# Parameter Learning using Approximate Model Counting

Lucile Dierckx[1,2][0000−0003−2855−1042], Alexandre Dubray[1][0000−0002−3302−870X], and Siegfried Nijssen[1,2][0000−0003−2678−1266]

[1] ICTEAM/INGI, UCLouvain, Louvain-la-Neuve, Belgium
[2] TRAIL Institute, Louvain-la-Neuve, Belgium
{firstname.lastname}@uclouvain.be

**Abstract.** An emerging class of neurosymbolic methods relies on the use of neural networks to determine the parameters of symbolic probabilistic models. To train these hybrid models, these methods use a knowledge compiler to turn the symbolic model into a differentiable arithmetic circuit, after which gradient descent can be performed. However, these methods require compiling a reasonably sized circuit, which is not always possible, as for many symbolic probabilistic models calculating a gradient towards the parameters is $\#P$-hard. We introduce a new approach for learning parameters using *partially* compiled circuits with approximation nodes. We show that, if the errors made in the approximation nodes are bounded, the error on the gradient of partially compiled circuits can also be bounded. We evaluate the impact of various approximation guarantees on this approach's learning and generalization performance. Using approximation allows more complex queries to be compiled and our experiments show that their addition helps reduce the training loss. However, we observe that there is a limit to the addition of partial circuits after which there is no more improvement.

**Keywords:** Model Counting · Parameter Learning · Knowledge Compilation

## 1 Introduction

An emerging class of machine learning methods relies on the use of differentiable machine learning models, such as Neural Networks, to fill in the parameters of symbolic probabilistic models, such as Bayesian Networks or probabilistic logic programs [16]. Training in these methods requires an ability to calculate a gradient towards the parameters of the symbolic model. Unfortunately, this can be challenging: calculating an exact gradient from an error on training instances requires solving a $\#P$-hard *weighted model counting* (WMC) problem [4].

Many algorithms exist to perform WMC tasks [2,3,9,21]. One family of such algorithms is based on compiling an arithmetic circuit (AC), which, when evaluated, produces the exact desired probability. In practice, these circuits perform product and sum operations, which makes them differentiable [4]. The parameters of the circuits (i.e., their inputs) can then be learned by gradient descent.

However, given the #P-hardness of WMC, compiling complete circuits can be too computationally expensive for hard training instances. In this work, we develop a new learning approach that can be used on such harder instances. It is based on *partially* compiling circuits, in which some sub-circuits are replaced by an approximate WMC, effectively reducing the size and compile time of ACs. Such partial ACs provide an approximate probability; hence, their gradients differ from fully compiled ACs, but doing so allows to consider more instances for training. We analyze the impact of this trade-off and show that adding more instances, even though approximated, benefits learning. Moreover, we show that, if the WMC calculated for approximation nodes is obtained with an $\varepsilon$-approximation quality guarantee, the error on the gradient is also bounded.

## 2 Related Work

Various learning methods rely on ACs for learning. In DeepProbLog [16] a probabilistic logic program is compiled into an AC, which is then combined with neural networks. As the outputs of the network are used as the input of the differentiable AC, the training of the neural net can be done through the logical circuit directly. While the authors of DeepProbLog proposed the use of compilers for Sentential Decision Diagrams [5], any tool that produces an AC can be used. It has for example been shown that gradients can be efficiently computed through Ordered Binary Decision Diagrams [15]. Alternatively, *Expectation Maximization* (EM) has been used to learn parameters in ProbLog [6] or in *Probabilistic Sentential Decision Diagrams* [12]. *Probabilistic Circuits* are generally learned with EM; for example, Einsum networks combine EM with automatic differentiation to speed up the learning process [19].

ACs allow to perform model counting that easily integrates with neural networks. ACs are typically constructed by algorithms that are modifications of model counting solvers. A number of such solvers exist. In this work, we build on Schlandals [7], which efficiently handles projected WMC and multi-valued distributions. Traditional model counters perform exact inference, but various methods exist to perform *approximate* WMC, each offering various guarantees. For example, sampling-based methods [10,14] provide statistical guarantees, which get stronger with the methods' runtime. On the other hand, methods providing $(\varepsilon, \delta)$-guarantees [1,2,8,22,23] return an $\varepsilon$-approximation (i.e., with bounded error) with probability $1 - \delta$. Combining these approximate techniques with parameter learning is much less explored. One approach for DeepProbLog [17] is based on proofs of the logic program and does not provide approximation guarantees.

## 3 Learning with Partial Circuits

This section presents our framework for learning from partially compiled circuits. We describe the problem setting considered in this work and show that

it is possible to learn from such circuits. Furthermore, we show that the gradient error, compared to a fully compiled circuit, can be bounded under certain approximation guarantees.

**Problem Setting** We consider a machine learning setting with labeled data $Q$ consisting of *queries* made of logical formulas $F_q$ in *conjunctive normal form* over boolean variables $\mathcal{V}$. Each formula $F_q$ encodes an inference task on a symbolic probabilistic model (such as a Bayesian Network or a probabilistic logic program) and is associated with a desired probability $y_q$; hence, $Q = \{(F_1, y_1), ..., (F_m, y_m)\}$. Let $\mathcal{W}$ be a set of *parameters* of the model, each giving a weight to a variable of $\mathcal{V}$; we define for each query $F_q$ a *weighted model count* $P_{\mathcal{W}}[F_q]$. The goal is to learn the set of parameters minimizing the difference between $P_{\mathcal{W}}[F_q]$ and $y_q$ for all queries, where we assume that not all parameters need to be learned: some belong to known *distributions* $\mathcal{D}_{NT}$; only parameters that belong to a set of trainable parameters $\mathcal{D}_T$ need to be learned.

