Where to Approximate in Neurosymbolic Inference?

Samy Badreddine [1 2 3] **Emile van Krieken** [4] **Luciano Serafini** [2 3] **Antonio Vergari** [5]

Abstract

Probabilistic neurosymbolic methods rely on weighted model counting (WMC) to combine neural predictors with symbolic constraints. Exact WMC is #P-hard and typically scales poorly, so many methods resort to approximations. By unifying existing approaches under a three-step bottom-up *approximate compilation* pipeline, we study *where* approximation budget is best allocated. Inspired by tensor networks, we instantiate two steps with tensor train decompositions and derive new pipelines that exactly multiply approximated factors and recompress their products via SVD- or interpolation-based schemes. On Sudokus of increasing size, these pipelines produce WMC estimates many orders-of-magnitude more accurate than prior methods. Yet, when it comes to neurosymbolic learning, even crude approximations reach competitive accuracy, suggesting that approximation quality matters far more for inference fidelity than for downstream learning. Our pipelines open significant design space for effective approximate compilation of challenging probabilistic reasoning tasks.

1. Introduction

Probabilistic neurosymbolic (NeSy) systems combine neural perception with symbolic constraints, to enable learning from weak supervision (Manhaeve et al., 2018; Xu et al., 2018) or to inject symbolic knowledge into structured prediction pipelines (Ahmed et al., 2022). Inference involves solving the *weighted model count* (WMC) problem (Chavira & Darwiche, 2008): computing the probability that a neural distribution over discrete concepts satisfies a constraint. Computing the WMC exactly is #P-hard in general (Valiant, 1979). One approach for solving practical instances tractably is *circuit compilation* which compiles the constraint into a logical circuit, a process also called *knowledge compilation* (Darwiche & Marquis, 2002), while the neural distribution is turned into a probabilistic circuit (Vergari et al., 2019; Choi et al., 2020; Ahmed et al., 2022). Then, under certain structural properties, these circuits can be efficiently multiplied and summed over to yield the exact WMC (Shen et al., 2016; Vergari et al., 2021). The same structural properties and multiplication operation are also behind popular bottom-up strategies for knowledge compilation (Pipatsrisawat & Darwiche, 2008; Darwiche, 2011).

However, compilation is itself hard, and exactly multiplying several compiled circuits blows up exponentially in the number of factors (Vergari et al., 2021). This implies that compilation strategies can hit a wall on many real-world NeSy problems. Even compiling a 9×9 Sudoku board, often used in benchmarks (Augustine et al., 2022; Wang et al., 2019), as a constraint is out of reach for all exact compilers. Without surprise, many NeSy methods focused on approximate computation of the WMC (van Krieken et al., 2022b; Smet et al., 2023b; Choi et al., 2025; Dierckx et al., 2025). However, these approaches have carried over with different perspectives and from different communities. As such, the problem of understanding *where* and *when* to approximate in a systematic way is still open.

In this paper, we provide a unified view over many NeSy pipelines and we identify three stages at which approximations can enter (Figure 1): **(I)** compiling each factor exactly or with a relaxed circuit, **(II)** multiplying factors exactly or via a recompression that controls size growth, and **(III)** combining remaining factors under an independence assumption. Many existing methods fit in this categorization, sometimes mixing several approximations. *Independence* (fuzzy) approximations are by far the most common (Badreddine et al., 2020; van Krieken et al., 2022a; Ahmed et al., 2023) and are well known to support effective learning (van Krieken et al., 2022a; Maene et al., 2024), yet their inference fidelity is not well-studied. Stages (I) and (II) are far less explored. Drawing on the close connection between tensor networks and circuits (Loconte et al., 2024b), we instantiate stages (I) and (II) using tensor train (TT) decompositions. TTs have been used in NeSy to approximate and compress isolated factors (Choi et al., 2025) but not in a full compilation

[1]Sony AI [2]Fondazione Bruno Kessler, Trento, Italy [3]University of Trento, Trento, Italy [4]Vrije Universiteit Amsterdam, Amsterdam, Netherlands [5]University of Edinburgh, Edinburgh, UK. Correspondence to: Samy Badreddine <badreddine.samy@gmail.com>.

43rd International Conference on Machine Learning (ICML 2026). Workshop: The 2nd Workshop on Connecting Low-rank Representations in AI.

pipeline. We show how **a**pproximate **t**ensor **t**rains can be used for **i**nference and **c**ompilation, deriving novel **ATTIC** pipelines.

Contributions. **(i)** We unify existing bottom-up compilation methods under a three-stage approximate-compilation pipeline and analyze how error and cost compose across strategies. **(ii)** Within this framework, we derive new approximation strategies based on TN recompressions, previously unexplored in NeSy or compilation. **(iii)** We empirically map out the design space on several challenging Visual Sudoku settings and show that several new ATTIC configurations produce WMC estimates orders of magnitude more accurate than prior methods. At the same time, when it comes to *learning*, we note that most approximations are competitive, consistent with prior findings that even simple independence approximations work surprisingly well as gradient estimators (Maene et al., 2024).

2. Probabilistic NeSy Models

A probabilistic NeSy model combines (i) a parameterized (neural) model $p_{\boldsymbol{\theta}}(\mathbf{c}|\mathbf{x})$ which maps the input $\mathbf{x}$ to a vector of discrete, symbolic *concepts* $\mathbf{c} = [c_i]_{i=1,\ldots,d}$, $c_i \in \mathcal{C}_i$, with (ii) a symbolic *constraint* $\phi : \mathcal{C}_1 \times \cdots \times \mathcal{C}_d \to \{0, 1\}$ that assesses the validity of the predicted concepts. In Sudoku, for example, $\mathbf{x}$ is the image of the board, $\mathbf{c}$ are the predicted digits in each cell, and ϕ checks that the predicted digits satisfy the Sudoku rules. This involves computing $p_{\boldsymbol{\theta}}(\phi|\mathbf{x})$ which maps the input $\mathbf{x}$ to the probability of satisfying the constraint ϕ. This is known as a conditional *weighted model count* (WMC) problem, defined as [1]

$$p_{\boldsymbol{\theta}}(\phi|\mathbf{x}) := \sum\nolimits_{\mathbf{c}\in\mathcal{C}} p_{\boldsymbol{\theta}}(\mathbf{c}|\mathbf{x})\phi(\mathbf{c}). \tag{1}$$

Often, ϕ decomposes into a conjunction of subformulas $\phi_1(\mathbf{c}_1) \wedge \cdots \wedge \phi_N(\mathbf{c}_N)$, where each ϕ_j is called a *factor* with scope $\mathbf{c}_j \subseteq \mathbf{c}$. For example, the 9×9 Sudoku rules decompose into $N = 810$ factors that check pairwise constraints $c_i \neq c_j$ for cells i, j in the same row, column, or box. Computing the WMC is #P-hard in general (Valiant, 1979), and this is also due to the large combinatoric space of NeSy problems. E.g., in a 9×9 Sudoku there are 9^{81} possible configurations, out of which *only* 6.67×10^{21} are valid boards (Felgenhauer & Jarvis, 2006). To solve this issue, probabilistic NeSy methods must use compilation techniques and/or relaxation strategies to make the WMC tractable.

[1]The problem can also be stated with $\phi(\mathbf{c}, \mathbf{y})$ being a program that maps concepts $\mathbf{c}$ to a structured output $\mathbf{y}$. See the *Sudoku completion* example in Section 5.

2.1. Exact circuit compilation

A prominent approach to computing the WMC tractably is to represent both ϕ and $p_{\boldsymbol{\theta}}(\mathbf{c}|\mathbf{x})$ as *circuits* and multiply them (Kisa et al., 2014; Manhaeve et al., 2018; Xu et al., 2018; Ahmed et al., 2022). We define the language of circuits next.

Definition 2.1 (Circuit (Choi et al., 2020; Vergari et al., 2021))**.** A *circuit* $f(\mathbf{z})$ is a directed acyclic computational graph of *weighted sum* and *product* operations, represented as intermediate units, over tractable functions over variables $\mathbf{z}$, e.g., indicator functions. [2]

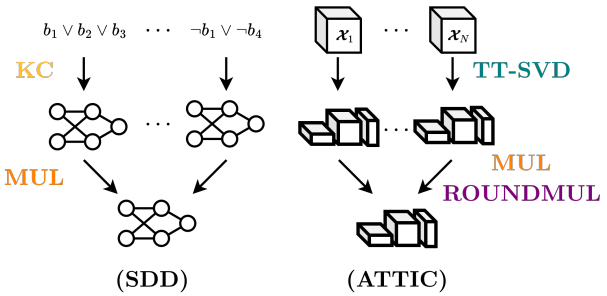
Probabilistic circuits (PCs) (Darwiche, 2003; Vergari et al., 2019; Choi et al., 2020) are circuits encoding a non-negative function and whose sum unit weights are generally assumed non-negative (Loconte et al., 2024a;b). *Logical circuits* are special PCs encoding a Boolean function (Darwiche & Marquis, 2002), and have sum unit weights equal to 1, while input units encode indicator functions corresponding to logical literals. Tractable computation of multiplications and summations over circuits, two key operations in the WMC (Equation (11)), require certain structural properties, which we briefly discuss next. We refer the reader to Choi et al. (2020) for details.

We focus on circuits that are *smooth*, i.e., all inputs of sum units are defined over the same variables, and *decomposable*, i.e., inputs of product units are defined over disjoint sets of variables. These circuits support tractable marginalization, e.g., summing over all possible concept assignments, in time linear in the circuit size (Choi et al., 2020). To enable tractable multiplication, we need circuits that are *compatible* (Vergari et al., 2021), i.e., all product units defined over the same variables decompose in the same way. A circuit compatible with itself is *structured-decomposable*. Compatible circuits can be multiplied in time polynomial in their sizes while preserving smoothness and decomposability. In other words, if $p_{\boldsymbol{\theta}}(\mathbf{c}|\mathbf{x})$ is a PC parameterized by a neural network, and we can represent ϕ as a logical circuit, and both are compatible, we can compute Equation (11) exactly and efficiently.

A first bottleneck comes from representing ϕ as a logical circuit with the right structural properties, a task that is known as **knowledge compilation** (KC) and is at least NP-hard (Darwiche & Marquis, 2002; Chavira & Darwiche, 2008). Popular knowledge compilers output logical circuits that are defined over logical OR/AND gates rather than

[2]Formulas over categorical variables can be handled by encoding each variable c_i with a set of boolean indicators $b_{i,j} = \mathbb{1}[c_i = j]$ and adding mutual-exclusivity constraints to ensure that exactly one of the $b_{i,j}$ is true for each i. This can significantly increase the size of a formula, see Sudoku 9x9 where ϕ goes from $d = 81$ variables and $N = 810$ factors to $d = 729$ variables and $N = 10287$ factors when propositionalized with mutex constraints (Section D).

Baseline	SI	SII	SIII
Exact methods (SL, DPL)	KC	MUL	–
Fuzzy methods (LTN, DFL)	–	–	INDEP
Sem. Strength.	KC	MUL	INDEP
CTSKETCH	TT-SVD	–	INDEP
ATTIC A	PREMUL, TT-SVD	MUL	INDEP
ATTIC B	PREMUL, TT-SVD	MUL, ROUNDMUL	INDEP



Exact operations: Approximate operations:

Figure 1. **Where can we approximate** when compiling structured-decomposable circuits? ATTIC unifies bottom-up approaches in a three-step pipeline: **(Step I)** compiles each factor ϕ_j into an **exact** or **relaxed** circuit, **(Step II)** multiplies the resulting N circuits down to K **exactly** or using **recompression strategies**, and **(Step III)** combines any $K > 1$ remaining circuits with an **independence** (fuzzy) approximation. The framework unifies existing methods and derives new ones (ATTIC), yet to be explored.

sum/product units. However, we can easily switch from the Boolean to the real semiring (Kimmig et al., 2017) by noting that these circuits are not only smooth and (structured-)decomposable but also *deterministic*, i.e., OR units are defined over inputs with disjoint supports, or in other words, encode logical XORs and can be safely turned to sums. Sentential decision diagrams (SDDs) (Darwiche, 2011) are a popular class of smooth, deterministic and structured-decomposable circuits for KC. SDDs can be multiplied with compatible PCs $p_{\boldsymbol{\theta}}(\mathbf{c}|\mathbf{x})$, and can also be compiled in a bottom-up fashion by **multiplying** the circuits of factors ϕ_j. Let $\mathrm{KC}(\phi_j)$ be the logic circuit for factor ϕ_j. We rewrite the WMC using the symbolic operator MUL, which takes as input a set of circuits and outputs a new circuit representing their multiplication.

$$\sum\nolimits_{\mathbf{c}\in\mathcal{C}} p_{\boldsymbol{\theta}}(\mathbf{c}|\mathbf{x})\mathrm{MUL}(\mathrm{KC}(\phi_1),\ldots,\mathrm{KC}(\phi_N))(\mathbf{c}). \quad (2)$$

This is the classical recipe behind many NeSy approaches including the semantic loss (SL) (Xu et al., 2018) and DeepProblog (DPL) (Manhaeve et al., 2018). However, to the best of our knowledge there is no KC pipeline that can output a circuit of polynomial size in reasonable time for a 9×9 Sudoku board, even relaxing structured-decomposability (Darwiche, 2001; Lagniez & Marquis, 2017). Therefore, a common practice is to consider *approximations*. We focus on approximations which (i) apply to bottom-up circuit compilation, (ii) are differentiable. In Section A, we discuss other approaches to WMC in NeSy, such as amortized or sampling-based methods, and how they relate to our work.

2.2. Approximations

Independence approximations. A common approximation when multiplying two circuits is to assume they are independent, even if they are defined over the same set of variables. If we assume that, the WMC of their conjunction factorizes into the product of their WMCs. More formally,

$$\begin{aligned}&\sum\nolimits_{\mathbf{c}\in\mathcal{C}} p_{\boldsymbol{\theta}}(\mathbf{c}|\mathbf{x})\mathrm{MUL}(\mathrm{KC}(\phi_1),\ldots,\mathrm{KC}(\phi_N))(\mathbf{c})\\ &\overset{\mathrm{I}}{\approx} \prod\nolimits_j \left(\sum\nolimits_{\mathbf{c}_j\in\mathcal{C}_j} p_{\boldsymbol{\theta}}(\mathbf{c}_j|\mathbf{x})\mathrm{KC}(\phi_j)(\mathbf{c}_j)\right).\end{aligned} \quad (3)$$

Often, $|\mathcal{C}_j| \ll |\mathcal{C}|$, which makes the factor WMCs much easier to compute. *Fuzzy* NeSy approaches with product semantics like LTN (Badreddine et al., 2020; Donadello et al., 2017) or DFL (van Krieken et al., 2022a) can be understood as probabilistic approaches relaxing *all* dependencies (van Krieken et al., 2019). [3] The approximation is only correct when the factors ϕ_j are independent, i.e., when the scopes $\mathbf{c}_j$ are pairwise disjoint. In general, this is not the case and the approximation can be arbitrarily bad, as we will show in our experiments. Still, in the context of differentiable learning, fuzzy relaxations are known to be effective in practice (van Krieken et al., 2022a).

