Learning from Logical Constraints with Lower- and Upper-Bound Arithmetic Circuits

Lucile Dierckx [1,2] , **Alexandre Dubray**[1] , **Siegfried Nijssen**[1,2,3]
[1] ICTEAM, UCLouvain, Belgium
[2] TRAIL Institute, Belgium
[3] DTAI, KU Leuven, Leuven, Belgium
{lucile.dierckx, alexandre.dubray, siegfried.nijssen}@uclouvain.be

Abstract

An important class of neuro-symbolic (NeSy) methods relies on knowledge compilation (KC) techniques to transform logical constraints into a differentiable exact arithmetic circuit (AC) that represents all models of a logical formula. However, given the complexity of KC, compiling such exact circuits can be infeasible. Previous works in such cases proposed to compile a circuit for a subset of models. In this work, we will show that gradients calculated on a subset of models can be very far from true gradients. We propose a new framework that calculates gradients based on compiling logical constraints partially in not only a *lower-bound* circuit but also an *upper-bound* circuit. We prove that from this pair of ACs, gradients that are within a bounded distance from true gradients can be calculated. Our experiments show that adding the upper-bound AC also helps the learning process in practice, allowing for similar or better generalisation than working solely with fully compiled ACs, even with less than 150 seconds of partial compilation.

1 Introduction

Neuro-symbolic AI (NeSy) integrates symbolic with sub-symbolic artificial intelligence [Garcez and Lamb, 2023; Hitzler and Sarker, 2022]. An important subclass of NeSy approaches combines neural networks with some form of logic [Giunchiglia *et al.*, 2022]. These approaches typically combine these ideas: (I) A neural network is used to calculate the parameters of a probabilistic model; (II) An inference task is defined on this probabilistic model; (III) This inference task is reduced to an inference task on a weighted logical formula in Conjunctive Normal Form (CNF), where we refer to this formula as a *query*; (IV) The actual outcome of this inference task on the weighted CNF is calculated for the current parameters; (V) A gradient-descent step is executed to learn values of the probabilistic parameters that bring the actual outcome closer to a desired outcome. Well-known examples of such approaches are `DeepProbLog` and semantic loss [Manhaeve *et al.*, 2018; Xu *et al.*, 2018].

For many inference tasks on probabilistic models, it has been found that step (III) requires a reduction to *Weighted Model Counting* (WMC) on logical formulas in CNF [Chavira and Darwiche, 2008; Fierens *et al.*, 2012; Fierens *et al.*, 2015]. In WMC, we calculate a *weighted* sum over *all* satisfying assignments of the logical formula (referred to as *models* of the formula). To execute step (V), the most common approaches are based on *Knowledge Compilation* (KC): a knowledge compiler is used to compile the CNF formula into an *Arithmetic Circuit* (AC) that represents the WMC calculation. Subsequently, standard differentiation techniques are used to calculate a derivative over this circuit and through the neural network. However, compiling a CNF formula into an AC, allowing an exact WMC, is a #P-hard task [Valiant, 1979], making it infeasible for some formulas to be compiled exactly, or resulting in very large ACs. To address this, many existing learning methods rely on approximations of WMC instead, that are based on *subsets* of satisfying assignments [Maene *et al.*, 2024; Manhaeve *et al.*, 2021]. For instance, the k most probable models are calculated and compiled into an AC.

The contributions of this paper focus on the gradient computation through ACs that can not be fully compiled, and are as follows. First, we show that using subsets of models can lead to gradients that are not a good approximation of exactly calculated gradients. Subsequently, we propose the *Lower- and Upper-Bound ACs (`LUBAC`) framework*[1] for calculating approximate gradients that address this weakness. The core novelty in this framework is that we propose to compile two ACs per query: not only a lower-bound circuit representing a subset of models of a formula, *but also an upper-bound* AC representing a subset of non-satisfying assignments. We show that the overhead of calculating both ACs in a modified WMC solver is negligible in theory and practice, while together, these two ACs contain significantly more information than one AC represents. Among others, we will show that, using both ACs, we can not only calculate a new form of approximate gradient, but we can also calculate an *optimality gap* between true and approximated gradients; we can show that this optimality gap is small in theory and practice; and we can obtain ACs that are smaller by carefully selecting which lower-bound and upper-bound ACs we use. Moreover, we show that the resulting ACs can easily be used in existing NeSy pipelines.

[1] https://github.com/aia-uclouvain/schlandals

2 Problem Setting & Related Work

We work in a machine learning setting with labelled examples. Our data consists of *queries* (i.e., logical formulas) associated with a desired probability $\mathcal{Q} = \{(F_1, y_1), \ldots, (F_m, y_m)\}$. Given a set of parameters $\mathcal{W}$ of a model, the learning task is to find values for $\mathcal{W}$ minimising the difference between y_q and $P_{\mathcal{W}}[F_q]$, the probability that each query F_q is satisfied according to this model.

Queries. A query is a propositional logical formula F, in *CNF*, defined over a set $\mathcal{X} = \{x_1, \ldots, x_n\}$ of boolean variables. Such queries can be obtained by grounding probabilistic logic queries, from Bayesian Networks, and more; their origin is not this work's focus. A literal over a variable $x \in \mathcal{X}$ is the variable x or its negation $\neg x$. We study the probabilistic setting in which F is weighted on its literals: a weight w_l is defined for each literal l of the formula, $w_x \in [0, 1]$ and $w_{\neg x} = 1 - w_x$. For simplicity, we define the parameters to learn as $\mathcal{W} = \{w_x \mid x \in \mathcal{X}\}$. We use $w \in \mathcal{W}$ to indicate a parameter that must be learned as well as the current weight of that parameter (i.e., a numerical value).

Model Counting. An interpretation $I = \{l_1, \ldots, l_n\}$ is a truth assignment to each variable (i.e., a choice of one literal per variable). We denote by $F[I]$ the evaluation of F under I, using the standard way of interpreting logical formulas. Let $\mathcal{I}_F$ be the set of all interpretations of F, then the *weighted model counting* (WMC) problem is to compute the weighted sum of F's models, defined as follows.

$$\text{WMC}_{\mathcal{W}}(F) = \sum_{I \in \mathcal{I}_F \mid F[I] = \top} \prod_{l \in I} w_l$$

Given parameter set $\mathcal{W}$, we define $P^*_{\mathcal{W}}[F] = \text{WMC}_{\mathcal{W}}(F)$. $P^*_{\mathcal{W}}[F]$ represents the exact probability that formula F is true, assuming we make choices for each variable $x \in \mathcal{X}$.