**Partial Compilation** Our contribution is an approach for creating a *partially* compiled AC. The core idea is that, for calculating gradients for trainable parameters, an AC is sufficient in which only trainable parameters are leaves. One approach for building such as ACs would be: (1) compile an AC over all parameters; (2) fill in all known parameters; (3) simplify the resulting AC, such that leaves are only either trainable parameters or constants; (4) use this simplified AC for calculating gradients. However, this would still require a complete and exact AC compilation, which we wish to avoid. We propose a different approach that obtains the same output.

For this purpose, we modify a DPLL-style, search-based model counter, as summarized in Algorithm 1. This algorithm is similar to other model counters. The probability $P[F]$[3] is computed recursively. First, a variable $v$ is selected to branch on (line 5), and the two possibilities are explored (lines 6-11). For each assignment to $v$, boolean unit propagation is applied (line 7), and if the resulting formula is not unsatisfiable, the residual formula is decomposed into independent components (line 8) that are solved recursively (line 9). We store the *trace* of this algorithm (details not shown) to compile an AC [11] that alternates between sum nodes (branching) and product nodes (independent components decomposition).

The important difference in line 3: when there are no more variables whose weight must be learned (i.e., only variables from $\mathcal{D}_{NT}$ are in the formula), then we stop the AC compilation process for that formula, and determine a constant that is put in the AC. A second core idea is that to calculate this constant, we can also use any *approximate* model counter `Approx-WMC` from the literature to calculate an approximation $\tilde{P}[F]$ of $P[F]$, including exact methods. (in such case $\tilde{P}[F] = P[F]$).

We can prove that under certain conditions on $\tilde{P}[F]$, it is possible to obtain guarantees on the quality of a gradient calculated based on the partial AC for the queries original. If $\tilde{P}[F]$ is an $\varepsilon-$approximation, that is $\frac{P[F]}{1+\varepsilon} \leq \tilde{P}[F] \leq$

[3] We dropped the $\mathcal{W}$ and $q$ notation for conciseness.

**Algorithm 1:** DPLL-style Weighted model Counter

```
1  Function WMC(F)
       input  : A boolean formula F over variables 𝒱
       output : P[F] the weighted model count of F
2      if |𝒱| = 0 then return 1
3      if 𝒱 ∩ 𝒟_T = ∅ then return Approx-WMC(F)
4      if F ↦ p in cache then return p
5      P[F] ← 0; v ← choose_variable()
6      foreach x ∈ {⊤, ⊥} do
7          if F[v = x] = ⊥ then continue // Applies propagation
8          C ← components of F[v = x]            // Independent components
9          foreach F_c ∈ C do p^c ← F_c
10         P[F] += w_v × ∏_c p^c                                  // w_v ∈ 𝒲
11     end
12     Add F ↦ p in cache and return p
13 return WMC(F)
```

$P[F](1+\varepsilon)$, then, Theorem 1 bounds the distance between the approximate gradient $\partial\tilde{P}[F]/\partial w$ and the exact gradient $\partial P[F]/\partial w$. The larger the $\varepsilon$, the larger the distance from the true gradient can be.

**Theorem 1.** *Let $\tilde{P}[F]$ represent $P[F]$ calculated by approximating subformulas of $F$ over non-trainable parameters with error rate $\varepsilon$. Then*

$$|\frac{\partial\tilde{P}[F]}{\partial w}| \leq |\frac{\partial P[F]}{\partial w}| + \varepsilon(\frac{3}{4} + \frac{1}{2}\varepsilon)$$

*(Proof in Appendix A).*

## 4 Experimental Results

This section describes our datasets and our training protocol. We answer these questions: I) What is the impact of the partial compilation on convergence? II) How well does the approximate learning process generalize to unseen examples? III) How does the number of queries used for training impact the learning?

### 4.1 Datasets and Training Protocol

**Dataset** Two types of probabilistic queries are used in our experiments: Computing the marginal probabilities in Bayesian Networks, and Reliability Estimation in probabilistic graphs. The former type is well known, and we use networks from the bnlearn R package [20]. One dataset is created for each network, and the instances are created as follows. For each network variable $N$, we create, for each of its values $v$, a query computing $P[N = v]$ (without evidence).

The second type aims at computing the probability that two nodes are connected in a graph in which each edge has a probability of being present. The

datasets used for that problem come from two sources. First, we use graphs extracted from the GridKit tool [18,24], representing high-voltage networks in Europe and the USA. One dataset is created for each country (Europe) or state (USA). Since the graphs are not directed, the instances are created by taking random source-target pairs until all nodes are ensured to be in a query. By doing so, we ensure that there is a possibility of learning all distributions in the networks (even if, in practice, not all queries can be compiled). We also use water supply networks coming from [13]. For these (directed) graphs, a query is created for each source (a parent-less node) and sink (a child-less node) in the graph, removing queries in which no path exists between the nodes.

**Training Protocol** We build the partial ACs using Schlandals [7] which was modified[4] to save its search trace and create approximation nodes as shown in Algorithm 1. This work considers partial ACs producing $(\varepsilon, \delta)-$approximations. We do not consider guarantee-less methods as it can be seen as a special case of $(\varepsilon, \delta)-$guarantees with a large $\varepsilon$. We simulated the approximation process by first computing the exact probability of the node (i.e., equivalent to a $(0, 0)$-approximation) and then returning an approximation taken at random in the allowed interval (method detailed in Appendix B). All experiments are repeated 10 times and the mean results are shown for a subset of the data sets[5]. The distributions are split such that 70% are known and the remaining 30% are to be learned. The datasets are limited to the queries that can be compiled in 10 minutes.

The training uses the Mean Absolute Error (MAE) as a loss and the optimizer used to update the distribution values is the Stochastic Gradient Descent without momentum, as our study here focuses on gradient computation and not on finding the most efficient optimizer. We set a limit of 6000 epochs and apply early stopping if the loss improvement is below $10^{-5}$ during five epochs or if the average loss is under $10^{-4}$. Additionally, a one-hour training time limit is set to stop the learning after that duration. The learning rate is initially set to 0.3 and follows a step-wise decrease every 100 epoch with a decay factor of 0.75.

