first line is a *neural predicate*, which represents a neural network that yields 10 probabilities for the image `I` to be classified as any of the 10 classes `[0, ..., 9]`. In general, a neural predicate is of the form

```
nn(id, [Input], Output, Domain) ::
    predicate(Input, Output).
```

where `id` is a unique identifier, `Input` is the input to the neural network and `Output` $\in$ `Domain` is the variable that gets unified with elements of the domain. DeepProbLog is end-to-end differentiable, meaning that a set of example ground atoms can be used to optimise all neural networks involved in those atoms.

Given the program $\mathbb{P}$ from Example 2.1 and the ground atom `q = addition(, , 1)`, the first step in inferring the probability $P(\mathtt{q})$ is to ground $\mathbb{P}$ with respect to `q`. In the case of $\mathbb{P}$ and `q`, grounding requires considering all possible combinations of values for `D1` and `D2` to see which are compatible with `Sum = 1`. Hence, only combinations where `D1` and `D2` are equal to 0 and 1 are possible, leading to the ground program

```
0.7 :: digit(, 0); 0.3 :: digit(, 1).
0.1 :: digit(, 0); 0.9 :: digit(, 1).

addition(, , 1) :-
    digit(, 0), digit(, 1).
addition(, , 1) :-
    digit(, 1), digit(, 0).
```

where the probabilities for the instances of the probabilistic fact `digit` are given by evaluating the neural network `classifier`. In the next step, the ground program is rewritten into a propositional logic formula.

```
(digit(, 0) ∧ digit(, 1)) ∨
(digit(, 1) ∧ digit(, 0)).
```

Finally, knowledge compilation [2] is used to convert this propositional logic formula into an effective representation for inference, which we will refer to as a *logic circuit*. Such a logic circuit is transformed according to a certain *commutative semiring* into an Arithmetic Circuit (AC). This is done by replacing all probabilistic facts in the logic circuit with their probability of being true and all occurrences of $\vee$ and $\wedge$ with the addition and multiplication operations of the semiring. The AC for `q` transformed according to the probability semiring $\mathcal{P} = ([0,1], +, \cdot)$ is given in Figure 1. An arithmetic circuit can itself be evaluated by working through its nodes in a bottom-up fashion. If the AC was obtained through a transformation of the logic circuit via the probability semiring $\mathcal{P}$, then the result of evaluating the AC is exactly the probability $P(\mathtt{q})$.

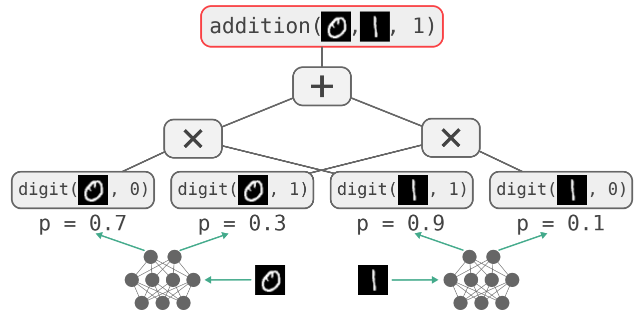

Figure 1: Arithmetic circuit for Example 2.1. The logic circuit is completely identical, but with $+$ and $\times$ replaced by $\vee$ and $\wedge$, respectively.

3 TENSORISATION

Figure 1 illustrates that computing the probability of a ground atom `q` reduces to the evaluation of a computational graph that combines neural networks with an arithmetic circuit. Such a graph could be parallelised to speed up inference and learning when multiple evaluations of the same atom are required, for example during learning. Unfortunately, the logic circuit structure can depend on the arguments of `q`. For example, the value of `Sum = 1` in Example 2.1 determines a ground program where `D1` and `D2` can only take the values 0 and 1, while they can generally be any integer between 0 and 9. The other possible values for `D1` and `D2` are pruned, resulting in a smaller and hence more efficient subsequent arithmetic circuit. As a consequence, such a circuit will be of no use for answering queries with `Sum` different from `1`. While the ACs of all possible queries can be cached and reused, that still necessitates separate groundings of the program for each circuit. Moreover, it still prevents data-parallelisation since different inputs will have different logic circuits and, consequently, different computational graphs. Neural networks exploit data-parallelism and so a data-parallel arithmetic circuit would extend this parallelism to the reasoning component. Hence, we will opt to construct a *tensor-lifted* arithmetic circuit that is valid for any set of input tensors. By explicitly attaching the neural architectures to this lifted AC, we obtain a single graph that can compute probabilities for any set of input tensors. Additionally, only a single grounding step is required during construction of the AC.