Example 1. *Let F be a formula, with three binary variables $\mathcal{X} = \{x_1, x_2, x_3\}$, defined as follows.*

$$F = (\neg x_1 \vee x_3) \wedge (x_2 \vee x_3)$$

Let the weights be $w_{x_1} = 0.99, w_{x_2} = 0.5, w_{x_3} = 0.65$. The interpretation $\{x_1, \neg x_2, x_3\}$, weighted 0.32175, is a model of F and WMC$_{\mathcal{W}}(F) = 0.65175$.

Related Work. A lot of languages have been proposed as targets for knowledge compilation algorithms [Darwiche, 2011; Kisa *et al.*, 2014; Darwiche and Marquis, 2002; Lai *et al.*, 2021]. Most compilers produce a complete representation of the input formula [Oztok and Darwiche, 2015; Lagniez and Marquis, 2017; Lai *et al.*, 2021; Muise *et al.*, 2010]. On the other hand, some methods exist that perform anytime (approximate) knowledge compilation. `PartialKC`, a `CCDD` compiler, iteratively produces partial `CCDD`s until timeout [Lai *et al.*, 2023]. Sampling-based methods can also extract a subset of the models [Gomes *et al.*, 2007]. However, these only provide statistical guarantees on the approximation quality and consider the *unweighted* model counting problem. Our approach is related to the approach taken in the `Schlandals` weighted model counter [Dubray *et al.*, 2023]. `Schlandals` is, to the best of our knowledge, the only weighted model counter providing deterministic lower- and upper-bound on the true WMC [Dubray *et al.*, 2024]. `ProbLog` provides an anytime inference algorithm with lower- and upper-bound by interleaving compilation and reasoning [De Raedt *et al.*, 2007; Vlasselaer *et al.*, 2015].

Previous works have considered learning in a setting where logical formulas are too complex to be completely compiled into an AC. `DeepProbLog` allows approximate learning by only considering a subset of the `ProbLog` proofs. Another framework, `A-NeSI`, proposed to approximate the WMC problem by a neural network [van Krieken *et al.*, 2023]. Dierckx et al. experimented with learning with partial circuits for a subset of the parameters [Dierckx *et al.*, 2024].

3 Compiling Partial Circuits

This section describes how to compile upper- and lower-bound circuits. We first describe how classical compilers work, and then explain our approach for calculating two ACs.

3.1 From Exhaustive Search to Arithmetic Circuit

It is known that the traces of exhaustive DPLL search, with decomposition, correspond to a subset of d-DNNF language [Darwiche, 2001; Huang and Darwiche, 2007] and can be converted to ACs. This has been used in popular compilers such as `dsharp` or `d4` [Muise *et al.*, 2010; Lagniez and Marquis, 2017]. Those adapt the behaviour of classical DPLL model counters to provide a d-DNNF diagram instead of the WMC.

Classical DPLL-style model counters calculate WMC(F) by selecting a variable $x \in \mathcal{X}$, exploring both truth values, propagating the assignments using *Boolean Unit Propagation* (BUP), and recursively exploring the residual formulas.

Example 2. *Figure 1 shows, on the left, a search tree for the formula of Example 1 and, in the centre, an arithmetic circuit computing WMC(F) as derived by a traditional compiler.*

At the root, forcing $x_1 = \top$ (left branch) reduces the first clause to x_3, forcing $x_3 = \top$. This latter assignment satisfies the second clause and reduces F to $\top$. The two models considered in that branch are $\{x_1, x_2, x_3\}$ and $\{x_1, \neg x_2, x_3\}$ and their weighted count is $w_{x_1} w_{x_2} w_{x_3} + w_{x_1} w_{\neg x_2} w_{x_3} = w_{x_1} w_{x_3}$, which corresponds to the AC's left-most part.

To be efficient, such counters integrate a caching system: when the count of a sub-formula F' is computed, it is stored that $C[F'] \mapsto \text{WMC}(F')$. Briefly, classical search-based compilers such as `dSharp` or `d4` build a d-DNNF diagram by replacing the sum with disjunction and the product with conjunction nodes. Accordingly, the cache does not store the solved sub-formula's WMC but the root of their d-DNNF.

A key observation for our work is that such algorithms can easily be modified to halt during their execution, for instance, when a timeout is reached. In such a case, we can still compile a partial d-DNNF representing models encountered until that moment, providing a circuit from which a lower-bound on the true WMC can be calculated. This is illustrated in Figure 1: if we interrupt the search after considering $x_2 = \top$, we have considered the blue part of the search tree, from which we can still compile the blue part of the middle AC. However,