### 4.2 Results

**I) What is the impact of the partial compilation on the convergence?** First, we analyze the convergence of the learning process when partially compiled ACs are used. Table 1 summarizes the average loss after convergence for various configurations of $(\varepsilon, \delta)$. We have $(\varepsilon, \delta) = (0, 0)$ as the baseline since it mimics the learning of the fully compiled AC. With looser guarantees, the loss on the training set is usually larger. Interestingly, there seems to be a relationship between the training error loss and the $\varepsilon$ parameter: the loss increases roughly linearly with the $\varepsilon$. For all values of $\delta$ there is a greater increase in final loss.

[4]The modifications are available on Schlandals GitHub `https://github.com/aia-uclouvain/schlandals`.

[5]The observed behaviors also apply to the other data sets, but we do not include them for conciseness reasons.

Table 1: Average MAE loss ± its standard deviation at convergence over 10 runs for three different datasets, using various approximation guarantees.

| | | $\varepsilon = 0.0$ (exact) | $\varepsilon = 0.05$ | $\varepsilon = 0.3$ | $\varepsilon = 0.8$ |
|---|---|---|---|---|---|
| Munin | $\delta = 0.0$ | 0.036 ± 0.0 | 0.037 ± 1.7e-04 | 0.042 ± 1.2e-03 | 0.052 ± 2.6e-03 |
| | $\delta = 0.2$ | | 0.090 ± 9.1e-03 | 0.095 ± 8.9e-03 | 0.110 ± 9.7e-03 |
| | $\delta = 0.5$ | | 0.170 ± 0.010 | 0.170 ± 0.010 | 0.180 ± 0.010 |
| | $\delta = 0.8$ | | 0.250 ± 0.012 | 0.260 ± 0.012 | 0.270 ± 0.011 |
| Tennessee | $\delta = 0.0$ | 9.1e-04 ± 1.1e-19 | 5.0e-03 ± 1.8e-03 | 0.018 ± 9.5e-03 | 0.036 ± 0.018 |
| | $\delta = 0.2$ | | 0.015 ± 0.015 | 0.027 ± 0.023 | 0.044 ± 0.031 |
| | $\delta = 0.5$ | | 0.066 ± 0.046 | 0.079 ± 0.049 | 0.093 ± 0.050 |
| | $\delta = 0.8$ | | 0.077 ± 0.037 | 0.091 ± 0.037 | 0.110 ± 0.036 |
| Net3 | $\delta = 0.0$ | 8.0e-04 ± 1.1e-19 | 2.5e-03 ± 3.5e-04 | 8.5e-03 ± 2.8e-03 | 0.017 ± 5.3e-03 |
| | $\delta = 0.2$ | | 8.1e-03 ± 5.9e-03 | 0.015 ± 6.9e-03 | 0.023 ± 8.7e-03 |
| | $\delta = 0.5$ | | 0.019 ± 0.013 | 0.025 ± 0.014 | 0.032 ± 0.014 |
| | $\delta = 0.8$ | | 0.035 ± 0.017 | 0.042 ± 0.016 | 0.049 ± 0.014 |

Table 2: The MAE loss computed on a test set *before* and *after* the learning of the distributions on a training set. The values are presented as: *before* → *after*.

| | | $\varepsilon = 0.0$ (exact) | $\varepsilon = 0.05$ | $\varepsilon = 0.3$ | $\varepsilon = 0.8$ |
|---|---|---|---|---|---|
| Munin | $\delta = 0.0$ | 0.150 → 0.094 | 0.160 → 0.096 | 0.160 → 0.099 | 0.170 → 0.110 |
| | $\delta = 0.2$ | | 0.200 → 0.130 | 0.200 → 0.130 | 0.210 → 0.140 |
| | $\delta = 0.5$ | | 0.260 → 0.190 | 0.260 → 0.190 | 0.270 → 0.200 |
| | $\delta = 0.8$ | | 0.330 → 0.260 | 0.340 → 0.260 | 0.340 → 0.270 |
| Hepar2 | $\delta = 0.0$ | 0.051 → 5.0e-04 | 0.053 → 3.0e-03 | 0.061 → 0.012 | 0.072 → 0.025 |
| | $\delta = 0.2$ | | 0.088 → 0.062 | 0.096 → 0.070 | 0.110 → 0.082 |
| | $\delta = 0.5$ | | 0.160 → 0.130 | 0.170 → 0.140 | 0.170 → 0.150 |
| | $\delta = 0.8$ | | 0.220 → 0.210 | 0.230 → 0.210 | 0.240 → 0.220 |
| Water | $\delta = 0.0$ | 0.160 → 0.024 | 0.160 → 0.024 | 0.160 → 0.025 | 0.160 → 0.025 |
| | $\delta = 0.2$ | | 0.190 → 0.080 | 0.190 → 0.081 | 0.190 → 0.100 |
| | $\delta = 0.5$ | | 0.250 → 0.200 | 0.250 → 0.200 | 0.250 → 0.200 |
| | $\delta = 0.8$ | | 0.320 → 0.310 | 0.330 → 0.310 | 0.330 → 0.320 |

**II) How good is the learning process on unseen examples?** To assess the generalization capability of the proposed method, we also learn distributions in a train-test (80%-20%) setting. This experiment is only done with the Bayesian Networks as there are multiple queries on the same nodes, which allows us to split them into a training and a test set while assuring that we do not perform testing on distributions that were never seen during the training. Table 2 shows the loss *on the test set* before and after having learned the model's distributions *on the train set*. Overall, the $\varepsilon$ parameter has little impact on the generalization of the learning process, unlike the $\delta$ parameter, which brings the approximation closer to a random guess and hinders the learning process.