3.1 LIFTED ARITHMETIC CIRCUIT

Instead of constructing an arithmetic circuit for a ground atom, the goal is to do so for an atom where some variables are allowed in the arguments. Specifically, only variables that unify with tensors are allowed, which we will call *tensor variables*. For example, instead of the ground atom `addition( , , 1)`, we wish to have an AC valid for the atom `addition(I1, I2, Sum)`. The variables `I1` and `I2` are tensor inputs to the neural predicate and `Sum` is a tensor of target values.

Obtaining a logic circuit that is valid for all instantiations of the tensor variables seems straightforward; we simply do not want to prune any of the possible values during program grounding. The problem is that, without a properly delimited, finite set of possible values, this idea would lead to an infinite logic circuit. In the case of `addition(I1, I2, Sum)`, `Sum` could, a priori, take an infinite number of values. However, the outputs of all neural predicates in a program have a specific and finite set of possible values. Because all tensors are either input to such a neural predicate or the result of a series of operations on the output of a set of neural predicates, they always correspond to a finite set of possible values. Indeed, `Sum` combines two digits in the set `[0, ..., 9]` into their sum that belongs to the set `[0, ..., 18]`. As such, we will compute all necessary domains of target tensors from the domains of the neural predicates during program grounding and prevent that any branches are pruned. This idea is related to lifted first-order inference [3], but in a neural-symbolic setting limited to tensor variables.

Keeping track of all values will ensure a logic circuit that is valid for all tensors. It does not mean that the subsequent tensor-lifted arithmetic circuit, filled with symbolic probabilities, is very efficient in computing the probability $P(\mathtt{q})$. Indeed, the result would be a repetitive AC with many similar branches, since every operation applied to neural predicate outputs will be repeated for each of the possible values of those outputs (Figure 2). Fortunately, exactly because they have a similar or equal format, it is possible to do them in parallel.

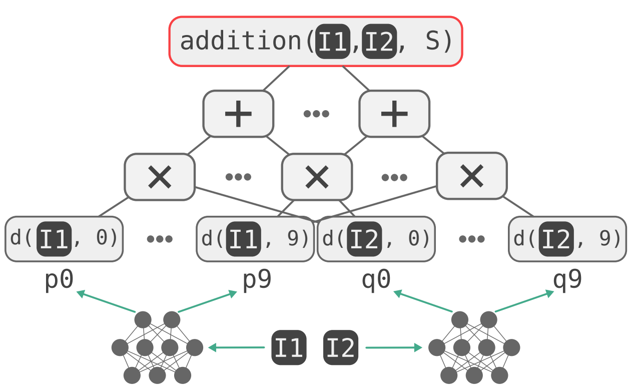

Figure 2: A slice of the general AC for the non-ground query `addition(I1, I2, Sum)`. Note that the top row of additions does depend on a given value of `Sum`. The full AC has 19 of such rows, one for each element of the domain of `Sum`.

In the addition example, there is only one operator that acts on the neural predicate outputs, being the addition of two digits. This operator combines the various possible summands through the conjuncts to then accumulate all combinations that lead to the same value through the disjuncts (Figure 2). To exploit the symmetry of the tensor-lifted AC, the 10 probabilities P_I^D of `digit(I, D)` that image `I` encodes the digit `D` are modelled as a rank-1 tensor $\boldsymbol{P}_{\mathtt{I}}$. Given the two images `I1` and `I2` with probabilities $\boldsymbol{P}_{\mathtt{I1}}$ and $\boldsymbol{P}_{\mathtt{I2}}$ we can compactly represent the probabilities of all combinations, i.e., all conjuncts, via the tensor product $\boldsymbol{P}_{\otimes} = \boldsymbol{P}_{\mathtt{I1}} \otimes \boldsymbol{P}_{\mathtt{I2}}$. Concretely, $\boldsymbol{P}_{\otimes}$ is a rank-2 tensor where the $(\mu, \nu)^{\text{th}}$ component $P_{\otimes}^{\mu\nu}$ corresponds to the probability of combining the digits μ and ν. The aggregation of combinations leading to the same value in the disjuncts can now also be expressed as a summing operation on the tensor $\boldsymbol{P}_{\otimes}$. In particular, the rank-1 tensor of probabilities for the sum result $\boldsymbol{P}_{+}$ is given componentwise as