Fuzzy relaxations can be used in combination with exact circuit compilation. Semantic Strengthening (Ahmed et al., 2023) compiles SDD circuits for factor groups under a certain budget, prioritizing groups of factors with high mutual information, and applies the independence approximation between groups. SDDs are particularly suitable for this approach as they can iteratively multiply and analyze factors, making informed compilation or stopping decisions within the budget.

Factor approximations. Instead of approximating the marginalization step, one can approximate the factors themselves. This requires approximate compilation (AC) step, i.e.,

$$\mathrm{MUL}(\mathrm{KC}(\phi_1),\ldots,\mathrm{KC}(\phi_N)) \overset{\mathrm{F}}{\approx} \mathrm{MUL}(\mathrm{AC}(\phi_1),\ldots,\mathrm{AC}(\phi_N)) \quad (4)$$

[3] In the rest of the paper, we sometimes refer to the independence approximation as "fuzzy", though it is important to note that not all fuzzy approaches use product semantics understood as an independence relaxation of probabilistic NeSy.

where AC compiles an *approximated circuit* in the real semiring for factor ϕ_j, i.e., a circuit that does not encode a Boolean function but closely approximates it up to an ϵ-error. Crucially, the circuit retains all the structural properties needed to tractably compute MUL. In Section 3, we discuss how each factor can be approximated with *tensor trains* (TTs) (Oseledets, 2011), a special family of tensorized circuits (Loconte et al., 2024b) for which there is a rich literature on function approximation (Orús, 2013). Such a TT-approximated factor $\mathrm{AC}(\phi_j)$ can be a representation that is significantly smaller than the exact circuit $\mathrm{KC}(\phi_j)$, given that it can introduce negative parameters (Loconte et al., 2024a). Additionally, TTs inherently handle categorical variables and do not need Boolean encoding.

CTSketch (Choi et al., 2025) is a recent approach that approximates factors ϕ_j with low-rank TTs. However, CTSketch only considers programs that follow a certain tree-like independence structure, where any variable c_i can appear in at most one factor. This applies to simple problems such as MNISTAdd (Manhaeve et al., 2018), but most programs, like Sudoku boards, have overlapping factors and violate this assumption. Therefore, CTSketch mixes factor approximations with fuzzy independence approximations $\prod_j(\sum_{\mathbf{c}_j} p_{\boldsymbol{\theta}}(\mathbf{c}_j|\mathbf{x})\mathrm{AC}(\phi_j)(\mathbf{c}_j))$. Section C details this. In Section 3, we show how to exactly multiply TT-approximated factors under more complex dependence structures.

Product approximations. Finally, we consider approximating the multiplication step directly, without going through intermediate approximations of the factors themselves, i.e.,

$$\mathrm{MUL}(\mathrm{KC}(\phi_1),\ldots,\mathrm{KC}(\phi_N)) \overset{\mathrm{P}}{\approx} \mathrm{ROUNDMUL}(\mathrm{KC}(\phi_1),\ldots,\mathrm{KC}(\phi_N)). \quad (5)$$

Drawing on TN theory, we show that it is possible to derive algebraically-principled approximations which compress the product of circuits into smaller circuits. We denote such operations ROUNDMUL for *rounded multiplications*, inspired by the TT-Round (Oseledets, 2011) algorithm discussed in Section 3.3. This direction is, to the best of our knowledge, unexplored in the NeSy or compilation literature, and can prove useful to approximate many operations on circuits.

3. Approximate Tensor Trains for Inference and Compilation (ATTIC)

TTs are a framework for efficiently representing and approximating high-dimensional tensors, and are a special case of not only circuits, but also tensor networks (TNs) (Orús, 2013). TNs and tensorized circuits allow for highly parallelized sum and product operations (Peharz et al., 2020; Mari et al., 2023; Loconte et al., 2024b). The tensor perspective applies to the WMC problem in Equation (1) by considering the full tensor $\boldsymbol{\mathcal{X}} \in \{0,1\}^{|\mathcal{C}_1|\times\cdots\times|\mathcal{C}_d|}$ where $\boldsymbol{\mathcal{X}}(\mathbf{c}) := \phi(\mathbf{c})$ encodes the value of the formula at index $\mathbf{c}$.

For example, a program $\phi(c_1, c_2)$ checking that two variables c_1 and c_2 are distinct is summarized in a two-dimensional tensor of all ones except on the diagonal, where entries are zero. However, the tensor for the full 9x9 Sudoku program has size 9^{81} and cannot be materialized, motivating a TN representation. As mentioned in the previous section, TTs have favorable properties for approximating not only factor tensors (Section 3.1) but also their multiplication (Section 3.3). These aspects, which are crucial to scale NeSy problems, have been overlooked so far in the NeSy literature.

3.1. TT Factor Approximations

TTs decompose a high-dimensional tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{n_1\times\cdots\times n_d}$ (which we will call the *dense* tensor) as a product of tensor cores $\boldsymbol{\mathcal{T}}_{\boldsymbol{\mathcal{X}},1},\ldots,\boldsymbol{\mathcal{T}}_{\boldsymbol{\mathcal{X}},d}$ such that

$$\boldsymbol{\mathcal{X}}(c_1,\ldots,c_d) \approx \boldsymbol{\mathcal{T}}_{\boldsymbol{\mathcal{X}},1}(c_1)\boldsymbol{\mathcal{T}}_{\boldsymbol{\mathcal{X}},2}(c_2)\ldots\boldsymbol{\mathcal{T}}_{\boldsymbol{\mathcal{X}},d}(c_d).$$

See also Figure 2a. Each $\boldsymbol{\mathcal{T}}_{\boldsymbol{\mathcal{X}},k}$ is a 3-dimensional tensor of shape $r_{k-1}\times n_k\times r_k$, and we denote its matrix slices $\boldsymbol{\mathcal{T}}_{\boldsymbol{\mathcal{X}},k}(i_k) := \boldsymbol{\mathcal{T}}_{\boldsymbol{\mathcal{X}},k}(:,i_k,:)$. The ranks r_k are called the TT ranks, with $r_0 = r_d = 1$ such that multiplying all sliced matrices gives a scalar. Let $n = \max_k n_k$ and $r = \max_k r_k$. The space complexity of a TT is $\mathcal{O}(dnr^2)$, which can be significantly smaller than the $\mathcal{O}(n^d)$ space complexity of the full tensor when r is sufficiently small.

The TT-SVD algorithm (Oseledets, 2011) derives a TT approximation of $\boldsymbol{\mathcal{X}}$ either within a Frobenius error criterion $\|\boldsymbol{\mathcal{X}} - \tilde{\boldsymbol{\mathcal{X}}}\|_F \leq \epsilon$ or within a rank budget R. TT-SVD is quasi-optimal: its approximation error is within $\sqrt{d-1}$ of the best rank-R approximation of $\boldsymbol{\mathcal{X}}$. A major limitation of TT-SVD is that it requires the full tensor $\boldsymbol{\mathcal{X}}$ to be materialized, which has $\mathcal{O}(n^d)$ space complexity. This is generally feasible for factors ϕ_j with small scopes, but not for the full formula ϕ. [4] We therefore assume that TT-SVD derives approximated $\boldsymbol{\mathcal{X}}_j$ for each factor ϕ_j, as in CTSketch (Choi et al., 2025), but that these factors remain to be combined to derive an approximation of the full ϕ, which has not yet been explored.

3.2. Exact multiplication of TTs

TTs are closed under several operations including element-wise multiplication (see Figure 2c and Section E). The multiplication of two TTs $\boldsymbol{\mathcal{X}}_j$ and $\boldsymbol{\mathcal{X}}_l$ gives a new TT

$$\boldsymbol{\mathcal{X}}^*(c_1,\ldots,c_d) = \boldsymbol{\mathcal{T}}_{\boldsymbol{\mathcal{X}}^*,1}(c_1)\ldots\boldsymbol{\mathcal{T}}_{\boldsymbol{\mathcal{X}}^*,d}(c_d) \quad (6)$$

[4] If TT-SVD is not applicable to factors too, one can use black-box algorithms such as tensor cross interpolation (Oseledets & Tyrtyshnikov, 2010) or gradient descent using samples, which by extension provide generalization capabilities but no error bounds.

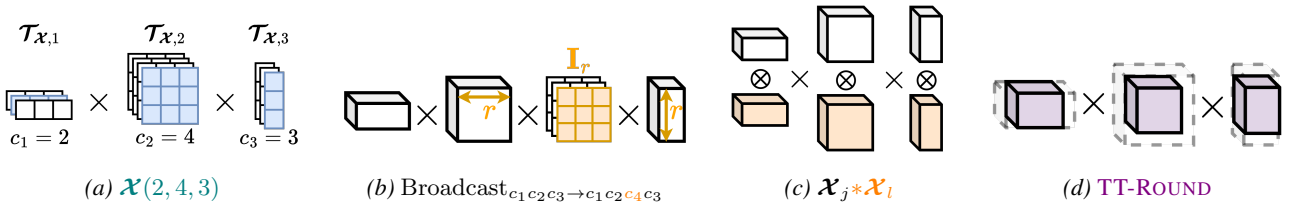


Figure 2. Closed algebra over TTs. TTs (a) can **approximate discrete functions** under an error criterion and (b-c) are closed under several **exact operations** including multiplications. As multiplications involve Kronecker products $\otimes$ of the cores resulting in an exponential increase in rank, algebraic recompression strategies (d) can **approximate the result of the operations**.

where $\mathcal{T}_{\mathcal{X}^*,k} = \mathcal{T}_{\mathcal{X}_j,k} \otimes \mathcal{T}_{\mathcal{X}_l,k}$ is the Kronecker product of the input cores. This can be seen as a special case of the MUL operation for structured-decomposable circuits which maintains a TT structure. Note that the Kronecker cores $\mathcal{T}_{\mathcal{X}_j,k} \otimes \mathcal{T}_{\mathcal{X}_l,k}$ have a quadratic rank increase. Unfortunately, this means that the multiplication of m TTs gives an exponential increase in rank, with space complexity $\mathcal{O}(ndr^{2m})$. [5] This matches the worst-case exponential complexity of circuit multiplication (Vergari et al., 2021). In Section 3.3, we discuss recompression strategies, new to circuits, to control the rank explosion of TT multiplications.

Broadcasting. Equation (6) assumes that the TTs $\mathcal{X}_j$ and $\mathcal{X}_l$ share (i) the same scope (the set of variables on which they depend), (ii) the same core-to-variable order. Assumption (ii) is easy to enforce: the order of the variables in a TT decomposition is arbitrary and can be set globally for all TTs. Assumption (i) is more restrictive as different factors ϕ_j usually have different scopes $\mathbf{c}_j$. We propose a simple trick for *broadcasting* TTs. For every missing variable (index) i_k in the scope of a TT, we can insert an *identity core* $\mathcal{I}_k$ in the TT order, where $\mathcal{I}_k(i_k) = \mathbf{I}_{r_{k-1}}$ is the identity matrix of dimension equal to the left bond rank previously at k. See also Figure 2b. The position at which the identity core is inserted influences the resulting ranks and thus the efficiency of the resulting TT. In Section E.2, we describe this in detail and derive a heuristic strategy to find a good TT variable order that minimizes the ranks after broadcast.

3.3. Approximate multiplications of TTs

The exponential rank increase from TT multiplication can be controlled by *recompressing* the result under a target rank R or a Frobenius tolerance ϵ. We consider four strategies (**A1-4**) along two axes: (i) the compression mechanism (SVD-based or interpolation-based), and (ii) whether the compression directions are chosen deterministically or via random projection. For simplicity, we describe each method for two factors $\mathcal{X} = \mathcal{X}_j * \mathcal{X}_l$ with uniform input and target ranks r. Below is a brief introduction of the four strategies. More details in Section H.

TT-Round (Oseledets, 2011) *(SVD, deterministic/site-randomized)* applies a sequence of truncated SVDs to the Kronecker cores of $\mathcal{X}$ to avoid materializing them.

HATT (Sun et al., 2025) *(SVD, randomized)* replaces the per-site SVDs of TT-Round by a global random TT sketch (Daas et al., 2021) of width $s = r + r_{\text{oversample}}$, drawn once and reused across sites via cached environments.

TCI (Núñez Fernández et al., 2025; Oseledets, 2011) *(interpolation, deterministic)* learns $\tilde{\mathcal{X}}$ from a small subset of its entries via maxvol-style cross interpolation (Goreinov et al.). At each site, a *supercore* of two (or more) consecutive cores is decomposed by a partially rank-revealing LU (Savostyanov, 2014) which selects the rows or columns that best represent it. TCI never materializes a Kronecker core nor performs a global SVD.

RSI (Meng et al., 2026) *(interpolation, randomized)* keeps TCI's supercore sweeps but replaces maxvol with a per-factor Khatri-Rao product sketch on each input TT (Camaño et al., 2026). By sidestepping the combinatorial maxvol search, it remains efficient for Hadamard multiplication over multiple TTs. The original RSI uses a single L2R sweep; we extend it to alternating L2R/R2L sweeps following the natural TCI connection (details in Section H.4).

4. A Unified Bottom-up Approximate Compilation Pipeline

We now combine the ingredients described so far into a modular bottom-up compilation pipeline for structured-decomposable circuits (Figure 1). We use the coloring scheme of Section 2 to distinguish the type of approximation involved (**factor**, **product**, **independence**) as opposed to exact factor **compilation** and **multiplication**. We use SDDs and TTs as representative exact and approximate circuits, but other structured-decomposable representations could technically be supported. The pipeline accepts a formula $\phi = \phi_1 \wedge \cdots \wedge \phi_N$ and exposes stages where different approximations can be applied. It recovers Semantic Strength-

[5] For WMC, a Matrix-Kronecker simplification exists for the inner product of m TTs (i.e., multiplication followed by a marginalization), which gives space complexity $\mathcal{O}(ndr^m)$. Therefore, multiplying two TTs is "free", as it matches their storage complexity, but the general case remains exponential in m.

ening (Ahmed et al., 2023) and CTSKETCH (Choi et al., 2025) as special cases, while providing new strategies like multiplications of TT-approximated factors (ATTIC).

4.1. Step I: Factor compilation (N)

For exact pipelines, each factor ϕ_j is compiled into an SDD circuit $\mathrm{KC}(\phi_j)$ (Darwiche, 2011). This requires ϕ_j to be propositionally encoded as a *CNF clause* which often introduces additional variables and mutual exclusivity constraints. For example, Sudoku 9x9 is transformed from a categorical program of 810 constraints over 81 variables to one of $10,287$ over 729 variables. ATTIC pipelines materialize ϕ_j as a boolean-valued tensor $\boldsymbol{\mathcal{X}}_j$ and apply **TT-SVD** to derive a TT approximation parameterized by an initial rank budget R_{init} or a Frobenius tolerance ϵ (see Section 3.3). ATTIC natively handles categorical variables. Optionally, before applying **TT-SVD**, ATTIC can start with an **optional pre-multiplication** step where small factors are grouped and multiplied *exactly in dense tensor space*: while $\boldsymbol{\mathcal{X}}_j$ has size $\mathcal{O}(n^{|\mathbf{c}_j|})$, in practice many factors are small enough that dense Hadamard products fit in memory. We multiply factors with similar scopes to maximize captured dependencies and minimize dense product sizes (see Section F for details).