**III) How does the number of queries used for training impact the learning?** Finally, we evaluate if the number (and difficulty) of the partially compiled queries used can impact the learning. We analyzed, for the Bayesian

network `Munin`, which is the largest in our experiment, the generalization on the test set when the distributions are learned on a fraction of the training set. We learned the distribution for $\varepsilon = 0.0, 0.1, 0.5, 0.8$ on, respectively, 40%, 60%, 80%, and 100% of the train data set, using the instances producing the smallest ACs first. For $\varepsilon = 0.0$, considering only the easiest instances does not give any generalization at all, and *the test loss* increases from 0.15 before the training to 0.17 after the model has been trained. Indeed, only a small part of the distributions are in the easiest instances, and it is impossible to generalize well. Using $\varepsilon = 0.1$ and 60% of the train set, the average loss decreases from 0.16 to 0.15; however, the gain is small. On the other hand, with $\varepsilon = 0.5$ and $\varepsilon = 0.8$, the average loss decreases respectively from 0.16 to 0.1 and from 0.17 to 0.11. Interestingly, the learning process seems to generalize similarly for both values. It means that at a certain point, adding more queries does not benefit the learning; the cost of the approximation is too high. Hence, depending on the data set, there must be an optimal value for $\varepsilon$, allowing the compilation of more queries, that cannot be increased and produce better generalization.

## 5 Conclusion

This work presented an approach for learning probability distribution parameters with partial compilation to learn the model's parameter when the complete compilation is impossible. This work opens many paths for further research. The approximate inference technique presented in this paper is still a simple extension of a DPLL-style search; enhancing this method could allow it to handle more difficult queries and improve the learning process with more accurate approximation. The gradient bounds using $\delta$ should also be investigated. Additionally, further work must be done to learn all the parameters with partial compilation.

**Acknowledgments** This work was supported by Service Public de Wallonie Recherche under grant n°2010235 – ARIAC by DIGITALWALLONIA4.AI.

## References

1. Chakraborty, S., Fremont, D., Meel, K., Seshia, S., Vardi, M.: Distribution-aware sampling and weighted model counting for SAT. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 28 (2014)
2. Chakraborty, S., Meel, K.S., Vardi, M.Y.: Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic SAT calls. Tech. rep. (2016)
3. Chavira, M., Darwiche, A.: On probabilistic inference by weighted model counting. Artificial Intelligence **172**(6-7) (2008)
4. Darwiche, A.: A differential approach to inference in bayesian networks. Journal of the ACM (JACM) **50**(3), 280–305 (2003)
5. Darwiche, A.: SDD: A new canonical representation of propositional knowledge bases. In: Twenty-Second International Joint Conference on Artificial Intelligence (2011)

6. De Raedt, L., Kimmig, A., Toivonen, H.: Problog: A probabilistic prolog and its application in link discovery. In: IJCAI 2007, Proceedings of the 20th international joint conference on artificial intelligence. pp. 2462–2467. IJCAI-INT JOINT CONF ARTIF INTELL (2007)
7. Dubray, A., Schaus, P., Nijssen, S.: Probabilistic Inference by Projected Weighted Model Counting on Horn Clauses. In: 29th International Conference on Principles and Practice of Constraint Programming (CP 2023) (2023)
8. Dubray, A., Schaus, P., Nijssen, S.: Anytime Weighted Model Counting with Approximation Guarantees for Probabilistic Inference. In: 30th International Conference on Principles and Practice of Constraint Programming (CP 2024) (2024)
9. Fierens, D., Van den Broeck, G., Renkens, J., Shterionov, D., Gutmann, B., Thon, I., Janssens, G., De Raedt, L.: Inference and learning in probabilistic logic programs using weighted Boolean formulas. Theory and Practice of Logic Programming **15**(3) (2015)
10. Gomes, C.P., Hoffmann, J., Sabharwal, A., Selman, B.: From Sampling to Model Counting. In: IJCAI. vol. 2007 (2007)
11. Huang, J., Darwiche, A.: The language of search. Journal of Artificial Intelligence Research **29** (2007)
12. Kisa, D., Van den Broeck, G., Choi, A., Darwiche, A.: Probabilistic sentential decision diagrams. In: Fourteenth International Conference on the Principles of Knowledge Representation and Reasoning (2014)
13. Klise, K.A., Bynum, M., Moriarty, D., Murray, R.: A software framework for assessing the resilience of drinking water systems to disasters with an example earthquake case study. Environmental modelling & software **95** (2017)
14. Lai, Y., Meel, K.S., Yap, R.H.: Fast Converging Anytime Model Counting. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 37 (2023)
15. Latour, A.L., Babaki, B., Fokkinga, D., Anastacio, M., Hoos, H.H., Nijssen, S.: Exact stochastic constraint optimisation with applications in network analysis. Artificial Intelligence **304**, 103650 (2022)
16. Manhaeve, R., Dumancic, S., Kimmig, A., Demeester, T., De Raedt, L.: Deepproblog: Neural probabilistic logic programming. advances in neural information processing systems **31** (2018)
17. Manhaeve, R., Marra, G., De Raedt, L.: Approximate inference for neural probabilistic logic programming. In: Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning. pp. 475–486. IJCAI Organization (2021)
18. Medjroubi, W., Müller, U.P., Scharf, M., Matke, C., Kleinhans, D.: Open data in power grid modelling: New approaches towards transparent grid models. Energy Reports **3** (2017)
19. Peharz, R., Lang, S., Vergari, A., Stelzner, K., Molina, A., Trapp, M., Van den Broeck, G., Kersting, K., Ghahramani, Z.: Einsum networks: Fast and scalable learning of tractable probabilistic circuits. In: International Conference on Machine Learning. PMLR (2020)
20. Scutari, M.: Learning bayesian networks with the bnlearn r package. arXiv preprint arXiv:0908.3817 (2009)
21. Shafer, G.R., Shenoy, P.P.: Probability propagation. Annals of mathematics and Artificial Intelligence **2**, 327–351 (1990)
22. Soos, M., Gocht, S., Meel, K.S.: Tinted, Detached, and Lazy CNF-XOR solving and its Applications to Counting and Sampling