$$P_{+}^{\gamma} = \sum_{\mu+\nu=\gamma} P_{\otimes}^{\mu\nu}. \tag{1}$$

Replacing the multitude of multiplications originating from the different conjuncts with the tensor product and the various additions corresponding to the disjuncts with the summation in Equation 1 leads to an optimised tensor-lifted AC (Figure 3). While we have used the addition example to guide the discussion of how to exploit the symmetry in a general logic circuit, the provided procedures are valid for any general program $\mathbb{P}$ and atom $\mathtt{q}$ containing tensor variables. More details on general operations are given in Appendix A. A related approach to tensorising probabilistic logic inference based on symbolic variable elimination for undirected factor graphs was proposed by Darwiche [1].

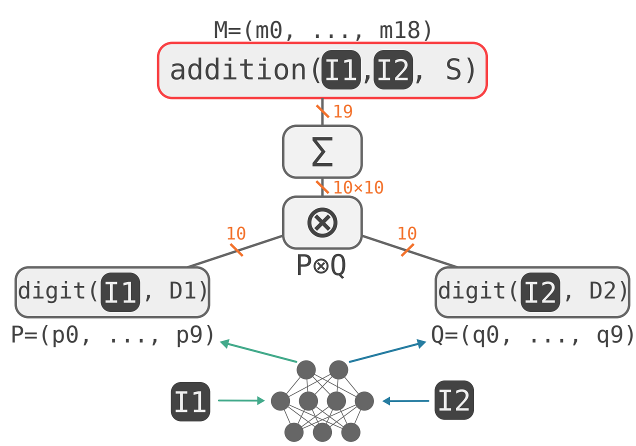

Figure 3: All conjunctions (AND) of digits and their disjunctions (OR) are taken into account by respectively taking a tensor product and summing over the result. All branches are annotated with the dimensionality of the signal that they carry.

Instead of grounding a program for every possible logic circuit, we perform a single, expensive grounding step that symbolically tracks all possibilities in a tensor format. Because of the efficacy of tensor computing, we will manage to achieve more time-efficient inference and learning. Of course, storing all combinatorial information in a tensor format will lead to a combinatorial increase in memory requirements. However, regular AMC implementations do already exploit caching of the constructed ACs to prevent having to perform the expensive grounding step unnecessarily. Hence, they are also putting a combinatorially increasing strain on the memory.

3.2 BATCHING AND OPTIMISATION

We can now explicitly link the tensor-lifted and optimised AC to the neural networks present in the program to obtain the full computational graph of an atom with tensor variables in its arguments. With a computational graph in place, we can naturally exploit data-parallelism through batching of its inputs. This is possible since all equations in the previous section are tensor equations, meaning that we can simply add a batching dimension. For example, computing the probability of all combinations of values for a batch of tensor inputs of the addition operation would now be a rank-3 tensor with components $P_{\otimes}^{b\mu\nu} = P_{\texttt{I1}}^{b\mu} \otimes P_{\texttt{I2}}^{b\nu}$, where b denotes the batching dimension. As for optimisation, the supervision should be a rank-2 tensor $T^{b\gamma}$ that gives, for each batch sample b, the rank-1 tensor of probabilities for each of the elements of the output domain. In the case of the addition example, the supervision would be a one-hot encoded tensor of the sum labels.

3.3 EVALUATION

We compare the learning time in DeepProbLog using the tensor perspective to the standard implementation, which already caches the arithmetic circuits whenever possible to minimise the number of program groundings. Additionally, it also supports batching in the sense that it averages the gradient over a number of executions of the different cached arithmetic circuits. The main point of improvement is thus expected to be the grounding phase, which needs to be repeated for each cached AC. To illustrate the speed-up, we will look at an extension of Example 2.1 to addition on 2-digit numbers [10], with 199 possible logic circuits, one for each possible output sum. The explicit program is given in Appendix B.