4.2. Step II: Multiplication ($N \to K$)

The N factor circuits obtained after Step I are reduced to K circuits using **exact** or **approximate** multiplication strategies, where $1 \leq K \leq N$ can be fixed or adapted based on available memory (or error tolerance in the case of approximate strategies). **Exact multiplications** use the corresponding multiplication algorithm of the circuit representation: SDD Apply for SDDs (Darwiche, 2011) and TT multiplication for TTs (Section 3.2). We show in Theorem B.1 (Section B) that exact multiplications of approximated factors have a controlled error propagation which grows linearly with the number of factors multiplied.

Approximate multiplications are only applicable to the TN approximated pipeline. For TTs, we can use any of the recompression strategies described in Section 3.3 (**TT-ROUND, HATT, TCI, RSI**). These methods introduce additional errors specific to each strategy. However, SVD-based methods like TT-ROUND maintain guarantees on the Frobenius error of the recompressed product, which adds to the error of the exact result (Oseledets, 2011). This guaranteed error depends, as usual, on the rank budget R. Another design choice is the order of multiplication, which matters particularly when $K > 1$ as this determines which factors will be multiplied independently in the next step. We choose to organize multiplications in a *binary tree structure*. We prioritize multiplying factors *sharing the most variables*, as they are likely to be the most dependent and thus benefit the most from exact multiplication. See Section F for details. This is a simplification of the idea proposed in Semantic Strengthening (Ahmed et al., 2023) to prioritize groups of factors with high mutual information.

4.3. Step III: Final independent reduction ($K \to 1$)

Any $K > 1$ residual factors are combined via the **independence approximation**. $K = 1$ disables this step. ATTIC can typically afford $K = 1$ using recompression strategies, at the cost of larger errors. SDD pipelines usually require $K > 1$ in practice. The **independence** approximation is crude, with an unbounded error. This approximation depends not only on the program semantics but also on how it is represented, which can lead to unpredictable behavior. As we will see in our results and as we derive analytically in Section D, the independence approximation for Sudoku derives estimates that vary by many orders of magnitude depending on the encoding used (categorical or binary).

5. Experiments

We evaluate approximate WMC approaches in two contexts: *inference* and *learning*. We construct our experiments around the Sudoku problem because (i) its complex dependence structure makes compilation non-trivial, unlike NeSy benchmarks like MNISTADD (Manhaeve et al., 2018) (see (Choi et al., 2025), Section C), (ii) its difficulty can easily be scaled with grid size, and (iii) it is a popular NeSy learning benchmark (Wang et al., 2019; Augustine et al., 2022).

Inference. We evaluate the model count (uniform WMC) capabilities as ground truth counts are available at large grid sizes. Specifically, we use: Sudoku 4x4 as an *easy* instance, where exact compilation with SDDs is possible; Sudoku 6x6 as a *challenging* instance, where compilation is only possible with the best non-structured-decomposable compilers like d4 (Lagniez & Marquis, 2017); and Sudoku 9x9 as a *very challenging* instance, where exact compilation is not possible with any existing compiler (the groundtruth MC is derived analytically (Felgenhauer & Jarvis, 2006)).

Learning. We evaluate two Visual Sudoku NeSy tasks. **(ViSudo-Classification)** We use the ViSudo-PC dataset (Augustine et al., 2022) to predict whether 9x9 boards of MNIST images $\mathbf{x}$ are valid Sudoku boards or not (100 train / 100 test examples). **(ViSudo-Completion)** Given predictions $\mathbf{c}$ for filled cells $\mathbf{x}$ in incomplete boards, we infer the missing cells $\mathbf{y}$ (see Section G.1 for the corresponding conditional WMC $p(\mathbf{y}|\mathbf{x})$ formulation). We use the SatNet (Wang et al., 2019) dataset (9000 train / 500 test examples), fixing the data leak observed in (Chang et al., 2023).

ATTIC setup. For the inference experiments, we sample many seeded points from the design space of Section 4. In step I of ATTIC, for the optional pre-multiplication of the

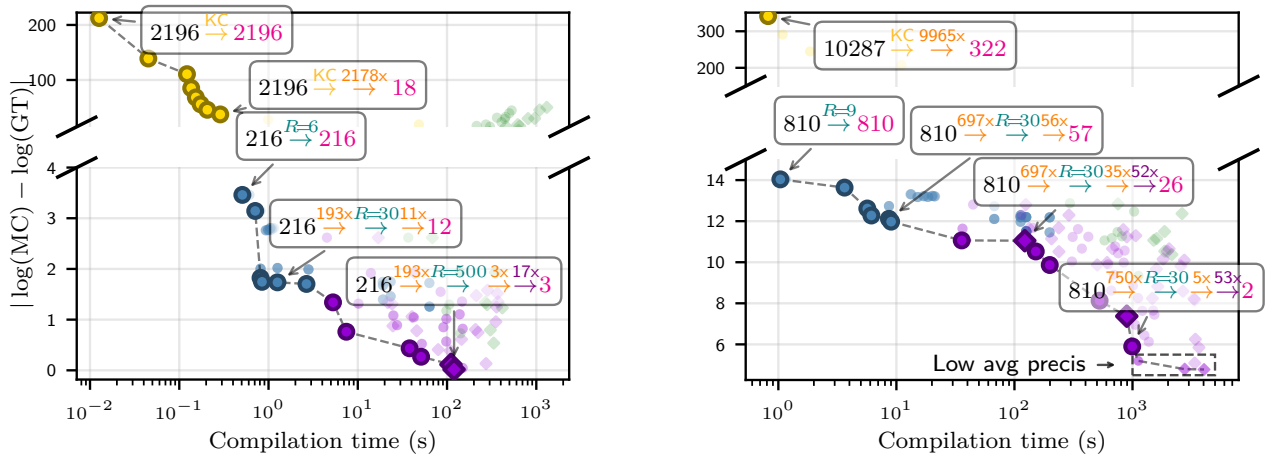


Figure 3. Pareto front of log inference error vs. compilation time for Sudoku 6x6 (left) and 9x9 (right). SDD pipelines are in yellow ●. ATTIC pipelines are in blue ● when they do not use recompressions, in violet when they use SVD-based recompressions (TT-ROUND ● and HATT ◆) and in green for interpolation-based strategies (TCI ● and RSI ◆).

N factors, we test the size budgets (as number of entries in the dense tensor) ($\{\text{skip}, 10^6, 10^9\}$). We share the resulting number of pre-multiplications #PREMUL. For TT-SVD, we test different initial rank budgets $R_{\text{init}} \in \{10, 30, 500\}$ – not necessarily fully exploited for tiny factors. In step II, we test different K reduction targets, to be followed by an independent reduction (step III) when $K > 1$. We start with # MUL exact multiplications, as much as fits within a rank budget of $R = 500$, followed by # ROUNDMUL compressed multiplications within the same budget – testing the four strategies described in Section 3.3. We summarize each point with a tag $N \xrightarrow{\#\text{PREMUL}} \xrightarrow{R_{\text{init}}} \xrightarrow{\#\text{MUL}} \xrightarrow{\#\text{RMUL}} K$. For example, $810 \xrightarrow{9} 810$ is CTSKETCH transforming the factors into TTs with rank 9 then fuzzily multiplying their WMC. $810 \xrightarrow{9} \xrightarrow{405\text{x}} 405$ adds 405 paired exact multiplications.

SDD setup. We test different K reduction targets within our memory budget. Step III applies when $K > 1$. We summarize each point with a tag $N \xrightarrow{\text{KC}} \xrightarrow{\#\text{MUL}} K$.

For learning, testing all (hundreds of) points is infeasible, so we select strong representative ones. We implement all TT algorithms in PyTorch (Paszke et al., 2019) for GPU acceleration. We use PySDD (Meert & Choi, 2017) for SDD compilation and KLay (Maene et al., 2025) for GPU inference on SDDs. We conduct our experiments on a single A100 GPU with 40 GB memory. Learning results are averaged over six (ViSudo-Classification) and three (ViSudo-Completion) runs to account for data-size variance. Hyperparameters (batch size, learning rate) are described in Section G.2.

5.1. Inference Results

The ground-truth model counts are 288 (4x4), 28, 200, 960 (6x6) and 6.67×10^{21} (9x9). On 4x4, many pipelines recover the exact model count with small ϵ errors instantly (Section G.3), as expected: SDD compiles the full formula exactly, so approximate approaches should reduce to $K = 1$ with negligible error. 6x6 and 9x9 are more informative.

In Figure 3, we plot the Pareto front of the different sampled strategies. Because of the large model counts, we report the log error $\log_{10}(\text{estimated MC}) - \log_{10}(\text{true MC})$. Overall, ATTIC pipelines achieve strong error-time trade-offs, with Pareto points several orders of magnitude more accurate than the best SDD pipelines. We highlight trends by approximation type.

On independence approximations. As predicted by our discussion on fuzzy errors (Section B), the *fuzzy WMC approximations behave unpredictably*, varying with the encoding. Even at matched K, SDD (yellow) and ATTIC behave very differently. In practice, the MC estimate of SDD+Fuzzy is near zero and hardly usable.

On exact multiplications. In both cases, *multiplying factors always improves the error by orders of magnitude.* The fully exact SDD pipelines can reduce the program to $K = 5$ factors for 6x6 and $K = 41$ for 9x9 before running out of memory on our system. **Factor** approximations *achieve a comparable number of feasible multiplications*, but not much further: TT ranks *still grow exponentially*. One advantage, however, is that factor growth is predictable, whereas SDD multiplication cost varies with circuit structure. Although it seems SDD runs are consistently fast, we only display finished runs. Other configurations either went out of space or timed out. ATTIC *multiplications are more*

Table 1. Compilation pipelines and baselines they reduce to on two Visual Sudoku tasks.

Pipeline	Method	ViSudo-Classification Acc (%) ↑	ViSudo-Completion $\mathbf{c}$ Acc (%) ↑	ViSudo-Completion $\mathbf{y}$ Acc (%) ↑
801	LTN, DFL	63.7 ± 1.6	99.0 ± 0.2	69.6 ± 5.8
● $801 \xrightarrow{R=9} 801$	CTSKETCH	62.0 ± 1.1	99.1 ± 0.0	71.5 ± 0.0
● $801 \xrightarrow{697x} \xrightarrow{R=3} 056x \rightarrow 57$	ATTIC A	62.3 ± 1.0	99.1 ± 0.0	72.3 ± 0.2
● $801 \xrightarrow{697x} \xrightarrow{R=3} 015x \rightarrow 91x \rightarrow 7$	ATTIC B	60.5 ± 0.0	99.2 ± 0.0	73.6 ± 0.7
● $10287 \xrightarrow{KC} 10287$	∼ SemStrength[a]	61.8 ± 2.5	98.9 ± 0.0	67.5 ± 1.1
● $10287 \xrightarrow{KC} \xrightarrow{10126x} 41$	∼ SemStrength	61.7 ± 1.8	99.0 ± 0.0	69.6 ± 0.4
$10287 \xrightarrow{KC} \xrightarrow{10287x} 1$	SL, DPL	oot	oot	oot

[a] Semantic strengthening chooses multiplications by KL divergence; we use scope overlap.

controllable and always fast thanks to GPU parallelization.

On approximate multiplications. Approximations can push the limit of feasible multiplications, but their error behavior and compilation time depend on the reduction target K and the strategy used. *The best strategies are the SVD-based ones* (TT-ROUND and HATT in violet), with no significant difference between randomization strategies. On the other hand, the interpolation-based strategies (TCI and RSI in green) are much less stable, as many points have worse error than stopping at exact multiplications. Interpolation lacks guarantees and struggles on the sparse boolean TTs here, unlike on smooth functions. Even with SVD-based strategies, too many multiplications under a tight rank budget can significantly degrade precision. For every Pareto point we check whether 1000 model samples obtained from www.menneske.no/sudoku score above non-model samples when evaluating $\mathcal{X}(\mathbf{c})$ (average precision, expected 1). All Pareto points have AP $= 1$ except three (marked "low AP") which, after too many multiplications, achieve good global error but miss the actual models. This failure is controllable: SVD-based recompressions expose a Frobenius error estimate at each truncation, so compilation can halt once the accumulated bound (Section B) exceeds a tolerance.

5.2. Learning Results

In ViSudo-Completion, we measure concept accuracy ($\mathbf{c}$ MNIST digit classification) and completion accuracy ($\mathbf{y}$: whole-grid correctness, a stricter metric). In ViSudo-Classification, we omit $\mathbf{c}$ accuracy as the task gives no signal for it: board validity depends on cell relations, not on cell values, so any permutation of the correct digits works (a "reasoning shortcut" (Marconato et al., 2023; 2026)).

In learning, we observe that *all pipelines achieve similar performance*. It might appear that some points slightly outperform others on one of the two tasks, but the difference is not statistically significant across runs nor consistent across tasks. In fact, concept accuracy is around 99% across all pipelines, leaving little room for improvement. Even fuzzy approximations are competitive, confirming previous findings that fuzzy systems learn well in many use cases (Badreddine et al., 2020) and behave surprisingly well as gradient estimators (Maene et al., 2024), despite their wildly inaccurate WMC approximations.

6. Conclusions

We unify existing bottom-up compilation methods for WMC under a modular pipeline (Figure 1) and instantiate previously unexplored configurations via tensor-train decompositions (ATTIC). Empirically, several ATTIC configurations achieve WMC estimates orders of magnitude more accurate than prior methods, while *where* approximation enters barely affects downstream learning, where even crude fuzzy relaxations remain competitive. Practically, it lets us pick configurations that match prior methods at training time while producing reliable WMC estimates at inference time.

Limitations. The TT decompositions and recompressions we use – particularly the interpolation-based variants (TCI, RSI) – are not tailored to the sparse boolean tensors of logical constraints. Designing recompressions specific to this regime is a natural direction for future work. On the learning side, all pipelines reach near-saturated concept accuracy on Visual Sudoku, leaving open whether more faithful WMC estimates translate into better downstream behavior. We believe answering this will require designing new benchmarks beyond those currently available in the NeSy literature.

Acknowledgements

We would like to thank Lorenzo Loconte, Christoph Staudt, Andreas Grivas and Joan Serrà for fruitful discussions during the writing of this paper. Emile van Krieken was funded by the NWO AiNed project "Human-Centric AI Agents with Common Sense" (NGF.1607.22.044). Luciano Serafini was funded by the NRRP project Future AI Research (FAIR - PE00000013) under the NRRP MUR program, funded by

NextGenerationEU. Antonio Vergari was supported by the "UNREAL: Unified Reasoning Layer for Trustworthy ML" project (EP/Y023838/1) selected by the ERC and funded by UKRI EPSRC.

References

Ahmed, K., Teso, S., Chang, K.-W., Van den Broeck, G., and Vergari, A. Semantic probabilistic layers for neuro-symbolic learning. In *Advances in Neural Information Processing Systems 35 (NeurIPS)*, volume 35, pp. 29944–29959. Curran Associates, Inc., 2022.

Ahmed, K., Chang, K.-W., and den Broeck, G. V. Semantic strengthening of neuro-symbolic learning, 2023. URL https://arxiv.org/abs/2302.14207.

Atkins, J. E., Boman, E. G., and Hendrickson, B. A spectral algorithm for seriation and the consecutive ones problem. *SIAM Journal on Computing*, 28(1): 297–310, 2003. ISSN 0097-5397. doi: 10.1137/s0097539795285771. URL https://doi.org/10.1137/s0097539795285771.