23. Soos, M., Meel, K.S.: BIRD: Engineering an efficient CNF-XOR SAT solver and its applications to approximate model counting. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 33 (2019)
24. Wiegmans, B.: Gridkit: European And North-American Extracts (Mar 2016). https://doi.org/10.5281/ZENODO.47317

## A Proof of the Theorems

In this section, we prove Theorem 1, and we assume that the AC is produced by `Schlandals` [7]. The main difference is that the branching is done on multi-valued variables; sum nodes have more than two children. The results and proof can easily be adapted to classical algorithms based on binary variables.

Before proving Theorem 1, we need the following results that give a procedural way of ensuring an $\varepsilon$-bound approximation at node $(n)$ of an AC. Let us first introduce the notation $F^{(n)}$ that represents the formula of the sub-problem at node $n$ of an AC.

**Theorem 2.** *Let $(n)$ be a node in an AC and $(c_1), \dots, (c_k)$ be its children. If it holds that $\forall i = 1, \dots k$,*

$$\begin{cases} \frac{P[F^{(c_i)}]}{1+\varepsilon} \leq \tilde{P}[F^{(c_i)}] \leq P[F^{(c_i)}](1+\varepsilon) & \text{if } n \text{ is a } \texttt{sum} \text{ node} \\ \frac{P[F^{(c_i)}]}{\sqrt[k]{1+\varepsilon}} \leq \tilde{P}[F^{(c_i)}] \leq P[F^{(c_i)}]\sqrt[k]{1+\varepsilon} & \text{if } n \text{ is a } \texttt{product} \text{ node} \end{cases}$$

*then it holds that $\frac{P[F^{(n)}]}{1+\varepsilon} \leq \tilde{P}[F^{(n)}] \leq P[F^{(n)}](1+\varepsilon)$.*

*Proof.* Let us consider the two cases separately
**Case** $(n)$ is a `sum` node
Let $v_1, \dots, v_k$ be the values of the parameter being branched on (line 5 of Algorithm 1) and $P[v_i]$ their respective probability with $P[v_i] \in [0, 1]$ $\forall i = 1, \dots, k$. Then we can compute $P[F^{(n)}]$ as

$$P[F^{(n)}] = \sum_{i=1}^{k} \Big( P(v_i) \cdot P[F^{(c_i)}] \Big)$$

and, in the same way, we have

$$\tilde{P}[F^{(n)}] = \sum_{i=1}^{k} \Big( P(v_i) \cdot \tilde{P}[F^{(c_i)}] \Big)$$

We can now derive the upper bound on $\tilde{P}[F^{(n)}]$

$$\tilde{P}[F^{(n)}] = \sum_{i=1}^{k} \Big( P(v_i) \cdot \tilde{P}[F^{(c_i)}] \Big)$$

$$\tilde{P}[F^{(n)}] \leq \sum_{i=1}^{k} \Big( P(v_i) \cdot P[F^{(c_i)}] \cdot (1+\varepsilon) \Big) = (1+\varepsilon) \sum_{i=1}^{k} \Big( P(v_i) \cdot P[F^{(c_i)}] \Big) = (1+\varepsilon) \cdot P[F^{(n)}]$$

A lower bound on $\tilde{P}[F^{(n)}]$ is given by

$$\tilde{P}[F^{(n)}] = \sum_{i=1}^{k} \Big( P(v_i) \cdot \tilde{P}[F^{(c_i)}] \Big) \geq \sum_{i=1}^{k} \left( P(v_i) \cdot \frac{P[F^{(c_i)}]}{1+\varepsilon} \right) = \frac{1}{1+\varepsilon} \sum_{i=1}^{k} \Big( P(v_i) \cdot P[F^{(c_i)}] \Big) = \frac{1}{1+\varepsilon} P[F^{(n)}]$$

**Case** $(n)$ is an `product` node
We can compute $P[F^{(n)}]$ as

$$P[F^{(n)}] = \prod_{i=1}^{k} P[F^{(c_i)}]$$

and, in the same way, we have

$$\tilde{P}[F^{(n)}] = \prod_{i=1}^{k} \tilde{P}[F^{(c_i)}]$$

An upper bound for $\tilde{P}[F^{(n)}]$ is

$$\tilde{P}[F^{(n)}] = \prod_{i=1}^{k} \tilde{P}[F^{(c_i)}] \leq \prod_{i=1}^{k} \left( P[F^{(c_i)}] \cdot \sqrt[k]{1+\varepsilon} \right) = (1+\varepsilon) \prod_{i=1}^{k} P[F^{(c_i)}] = (1+\varepsilon) \cdot P[F^{(n)}]$$

A lower bound is given by

$$\tilde{P}[F^{(n)}] = \prod_{i=1}^{k} \tilde{P}[F^{(c_i)}] \geq \prod_{i=1}^{k} \frac{P[F^{(c_i)}]}{\sqrt[k]{1+\varepsilon}} = \frac{1}{1+\varepsilon} \prod_{i=1}^{k} P[F^{(c_i)}] = \frac{1}{1+\varepsilon} \cdot P[F^{(n)}]$$

### A.1 Theorem 1

Theorem 1 states that the bound on the error of the approximated gradient is

$$\left| \frac{\partial \tilde{P}[F]}{\partial w} \right| \leq \left| \frac{\partial P[F]}{\partial w} \right| + \varepsilon \left( \frac{3}{4} + \frac{1}{2} \varepsilon \right)$$

when the approximation method used gives $\varepsilon$ guarantees.