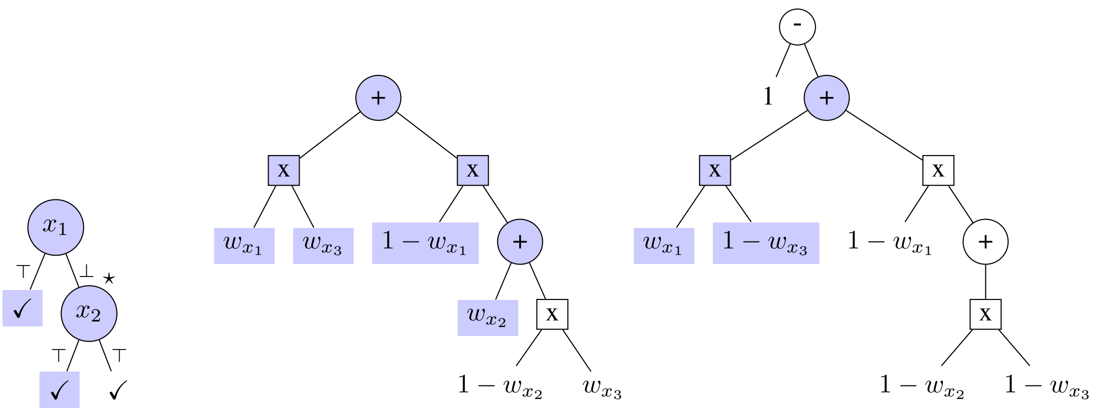

Figure 1: **Left**: A search tree for $F = (\neg x_1 \vee x_3) \wedge (x_2 \vee x_3)$. Leaves corresponding to a model are marked with ✓. **Center**: An AC computing WMC(F), corresponding to the polynomial $w_{x_1} w_{x_3} + (1 - \omega_{x_1})(w_{x_2} + (1 - w_{x_2}) w_{x_3}) = 0.65175$. **Right**: An AC computing the complement of the nonsatisfying assignments' WMC, corresponding to the polynomial $1 - (w_{x_1}(1 - w_{x_3}) + (1 - w_{x_1})(1 - w_{x_2})(1 - w_{x_3}))$. **In Blue**: Matching parts between the ACs and the search tree.

existing compilers are not designed to also produce an upper-bound AC. Next, we discuss a new caching system that allows compiling both lower- and upper-bound (partial) circuits.

3.2 Compilation as a Post-Processing Step

Instead of modifying the search algorithms to produce an AC, we propose storing additional information in the cache and constructing two ACs as a post-processing. Intuitively, the cache contains information about the satisfying *and nonsatisfying assignments* explored with the search. More formally, let F be a boolean formula, x a variable of F, and $F[x]$ and $F[\neg x]$ the two sub-formulas obtained by branching on x. We denote $\text{prop}_x^\top$ (resp. $\text{prop}_{\neg x}^\top$) the set of variables (except x) set to $\top$ when applying BUP with $x = \top$ (resp. $x = \bot$). Our modified cache entry has the following structure:

$$C[F] \mapsto \left(x, (F[x], \text{prop}_x^\top), (F[\neg x], \text{prop}_{\neg x}^\top) \right) \qquad (1)$$

Intuitively, the $\text{prop}^\top$ sets record the interpretations built alongside a branch. The sub-formulas $F[x]$ and $F[\neg x]$ are not stored as-is as values in the cache; they are stored using a hashable representation. For example, the cache entry for the root node of the search tree in Figure 1 would be as follows:

$$(x_1, (\top, [x_3]), (x_2 \vee x_3, [\,]))$$

If the search is partial (i.e., an assignment is not explored), the special marker † replaces the formula in the cache. Hence, if only the left branch of the root is explored, its entry is $(x_1, (\top, [x_3]), (\dagger, [\,]))$. Algorithm 1 shows how the cache can be parsed to produce an AC from the models seen during the search. When computing the (sub-)circuit for a formula F, Algorithm 1 creates a sum node that link the two branches (line 2). Then, it considers both branches of the search tree (lines 4–11), producing a product node (line 6) with the variables propagated to $\top$ during the search (line 8) and the sub-circuit computing the WMC of the residual formula (line 9). For simplicity, we omit static decomposition in Equation (1) and Algorithm 1. However, the structure defined by Equation (1) is easily adaptable for such cases by storing a list of sub-formulas (i.e., $F^1[x], F^2[x], \ldots$) instead of a single residual formula. This caching system is as costly as constructing a d-DNNF during the search. Indeed, the sets $\text{prop}^\top$ correspond to pointers to leaves in a d-DNNF; hence, compilers constructing the diagram directly must store as many edges as there are elements in the $\text{prop}^\top$ sets.

Algorithm 1 Compilation algorithm from a search trace

```
Require: Cache $C$ with entries as defined by Equation (1)
Ensure: An AC computing WMC($F$)
 1: procedure COMPILE($F, C$)
 2:     $N^+ \leftarrow$ new sum node
 3:     $(x, (F[x], \text{prop}_x^\top), (F[\neg x], \text{prop}_{\neg x}^\top)) \leftarrow C[F]$
 4:     for $l \in \{x, \neg x\}$ do
 5:         if $F[l] = \dagger$ then skip $l$
 6:         $N^\times \leftarrow$ new product node
 7:         add $w_l$ as child of $N^\times$
 8:         $\forall x' \in \text{prop}_l^\top$ add $w_{x'}$ as child of $N^\times$
 9:         add Compile($F[l], C$) as child of $N^\times$
10:         add $N^\times$ as child of $N^+$
11:     end for
12:     return $N^+$
13: end procedure
```

To calculate a (partial) circuit representing the nonsatisfying assignments, Algorithm 1 can easily be modified. The main difference is in lines 6–8 of Algorithm 1, for which the lower-bound calculation creates an AC representation of $w_l \times \prod_{x' \in \text{prop}_l^\top} w_{x'}$. For the upper-bound AC, if $\text{prop}_l^\top \neq \emptyset$ we add an AC representation of $w_l \times (1 - \prod_{x' \in \text{prop}_l^\top} w_{x'})$ to the parent summation node; and no node if $\text{prop}_l^\top = \emptyset$. The intuition behind this formula is the following. Since $\prod_{x' \in \text{prop}_l^\top} w_{x'}$ represent the weights of the literal forced to be true by propagation, it means that they *cannot* all be part of a nonsatisfying assignment. The weighted sum of the nonsatisfying interpretations resulting from propagation is given by $1 - \prod_{x' \in \text{prop}_l^\top} w_{x'}$.

In Figure 1, the AC resulting from this compilation on a fully traversed search space is illustrated at the right; in blue is shown the part that is compiled when the search is interrupted before traversing the assignment $x_2 = \perp$. Note that instead of w_{x_3} in the lower-bound circuit, we have $1 - w_{x_3}$ in the upper-bound circuit. When the formula is fully compiled, both ACs calculate exactly the same model count. However, importantly, when the search is interrupted, both circuits represent different model counts, where one corresponds to a lower-bound and the other to an upper-bound. Having both circuits can be valuable: by calculating a probability using both, we can calculate an *optimality gap* that provides information with respect to how good the approximation is.