Augustine, E., Pryor, C., Dickens, C., Pujara, J., Wang, W. Y., and Getoor, L. Visual sudoku puzzle classification: A suite of collective neuro-symbolic tasks. In *International Workshop on Neural-Symbolic Learning and Reasoning (NeSy)*, Windsor, United Kingdom, 2022.

Badreddine, S., d'Avila Garcez, A. S., Serafini, L., and Spranger, M. Logic tensor networks. *CoRR*, abs/2012.13635, 2020. URL https://arxiv.org/abs/2012.13635.

Camaño, C., Epperly, E. N., and Tropp, J. A. Successive randomized compression: A randomized algorithm for the compressed mpo-mps product. *Quantum*, 10:2022, March 2026. ISSN 2521-327X. doi: 10.22331/q-2026-03-10-2022. URL http://dx.doi.org/10.22331/q-2026-03-10-2022.

Chang, O., Flokas, L., Lipson, H., and Spranger, M. Assessing satnet's ability to solve the symbol grounding problem, 2023. URL https://arxiv.org/abs/2312.11522.

Chavira, M. and Darwiche, A. On probabilistic inference by weighted model counting. *Artificial Intelligence.*, 172 (6-7):772–799, 2008.

Choi, S., Solko-Breslin, A., Alur, R., and Wong, E. CTSketch: Compositional Tensor Sketching for Scalable Neurosymbolic Learning, 2025. URL https://arxiv.org/abs/2503.24123. Version Number: 1.

Choi, Y., Vergari, A., and Van den Broeck, G. Probabilistic circuits: A unifying framework for tractable probabilistic modeling. Technical report, University of California, Los Angeles (UCLA), 2020.

Daas, H. A., Ballard, G., Cazeaux, P., Hallman, E., Miedlar, A., Pasha, M., Reid, T. W., and Saibaba, A. K. Randomized algorithms for rounding in the tensor-train format, 2021. URL https://arxiv.org/abs/2110.04393.

Darwiche, A. Decomposable negation normal form. *Journal of the ACM (JACM)*, 48:608–647, 2001.

Darwiche, A. A differential approach to inference in bayesian networks. *Journal of the ACM (JACM)*, 50: 280–305, 2003.

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

Darwiche, A. and Marquis, P. A knowledge compilation map. *Journal of Artificial Intelligence Research (JAIR)*, 17:229–264, 2002.

Dierckx, L., Dubray, A., Nijssen, S., and Kwok, J. Learning from logical constraints with lower-and upper-bound arithmetic circuits. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence*, pp. 5012–5020. International Joint Conferences on Artificial Intelligence, 2025.

Donadello, I., Serafini, L., and d'Avila Garcez, A. Logic tensor networks for semantic image interpretation, 2017. URL https://arxiv.org/abs/1705.08968.

Edmonds, J. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 2010. ISSN 0008-414X. doi: 10.4153/cjm-1965-045-4. URL https://doi.org/10.4153/cjm-1965-045-4.

Felgenhauer, B. and Jarvis, F. Mathematics of sudoku i. 2006. URL https://api.semanticscholar.org/CorpusID:122734246.

Goreinov, S. A., Oseledets, I. V., Savostyanov, D. V., Tyrtyshnikov, E. E., and Zamarashkin, N. L. *How to Find a Good Submatrix*, pp. 247–256. doi: 10.1142/9789812836021_0015. URL https://www.worldscientific.com/doi/abs/10.1142/9789812836021_0015.

Hagberg, A. A., Schult, D. A., and Swart, P. J. Exploring network structure, dynamics, and function using networkx. In *Proceedings of the 7th Python in Science Conference*, pp. 11–15, Pasadena, CA USA, 2008.

Halko, N., Martinsson, P.-G., and Tropp, J. A. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions, 2010. URL https://arxiv.org/abs/0909.4061.

Kimmig, A., Van den Broeck, G., and De Raedt, L. Algebraic model counting. *Journal of Applied Logic*, 22: 46–62, 2017.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations (ICLR)*, 2015.

Kisa, D. G., den Broeck, G. V., Choi, A., and Darwiche, A. Probabilistic sentential decision diagrams. In *International Conference on Principles of Knowledge Representation and Reasoning*, 2014.

Lagniez, J.-M. and Marquis, P. An improved decision-dnnf compiler. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pp. 667–673, 2017. doi: 10.24963/ijcai.2017/93. URL https://doi.org/10.24963/ijcai.2017/93.

Loconte, L., Aleksanteri, M. S., Mengel, S., Trapp, M., Solin, A., Gillis, N., and Vergari, A. Subtractive mixture models via squaring: Representation and learning. In *The Twelfth International Conference on Learning Representations (ICLR)*, 2024a.

Loconte, L., Mari, A., Gala, G., Peharz, R., de Campos, C., Quaeghebeur, E., Vessio, G., and Vergari, A. What is the relationship between tensor factorizations and circuits (and how can we exploit it)? *arXiv preprint arXiv:2409.07953*, 2024b.

Maene, J., Derkinderen, V., and De Raedt, L. On the hardness of probabilistic neurosymbolic learning. In Salakhutdinov, R., Kolter, Z., Heller, K., Weller, A., Oliver, N., Scarlett, J., and Berkenkamp, F. (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 34203–34218. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/maene24a.html.

Maene, J., Derkinderen, V., and Zuidberg Dos Martires, P. KLay: Accelerating arithmetic circuits for neurosymbolic AI. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=Zes7Wyif8G.

Manhaeve, R., Dumančić, S., Kimmig, A., Demeester, T., and De Raedt, L. Deepproblog: Neural probabilistic logic programming. *arXiv preprint arXiv:1805.10872*, 2018.

Marconato, E., Teso, S., and Passerini, A. Neuro-symbolic reasoning shortcuts: Mitigation strategies and their limitations, 2023. URL https://arxiv.org/abs/2303.12578.

Marconato, E., Bortolotti, S., van Krieken, E., Morettin, P., Umili, E., Vergari, A., Tsamoura, E., Passerini, A., and Teso, S. Symbol grounding in neuro-symbolic ai: A gentle introduction to reasoning shortcuts, 2026. URL https://arxiv.org/abs/2510.14538.

Mari, A., Vessio, G., and Vergari, A. Unifying and understanding overparameterized circuit representations via low-rank tensor decompositions. In *6th Workshop on Tractable Probabilistic Modeling*, 2023.

Meert, W. and Choi, A. Pysdd. In *Recent Trends in Knowledge Compilation*, number 17381 in Dagstuhl Reports, 2017.

Meng, Z., Khoo, Y., Li, J., and Stoudenmire, E. M. Recursive sketched interpolation: Efficient hadamard products of tensor trains, 2026. URL https://arxiv.org/abs/2602.17974.

Núñez Fernández, Y., Ritter, M. K., Jeannin, M., Li, J.-W., Kloss, T., Louvet, T., Terasaki, S., Parcollet, O., von Delft, J., Shinaoka, H., and Waintal, X. Learning tensor networks with tensor cross interpolation: New algorithms and libraries. *SciPost Physics*, 18(3), March 2025. ISSN 2542-4653. doi: 10.21468/scipostphys.18.3.104. URL http://dx.doi.org/10.21468/SciPostPhys.18.3.104.

Onaka, R., Nakamura, K., Nishino, M., and Yasuda, N. Tensor decomposition meets knowledge compilation: A study comparing tensor trains with obdds, 2025. URL https://arxiv.org/abs/2502.03702.

Orús, R. A practical introduction to tensor networks: Matrix product states and projected entangled pair states. *Annals of Physics*, 349:117–158, 2013.

Oseledets, I. and Tyrtyshnikov, E. TT-cross approximation for multidimensional arrays. *Linear Algebra and its Applications*, 432(1):70–88, January 2010. ISSN 00243795. doi: 10.1016/j.laa.2009.07.024. URL https://linkinghub.elsevier.com/retrieve/pii/S0024379509003747.

Oseledets, I. V. Tensor-train decomposition. *SIAM Journal on Scientific Computing*, 33:2295–2317, 2011.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E. Z., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative

style, high-performance deep learning library. *CoRR*, abs/1912.01703, 2019. URL http://arxiv.org/abs/1912.01703.

Peharz, R., Lang, S., Vergari, A., Stelzner, K., Molina, A., Trapp, M., Van den Broeck, G., Kersting, K., and Ghahramani, Z. Einsum networks: Fast and scalable learning of tractable probabilistic circuits. In *International Conference on Machine Learning*, pp. 7563–7574. PMLR, 2020.

Pipatsrisawat, K. and Darwiche, A. New compilation languages based on structured decomposability. In *23rd Conference on Artificial Intelligence (AAAI)*, volume 8, pp. 517–522, 2008.

Rogozhnikov, A. Einops: Clear and reliable tensor manipulations with einstein-like notation. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=oapKSVM2bcj.

Savostyanov, D. V. Quasioptimality of maximum-volume cross interpolation of tensors. *Linear Algebra and its Applications*, 458:217–244, October 2014. ISSN 0024-3795. doi: 10.1016/j.laa.2014.06.006. URL http://dx.doi.org/10.1016/j.laa.2014.06.006.

Schlag, S., Heuer, T., Gottesbüren, L., Akhremtsev, Y., Schulz, C., and Sanders, P. High-quality hypergraph partitioning, 2021. URL https://arxiv.org/abs/2106.08696.

Shen, Y., Choi, A., and Darwiche, A. Tractable operations for arithmetic circuits of probabilistic models. In *Advances in Neural Information Processing Systems 29 (NeurIPS)*. Curran Associates, Inc., 2016.

Smet, L. D., Martires, P. Z. D., Manhaeve, R., Marra, G., Kimmig, A., and Raedt, L. D. Neural probabilistic logic programming in discrete-continuous domains. *ArXiv*, abs/2303.04660, 2023a.

Smet, L. D., Sansone, E., and Martires, P. Z. D. Differentiable sampling of categorical distributions using the catlog-derivative trick. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023b. URL https://openreview.net/forum?id=AQyqxXctsN.

Sun, Z., Huang, J., Xiao, C., and Yang, C. Hatt: Hadamard avoiding tt recompression, 2025. URL https://arxiv.org/abs/2410.04385.

Valiant, L. G. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.

van Krieken, E., Acar, E., and van Harmelen, F. Semi-supervised learning using differentiable reasoning. *CoRR*, abs/1908.04700, 2019. URL http://arxiv.org/abs/1908.04700.

van Krieken, E., Acar, E., and van Harmelen, F. Analyzing differentiable fuzzy logic operators. *Artif. Intell.*, 302: 103602, 2022a.

van Krieken, E., Thanapalasingam, T., Tomczak, J. M., Harmelen, F. V., and ten Teije, A. A-nesi: A scalable approximate method for probabilistic neurosymbolic inference. *ArXiv*, abs/2212.12393, 2022b.

van Krieken, E., Minervini, P., Ponti, E. M., and Vergari, A. On the independence assumption in neurosymbolic learning, 2024. URL https://arxiv.org/abs/2404.08458.

van Krieken, E., Minervini, P., Ponti, E., and Vergari, A. Neurosymbolic reasoning shortcuts under the independence assumption, 2025a. URL https://arxiv.org/abs/2507.11357.

van Krieken, E., Minervini, P., Ponti, E., and Vergari, A. Neurosymbolic diffusion models, 2025b. URL https://arxiv.org/abs/2505.13138.

Vergari, A., Di Mauro, N., and Van den Broeck, G. Tractable probabilistic models: Representations, algorithms, learning, and applications. *Tutorial at the 35th Conference on Uncertainty in Artificial Intelligence (UAI)*, 2019.

Vergari, A., Choi, Y., Liu, A., Teso, S., and Van den Broeck, G. A compositional atlas of tractable circuit operations for probabilistic inference. In *Advances in Neural Information Processing Systems 34 (NeurIPS)*, pp. 13189–13201. Curran Associates, Inc., 2021.

Verreet, V., De Smet, L., De Raedt, L., and Sansone, E. Explain, agree, learn: scaling learning for neural probabilistic logic. In *ECAI 2024: 27th European Conference on Artificial Intelligence, 19–24 October 2024, Santiago de Compostela, Spain–Including 13th Conference on Prestigious Applications of Intelligent Systems (PAIS 2024)*, pp. 1349–1356. SAGE Publications Pvt. Ltd 1 Oliver's Yard, 55 City Road, London, EC1Y 1SP, 2024.

Wang, P., Donti, P. L., Wilder, B., and Kolter, J. Z. Satnet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver. *CoRR*, abs/1905.12149, 2019. URL http://arxiv.org/abs/1905.12149.

Xu, J., Zhang, Z., Friedman, T., Liang, Y., and Broeck, G. A semantic loss function for deep learning with symbolic knowledge. In *37th International conference on machine learning (ICML)*, pp. 5502–5511. PMLR, 2018.

A. Related Work

Knowledge compilation. Compilers more permissive than SDDs handle larger formulas at the cost of compositionality with other circuits. d4 (Lagniez & Marquis, 2017) is an exact compiler for d-DNNF, a broader class than SDDs that drops structured-decomposability. It scales to formulas no SDD compiler can handle (e.g., 6x6 Sudoku, though still not 9x9), but the resulting circuits can only be weighted by fully-factorized probabilities, an independence assumption known to lack expressivity in practice (Ahmed et al., 2022; van Krieken et al., 2024; 2025a). Dierckx et al. (2025) propose an approximate variant of d-DNNF compilation, which enumerates a budget-limited set of models into a DNF to derive lower- and upper-bound circuits. While these provide provable intervals, the resulting circuits are again not structured-decomposable, making them orthogonal to our target. Onaka et al. (2025) treat TTs themselves as a target for **exact factor compilation**, using them in the knowledge-compilation map alongside NNF subsets and showing they are strictly more succinct than OBDDs while supporting the same polytime operations. This corresponds to combining TT factors via exact multiplications without **TT-SVD approximation** or **product recompression**, and inherits the same exponential blow-up under multiplication that motivates the approximate strategies we study.

Scalable inference in NeSy. Other approaches avoid circuit compilation entirely. A-NeSI (van Krieken et al., 2022b) trains a *neural network* to amortize WMC inference, which are more expressive than circuits. Like ATTIC (but unlike SDD pipelines), A-NeSI can also learn the logic of black-box programs using input-output examples. However, the neural approach offers no formal error guarantees and gives a representation that does not support downstream tractable operations such as multiplication with other circuits or conditioning on evidence. *Monte Carlo methods* (Maene et al., 2024; Smet et al., 2023a;b; Verreet et al., 2024; van Krieken et al., 2025b) sample concept assignments to estimate gradients, which also extends to continuous settings, but Maene et al. (2024) show that the underlying gradient estimation from samples is itself #P-hard and can require an exponential number of samples to converge. Naturally, they also do not produce circuits that can be reused for downstream operations, targeting a different use case than ours.

B. Error bounds

We investigate how the different approximations impact the final WMC error. The **independence** approximation is crude, with an unbounded error. This approximation depends not only on the program semantics but also on how it is represented, which can lead to unpredictable behavior. As we will see in our results and as we derive analytically in Section D, the independence approximation for Sudoku derives estimates that vary by many orders of magnitude depending on the encoding used (categorical or binary).

The other approximations have more controlled error propagation. First, the ϵ error bound on the **factor** approximation introduced by TT-SVD in ATTIC pipelines propagates linearly with the number of factors when **multiplying them exactly**.