*Proof.* In the partial arithmetic circuit, the output probability of a node $(n)$ with children $(c_1), ..., (c_k)$ is

$$\tilde{P}[F^{(n)}] = \begin{cases} \sum_{i=1}^{k} \tilde{P}[F^{(c_i)}] & \text{if (n) is an } \texttt{sum} \text{ node} \\ \prod_{i=1}^{k} \tilde{P}[F^{(c_i)}] & \text{if (n) is an } \texttt{product} \text{ node} \end{cases}$$

For any parameter $w$ from distribution $D$ present in the partially compiled circuit, we have to compute

$$\frac{\partial \tilde{P}[F]}{\partial w}$$

Without loss of generality, let us assume that the root $(r)$ is an `sum` node, then

$$\frac{\partial \tilde{P}[F^{(r)}]}{\partial w} = \frac{\partial \sum_{i=1}^{k} \tilde{P}[F^{(c_i)}]}{\partial w} = \sum_{i=1}^{k} \frac{\partial \tilde{P}[F^{(c_i)}]}{\partial w}$$

Let $(c_1), ..., (c_{k'})$ be the children of $(r)$ for which the subcircuit rooted at $c_i, 1 \leq i \leq k'$ contains an input node sending a value of $D$. Then

$$\sum_{i=1}^{k} \frac{\partial \tilde{P}[F^{(c_i)}]}{\partial w} = \sum_{i=1}^{k'} \frac{\partial \tilde{P}[F^{(c_i)}]}{\partial w} \tag{1}$$

As the circuit alternates between `product` and `sum` nodes, the children $c_i$ are `product` nodes and the derivatives become

$$\sum_{i=1}^{k'} \frac{\partial \tilde{P}[F^{(c_i)}]}{\partial w} = \sum_{i=1}^{k'} \frac{\partial \prod_{j=1}^{k_i} \tilde{P}[F^{(c_j^i)}]}{\partial w} = \sum_{i=1}^{k'} \sum_{j=1}^{k_i} \left( \frac{\partial \tilde{P}[F^{(c_j^i)}]}{\partial w} \prod_{l=1, l \neq j}^{k_i} \tilde{P}[F^{(c_l^i)}] \right) \tag{2}$$

with $(c_j^i), 1 \leq j \leq k_i$ the children of $(c_i)$.

In the case where the subcircuit rooted at $c_j^i$ does not contain distribution $D$, its derivative will be zero and it does not contribute to the sum. Otherwise, the gradient must be computed. If it is an `sum` node, the formula is recursively unfolded using Equation (1). If it is a parameter from the domain of $D$, the derivative of the node is the derivative of the softmax function with respect to $w$. By abuse of notation, we denote $P(c_j^i)$ as the probability of the parameter in node $(c_j^i)$. The derivative of the softmax is given by

$$\frac{\partial \tilde{P}[F^{(c_j^i)}]}{\partial w} = \begin{cases} P(w) \cdot (1 - P(w)) & \text{if } c_j^i = w \\ -P(w) \cdot P(c_j^i) & \text{if } c_j^i \neq w \end{cases}$$

Given that, for any node that does not contain $D$ in its subcircuit the derivative is 0, the total derivative for $w$ is only influenced by nodes present on a path between the root and the leaves related to $D$.

For any parameter $v \in dom(D)$, let $C_v^i =< n_1, ..., n_m >$ be the sequence of nodes on the i-th path from $n_1$ (being the root $(r)$) to node $n_m$, the `sum` node that branches on $(v)$, and $\mathcal{C}_D$ the set of all paths going from $(r)$ to any of the parameters of $D$. Let $\tilde{pv} : \mathcal{C}_D \mapsto [0; 1]$ be a function that maps a path to the multiplicative factor of the path, evaluated as

$$\tilde{pv}(C_v^i) = \prod_{i=1}^{m} \begin{cases} 1 & \text{if } n_i \text{ is an } \texttt{sum} \text{ node} \\ \prod_{\substack{c \in children(n_i) \\ c \neq n_{i+1}}} \tilde{P}[F^{(c)}] & \text{if } n_i \text{ is an } \texttt{product} \text{ node} \end{cases}$$

Moreover, $pv(C_v^i)$ is the analogous function in the fully compiled circuit, with no approximation.

The derivative of the AC circuit with respect to $w$ can thus be expressed as

$$\frac{\partial \tilde{P}[F]}{\partial w} = \sum_{C \in \mathcal{C}_D} \left( \tilde{pv}(C) P(w) \sum_{\substack{u \in dom(D) \\ u \neq w}} P(u) (\tilde{P}[F|_{C,w}] - \tilde{P}[F|_{C,u}]) \right) \tag{3}$$

Let $P[F|_{C,w}]$ be the value of the subcircuit after following path $C$ and branching on the parameter $w$ of distribution $D$.

We will show that this expression of the partial derivative can be bounded as

$$|\frac{\partial \tilde{P}[F]}{\partial w}| \leq |\frac{\partial P[F]}{\partial w}| + \varepsilon(\frac{3}{4} + \frac{1}{2}\varepsilon)$$

Let us consider the case where $\forall u \in dom(D) : P[F|_{C,w}] \geq P[F|_{C,u}]$:

Let us remember that

$$\frac{P[F|_{C,w}]}{1+\varepsilon} \leq \tilde{P}[F|_{C,w}] \leq P[F|_{C,w}](1+\varepsilon) \tag{4}$$