For the approximation quality, it can be crucial in which order we traverse the search space. In our experiments, we will build on the `Schlandals` WMC solver, which proposed a specific strategy for this.

4 Combining Lower- and Upper-Bound Arithmetic Circuits for Learning

This section motivates using both lower- and upper-bound ACs for gradient approximation and derives its error bound. Then it discusses their integration into a learning setting.

4.1 Analysis of Gradients

Let us show that there are scenarios where the addition of the upper-bound circuit corrects the gradient obtained solely from the lower-bound circuit.

Example 3. *Let us consider the two partial circuits, denoted $P^{lb}_{\mathcal{W}}[F]$ and $P^{ub}_{\mathcal{W}}[F]$, highlighted in blue in Figure 1 and the complete circuit in the centre, denoted $P^{*}_{\mathcal{W}}[F]$. These circuits can be written as polynomials in the following way.*

$$P^{lb}_{\mathcal{W}}[F] = w_{x_1}w_{x_3} + (1 - w_{x_1})w_{x_2}$$
$$P^{ub}_{\mathcal{W}}[F] = 1 - w_{x_1}(1 - w_{x_3})$$
$$P^{*}_{\mathcal{W}}[F] = w_{x_1}w_{x_3} + (1 - w_{x_1})(w_{x_2} + (1 - w_{x_2})w_{x_3})$$

Note that, as $P^{*}_{\mathcal{W}}[F]$ is obtained from the exact AC, it is the most accurate form for the WMC, but cannot always be derived within a reasonable time for complex queries.

The gradients w.r.t. parameter w_{x_1} for each circuits are:

$$\partial P^{lb}_{\mathcal{W}}[F]/\partial w_{x_1} = w_{x_3} - w_{x_2} = 0.15$$
$$\partial P^{ub}_{\mathcal{W}}[F]/\partial w_{x_1} = -1 + w_{x_3} = -0.35$$
$$\partial P^{*}_{\mathcal{W}}[F]/\partial w_{x_1} = w_{x_3} - w_{x_2} - (1 - w_{x_2})w_{x_3} = -0.175$$

This example highlights that even when the absolute difference between the lower bound (0.6485) and the true WMC (0.65175) is small, differentiating only through one partial AC might be insufficient for learning. Specifically, in this case, learning from the lower bound would push the parameter in the opposite direction of the true gradient. This issue also applies to using the upper bound alone. On the other hand, combining the gradients of both bounds through their mean results in a value closer to the true gradient behaviour.

Although the true gradient towards a parameter cannot be computed without a complete AC, the gradients derived from the two partial ACs give an interval in which it lies. Theorem 1 states that the weighted sum of the partial ACs' gradients bound the true gradient and that the size of the interval depends on the bounds' gap.

Theorem 1. *Let F be a boolean formula in CNF over a set of variables weighted by parameters $\mathcal{W}$. Let $P^{*}_{\mathcal{W}}[F]$, $P^{lb}_{\mathcal{W}}[F]$, and $P^{ub}_{\mathcal{W}}[F]$ respectively be the exact, lower-, and upper-bound WMC for formula F. The following inequalities hold for every parameter $w \in \mathcal{W}$:*

$$(1-w)\frac{\partial P^{lb}_{\mathcal{W}}[F]}{\partial w} + w\frac{\partial P^{ub}_{\mathcal{W}}[F]}{\partial w} - \left(P^{ub}_{\mathcal{W}}[F] - P^{lb}_{\mathcal{W}}[F]\right) \leq \frac{\partial P^{*}_{\mathcal{W}}[F]}{\partial w} \leq w\frac{\partial P^{lb}_{\mathcal{W}}[F]}{\partial w} + (1-w)\frac{\partial P^{ub}_{\mathcal{W}}[F]}{\partial w} + \left(P^{ub}_{\mathcal{W}}[F] - P^{lb}_{\mathcal{W}}[F]\right) \quad (2)$$

(Proof in Appendix A available here).

Example 4. *Let us continue with our small example, whose bounds are $P^{lb}_{\mathcal{W}}[F] = 0.6485$ and $P^{ub}_{\mathcal{W}}[F] = 0.6535$. Hence, we have that $P^{ub}_{\mathcal{W}}[F] - P^{lb}_{\mathcal{W}}[F] = 0.005$. The true gradient w.r.t. parameter $w_{x_1} = 0.99$ is bounded as follows. The lower bound is given by*

$$0.01 \times 0.15 + 0.99 \times -0.35 - 0.005 = -0.35$$

The upper bound is given by

$$0.99 \times 0.15 + 0.01 \times -0.35 + 0.005 = 0.15$$

The true gradient, -0.175, is within the bounds.

An interesting property derived from Theorem 1 is that it is possible to bound the distance between the exact gradient and the mean of the lower- and upper-bound gradients. Corollary 1 demonstrates that this bound mainly depends on the two following elements: the bounds' gap and the difference between the partial gradients.

Corollary 1. *The distance between the true gradient, $\partial P^{*}_{\mathcal{W}}[F]/\partial w$, and the mean of the bounds' gradients is bounded as follows:*

$$\left|\frac{\partial P^{*}_{\mathcal{W}}[F]}{\partial w} - \frac{1}{2}\left(\frac{\partial P^{lb}_{\mathcal{W}}[F]}{\partial w} + \frac{\partial P^{ub}_{\mathcal{W}}[F]}{\partial w}\right)\right| \leq \left|(\frac{1}{2} - w)\left(\frac{\partial P^{ub}_{\mathcal{W}}[F]}{\partial w} - \frac{\partial P^{lb}_{\mathcal{W}}[F]}{\partial w}\right) + (P^{ub}_{\mathcal{W}}[F] - P^{lb}_{\mathcal{W}}[F])\right|$$

(Proof in Appendix B available here).

This corollary suggests that, if the gap between the bounds and/or the partial gradients is small, the mean of the partial gradients is a good approximation of the true gradient. We can exploit this in a learning algorithm, as discussed next. Note that in our running example, the approximate gradient would be $(-0.35 + 0.15)/2 = -0.1$, which is closer to the true gradient than either the upper- or lower-bound gradient.