Theorem B.1. *Let $\mathcal{X}^* = \mathcal{X}_j * \mathcal{X}_l$ the element-wise product of two tensors. Let $\tilde{\mathcal{X}}^* = \tilde{\mathcal{X}}_j * \tilde{\mathcal{X}}_l$, where $\|\mathcal{X}_j - \tilde{\mathcal{X}}_j\|_F \leq \epsilon_j$ and $\|\mathcal{X}_l - \tilde{\mathcal{X}}_l\|_F \leq \epsilon_l$. Then, $\|\mathcal{X}^* - \tilde{\mathcal{X}}^*\|_F \leq \|\mathcal{X}_j\|_\infty \epsilon_l + \|\mathcal{X}_l\|_\infty \epsilon_j + \epsilon_j \epsilon_l$ where $\|\mathcal{X}_j\|_\infty = \max_{c_1,\ldots,c_d} |\mathcal{X}_j(c_1,\ldots,c_d)|$ is the max norm.*

Proof. Let $\Delta_j = \tilde{\mathcal{X}}_j - \mathcal{X}_j$ and $\Delta_l = \tilde{\mathcal{X}}_l - \mathcal{X}_l$, so that $\|\Delta_j\|_F \leq \epsilon_j$ and $\|\Delta_l\|_F \leq \epsilon_l$. Expanding the product,

$$\begin{aligned}\mathcal{X}^* - \tilde{\mathcal{X}}^* &= \mathcal{X}_j * \mathcal{X}_l - (\mathcal{X}_j + \Delta_j) * (\mathcal{X}_l + \Delta_l) \\ &= -\mathcal{X}_j * \Delta_l - \Delta_j * \mathcal{X}_l - \Delta_j * \Delta_l.\end{aligned}$$

By the triangle inequality,

$$\|\mathcal{X}^* - \tilde{\mathcal{X}}^*\|_F \leq \|\mathcal{X}_j * \Delta_l\|_F + \|\Delta_j * \mathcal{X}_l\|_F + \|\Delta_j * \Delta_l\|_F.$$

For any two tensors $\mathcal{A}, \mathcal{B}$ of matching shape, the Hadamard product satisfies $\|\mathcal{A} * \mathcal{B}\|_F \leq \|\mathcal{A}\|_\infty \|\mathcal{B}\|_F$ – follows from $\sum_{\mathbf{c}} (\mathcal{A}(\mathbf{c})\mathcal{B}(\mathbf{c}))^2 \leq \max_{\mathbf{c}} \mathcal{A}(\mathbf{c})^2 \sum_{\mathbf{c}} \mathcal{B}(\mathbf{c})^2 = \|\mathcal{A}\|_\infty^2 \|\mathcal{B}\|_F^2$. Applied to the first two terms,

$$\begin{aligned}\|\mathcal{X}_j * \Delta_l\|_F &\leq \|\mathcal{X}_j\|_\infty \|\Delta_l\|_F \leq \|\mathcal{X}_j\|_\infty \epsilon_l, \\ \|\Delta_j * \mathcal{X}_l\|_F &\leq \|\mathcal{X}_l\|_\infty \|\Delta_j\|_F \leq \|\mathcal{X}_l\|_\infty \epsilon_j.\end{aligned}$$

For the third term, the L^p-norm inequality $\|\Delta_l\|_\infty \leq \|\Delta_l\|_F$ (treating the tensor as a vector) gives

$$\|\Delta_j * \Delta_l\|_F \leq \|\Delta_l\|_\infty \|\Delta_j\|_F \leq \epsilon_l \, \epsilon_j.$$

Summing the three bounds yields the claim. □

As we study boolean tensors, small ϵ_j implies $\|\boldsymbol{\mathcal{X}}_j\|_\infty \approx 1$ and thus the error on $\boldsymbol{\mathcal{X}}^*$ is $\mathcal{O}(\epsilon_i + \epsilon_j)$. By extension, for N factors, the error on $\boldsymbol{\mathcal{X}}^*$ is $\mathcal{O}(N\epsilon)$ where $\epsilon = \max_j \epsilon_j$.

Recompression after multiplication (Section 3.3) introduces additional errors specific to each strategy. Deterministic TT-ROUND, in particular, can maintain guarantees on the Frobenius error of the recompressed product, which adds to the error of the exact result (Oseledets, 2011).

Reusing Theorem B.1 and considering the probability tensor $\boldsymbol{\mathcal{P}}$ such that $\boldsymbol{\mathcal{P}}(\mathbf{c}) = p_{\boldsymbol{\theta}}(\mathbf{c}|\mathbf{x})$ with $\epsilon = 0$ error, we can finally bound the error on the WMC as follows.

Corollary B.2. *Let $\tilde{\boldsymbol{\mathcal{X}}}$ an ϵ-approximation of $\boldsymbol{\mathcal{X}}$. Let $\boldsymbol{\mathcal{P}}$ a tensor of probabilities. Then, the Frobenius error of the weighted tensor is bounded by $\|\boldsymbol{\mathcal{P}} * \tilde{\boldsymbol{\mathcal{X}}} - \boldsymbol{\mathcal{P}} * \boldsymbol{\mathcal{X}}\|_F \leq \|\boldsymbol{\mathcal{P}}\|_\infty \epsilon$.*

C. CTSKETCH and Program Dependencies

CTSKETCH (Choi et al., 2025) approximates each factor ϕ_j with a tensor train (Section 3.1), computes the factor WMC independently, and multiplies them progressively following a "tree-like" structure. We formalize this using factor graphs and show how the assumption is violated by most NeSy programs, including the Sudoku tasks already targeted by Choi et al. (2025), where we find that CTSKETCH introduces independence approximations ((3)).

C.1. Tree-like factorizations

Example C.1 (sum$_4$ (Choi et al., 2025))**.** Consider the program sum$_4$ which takes as input four digits $c_1, c_2, c_3, c_4 \in \{0, \ldots, 9\}$ and outputs their sum $y = c_1 + c_2 + c_3 + c_4$. This program can be decomposed using two intermediate sums $y_1 = c_1 + c_2$ and $y_2 = c_3 + c_4$, followed by a final addition $y = y_1 + y_2$. The constraint factorizes as

$$\phi(\mathbf{c}, y) = \sum_{y_1, y_2} \underbrace{\mathbb{1}[c_1 + c_2 = y_1]}_{\phi_1(c_1, c_2, y_1)} \underbrace{\mathbb{1}[c_3 + c_4 = y_2]}_{\phi_2(c_3, c_4, y_2)} \underbrace{\mathbb{1}[y_1 + y_2 = y]}_{\phi_3(y_1, y_2, y)},$$

with the corresponding factor graph shown in Figure 4a.

Let $\mathbf{c}_1 = \{c_1, c_2\}$ and $\mathbf{c}_2 = \{c_3, c_4\}$ denote the two disjoint subsets of $\mathbf{c}$ associated with ϕ_1 and ϕ_2 respectively. The WMC rewrites as:

$$p_{\boldsymbol{\theta}}(y|\mathbf{x}) = \sum_{\mathbf{c}} p_{\boldsymbol{\theta}}(\mathbf{c}|\mathbf{x}) \left(\sum_{y_1, y_2} \phi_1(\mathbf{c}_1, y_1) \, \phi_2(\mathbf{c}_2, y_2) \, \phi_3(y_1, y_2, y) \right).$$

Assuming $p_{\boldsymbol{\theta}}(\mathbf{c}|\mathbf{x}) = \prod_i p_{\boldsymbol{\theta}}(c_i|\mathbf{x})$, CTSKETCH simplifies this marginalization in two steps. **First**, since ϕ_3 does not depend on $\mathbf{c}$ and $p_{\boldsymbol{\theta}}(\mathbf{c}|\mathbf{x})$ does not depend on y_1, y_2, the two sums commute and ϕ_3 can be moved outside:

$$\begin{aligned} p_{\boldsymbol{\theta}}(y|\mathbf{x}) &= \sum_{y_1, y_2} \phi_3(y_1, y_2, y) \left(\sum_{\mathbf{c}} p_{\boldsymbol{\theta}}(\mathbf{c}|\mathbf{x}) \phi_1(\mathbf{c}_1, y_1) \, \phi_2(\mathbf{c}_2, y_2) \right) \\ &= \sum_{y_1, y_2} \phi_3(y_1, y_2, y) \left(\sum_{\mathbf{c}} p_{\boldsymbol{\theta}}(\mathbf{c}|\mathbf{x}) \prod_{j=1}^{2} \phi_j(\mathbf{c}_j, y_j) \right) \end{aligned}$$

Second, the tree structure of the factor graph and disjoint scopes of ϕ_1, ϕ_2 let us push inner marginalizations down to the leaves.

$$p_{\boldsymbol{\theta}}(y|\mathbf{x}) = \sum_{y_1, y_2} \phi_3(y_1, y_2, y) \prod_{j=1}^{2} \left(\sum_{\mathbf{c}_j} p_{\boldsymbol{\theta}}(\mathbf{c}_j|\mathbf{x}) \, \phi_j(\mathbf{c}_j, y_j) \right)$$

Read in isolation, this second step has exactly the form of the independence approximation of Equation (3): it computes a per-factor WMC and multiplies the results. The crucial difference is that here the equality is *exact* – because the leaf **c**-scopes are genuinely disjoint, the per-factor marginals are independent rather than approximately so.

Choi et al. (2025) exploit this factorization and additionally approximate each ϕ_j as a tensor train, but the same trick applies directly to the dense factor tensors $\boldsymbol{\mathcal{X}}_j$. MNISTADD (Manhaeve et al., 2018) satisfies this tree-like assumption, as do other programs that decompose into trees of intermediate variables.

C.2. Overlapping factors

In most NeSy programs the factor graph is not a tree: variables appear in multiple factors and the corresponding scopes overlap substantially.

Example C.2 (Sudoku 4x4)**.** Consider the 4×4 Sudoku classification task, where the program takes as input 16 concepts $c_i \in \{1, 2, 3, 4\}$ representing the digits in each cell and outputs whether the grid is valid. The program decomposes into 56 pairwise inequality constraints across rows, columns and blocks. For example, the first row alone factorizes as

$$\phi_{\text{row1}}(c_1, c_2, c_3, c_4) = \underbrace{\mathbb{1}[c_1 \neq c_2]}_{\phi_1} \underbrace{\mathbb{1}[c_1 \neq c_3]}_{\phi_2} \underbrace{\mathbb{1}[c_1 \neq c_4]}_{\phi_3} \underbrace{\mathbb{1}[c_2 \neq c_3]}_{\phi_4} \underbrace{\mathbb{1}[c_2 \neq c_4]}_{\phi_5} \underbrace{\mathbb{1}[c_3 \neq c_4]}_{\phi_6}.$$

The corresponding factor graph is shown in Figure 4b. Each variable already participates in three factors within a single row, and in even more once columns and blocks are considered. These overlapping scopes and cycles in the factor graph prevent us from pushing marginalization sums inside the product, as we did in Example C.1.

Despite the overlap, Choi et al. (2025) apply the same per-factor simplification to Sudoku, pushing each marginalization sum to its associated factor as if the leaves had disjoint **c**-scopes:

$$\sum_{\mathbf{c}} p_{\boldsymbol{\theta}}(\mathbf{c}|\mathbf{x}) \prod_j \phi_j(\mathbf{c}_j) \overset{\text{I,F}}{\approx} \prod_j \Big(\sum_{\mathbf{c}_j} p_{\boldsymbol{\theta}}(\mathbf{c}_j|\mathbf{x}) \, \mathrm{AC}(\phi_j)(\mathbf{c}_j) \Big).$$

The right-hand side has exactly the form of the (exact) factorization in Example C.1, but here the overlapping $\mathbf{c}_j$ turn the equality into an approximation: it is the composition of a **factor approximation** – each ϕ_j replaced by its TT $\mathrm{AC}(\phi_j)$ – with an **independence approximation**. The independence step is the source of the unbounded error analyzed in Section B and empirically illustrated in Section D. The ATTIC pipelines proposed in Section 4 avoid this approximation when the budget allows by combining TN-approximated factors via exact (or recompressed) multiplications.

D. Independence Approximation and Categorical vs. Binary Encoding

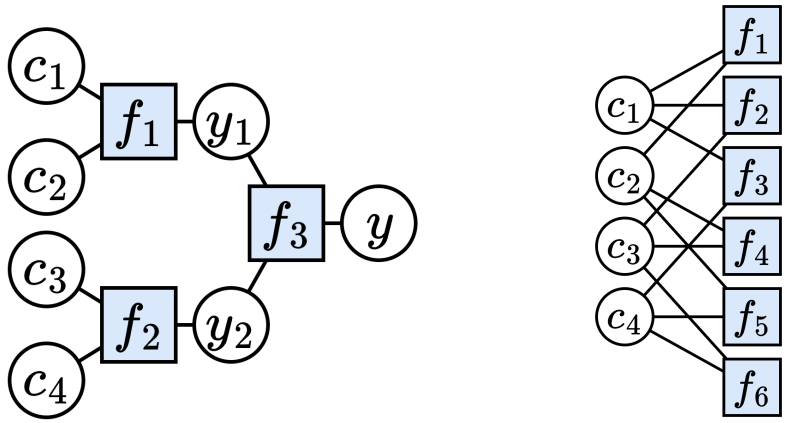
We illustrate how the **independence approximation** can yield drastically different model count estimates depending on whether ϕ is encoded over categorical or binary variables, even though the two encodings describe the same set of valid worlds. We use Sudoku as a concrete example, with the same constraint expressed once over categorical cells and once after the propositionalization required by SDD compilation (Section 2.1).

D.1. Independence Approximation under a Uniform Prior

Specializing the independence approximation to a uniform prior $p_{\boldsymbol{\theta}}(\mathbf{c}) = 1/|\mathcal{C}|$, the WMC reduces to the model count $\mathrm{MC} = \sum_{\mathbf{c}} \phi(\mathbf{c})$, and the approximation becomes

$$\widehat{\mathrm{MC}}^{\,\mathrm{I}} = |\mathcal{C}| \cdot \prod_{j=1}^{N} p_j, \qquad p_j := \frac{1}{|\mathcal{C}_j|} \sum_{\mathbf{c}_j \in \mathcal{C}_j} \phi_j(\mathbf{c}_j), \tag{7}$$

where p_j is the fraction of assignments over the scope $\mathbf{c}_j$ that satisfy ϕ_j. Both factors of Equation (7) depend on the encoding only through trivial counting, but their product is dominated by very different magnitudes in the two encodings.



(a) sum_4 program.

(b) First-row constraint of a 4×4 Sudoku program.

Figure 4. Factor graphs for (a) a tree-like program where marginalization sums can be pushed inside individual factors, and (b) a program with overlapping scopes and cycles where this simplification does not apply.

D.2. Sudoku 4x4: Categorical Encoding

The categorical formulation has variables $c_i \in \{1, \ldots, 4\}$ for $i = 1, \ldots, 16$, giving $|\mathcal{C}| = 4^{16}$. The constraint ϕ decomposes into $N = 56$ pairwise inequalities $\phi_j(c_i, c_l) = \mathbb{1}[c_i \neq c_l]$ over cell pairs (i, l) sharing a row, column, or box. Each factor has $|\mathcal{C}_j| = 16$ joint assignments of which 12 are satisfying, so $p_j = 3/4$. Plugging into Equation (7),

$$\widehat{\text{MC}}^{\text{I}}_{\text{cat}} \;=\; 4^{16} \cdot \left(\tfrac{3}{4}\right)^{56} \;\approx\; 432.9,$$

which overestimates the true MC of 288 by a factor of $\approx 1.5\times$.