The input in this case is two lists of two MNIST images representing two numbers consisting of two digits each together with their resulting sum. The goal is to optimise a neural MNIST classifier from a dataset of correct examples of the addition, such as + = `11`. The fact that we only require a single grounding step manifest itself in significantly reduced initial learning runtimes. Across 10 independent

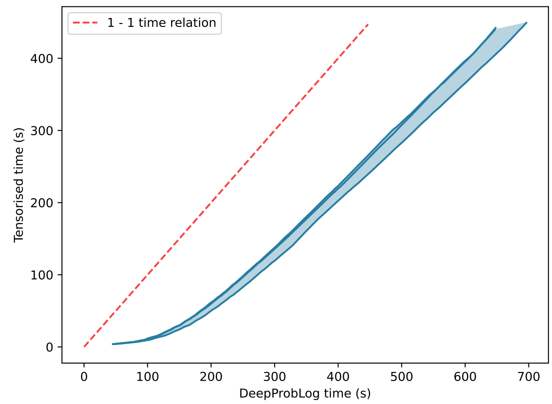

Figure 4: Median and quartiles of regular versus tensorised implementation learning time. Not only is the overall grounding time reduced, but even with all ACs cached, the computational graph is more efficient asymptotically too.

learning instances, the average grounding time was only $2.62s$. Regular DeepProbLog has to construct 199 separate ACs that require 199 expensive grounding steps, leading to an average estimated grounding time of around $200s$. Once all ACs are constructed and caching maximises its effect, the gap in time per iteration of updating closes, yet the tensorised perspective does seem to be slightly faster asymptotically. The latter is not a given, as we always evaluate the full arithmetic circuit while caching does allow evaluating more efficient circuits. It demonstrates the efficacy of tensorising the logical operators and running them in parallel on a GPU.

4 CONCLUSION

A tensor perspective on algebraic model counting for neural probabilistic logic programming was proposed. It allowed the construction of an optimised and lifted computational graph encoding both the neural and logic component of a neural probabilistic logic program. Our construction only requires a single expensive grounding step in contrast to the many ones for the state-of-the-art approach. This advantage leads to a noticeable speed-up in learning and inference.

In the future, we aim to further analyse the parallelisation potential of probabilistic logic. The focus of this paper was on variables that take tensor values and are involved with neural predicates, but it might be interesting to extend this to variables that can take general logic values. An immediate limitation of our approach is that the lifted circuit always takes all possible values into account, while it would make sense to limit to those present in the given dataset. Finally, this tensorisation is also easier to integrate with approximate inference methods based on Monte Carlo sampling [8, 9] through the addition of a separate sampling dimension to the tensors.

ACKNOWLEDGEMENTS

G. Marra has received funding from the Research Foundation-Flanders (FWO, 1239422N).

R. Manhaeve is supported by a KU Leuven post-doctoral mandate.

Pedro Zuidberg Dos Martires and acknowledges the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

L. De Smet wishes to thank Luc De Raedt for his meaningful comments that helped shape this paper into its final form.

References

[1] Adnan Darwiche. An advance on variable elimination with applications to tensor-based computation. In *ECAI 2020*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, pages 2559–2568. IOS Press, 2020.

[2] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *J. Artif. Intell. Res.*, 17:229–264, 2002.

[3] Guy Van den Broeck, Nima Taghipour, Wannes Meert, Jesse Davis, and Luc De Raedt. Lifted probabilistic inference by first-order knowledge compilation. In *IJCAI 2011*, pages 2178–2185. IJCAI/AAAI, 2011.

[4] Pedro M. Domingos and Daniel Lowd. Unifying logical and statistical AI with markov logic. *Commun. ACM*, 62(7):74–83, 2019.

[5] Anton Dries, Angelika Kimmig, Wannes Meert, Joris Renkens, Guy Van den Broeck, Jonas Vlasselaer, and Luc De Raedt. Problog2: Probabilistic logic programming. In *ECML - PKDD 2015*, volume 9286, pages 312–315. Springer, 2015.