4.2 Learning from Lower- and Upper-Bound ACs

Most gradient descent-based algorithms for learning neural networks are based on iteratively updating parameters in the direction that reduces loss. Let us first assume that we can determine an exact circuit for calculating a model count. Then the loss for a specific instance is written as $\mathcal{L}(y, P^*_{\mathcal{W}}[F])$, where y is the desired probability, $P^*_{\mathcal{W}}[F]$ is the predicted probability based on the exact model count, and $\mathcal{L}(y, \hat{y})$ is a loss function, for instance, $\mathcal{L}(y, \hat{y}) = (y - \hat{y})^2$. To update parameters, we need to calculate for every training instance a parameter update step with respect to the loss function:

$$\frac{\partial \mathcal{L}(y, P^*_{\mathcal{W}}[F])}{\partial w} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \left(y, P^*_{\mathcal{W}}[F]\right) \frac{\partial P^*_{\mathcal{W}}[F]}{\partial w}. \quad (3)$$

Let us now assume we do not have an exact circuit for $P^*_{\mathcal{W}}[F]$, then Corollary 1 suggests we can approximate $\frac{\partial P^*_{\mathcal{W}}[F]}{\partial w}$:

$$\frac{1}{2}\frac{\partial \mathcal{L}}{\partial \hat{y}} \left(y, P^*_{\mathcal{W}}[F]\right) \frac{\partial P^{lb}_{\mathcal{W}}[F]}{\partial w} + \frac{1}{2}\frac{\partial \mathcal{L}}{\partial \hat{y}} \left(y, P^*_{\mathcal{W}}[F]\right) \frac{\partial P^{ub}_{\mathcal{W}}[F]}{\partial w}. \quad (4)$$

The challenge is that we do not know $P^*_{\mathcal{W}}[F]$; hence we also need to approximate it. One approach to deal with this is as follows. Assuming that both circuits calculate probabilities that are close, then we could assume that both $P^*_{\mathcal{W}}[F] \approx P^{ub}_{\mathcal{W}}[F]$ and $P^*_{\mathcal{W}}[F] \approx P^{lb}_{\mathcal{W}}[F]$; our expression becomes

$$\frac{1}{2}\frac{\partial \mathcal{L}}{\partial \hat{y}} \left(y, P^{lb}_{\mathcal{W}}[F]\right) \frac{\partial P^{lb}_{\mathcal{W}}[F]}{\partial w} + \frac{1}{2}\frac{\partial \mathcal{L}}{\partial \hat{y}} \left(y, P^{ub}_{\mathcal{W}}[F]\right) \frac{\partial P^{ub}_{\mathcal{W}}[F]}{\partial w}, \quad (5)$$

which corresponds to

$$\frac{1}{2}\frac{\partial \mathcal{L}(y, P^{lb}_{\mathcal{W}}[F])}{\partial w} + \frac{1}{2}\frac{\partial \mathcal{L}(y, P^{ub}_{\mathcal{W}}[F])}{\partial w}. \quad (6)$$

This points to a straightforward solution in which we execute the following steps: (I) Initialise the weights of the machine learning model; (II) Compile every query, either with a timeout, or with a constraint on the quality of the resulting circuits, providing lower-bound and upper-bound circuits; (III) Treat every lower-bound and upper-bound circuit as a separate training instance during gradient descent.

The advantage of this approach is that any existing NeSy implementation calculating gradients from a given AC can be reused once for every lower-bound and once for every upper-bound AC. Other more complex approaches are also made possible by our framework, which is discussed in Section 6.

5 Experiments

This section describes the solver and datasets used in our experiments. Then, we answer the questions: (I) How well does LUBAC learning generalise? (II) How are the initial gradients impacted by which AC is used? (III) What is the time overhead to compile both lower- and upper-bound ACs?

5.1 Solver Used

We implemented the caching system and compilation algorithm from Section 3 into the Schlandals solver [Dubray *et al.*, 2023]. The latter has two advantages compared to other solvers. First, it natively supports the computation of bounds on the true WMC [Dubray *et al.*, 2024], making the implementation of the caching system relatively easy. Schlandals works directly with the distributions (i.e., non-binary variables); hence, we extended the caching system and compilation algorithm to work with such variables.

Then, it provides two search-based WMC algorithms: a classical exhaustive DPLL-style search and an incremental one. The latter is based on *Limited Discrepancy Search* (LDS), an iterative approach exploring progressively larger parts of the search space [Harvey and Ginsberg, 1995]. LDS's advantage over other anytime methods is that it is designed to make the bounds converge quickly toward the true WMC. Hence, partial ACs computed using LDS are expected to provide better bounds for a given timeout, leading to better gradient estimates from the partial ACs.

Both searches can generate complete or partial ACs from their cache. The classical DPLL search can be run until completion or stopped at a timeout. The LDS-based search, by nature, provides a way to compute partial ACs, as each iteration corresponds to a portion of the search space.

In our experiments, both search strategies will be used. The traditional search is referred to as a DFS search. By default, LDS is used. Additionally, when the DFS is used to produce exact ACs, only the queries that can be fully compiled within 10 minutes are used, excluding the others of the training set. This mimics the usage of a classical compiler in the literature that does not provide approximations or partial compilation.

As indicated, our contribution compared to the literature is the use of two bounding circuits in the training process. To evaluate the potential of using two circuits, we compare with approaches that only compile one circuit, while keeping all other parameters, such as BN encodings and heuristics used, constant. For this reason, we exclude methods working with approximation and learning algorithms such as [Manhaeve *et al.*, 2021; Maene *et al.*, 2024] that rely on other constraint encodings or heuristics.

5.2 Datasets

The probabilistic queries in our experiments originate from Bayesian Networks for which we wish to learn parameters that lead to desired marginal probabilities. This is a well-known problem, and we use networks from the bnlearn R package [Scutari, 2009]. For each network, one dataset of queries is created as follows. For each value v of each network variable N, we create a query $P[N = v]$ (without evidence). These queries sets, with their expected marginal probabilities, form each a dataset for parameter learning. This inference task is challenging, providing queries of various difficulties. Hence, it is possible to evaluate the LUBAC framework as some queries cannot be compiled exactly. Our results are presented on *munin* [Andreassen *et al.*, 1987], *pigs* and *water* [Jensen *et al.*, 1989] datasets except for the experiments in Section 6, covering learning improvements, in which case only the *munin* dataset is used.