D.3. Sudoku 4x4: Binary Encoding

SDD compilation requires propositionalizing ϕ via indicators $b_{i,v} = \mathbb{1}[c_i = v]$ for $i = 1, \ldots, 16$ and $v \in \{1, \ldots, 4\}$, resulting in 64 Boolean variables and $|\mathcal{C}| = 2^{64}$. The constraint becomes a conjunction of $N = 336$ CNF clauses of four types:

- *At-least-one* (16 clauses): $b_{i,1} \vee b_{i,2} \vee b_{i,3} \vee b_{i,4}$ per cell, with $p_j = 1 - (1/2)^4 = 15/16$.
- *At-most-one* (96 clauses): $\neg b_{i,v} \vee \neg b_{i,v'}$ for each cell i and value pair $v < v'$, with $p_j = 3/4$.
- *Distinct values across rows and columns* ($96 + 96$ clauses): $\neg b_{i,v} \vee \neg b_{l,v}$ for each row- or column-pair (i, l) and each value v, with $p_j = 3/4$.
- *Distinct values across boxes* (32 clauses): same form, restricted to box-pairs not already covered by rows or columns, with $p_j = 3/4$.

Plugging into Equation (7),

$$\widehat{\text{MC}}^{\text{I}}_{\text{bin}} \;=\; 2^{64} \cdot \left(\tfrac{15}{16}\right)^{16} \cdot \left(\tfrac{3}{4}\right)^{96+96+96+32} \;\approx\; 6.87 \times 10^{-22},$$

underestimating the true MC of 288 by roughly 24 orders of magnitude.

D.4. Discussion

The two estimates differ by approximately 24 orders of magnitude despite describing the same 288 valid boards. It is tempting to attribute this entirely to the N scaling of Equation (7), as propositionalization inflates the factor count from 56 to

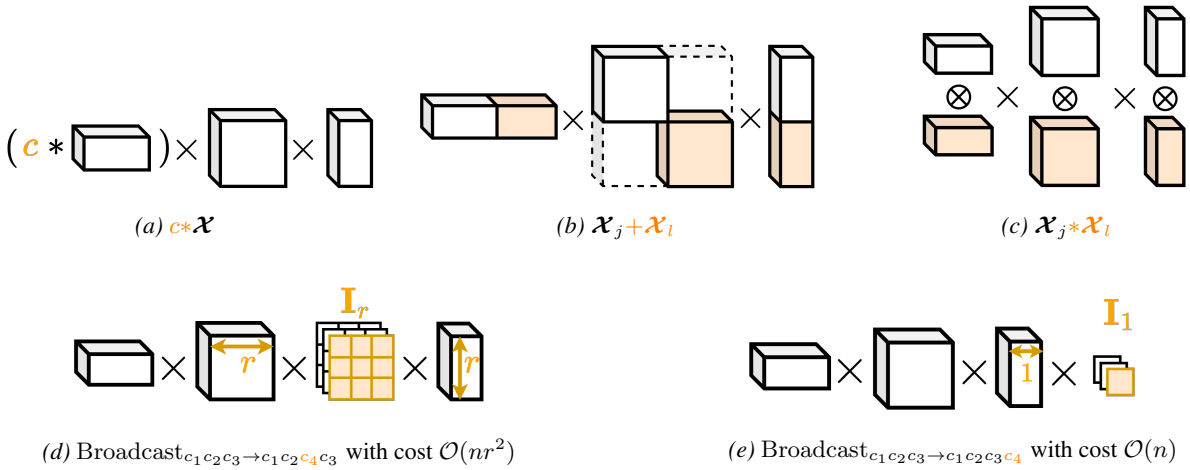


Figure 5. Closed algebra over tensor trains: (a) scalar multiplication, (b) addition, (c) Hadamard product, and (d-e) broadcasting with two different insertion positions. The cost of broadcasting is given as the size of the inserted *identity cores*.

336, and each additional sub-unit p_j pushes the product toward zero. Our inference experiments in Section 5 show that this picture is misleading: even when the SDD pipeline reduces the propositionalized factors back to a count K comparable to the categorical N via exact multiplications, the binary estimate continues to underestimate the true MC while the categorical one continues to overestimate it, and the two remain orders of magnitude apart.

The deeper issue is that the independence approximation, viewed as a fuzzy semantics with the product t-norm, is unbounded and *syntax-dependent*: its value depends not only on the set of models of ϕ but on how ϕ is written. As a minimal illustration, consider a single Boolean $a \in \{0, 1\}$. The formulas $\phi_1 := a \wedge a$ and $\phi_2 := a$ have the same single satisfying assignment (MC $= 1$), yet $\widehat{\mathrm{MC}}^{\mathrm{I}}(\phi_2) = 2 \cdot \frac{1}{2} = 1$ and $\widehat{\mathrm{MC}}^{\mathrm{I}}(\phi_1) = 2 \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{2}$.

Categorical and binary encodings of Sudoku express the same underlying constraint with very different syntactic profiles, and the product t-norm amplifies these differences in opposite directions: even after multiplying propositionalized factors back together, the residual factor structure is not the same as the categorical one and produces a systematically smaller $\prod_j p_j$. This contrasts with fuzzy semantics built on other t-norms (Gödel, Łukasiewicz), which have more established bounds and structural guarantees. (van Krieken et al., 2022a) Neither, however, corresponds to a probabilistic relaxation of Equation (1), which is what motivates the use of the product t-norm as the **independence approximation** in the first place.

E. Tensor Train Algebra

E.1. Addition, Scalar Multiplication and Hadamard Product

We briefly introduce the closed-form algebra of TTs (Oseledets, 2011). Figure 5 provides a graphical illustration. Let $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ denote two TTs over the same d variables in the same core-to-variable order, with cores $\boldsymbol{\mathcal{T}}_{\boldsymbol{\mathcal{A}},k}$ and $\boldsymbol{\mathcal{T}}_{\boldsymbol{\mathcal{B}},k}$ of TT ranks r_k^A and r_k^B respectively. We write $n = \max_k n_k$ for the maximal mode size and assume for simplicity that input TTs share a common rank r.

Scalar multiplication (Figure 5a). For $c \in \mathbb{R}$, the TT $c \cdot \boldsymbol{\mathcal{A}}$ is obtained by absorbing c into any single core, e.g. $\boldsymbol{\mathcal{T}}_{c\boldsymbol{\mathcal{A}},1}(c_1) = c\,\boldsymbol{\mathcal{T}}_{\boldsymbol{\mathcal{A}},1}(c_1)$. The ranks are unchanged and the resulting TT has the same $\mathcal{O}(dnr^2)$ storage as $\boldsymbol{\mathcal{A}}$.

Addition (Figure 5b). The sum $\boldsymbol{\mathcal{A}} + \boldsymbol{\mathcal{B}}$ is a TT whose cores stack the input cores block-diagonally,

$$\boldsymbol{\mathcal{T}}_{\boldsymbol{\mathcal{A}}+\boldsymbol{\mathcal{B}},k}(c_k) = \begin{pmatrix} \boldsymbol{\mathcal{T}}_{\boldsymbol{\mathcal{A}},k}(c_k) & 0 \\ 0 & \boldsymbol{\mathcal{T}}_{\boldsymbol{\mathcal{B}},k}(c_k) \end{pmatrix}, \qquad 1 < k < d,$$

with the boundary cores stacked horizontally ($k = 1$) and vertically ($k = d$) so that $r_0 = r_d = 1$ is preserved. The TT ranks add: $r_k^{A+B} = r_k^A + r_k^B$, giving total storage $\mathcal{O}(dn(r_k^A + r_k^B)^2) = \mathcal{O}(dnr^2)$ for two equal-rank inputs.

Hadamard product (Figure 5c). The element-wise product $\boldsymbol{\mathcal{A}} * \boldsymbol{\mathcal{B}}$ has cores given by the per-site Kronecker product of

Equation (6),

$$\mathcal{T}_{\mathcal{A}*\mathcal{B},k}(c_k) = \mathcal{T}_{\mathcal{A},k}(c_k) \otimes \mathcal{T}_{\mathcal{B},k}(c_k),$$

with multiplied ranks $r_k^{A*B} = r_k^A \cdot r_k^B$ and storage $\mathcal{O}(dn(r_k^A r_k^B)^2) = \mathcal{O}(dnr^4)$ for two equal-rank inputs. This quadratic per-product blow-up is the exponential rank growth discussed in Section 3.2 and motivates the recompression strategies of Section 3.3.

E.2. Broadcasting

The cores in Equation (6) assume that $\mathcal{X}_j$ and $\mathcal{X}_l$ are TTs over (i) the same scope and (ii) the same core-to-variable order. Assumption (ii) is easily enforced by fixing a global order σ of the variables in $\mathbf{c}$ and decomposing every factor in that order. Assumption (i) requires *broadcasting*: any TT whose scope $\mathbf{c}_j \subsetneq \mathbf{c}$ must be lifted to the common scope $\mathbf{c}$ before it can be multiplied with another TT.

Identity cores. For a TT $\mathcal{X}_j$ with scope $\mathbf{c}_j$ decomposed in the order induced by σ, broadcasting to $\mathbf{c}$ inserts, for every missing variable $c_k \notin \mathbf{c}_j$, an *identity core* $\mathcal{I}_k$ at the position dictated by σ:

$$\mathcal{I}_k(c_k) = \mathbf{I}_{r_{k-1}}, \qquad c_k = 1, \dots, |\mathcal{C}_k|,$$

where r_{k-1} is the rank of the bond at which $\mathcal{I}_k$ is inserted, so that $\mathcal{I}_k$ has shape $r_{k-1} \times |\mathcal{C}_k| \times r_{k-1}$. Since the slices of $\mathcal{I}_k$ are constant in c_k, the broadcasted TT evaluates to the same value as $\mathcal{X}_j$ for every assignment of the missing variables, as required.

Position matters. The position at which an identity core is inserted directly controls its size, and hence the storage and downstream multiplication cost of the broadcasted TT. If c_k is inserted *within the span* of $\mathbf{c}_j$ in σ – i.e., between two original cores – then $\mathcal{I}_k$ has size $\mathcal{O}(nr^2)$, where r is the local rank of $\mathcal{X}_j$ at that bond. If instead c_k is inserted *outside the span* – before the first or after the last variable of $\mathbf{c}_j$ – the surrounding bond rank is $r_0 = r_d = 1$ and $\mathcal{I}_k$ collapses to a $1 \times |\mathcal{C}_k| \times 1$ core of size $\mathcal{O}(n)$, leaving the storage essentially unchanged. Figures 5d and 5e contrast the two cases: inserting the same variable c_4 between c_2 and c_3 costs $\mathcal{O}(nr^2)$, while appending it after c_3 costs only $\mathcal{O}(n)$.

Choosing a global variable order. Different global orders σ yield different broadcasting costs because they change which missing variables fall inside or outside each factor's span. We seek a σ that minimizes the total cost of broadcasting all factors to their pairwise (or full) common scopes. We assume for simplicity a uniform rank r and uniform mode size n across all TTs. [6] The broadcasting cost of factor ϕ_j under σ scales with its number of *interior gaps*: the number of positions inside the span of $\mathbf{c}_j$ that are occupied by variables outside $\mathbf{c}_j$. Concretely, denoting by $\mathrm{pos}_\sigma(c_i)$ the position of c_i in σ, the interior gap length of ϕ_j is

$$\mathrm{gaplen}_j(\sigma) = \Big(\max_{c_i \in \mathbf{c}_j} \mathrm{pos}_\sigma(c_i) - \min_{c_i \in \mathbf{c}_j} \mathrm{pos}_\sigma(c_i) + 1 \Big) - |\mathbf{c}_j|, \tag{8}$$

where the parenthesised term is the span of $\mathbf{c}_j$ in σ and $|\mathbf{c}_j|$ is the number of own variables it contains. We seek the order minimising the total gap length,

$$\min_\sigma \sum_{j=1}^{N} \mathrm{gaplen}_j(\sigma). \tag{9}$$

Spectral ordering. Equation (9) is a variant of the (NP-hard) minimum linear arrangement problem on the hypergraph whose hyperedges are the factor scopes $\mathbf{c}_j$. We solve it heuristically via *spectral seriation*, which is fast, parameter-free, and reliably produces orders that are good enough for our pipelines.

We construct a weighted co-occurrence graph on the variables $\mathbf{c}$ with adjacency

$$\mathbf{W}_{ii'} = \sum_{j : c_i, c_{i'} \in \mathbf{c}_j} 1,$$

i.e., the number of factors in which c_i and $c_{i'}$ jointly appear. Variables that co-occur in many factors should sit close together in σ to keep factor spans short. Letting $D = \mathrm{diag}(\mathbf{W}\mathbf{1})$ be the degree matrix and $L = D - \mathbf{W}$ the graph Laplacian, the

[6] In practice, TT-SVD yields non-uniform ranks, but estimating the exact post-broadcast cost would require recompiling every factor for every candidate σ, which is prohibitive. The uniform-rank approximation is enough to capture the essential combinatorial structure of the problem.

Fiedler vector – the eigenvector associated with the second-smallest eigenvalue of L – provides a one-dimensional embedding of the variables that approximately minimises the weighted sum of squared distances between adjacent vertices (Atkins et al., 2003). We sort the variables by their Fiedler-vector entries to obtain σ.

This ordering is computed once per WMC instance, before any TT is built, and is then reused for all factors.

F. Prioritizing Multiplications

The bottom-up compilation pipeline of Section 4 pairs factors at two distinct stages: (i) the optional **pre-multiplication** of Step I, which groups dense factor tensors $\boldsymbol{\mathcal{X}}_j$ and multiplies them in dense space before **TT-SVD** (TT pipelines only); and (ii) the binary-tree **multiplication** of Step II, which iteratively pairs and multiplies the N compiled factor circuits into the K circuits consumed by Step III. The two stages share the same guiding intuition – *pair factors with the largest overlap in scope, so that within-pair multiplications consider dependencies that would otherwise be approximated away by Step III* – but face different constraints, and we therefore use two different strategies.

F.1. Dense Pre-multiplication: Hypergraph Partitioning

Pre-multiplication groups the N initial factors into K clusters $\mathcal{P}_1, \ldots, \mathcal{P}_K$, replacing each cluster by the dense Hadamard product

$$\boldsymbol{\mathcal{X}}_{\mathcal{P}_k} \;=\; \prod_{j \in \mathcal{P}_k} \boldsymbol{\mathcal{X}}_j,$$

with merged scope $\mathbf{c}_{\mathcal{P}_k} = \bigcup_{j \in \mathcal{P}_k} \mathbf{c}_j$ and dense size $\prod_{c_i \in \mathbf{c}_{\mathcal{P}_k}} |\mathcal{C}_i|$. We want clusters that are (a) as *decoupled* as possible – clusters that share few variables produce post-clustering factors that are close to independent, reducing the dependency burden on the rest of the pipeline – and (b) of *bounded dense size*, so that each $\boldsymbol{\mathcal{X}}_{\mathcal{P}_k}$ fits within a memory budget B (we use $B \in \{10^6, 10^9\}$ in our experiments).