We know that $\tilde{pv}(C)$ is a product of the children of the `product` nodes in path $C$. Additionally, Theorem 2 tells us that at each `product` node, the error bound is reduced to $(1+\varepsilon)^{1/k}$ with k being the number of independent subproblems solved at that node during compilation. Therefore, if our currently considered path is composed of $m$ `product` nodes $n_i$ each solving $k_i$ independent subproblems, we have that the approximation upper bound for $\tilde{pv}(C)$ is of the form

$$\begin{aligned}
\tilde{pv}(C_v^i) &\leq \prod_{i=1}^{m} \prod_{\substack{c \in children(n_i) \\ c \neq n_{i+1}}} (P[F^{(c)}] \cdot (1+\varepsilon)^{\prod_{j=1}^{i} 1/k_j}) \\
&= pv(C_v^i) \cdot \prod_{i=1}^{m} \prod_{\substack{c \in children(n_i) \\ c \neq n_{i+1}}} (1+\varepsilon)^{\prod_{j=1}^{i} 1/k_j} \\
&= pv(C_v^i) \cdot \prod_{i=1}^{m} (1+\varepsilon)^{(k_i - 1)/\prod_{j=1}^{i} k_j} \\
&= pv(C_v^i) \cdot \prod_{i=1}^{m} (1+\varepsilon)^{(k_i/(\prod_{j=1}^{i} k_j) - 1/(\prod_{j=1}^{i} k_j)} \\
&\leq pv(C_v^i) \cdot \prod_{i=1}^{m} (1+\varepsilon)^{k_i/\prod_{j=1}^{i} k_j} \\
&= pv(C_v^i) \cdot \prod_{i=1}^{m} (1+\varepsilon)^{1/\prod_{j=1}^{i-1} k_j} \\
&= pv(C_v^i) \cdot (1+\varepsilon)^{\sum_{i=1}^{m} (1/\prod_{j=1}^{i-1} k_j)}
\end{aligned}$$

We first simplify to a situation in which all nodes in the path solve 2 independent subproblems each, we obtain the following bounding factor $(1+\varepsilon)^{\sum_{i=1}^{m} 1/2^i}$. It is known that the sum $\sum_{i=1}^{m} 1/2^i$ converges to 1. If we now extend this to the case where the nodes solve more than two subproblems, the denominator of the summed terms will be bigger and therefore the total sum will never be greater than 1. Hence, we can use $(1+\varepsilon)$ as the bounding factor for the path value term $\tilde{pv}(C)$.

Let us define $AC$ as $\frac{\partial P[F]}{\partial w}$, with $\tilde{AC}$ its equivalent in the approximate circuit. We can then express an upper bound for $|\tilde{AC}|$. Using Equation (3) and the Inequalities (4) we have

$$
\begin{aligned}
|\tilde{AC}| &= \left| \sum_{C \in \mathcal{C}_D} \left( \tilde{pv}(C) P(w) \sum_{\substack{u \in dom(D) \\ u \neq w}} P(u) \left( \tilde{P}[F|_{C,w}] - \tilde{P}[F|_{C,u}] \right) \right) \right| \\
&\leq \left| \sum_{C \in \mathcal{C}_D} \left( pv(C)(1+\varepsilon) P(w) \sum_{\substack{u \in dom(D) \\ u \neq w}} P(u) \left( P[F|_{C,w}](1+\varepsilon) - \frac{P[F|_{C,u}]}{1+\varepsilon} \right) \right) \right|
\end{aligned}
$$

By distributing $(1+\varepsilon)$ in the second sum and by adding $-P[F|_{C,u}] + P[F|_{C,u}]$ (hence not changing the total value) we have

$$
\begin{aligned}
|\tilde{AC}| &\leq \left| \sum_{C \in \mathcal{C}_D} \left( pv(C)(1+\varepsilon) P(w) \sum_{\substack{u \in dom(D) \\ u \neq w}} P(u) \left( P[F|_{C,w}] + \varepsilon P[F|_{C,w}] - P[F|_{C,u}] + \frac{(1+\varepsilon)-1}{1+\varepsilon} P[F|_{C,u}] \right) \right) \right| \\
&= \left| \sum_{C \in \mathcal{C}_D} \left( pv(C)(1+\varepsilon) P(w) \sum_{\substack{u \in dom(D) \\ u \neq w}} P(u) \left( P[F|_{C,w}] - P[F|_{C,u}] + \varepsilon (P[F|_{C,w}] + \frac{P[F|_{C,u}]}{1+\varepsilon}) \right) \right) \right|
\end{aligned}
$$

Each probability is always bounded by 1, in particular, $P[F|_{C,w}]$ and $P[F|_{C,u}]$ can be replaced by 1 in the upper bound

$$
\begin{aligned}
|\tilde{AC}| &\leq \left| \sum_{C \in \mathcal{C}_D} \left( pv(C)(1+\varepsilon) P(w) \sum_{\substack{u \in dom(D) \\ u \neq w}} P(u) \left( P[F|_{C,w}] - P[F|_{C,u}] + \varepsilon (1 + \frac{1}{1+\varepsilon}) \right) \right) \right| \\
&= \left| \sum_{C \in \mathcal{C}_D} \left( pv(C)(1+\varepsilon) P(w) \left( \sum_{\substack{u \in dom(D) \\ u \neq w}} P(u)(P[F|_{C,w}] - P[F|_{C,u}]) + \sum_{\substack{u \in dom(D) \\ u \neq w}} P(u) \varepsilon (1 + \frac{1}{1+\varepsilon}) \right) \right) \right| \\
&= \left| \sum_{C \in \mathcal{C}_D} pv(C)(1+\varepsilon) P(w) \left( \sum_{\substack{u \in dom(D) \\ u \neq w}} P(u)(P[F|_{C,w}] - P[F|_{C,u}]) \right) \right| \\
&\qquad + \left| \sum_{C \in \mathcal{C}_D} pv(C)(1+\varepsilon) P(w) \left( \sum_{\substack{u \in dom(D) \\ u \neq w}} P(u) \varepsilon (\frac{2+\varepsilon}{1+\varepsilon}) \right) \right|
\end{aligned}
$$