5.3 Results

Below, we present our experimental results.

I) How well does LUBAC learning generalise? We first evaluate the generalisation of LUBAC learning and compare it to classical learning with exact ACs and learning with only one of the partial ACs (either lower- or upper-bound). Appendix C (available here) provides the full list of training parameters and train-test split.

		Munin (1384)		Pigs (282)		Water (21)	
Time-out (s)	AC type	Final test MAE	Learn t (s)	Final test MAE	Learn t (s)	Final test MAE	Learn t (s)
	Initial test MAE	0.127		0.093		0.111	
5	LB	0.125	152	0.054	1061	0.0009	51
	UB	**0.112**	987	0.083	2004	0.0179	50
	LUBAC	0.127	3614	**0.017**	2688	**0.0007**	208
50	LB	**0.036**	1331	0.020	1563	0.0110	118
	UB	0.070	3604	0.024	3601	0.0176	196
	LUBAC	0.066	3609	**0.003**	3604	**0.0074**	247
150	LB	**0.025**	2771	0.009	1998	0.0083	161
	UB	0.078	3608	0.012	3603	0.0049	530
	LUBAC	0.072	3624	**0.003**	3603	**0.0047**	798
Exact (600)	DFS	0.096	3600	0.044	1896	0.0473	1

Table 1: Learning generalisation on a test set, with different datasets, compilation timeouts, and AC types. For each combination, the test set MAE at the end of the training and the time taken by the learning loop are given. The first line, represents the test set MAE before training. The best final MAE for each timeout and dataset is in **bold**. The number of training instances is given next to the dataset name.

Table 1 shows the *Mean Absolute Error* (MAE) for the considered settings. It shows that exact ACs generalise poorly compared to partial methods, underperforming compared to 50-second partial compilations while requiring more compilation time. This is especially evident for the *water* dataset, where few queries are usable in the exact setting.

Interestingly, using both ACs does not always improve generalisation; however, it cannot be said beforehand that either an LB or UB circuit always performs better. While the LUBAC framework *bounds the approximate gradient's error*, the true gradient may align more closely with either bound. Our method's strength is ensuring the learning gradients do not significantly deviate from the true ones, reducing the risk of being misled by gradients derived from one bound only.

Additionally, increasing compilation time does not always improve generalisation. For instance, on the *water* dataset, LUBAC learning performs better with 5 seconds of compilation, and the learning time is reduced, as calculating the gradient through smaller circuits is much faster.

II) How are the initial gradients impacted by which AC is used? Corollary 1 states that our approximate gradient's error is bounded. If we use both ACs during training, as the learning progresses, the probabilities calculated by both partial ACs align with the true label values, reducing the ACs' bounds gap and the approximate gradient's error bound. The question remains how our approximate gradient compares to exactly calculated gradients early in the search, when the direction of the gradient descent is determined. Figure 2 shows how the approximate gradient compares to the true gradient for parameters in the *munin* and *pigs* datasets. Each data point corresponds to a parameter. If the data point is in the lower left or upper right quarter of the graph, then the approximate gradient is in the same direction as the true one. The approximate gradient is positively correlated with the true gradient, meaning that, for most parameters, they are in the same direction. The correlation coefficient supports the finding in the previous experiment; for the *pigs* dataset, the combination of ACs works significantly better than using either bound alone, while for *munin* the lower-bound works generally better than the other configurations.

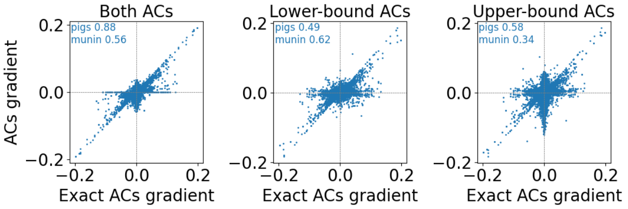

Figure 2: Exact and partial ACs parameter gradients for the first training epoch. The x-axis represents the exact gradient value, while the y-axis shows gradients computed from both (left), lower-bound (centre), or upper-bound (right) AC(s). Points correspond to a parameter gradient from the two datasets' queries. Points near the main diagonal are cases where the partial ACs gradients closely approximate the true gradients. The Pearson correlation coefficient of the datasets is indicated in the upper left of each subfigure.

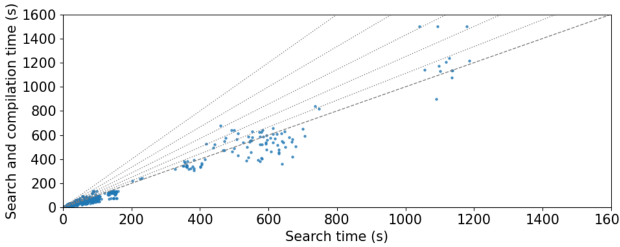

Figure 3: Search vs compilation time. Points are the processing time for the three datasets' queries. The x-axis presents the time needed for the classical search only while the y-axis is the time taken to perform both the search with the new caching system and the ACs compilation. The diagonals are guidelines ranging from $y = 1x$ to $y = 1.5x$ by steps of 0.1. Points on the main diagonal mean that both the search and compilation take the same time, points above it mean the compilation takes longer.

III) What is the time overhead to compile both lower- and upper-bound ACs? We evaluate the runtime overhead of transforming Schlandals' search algorithm into a compiler using the cache system and algorithm in Section 3. Queries solvable within 20 minutes via DFS were timed for two approaches: initial Schlandals search and the proposed compilation. Figure 3 shows that adding compilation has minimal runtime impact for most queries, with only a few requiring up to 40% more time. It can be seen that sometimes the compilation takes *less* time than the search. This variance is due to variances in hardware utilisation (e.g., L3 cache access) between the benchmark sets, which we ran in parallel. In any case, the overhead of the compilation is negligible.

6 Improvements of the Learning Algorithm

The LUBAC framework allows many different forms of learning algorithms in addition to the one discussed in Section 4. Below, we propose several different strategies for learning, as well as some early experimental results.