Hypergraph formulation. We encode this as a balanced hypergraph partitioning problem. We build the hypergraph $\mathcal{H} = (V, E)$ in which the nodes $V = \{1, \ldots, N\}$ correspond to factors and carry weights $w_j = \log \prod_{c_i \in \mathbf{c}_j} |\mathcal{C}_i|$ (the log of the dense size of $\boldsymbol{\mathcal{X}}_j$), and the hyperedges $E = \{e_i\}_{i : c_i \in \mathbf{c}}$ correspond to variables, with $e_i = \{j : c_i \in \mathbf{c}_j\}$ collecting the factors that contain c_i. A hyperedge e_i is *cut* by a partition if it spans multiple parts, i.e., c_i appears in factors assigned to different clusters. The total number of cut hyperedges – the *cut-net* objective – is the hypergraph analogue of the edge cut for graphs and counts exactly the variables shared between distinct $\mathbf{c}_{\mathcal{P}_k}$. Minimising it realises objective (a).

Objective (b) becomes a balance constraint on the node weights. Since the dense size of $\boldsymbol{\mathcal{X}}_{\mathcal{P}_k}$ is upper-bounded by $\prod_{j \in \mathcal{P}_k} \prod_{c_i \in \mathbf{c}_j} |\mathcal{C}_i|$,[7] the constraint $\prod_{j \in \mathcal{P}_k} \prod_{c_i \in \mathbf{c}_j} |\mathcal{C}_i| \leq B$ rewrites linearly as $\sum_{j \in \mathcal{P}_k} w_j \leq \log B$. Combining the two,

$$\min_{\mathcal{P}_1, \ldots, \mathcal{P}_K} \big|\{\, i : e_i \text{ is cut} \,\}\big| \quad \text{s.t.} \quad \sum_{j \in \mathcal{P}_k} w_j \;\leq\; \log B \quad \forall\, k = 1, \ldots, K. \tag{10}$$

Implementation. We solve Equation (10) with the multilevel hypergraph partitioner KaHyPar (Schlag et al., 2021), increasing K from $K = 1$ until a feasible partition is found. KaHyPar's runtime is negligible compared to the compilation products themselves.

F.2. Circuit Multiplication: Binary-Tree Pairing

After Step I, we have N compiled circuits – SDDs, or TTs that may already have been pre-multiplied. Step II reduces them to K via a sequence of pairwise multiplications organised as a binary tree.

Why not hypergraph partitioning? The formulation of Equation (10) relied on each factor having a predictable size w_j. For compiled circuits, the size of the multiplication $\boldsymbol{\mathcal{X}}_j * \boldsymbol{\mathcal{X}}_l$ is not predictable from $\boldsymbol{\mathcal{X}}_j$ and $\boldsymbol{\mathcal{X}}_l$ alone: SDD APPLY can blow up unpredictably depending on vtree alignment, and even for TTs the Kronecker ranks depend on the tensors involved due to their rank patterns, broadcasting and unpredictable recompressions. Without an a-priori size estimate per

[7] The bound is tight only when scopes inside $\mathcal{P}_k$ are disjoint; large within-partition overlap shrinks the actual merged size well below the product. The cut-net objective explicitly steers toward large within-partition overlap, so the constraint is conservative on the cuts and slack inside the partitions.

multiplication, we cannot set per-cluster hard constraints in advance. We instead pair factors greedily, level by level, using the same *shared-variables* proxy that motivated the cut-net objective in Step I.

Maximum-weight matching. At each level, we build a complete graph on the surviving circuits and weight each edge by the size of the scope intersection $w_{jl} = |\mathbf{c}_j \cap \mathbf{c}_l|$ between the corresponding factors. We then pair circuits via a maximum-weight matching of this graph, which maximises the total intersection size across pairs and thereby keeps the most strongly coupled factors together. Each pair is multiplied (exactly or via recompression, depending on the pipeline configuration), the unmatched factor (if the level has odd cardinality) is carried over, and the procedure recurses on the resulting half-sized set, stopping when only K factors remain.

Implementation. We compute the matching exactly with NetworkX's implementation (Hagberg et al., 2008) of the blossom algorithm (Edmonds, 2010), which costs $\mathcal{O}(N^3)$ per level and $\mathcal{O}(\log_2 N)$ levels. For large N (e.g., with SDD propositionalised constraints), we use a greedy approximation.

G. Experiments (cont.)

G.1. Conditional WMC

In the conditional WMC setting, we still use a parameterized model $p_{\boldsymbol{\theta}}(\mathbf{c}|\mathbf{x})$ over the symbolic concepts, but the logical constraint $\phi(\mathbf{c}, \mathbf{y}) : \mathcal{C} \times \mathcal{Y} \to \{0, 1\}$ now also depends on output variables $\mathbf{y}$. The goal is to use the formula to derive a probability of $\mathbf{y}$ given $\mathbf{x}$:

$$p_{\boldsymbol{\theta}}(\mathbf{y}|\mathbf{x}) := \frac{\sum_{\mathbf{c}\in\mathcal{C}} p_{\boldsymbol{\theta}}(\mathbf{c}|\mathbf{x})\phi(\mathbf{c}, \mathbf{y})}{\sum_{\mathbf{y}'\in\mathcal{Y},\mathbf{c}\in\mathcal{C}} p_{\boldsymbol{\theta}}(\mathbf{c}|\mathbf{x})\phi(\mathbf{c}, \mathbf{y}')} \tag{11}$$

which, given a label $\mathbf{l} \in \mathcal{Y}$, is optimized by maximizing $p_{\boldsymbol{\theta}}(\mathbf{l}|\mathbf{x})$.

The denominator is a normalizing constant that ensures $p_{\boldsymbol{\theta}}(\mathbf{y}|\mathbf{x})$ is a valid distribution over $\mathbf{y}$. If, for every $\mathbf{c} \in \mathcal{C}$, there exists one and only one $\mathbf{y} \in \mathcal{Y}$ such that $\phi(\mathbf{c}, \mathbf{y}) = 1$, then the denominator simplifies to 1. Many works (Choi et al., 2025; van Krieken et al., 2022b; 2025b) take this assumption and write the symbolic program as a function $\mathbf{y} = f(\mathbf{c})$. We note however that this is not the case for Visual Sudoku completion: some partial boards $\mathbf{c}$ can be completed in multiple ways, and some do not admit any completion. However, the SatNet dataset (Wang et al., 2019) is constructed so that every $\mathbf{c}$ has only one valid completion, such that in practice works that optimize the numerator alone (Choi et al., 2025) still achieve good results.

G.2. Training details

We use the dataset splits provided with the original benchmarks for both Visual Sudoku completion and Visual Sudoku classification. All models are trained with the Adam optimizer (Kingma & Ba, 2015) and a batch size of 8, kept small to accommodate the memory footprint of the large circuits we compile. For Visual Sudoku completion, we use a learning rate of 10^{-5} and train for 100 epochs. For Visual Sudoku classification, we use a learning rate of 10^{-3} and train for 1000 epochs.

G.3. Sudoku 4x4 inference

On Sudoku 4x4, we confirm that both SDD and ATTIC pipelines can derive the exact model count within a tiny ϵ error. As SDD can exactly reduce the formula into a single circuit, we also expect that approximate compilation can achieve a reduction to $K = 1$ with negligible error, which is what we observe in practice. We also notice that, as introducing approximate multiplications is slower than exact ones due to the overhead of recompression (though, they still compile within seconds), they are not beneficial in such small scale settings, and the best point on the Pareto front is achieved by the SDD pipelines.

H. Approximation Algorithms for TT Multiplication

In this appendix, we detail the four recompression strategies introduced in Section 3.3. All algorithms take as input two compatible TTs $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ (same scope, same core ordering, same mode dimensions n_k) and a target maximum rank R or Frobenius tolerance ϵ, and return a TT approximation $\tilde{\boldsymbol{\mathcal{X}}}$ of the Hadamard product $\boldsymbol{\mathcal{X}} = \boldsymbol{\mathcal{A}} * \boldsymbol{\mathcal{B}}$. We denote by $\boldsymbol{\mathcal{T}}_{\boldsymbol{\mathcal{A}},k} \in \mathbb{R}^{r^A_{k-1}\times n_k \times r^A_k}$ the k-th core of $\boldsymbol{\mathcal{A}}$, and similarly for $\boldsymbol{\mathcal{B}}$ and $\tilde{\boldsymbol{\mathcal{X}}}$.

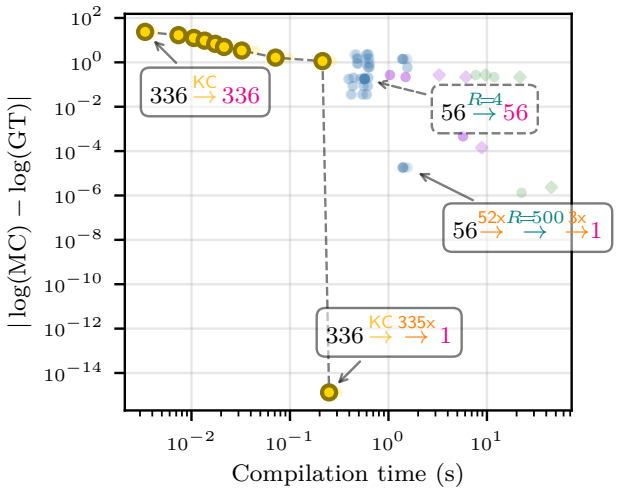


Figure 6. Pareto front of log inference error vs compilation time for Sudoku 4x4.

Notation. We follow einsum and einops conventions for tensor operations (Rogozhnikov, 2022). Given tensors $\mathcal{T}_1, \mathcal{T}_2$:

- EINSUM(`"abc, cde -> abde"`, $\mathcal{T}_1, \mathcal{T}_2$) denotes Einstein summation, contracting indices that appear in both inputs but not in the output.
- REARRANGE(`"(a b) c -> a b c"`, $\mathcal{T}$) denotes a reshape/permutation, where parentheses on the left split a single axis into multiple axes (with explicit sizes given as side arguments) and parentheses on the right merge multiple axes into a single axis.

H.1. TT-ROUND

TT-ROUND (Oseledets, 2011) computes $\tilde{\mathcal{X}} \approx \mathcal{A} * \mathcal{B}$ by interleaving the Kronecker product of cores with a sweep of truncated SVDs. The naive variant first materializes the inflated cores $\mathcal{T}_{\mathcal{X},k} = \mathcal{T}_{\mathcal{A},k} \otimes \mathcal{T}_{\mathcal{B},k}$ of rank $r_k^A r_k^B$ and then applies a standard rounding sweep, paying $\mathcal{O}(dnr^4)$ memory for the inflated cores. The *single-pass* variant presented here exploits the Matrix-Kronecker structure of the inflated cores to avoid this materialization, reducing the memory cost to $\mathcal{O}(r^3)$.

The algorithm proceeds in two phases:

1. **Right-orthogonalization.** Both $\mathcal{A}$ and $\mathcal{B}$ are first right-orthogonalized so that the right unfolding of every core is row-orthonormal. This ensures that the subsequent local SVD truncations are quasi-optimal with respect to the global Frobenius norm.
2. **Left-to-right SVD sweep.** A left environment $\mathcal{L} \in \mathbb{R}^{r_{k-1} \times r_{k-1}^A \times r_{k-1}^B}$ accumulates the already-compressed prefix. At each site k, the algorithm (i) contracts $\mathcal{L}$ with the (implicit) Kronecker slice $\mathcal{T}_{\mathcal{A},k} \otimes \mathcal{T}_{\mathcal{B},k}$ via a single einsum, producing a working tensor $\mathcal{W}$; (ii) flattens $\mathcal{W}$ into a matrix and performs a truncated SVD with target rank R (or tolerance ϵ); (iii) stores the left singular vectors as the new core $\mathcal{T}_{\tilde{\mathcal{X}},k}$ and absorbs $\mathrm{diag}(\mathbf{s})\mathbf{V}^\top$ into the next left environment.

The last core requires no SVD truncation since its rightmost rank is already 1 by definition.

Randomized per-site SVD. When the target rank R is small relative to the working tensor dimensions, the truncated SVD on $\mathcal{W}$ (Algorithm 1, line 8) can be replaced by a randomized low-rank SVD (Halko et al., 2010): $\mathcal{W}$ is multiplied by a Gaussian sketch of width $R + r_{\text{oversample}}$, and the dominant left singular vectors are recovered from the resulting tall-and-skinny matrix. We emphasize that this is a purely *local*, matrix-level acceleration — a fresh sketch is drawn independently at each site, and the sweep structure, orthogonalization, and per-site target ranks are unchanged. This is conceptually distinct from the TT-level sketching used by HATT (Section H.2), which draws a single random TT once and reuses it across all sites by precomputing right environments, so that no unfolding of $\mathcal{X}$ is ever formed.

Algorithm 1 Hadamard TT-ROUND

Require: TTs $\boldsymbol{\mathcal{A}}, \boldsymbol{\mathcal{B}}$ with cores $\{\boldsymbol{\mathcal{T}}_{\boldsymbol{\mathcal{A}},k}\}_{k=1}^{d}$, $\{\boldsymbol{\mathcal{T}}_{\boldsymbol{\mathcal{B}},k}\}_{k=1}^{d}$; max rank R; tolerance ϵ.
Ensure: TT approximation $\tilde{\boldsymbol{\mathcal{X}}} \approx \boldsymbol{\mathcal{A}} * \boldsymbol{\mathcal{B}}$.

$\boldsymbol{\mathcal{A}} \leftarrow$ RIGHTORTHOGONALIZE($\boldsymbol{\mathcal{A}}$)
$\boldsymbol{\mathcal{B}} \leftarrow$ RIGHTORTHOGONALIZE($\boldsymbol{\mathcal{B}}$)
$\boldsymbol{\mathcal{L}} \leftarrow \mathbf{1} \in \mathbb{R}^{1\times1\times1}$ ▷ initial left environment
for $k = 1, \ldots, d$ **do**
  $\boldsymbol{\mathcal{W}} \leftarrow$ EINSUM(`"r ra rb, ra n ra′, rb n rb′ -> r n ra′ rb′"`, $\boldsymbol{\mathcal{L}}$, $\boldsymbol{\mathcal{T}}_{\boldsymbol{\mathcal{A}},k}$, $\boldsymbol{\mathcal{T}}_{\boldsymbol{\mathcal{B}},k}$)
  $\boldsymbol{\mathcal{W}} \leftarrow$ REARRANGE(`"r n ra′ rb′ -> (r n) (ra′ rb′)"`, $\boldsymbol{\mathcal{W}}$)
  if $k < d$ **then**
    $\mathbf{U}, \mathbf{s}, \mathbf{V}^\top \leftarrow$ TRUNCATEDSVD($\boldsymbol{\mathcal{W}}$; ϵ, R)
    $\boldsymbol{\mathcal{T}}_{\tilde{\boldsymbol{\mathcal{X}}},k} \leftarrow$ REARRANGE(`"(r n) r′ -> r n r′"`, $\mathbf{U}$)
    $\boldsymbol{\mathcal{L}} \leftarrow$ REARRANGE(`"r′ (ra′ rb′) -> r′ ra′ rb′"`, $\mathrm{diag}(\mathbf{s})\,\mathbf{V}^\top$)
  else
    $\boldsymbol{\mathcal{T}}_{\tilde{\boldsymbol{\mathcal{X}}},d} \leftarrow$ REARRANGE(`"(r n) 1 -> r n 1"`, $\boldsymbol{\mathcal{W}}$)
  end if
end for
return TT $\tilde{\boldsymbol{\mathcal{X}}}$ with cores $\{\boldsymbol{\mathcal{T}}_{\tilde{\boldsymbol{\mathcal{X}}},k}\}_{k=1}^{d}$.