[6] Angelika Kimmig, Guy Van den Broeck, and Luc De Raedt. Algebraic model counting. *J. Appl. Log.*, 22: 46–62, 2017.

[7] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proc. IEEE*, 86(11):2278–2324, 1998.

[8] Daniel Lundén, Johannes Borgström, and David Broman. Correctness of sequential monte carlo inference for probabilistic programming languages. In *ESOP 2021*, volume 12648, pages 404–431. Springer, 2021.

[9] Daniel Lundén, Joey Öhman, Jan Kudlicka, Viktor Senderov, Fredrik Ronquist, and David Broman. Compiling universal probabilistic programming languages with efficient parallel sequential monte carlo inference. In *ESOP 2022*, volume 13240, pages 29–56. Springer, 2022.

[10] Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. Neural probabilistic logic programming in deepproblog. *Artif. Intell.*, 298:103504, 2021.

[11] Giuseppe Marra and Ondrej Kuzelka. Neural markov logic networks. In *UAI 2021*, volume 161 of *Proceedings of Machine Learning Research*, pages 908–917. AUAI Press, 2021.

[12] Luc De Raedt, Kristian Kersting, Sriraam Natarajan, and David Poole. *Statistical Relational Artificial Intelligence: Logic, Probability, and Computation*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2016.

[13] Taisuke Sato. A statistical learning method for logic programs with distribution semantics. In *Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming, 1995*, pages 715–729. MIT Press, 1995.

[14] Leon Weber, Pasquale Minervini, Jannes Münchmeyer, Ulf Leser, and Tim Rocktäschel. Nlprolog: Reasoning with weak unification for question answering in natural language. In *ACL 2019*, pages 6151–6161. Association for Computational Linguistics, 2019.

A AC OPTIMISATION DETAILS

Let T be a general n-ary operation in a DeepProbLog program that is applied to n outputs of neural predicates with their respective rank-1 tensors of probabilities $\boldsymbol{P}_i$ and domains $\mathtt{D}_i$. The components of each of these tensors can be indexed by the domain of the neural predicate, i.e., $P_i^{\mu_i}$ corresponds to the probability of the output of the i^{th} network being equal to $\mu_i \in \mathtt{D}_i$. Instead of a single tensor product, all combinations of neural predicate outputs are now efficiently modelled in the rank-n tensor

$$\boldsymbol{P}_{\bigotimes} = \bigotimes_{i=1}^{n} \boldsymbol{P}_i, \tag{2}$$

such that $P_{\bigotimes}^{\mu_1 \cdots \mu_n}$ is the probability of combining the outputs $\mu_i \in \mathtt{D}_i$ for $i \in \{1, \ldots, n\}$. Through the grounding, the domain of the output of T will be determined, say $\mathtt{D}_T$. Obtaining the probability P_T^{ν} that the output of T is equal to $\nu \in \mathtt{D}_T$ is then again given by aggregating the necessary probabilities from $\boldsymbol{P}_{\bigotimes}$ as

$$P_T^{\nu} = \sum_{T(\mu_1, \ldots, \mu_n) = \nu} P_{\bigotimes}^{\mu_1 \cdots \mu_n}. \tag{3}$$

Even more, such an n-ary operation on neural predicate outputs can itself be modelled as a tensor that acts on the tensor of combinations $\boldsymbol{P}_{\bigotimes}$. Specifically, we can generally rewrite Equation 3 in a complete tensorised form as

$$P_T^{\nu} = T^{\nu}_{\mu_1 \cdots \mu_n} P_{\bigotimes}^{\mu_1 \cdots \mu_n}. \tag{4}$$

This equation illustrates that we can fully tensorise the evaluation of an arithmetic circuit originating from an atom and program where neural predicate outputs are used in general n-ary operations.

B MULTI-DIGIT PROGRAM

```
nn(classifier, [I], D, [0, ..., 9]) ::
    digit(I, D).

number(I, N) :- number(I, 0, N).
number([], R, R).
number([H | T], Acc, R) :-
    digit(H, D), Acc2 is D + 10 * Acc,
    number(T, Acc2, R).

addition(I1, I2, Sum) :-
    number(I1, N1), number(I2, N2),
    Sum is N1 + N2.
```