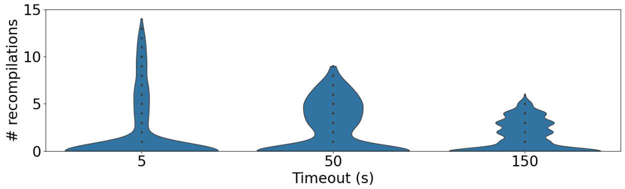

Figure 4: Number of times the gap between a lower- and upper-bound ACs pair increased instead of decreased during the training.

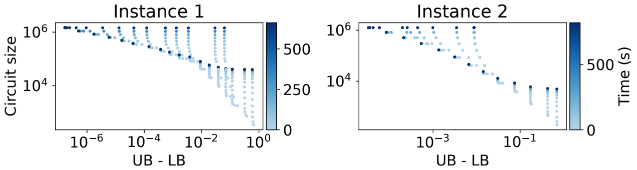

Figure 5: Trade-off between circuits size and bounds' gap. Points are pairs of possible discrepancies for lower- and upper-bound partial ACs, with an LDS timeout of 10 minutes. The y-axis represents the sum of the two circuits' sizes. The x-axis is the difference between the upper- and lower-bound. The colour indicates the time needed to obtain the two ACs.

Loss Computation. In our baseline framework, we made a specific choice for approximating the exact model count $P^*_{\mathcal{W}}[F]$ in Equation 4. While this choice leads to a practical learning algorithm, other choices may be more well-founded from a theoretical perspective. For instance, in [Dubray *et al.*, 2024], it was argued that a better approximation of $P^*_{\mathcal{W}}[F]$ is obtained by calculating the geometric mean $\sqrt{P^{lb}_{\mathcal{W}}[F] * P^{ub}_{\mathcal{W}}[F]}$. Our experiments with this alternative showed that its results were not significantly different from the more practical approach introduced earlier. Hence, we skip further results on this alternative.

Recompilation of Lower- and Upper-Bound AC Pairs. The first step of the LUBAC framework is to compile partially two ACs. In our experiments, we used Schlandals with LDS, which prioritises high-weight interpretations according to the initial choice of weights, enabling faster convergence of upper- and lower-bounds on the probability. However, since parameters change during learning, previously accurate ACs may no longer effectively approximate the true probability. This possible behaviour has also been considered in DeepProbLog and solved by considering alternative approximate compilation techniques [Manhaeve *et al.*, 2018; Manhaeve *et al.*, 2021]. Within LUBAC, maintaining both LB and UB ACs allows tracking the *optimality gap* by simply updating parameters in both circuits. If the bounds diverge significantly, recompilation might be necessary.

Figure 4 shows, for all pairs of corresponding LB and UB ACs, the number of times their bound difference increases by a non-negligible factor (10^{-4} in our experiment) during training for 10 consecutive epochs. The results indicate that recompilation is not necessary for most queries throughout training. The small number of queries that would benefit from it, decreases as the compilation timeout increases.

Trade-Off Between Time, Space, and Bounds Gap. In our baseline approach, we use the ACs produced at the end of the partial compilation algorithm under a per-query timeout. However, these ACs can already be very large, significantly slowing gradient calculations and the learning process, in particular for higher timeouts.

In practice, longer compilation times will not always substantially change the probabilities calculated for the lower- or upper-bound and do not always produce significantly better generalisation. Hence, there are strong advantages to using smaller circuits.

An important observation is that we can modify the caching algorithm relatively easily to roll back towards a partial circuit earlier in the search if that AC has a better precision-size trade-off. By adding timestamps to cache nodes, representing when they were added, we can limit compilation to nodes discovered before a certain point. In an LDS-style search, this timestamp can be the iteration in which the node was added. This can be done independently for the lower and upper bounds, choosing moments when the bounds are close enough and the ACs are small enough. This is a multi-objective optimisation problem; the goal is to find within reasonable time a pair of partial circuits whose bounds' gap and size are small enough.

Figure 5 shows this trade-off for two representative hard instances. Each point is a possible combination of lower-bound and upper-bound pair of partial ACs obtained using Schlandals' LDS search, where also pairs of circuits are considered found at different moments in time. The smallest bound gap (i.e., the leftmost point) is always achieved at the end of the search, requiring the most time (darkest point) and space (highest point). It can be seen that solutions on the Pareto front are not necessarily the ones with the highest compilation time. We leave it as future work on how to best solve this multi-objective optimisation problem; however, our result already demonstrates the potential of performing such an optimisation.

7 Conclusions

This work introduced the Lower- and Upper-Bound ACs (LUBAC) framework for learning from CNF formulas that cannot be compiled into a complete AC. We introduced a new algorithm for compiling two circuits by post-processing the cache of weighted model counters. Our experiment showed that the overhead of such compilation is negligible. We also proved that combining lower- and upper-bound partial ACs allows deriving an optimality gap between true and approximated gradients. We showed that it is straightforward to apply the LUBAC framework to a learning pipeline. Our experiments showed that the LUBAC framework can help the learning to generalise better than with only one partial AC.

The LUBAC framework makes many different learning algorithms possible, as introduced in Section 6. Future work can study these in depth; this includes testing various compilation methods, loss functions, and strategies for selecting only small ACs. Moreover, the LUBAC framework can be evaluated on other benchmarks of learning tasks and can be implemented in different NeSy systems.

Acknowledgments

This work was supported by Service Public de Wallonie Recherche under grant n°2010235 – ARIAC by DIGITAL-WALLONIA4.AI.

References

[Andreassen *et al.*, 1987] Steen Andreassen, Marianne Woldbye, Bjørn Falck, and Stig K Andersen. Munin: A causal probabilistic network for interpretation of electromyographic findings. In *Proceedings of the 10th international joint conference on Artificial intelligence-Volume 1*, pages 366–372, 1987.

[Chavira and Darwiche, 2008] Mark Chavira and Adnan Darwiche. On probabilistic inference by weighted model counting. *Artificial Intelligence*, 172(6-7), 2008.