We observe that the first sum can be rewritten as

$$\sum_{C\in\mathcal{C}_D} pv(C)(1+\varepsilon)P(w)\left(\sum_{\substack{u\in dom(D)\\ u\neq w}} P(u)(P[F|_{C,w}]-P[F|_{C,u}])\right)$$

$$=\sum_{C\in\mathcal{C}_D} pv(C)P(w)\left(\sum_{\substack{u\in dom(D)\\ u\neq w}} P(u)(P[F|_{C,w}]-P[F|_{C,u}])\right)$$

$$+\sum_{C\in\mathcal{C}_D} pv(C)\varepsilon P(w)\left(\sum_{\substack{u\in dom(D)\\ u\neq w}} P(u)(P[F|_{C,w}]-P[F|_{C,u}])\right)$$

Remark that the first term corresponds to Equation (3) in the non-partial case. Therefore, $\tilde{AC}$ is bounded by

$$|\tilde{AC}|\leq\left|\left(AC+\sum_{C\in\mathcal{C}_D} pv(C)\varepsilon P(w)\sum_{\substack{u\in dom(D)\\ u\neq w}} P(u)(P[F|_{C,w}]-P[F|_{C,u}])\right)\right|$$

$$+\left|\left(\sum_{C\in\mathcal{C}_D} pv(C)(1+\varepsilon)P(w)\sum_{\substack{u\in dom(D)\\ u\neq w}} P(u)\varepsilon(\frac{2+\varepsilon}{1+\varepsilon})\right)\right|$$

As we are in the case where $\forall u\in dom(D): P[F|_{C,w}]\geq P[F|_{C,u}]$, the biggest difference that we can have between $P[F|_{C,w}]$ and $P[F|_{C,u}]$ is 1

$$|\tilde{AC}|\leq\left|\left(AC+\sum_{C\in\mathcal{C}_D} pv(C)\varepsilon P(w)\sum_{\substack{u\in dom(D)\\ u\neq w}} P(u)+\sum_{C\in\mathcal{C}_D} pv(C)(1+\varepsilon)P(w)\sum_{\substack{u\in dom(D)\\ u\neq w}} P(u)\varepsilon(\frac{2+\varepsilon}{1+\varepsilon})\right)\right|$$

Since $\sum_{v\in dom(D)} P(v)=1$, we have that $\sum_{\substack{u\in dom(D)\\ u\neq w}} P(u)=1-P(w)$. It follows that

$$|\tilde{AC}|\leq\left|\left(AC+\sum_{C\in\mathcal{C}_D} pv(C)\varepsilon P(w)(1-P(w))+pv(C)(1+\varepsilon)\varepsilon(1-P(w))\frac{2+\varepsilon}{1+\varepsilon}\right)\right|$$

The highest value that $P(w)(1-P(w))$ can reach is when $P(w)$ is 0.5, which gives a value of 0.25 to the product. Hence,

$$|\tilde{AC}|\leq\left|\left(AC+\sum_{C\in\mathcal{C}_D} pv(C)\frac{1}{4}\varepsilon+pv(C)(1+\varepsilon)\varepsilon\frac{1}{4}\frac{2+\varepsilon}{1+\varepsilon}\right)\right|$$

Given that $0 \leq \varepsilon$, the highest value that $\frac{2+\varepsilon}{1+\varepsilon}$ can take is when $\varepsilon = 0$, which gives an upper bound of

$$\begin{aligned}
|\tilde{AC}| &\leq \left| \left( AC + \sum_{C \in \mathcal{C}_D} pv(C) \frac{1}{4} \varepsilon + pv(C)(1+\varepsilon)\varepsilon \frac{1}{4} 2 \right) \right| \\
&= \left| \left( AC + \sum_{C \in \mathcal{C}_D} pv(C)(\frac{\varepsilon}{4} + \frac{\varepsilon}{2} + \frac{\varepsilon^2}{2}) \right) \right| \\
&= |AC| + \sum_{C \in \mathcal{C}_D} pv(C)\varepsilon(\frac{3}{4} + \frac{\varepsilon}{2})
\end{aligned}$$

Each path $C$ from the root to a leaf in the circuit can be seen as a partial interpretation of the formula, with a probability $p_C$ equal to the product of the decisions on the path. As $p_C$ is a subproduct of $pv(C)$ and the additional terms in $pv(C)$ are less than 1, then $pv(C) \leq p_C$. In addition to that, the sum of all interpretations' probability of $F$ is no greater than 1. Hence,

$$\sum_{C \in \mathcal{C}_D} pv(C) \leq \sum_{C \in \mathcal{C}_D} p_C \leq 1$$

Which finally gives

$$\left| \frac{\partial \tilde{P}[F]}{\partial w} \right| \leq \left| \frac{\partial P[F]}{\partial w} \right| + \varepsilon(\frac{3}{4} + \frac{1}{2}\varepsilon)$$

The same formula can be derived from the cases in which $\forall u \in dom(D) : p(w) \geq p(u)$ does not hold following the same reasoning.

## B Approximations Simulation Method

The construction of partially compiled ACs with a guaranteed $\varepsilon$-bound *on the root node's value* necessitates consideration to constrain each of the approximated node in the circuit with the adapted $\varepsilon$-bound value. Indeed, Theorem 2 establishes how the $\varepsilon$-bound should be adjusted when encountering sum and product nodes. This means that, to ensure an $\varepsilon$-bound at the root, the effective $\varepsilon$ value assigned to the newly created approximated nodes must be adapted. The $\varepsilon$ bound of the root reduces as $\sqrt[k]{\varepsilon}$ at each encountered product node $(n)$ in the path between the root and the created approximated node, with $k$ the branching factor of that node. This scaling accounts for the potential amplification of error introduced by the product operation as values propagate through the circuit. The experiments we present take into account the adapted $\varepsilon$ value required at each approximation node to ensure the correct $\varepsilon$-bound at the root.