[Darwiche and Marquis, 2002] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

[Darwiche, 2001] Adnan Darwiche. On the tractable counting of theory models and its application to truth maintenance and belief revision. *Journal of Applied Non-Classical Logics*, 11(1-2):11–34, 2001.

[Darwiche, 2011] Adnan Darwiche. SDD: A new canonical representation of propositional knowledge bases. In *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

[De Raedt *et al.*, 2007] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. ProbLog: A Probabilistic Prolog and Its Application in Link Discovery. In *IJCAI*, volume 7. Hyderabad, 2007.

[Dierckx *et al.*, 2024] Lucile Dierckx, Alexandre Dubray, and Siegfried Nijssen. Parameter learning using approximate model counting. In *International Conference on Neural-Symbolic Learning and Reasoning*, pages 80–88. Springer, 2024.

[Dubray *et al.*, 2023] Alexandre Dubray, Pierre Schaus, and Siegfried Nijssen. Probabilistic Inference by Projected Weighted Model Counting on Horn Clauses. In *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*, 2023.

[Dubray *et al.*, 2024] Alexandre Dubray, Pierre Schaus, and Siegfried Nijssen. Anytime Weighted Model Counting with Approximation Guarantees for Probabilistic Inference. In *30th International Conference on Principles and Practice of Constraint Programming (CP 2024)*, 2024.

[Fierens *et al.*, 2012] Daan Fierens, Guy Van den Broeck, Ingo Thon, Bernd Gutmann, and Luc De Raedt. Inference in probabilistic logic programs using weighted cnf's. *arXiv preprint arXiv:1202.3719*, 2012.

[Fierens *et al.*, 2015] Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. Inference and learning in probabilistic logic programs using weighted Boolean formulas. *Theory and Practice of Logic Programming*, 15(3), 2015.

[Garcez and Lamb, 2023] Artur d'Avila Garcez and Luis C Lamb. Neurosymbolic ai: The 3 rd wave. *Artificial Intelligence Review*, 56(11):12387–12406, 2023.

[Giunchiglia *et al.*, 2022] Eleonora Giunchiglia, Mihaela Catalina Stoian, and Thomas Lukasiewicz. Deep learning with logical constraints. *arXiv preprint arXiv:2205.00523*, 2022.

[Gomes *et al.*, 2007] Carla P. Gomes, Jörg Hoffmann, Ashish Sabharwal, and Bart Selman. From Sampling to Model Counting. In *IJCAI*, volume 2007, 2007.

[Harvey and Ginsberg, 1995] William D Harvey and Matthew L Ginsberg. Limited discrepancy search. In *IJCAI (1)*, pages 607–615, 1995.

[Hitzler and Sarker, 2022] Pascal Hitzler and Md Kamruzzaman Sarker. Neuro-symbolic artificial intelligence: The state of the art. 2022.

[Huang and Darwiche, 2007] Jinbo Huang and Adnan Darwiche. The language of search. *Journal of Artificial Intelligence Research*, 29:191–219, 2007.

[Jensen *et al.*, 1989] FV Jensen, U Kjærulff, KG Olesen, and J Pedersen. An expert system for control of waste water treatment—a pilot project. Technical report, Technical report, Judex Datasystemer A/S, Aalborg, 1989. In Danish, 1989.

[Kisa *et al.*, 2014] Doga Kisa, Guy Van den Broeck, Arthur Choi, and Adnan Darwiche. Probabilistic sentential decision diagrams. In *Fourteenth International Conference on the Principles of Knowledge Representation and Reasoning*, 2014.

[Lagniez and Marquis, 2017] Jean-Marie Lagniez and Pierre Marquis. An Improved Decision-DNNF Compiler. In *IJCAI*, volume 17, 2017.

[Lai *et al.*, 2021] Yong Lai, Kuldeep S Meel, and Roland HC Yap. The power of literal equivalence in model counting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 3851–3859, 2021.

[Lai *et al.*, 2023] Yong Lai, Kuldeep S Meel, and Roland HC Yap. Fast converging anytime model counting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 4025–4034, 2023.

[Maene *et al.*, 2024] Jaron Maene, Vincent Derkinderen, and Luc De Raedt. On the hardness of probabilistic neurosymbolic learning. *arXiv preprint arXiv:2406.04472*, 2024.

[Manhaeve *et al.*, 2018] Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. Deepproblog: Neural probabilistic logic programming. *advances in neural information processing systems*, 31, 2018.

[Manhaeve *et al.*, 2021] Robin Manhaeve, Giuseppe Marra, and Luc De Raedt. Approximate inference for neural probabilistic logic programming. In *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning*, pages 475–486. IJCAI Organization, 2021.

[Muise *et al.*, 2010] Christian Muise, Sheila McIlraith, J Christopher Beck, and Eric Hsu. Fast d-dnnf compilation with sharpsat. In *Workshops at the twenty-fourth AAAI conference on artificial intelligence*, 2010.

[Oztok and Darwiche, 2015] Umut Oztok and Adnan Darwiche. A top-down compiler for sentential decision diagrams. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[Scutari, 2009] Marco Scutari. Learning bayesian networks with the bnlearn r package. *arXiv preprint arXiv:0908.3817*, 2009.

[Valiant, 1979] Leslie G Valiant. The complexity of computing the permanent. *Theoretical computer science*, 8(2):189–201, 1979.

[van Krieken *et al.*, 2023] Emile van Krieken, Thiviyan Thanapalasingam, Jakub Tomczak, Frank Van Harmelen, and Annette Ten Teije. A-nesi: A scalable approximate method for probabilistic neurosymbolic inference. *Advances in Neural Information Processing Systems*, 36:24586–24609, 2023.

[Vlasselaer *et al.*, 2015] Jonas Vlasselaer, Guy Van den Broeck, Angelika Kimmig, Wannes Meert, and Luc De Raedt. Anytime inference in probabilistic logic programs with tp-compilation. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[Xu *et al.*, 2018] Jingyi Xu, Zilu Zhang, Tal Friedman, Yitao Liang, and Guy Broeck. A semantic loss function for deep learning with symbolic knowledge. In *International conference on machine learning*, pages 5502–5511. PMLR, 2018.