Complexity. At site k, $\boldsymbol{\mathcal{W}}$ has shape $r_{k-1}n_k \times r_k^A r_k^B$ and its truncated SVD costs $\mathcal{O}(nr^4)$ for uniform ranks r. Summed over d sites, this gives $\mathcal{O}(dnr^4)$ time. Memory is dominated by storing $\boldsymbol{\mathcal{W}}$ and the left environment, both of which are $\mathcal{O}(r^3)$ since the Kronecker factors are never explicitly formed. When randomized per-site SVDs are used in place of deterministic ones, the per-site cost drops to $\mathcal{O}(nr^3)$ since the SVD operates on a sketched matrix of column dimension $\mathcal{O}(r)$ rather than on the full $r_k^A r_k^B = r^2$ unfolding, giving an overall time complexity of $\mathcal{O}(dnr^3)$ — matching that of HATT (Section H.2).

H.2. HATT

HATT (Sun et al., 2025) replaces the deterministic SVD-based truncation of TT-ROUND with a randomized sketching approach inspired by randomized linear algebra (Halko et al., 2010; Daas et al., 2021). Like TT-ROUND, it exploits the Matrix-Kronecker structure of the inflated cores $\boldsymbol{\mathcal{T}}_{\boldsymbol{\mathcal{A}},k} \otimes \boldsymbol{\mathcal{T}}_{\boldsymbol{\mathcal{B}},k}$ to avoid materializing them. Unlike TT-ROUND— whose per-site SVDs (deterministic or randomized) still operate on full unfoldings of the inflated working tensor $\boldsymbol{\mathcal{W}}$ — HATT projects $\boldsymbol{\mathcal{X}}$ against a TT sketch $\boldsymbol{\mathcal{S}}$ of width $s = R + r_{\text{oversample}}$ first and applies SVDs only on small sketched matrices of column dimension s.

The algorithm proceeds in two phases.

Phase 1: Sketched right environments. A random TT sketch $\boldsymbol{\mathcal{S}}$ with cores $\boldsymbol{\mathcal{T}}_{\boldsymbol{\mathcal{S}},k} \in \mathbb{R}^{s_{k-1}\times n_k \times s_k}$ of width s is drawn (e.g., with Gaussian or Rademacher entries) and left-orthogonalized for numerical stability. A right-to-left sweep (Algorithm 2) then produces the right environments $\boldsymbol{\mathcal{R}}_k \in \mathbb{R}^{r_k^A \times r_k^B \times s_k}$ for $k = 1, \ldots, d-1$. Each $\boldsymbol{\mathcal{R}}_k$ represents the contraction of the inflated cores at positions $k+1, \ldots, d$ projected against the right tail of $\boldsymbol{\mathcal{S}}$.

As an extension to the original HATT formulation, we additionally allow the user to provide a TT $\boldsymbol{\mathcal{G}}$ as an initial guess of the solution, which is then used in place of a random sketch. This can leverage prior knowledge about $\boldsymbol{\mathcal{X}}$ – for instance, the result of a previous Hadamard product in a bottom-up compilation – to bias the sketch towards the relevant subspace.

Phase 2: Left-to-right projection sweep. A left-to-right sweep analogous to that of TT-ROUND constructs the cores of $\tilde{\boldsymbol{\mathcal{X}}}$. At each site k, the algorithm contracts the left environment $\boldsymbol{\mathcal{L}}$, the inflated core $\boldsymbol{\mathcal{T}}_{\boldsymbol{\mathcal{A}},k} \otimes \boldsymbol{\mathcal{T}}_{\boldsymbol{\mathcal{B}},k}$, and the sketched right environment $\boldsymbol{\mathcal{R}}_k$ into a small working tensor $\mathbf{M}$ of shape $r_{k-1} \times n_k \times s_k$, where r_{k-1} is the running compressed rank. A truncated SVD on the unfolding of $\mathbf{M}$ then yields the left singular vectors used as the new core $\boldsymbol{\mathcal{T}}_{\tilde{\boldsymbol{\mathcal{X}}},k}$. The left environment is updated by contracting the new core back against the inflated cores, and the loop proceeds. As in TT-ROUND, the last core requires no SVD since $s_d = 1$.

Complexity. The dominant cost at each site is the truncated SVD on $\mathbf{M}$ of shape $r_{k-1}n_k \times s_k$, costing $\mathcal{O}(nr^2 s)$ under uniform ranks. Summed over the d sites, this gives $\mathcal{O}(dnr^2 s)$ time. Memory is dominated by storing the right environments, at $\mathcal{O}(r^2 s)$ each. The right-environment sweep (Algorithm 2) has the same asymptotic cost of $\mathcal{O}(dnr^2 s)$. For $s \approx r$, the

Algorithm 2 Sketched right environments for $\boldsymbol{\mathcal{X}} = \boldsymbol{\mathcal{A}} * \boldsymbol{\mathcal{B}}$

Require: TT cores $\{\boldsymbol{\mathcal{T}}_{\boldsymbol{\mathcal{A}},k}\}_{k=1}^{d}, \{\boldsymbol{\mathcal{T}}_{\boldsymbol{\mathcal{B}},k}\}_{k=1}^{d}$; sketch TT cores $\{\boldsymbol{\mathcal{T}}_{\boldsymbol{\mathcal{S}},k}\}_{k=1}^{d}$.
Ensure: Right environments $\{\boldsymbol{\mathcal{R}}_k\}_{k=1}^{d-1}$.

```
R_d ← 1 ∈ ℝ^{1×1×1}
for k = d, d−1, ..., 2 do
    R_{k−1} ← EINSUM("ra n ra', rb n rb', s n s', ra' rb' s' -> ra rb s",
                     T_{A,k}, T_{B,k}, T_{S,k}, R_k)
end for
return {R_k}_{k=1}^{d−1}
```

Algorithm 3 Hadamard HATT

Require: TTs $\boldsymbol{\mathcal{A}}, \boldsymbol{\mathcal{B}}$; max rank R; tolerance ϵ; sketch width $s = R + r_{\text{oversample}}$; (optional) initial guess TT $\boldsymbol{\mathcal{G}}$.
Ensure: TT approximation $\tilde{\boldsymbol{\mathcal{X}}} \approx \boldsymbol{\mathcal{A}} * \boldsymbol{\mathcal{B}}$.

```
A ← RIGHTORTHOGONALIZE(A)
B ← RIGHTORTHOGONALIZE(B)
S ← G if provided, else RANDOMTT(dims, s)
S ← LEFTORTHOGONALIZE(S)
{R_k}_{k=1}^{d−1} ← SKETCHEDRIGHTENVS(A, B, S)          ▷ Algorithm 2
R_d ← 1 ∈ ℝ^{1×1×1}
L ← 1 ∈ ℝ^{1×1×1}                                      ▷ initial left environment
for k = 1, ..., d do
    M ← EINSUM("r ra rb, ra n ra', rb n rb', ra' rb' s' -> r n s'",
               L, T_{A,k}, T_{B,k}, R_k)
    if k < d then
        M ← REARRANGE("r n s' -> (r n) s'", M)
        U, ·, · ← TRUNCATEDSVD(M; ε, R)
        T_{X̃,k} ← REARRANGE("(r n) r' -> r n r'", U)
        L ← EINSUM("r n r', r ra rb, ra n ra', rb n rb' -> r' ra' rb'",
                   T_{X̃,k}, L, T_{A,k}, T_{B,k})
    else
        T_{X̃,d} ← M                                    ▷ s_d = 1, so M already has shape r_{d−1} × n_d × 1
    end if
end for
return TT X̃ with cores {T_{X̃,k}}_{k=1}^{d}.
```

overall time becomes $\mathcal{O}(dnr^3)$.

H.3. TCI

Tensor Cross Interpolation (TCI) (Oseledets & Tyrtyshnikov, 2010; Núñez Fernández et al., 2025) approximates the Hadamard product $\boldsymbol{\mathcal{X}} = \boldsymbol{\mathcal{A}} * \boldsymbol{\mathcal{B}}$ by a TT $\tilde{\boldsymbol{\mathcal{X}}}$ that interpolates $\boldsymbol{\mathcal{X}}$ at a small set of carefully chosen entries. Unlike TT-ROUND and HATT, TCI never materializes a Kronecker core nor performs an SVD on a full unfolding of $\boldsymbol{\mathcal{X}}$: at each site, a small *supercore* is formed by contracting a w-site window of the input cores with their interpolated environments, and an interpolative decomposition picks the rows or columns of that supercore that best interpolate it. We present the algorithm with two-site sweeps ($w = 2$); the generalization to $K > 2$ factors and larger windows is straightforward.

Pivots and shared environments. Let $p \leq R$ denote the current number of pivots at a given bond. TCI maintains per-factor environments that share the pivot axis across factors:

- Left environments $\boldsymbol{\mathcal{L}}_i^{\boldsymbol{\mathcal{A}}} \in \mathbb{R}^{p \times r_{i-1}^A}$ and $\boldsymbol{\mathcal{L}}_i^{\boldsymbol{\mathcal{B}}} \in \mathbb{R}^{p \times r_{i-1}^B}$ holding p contracted slices of the left tail of $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ respectively.

Algorithm 4 Maxvol right environments for $\mathcal{X} = \mathcal{A} * \mathcal{B}$

Require: TTs $\mathcal{A}, \mathcal{B}$; max rank R.
Ensure: Right environments $\{\mathcal{R}_i^{\mathcal{A}}, \mathcal{R}_i^{\mathcal{B}}\}_{i=1}^{d-1}$.
$\mathcal{R}_d^{\mathcal{A}} \leftarrow \mathbf{1} \in \mathbb{R}^{1\times 1}$; $\mathcal{R}_d^{\mathcal{B}} \leftarrow \mathbf{1} \in \mathbb{R}^{1\times 1}$
for $i = d, d-1, \ldots, 2$ **do**
  for $\mathcal{Y} \in \{\mathcal{A}, \mathcal{B}\}$ **do**
    $\mathbf{\Phi}_{\mathcal{Y}} \leftarrow$ EINSUM(`"r n r', r' p -> r (n p)"`, $\mathcal{T}_{\mathcal{Y},i}$, $\mathcal{R}_i^{\mathcal{Y}}$)
  end for
  $\mathbf{\Phi} \leftarrow$ EINSUM(`"ra q, rb q -> (ra rb) q"`, $\mathbf{\Phi}_{\mathcal{A}}$, $\mathbf{\Phi}_{\mathcal{B}}$)
  pivot $\leftarrow$ MAXVOL($\mathbf{\Phi}$; R)
  for $\mathcal{Y} \in \{\mathcal{A}, \mathcal{B}\}$ **do**
    $\mathcal{R}_{i-1}^{\mathcal{Y}} \leftarrow \mathbf{\Phi}_{\mathcal{Y}}[:, \text{pivot}]$
  end for
end for
return $\{\mathcal{R}_i^{\mathcal{A}}, \mathcal{R}_i^{\mathcal{B}}\}_{i=1}^{d-1}$

- Right environments $\mathcal{R}_i^{\mathcal{A}} \in \mathbb{R}^{r_i^A \times p}$ and $\mathcal{R}_i^{\mathcal{B}} \in \mathbb{R}^{r_i^B \times p}$ holding p contracted slices of the right tails.

The shared pivot axis encodes the Hadamard product structure without ever forming Kronecker cores: at each bond, both factors are interpolated at the *same* set of physical-pivot indices.

Interpolative primitives. We use two primitives for selecting pivots and interpolating from them.

- MAXVOL (Goreinov et al.): given a matrix $\mathbf{M}$, MAXVOL($\mathbf{M}$; R) returns up to R column indices of $\mathbf{M}$ that form a near-maximum-volume submatrix.
- PRRLDU (Savostyanov, 2014): a partially rank-revealing LU decomposition. Its row-pivoted variant $\text{PRRLDU}_{\text{row}}(\mathbf{M}; \epsilon, R)$ returns an interpolation matrix $\mathcal{Z}$ and a pivot set such that $\mathbf{M} \approx \mathcal{Z} \cdot \mathbf{M}[\text{pivot}, :]$. The column-pivoted variant $\text{PRRLDU}_{\text{col}}$ analogously yields $\mathbf{M} \approx \mathbf{M}[:, \text{pivot}] \cdot \mathcal{Z}$.

Phase 1: Maxvol warm-up. A right-to-left sweep (Algorithm 4) initializes the right environments. At each site, per-factor supercores $\mathbf{\Phi}_{\mathcal{A}}, \mathbf{\Phi}_{\mathcal{B}}$ are formed by contracting the current right environment with the corresponding core; their joint Hadamard matrix $\mathbf{\Phi}$ has rows that range over the outer product of the per-factor left bonds and columns that range over the shared physical-pivot indices. MAXVOL selects up to R such columns, and the picked columns of each $\mathbf{\Phi}_{\mathcal{A}}, \mathbf{\Phi}_{\mathcal{B}}$ become the next right environments.

Phase 2: Alternating sweeps. Sweeps of Algorithm 5 (L2R) and Algorithm 6 (R2L) refine the pivots and emit the cores of $\tilde{\mathcal{X}}$. At each interior site of an L2R sweep, per-factor supercores are formed by contracting the left environment, two consecutive cores, and the right environment. They are combined element-wise into the joint supercore $\mathbf{\Phi}$, flattened into a matrix, and decomposed by $\text{PRRLDU}_{\text{row}}$ into an interpolation matrix $\mathcal{Z}$ and a row pivot. The interpolation matrix becomes the new core $\mathcal{T}_{\tilde{\mathcal{X}},i}$, and the pivot updates the left environments of $\mathcal{A}$ and $\mathcal{B}$. The R2L sweep is symmetric, processing sites from right to left and using $\text{PRRLDU}_{\text{col}}$ to update right environments.

Boundary cores. The L2R sweep additionally extracts the rightmost core $\mathcal{T}_{\tilde{\mathcal{X}},d}$ directly from the boundary supercore at $i = d-1$ (whose right rank is already $r_d = 1$). The R2L sweep mirrors this for the leftmost core.

Complexity. At each interior site, the dominant cost is forming the joint supercore $\mathbf{\Phi}$ and applying PRRLDU to its $(p \cdot n) \times (n \cdot p)$ unfolding. The per-factor four-tensor contractions cost $\mathcal{O}(n^2 p r^2)$, the elementwise Hadamard costs $\mathcal{O}(n^2 p^2)$, and PRRLDU costs $\mathcal{O}(n^2 p^3)$ in the worst case. With $p \leq R \approx r$, the per-site cost is $\mathcal{O}(n^2 r^3)$. Summing over the d sites and a constant number of sweeps gives an overall time complexity of $\mathcal{O}(d n^2 r^3)$ and memory $\mathcal{O}(n^2 r^2)$, dominated by $\mathbf{\Phi}$. The maxvol warm-up has lower per-site cost, $\mathcal{O}(n r^3)$, and is dominated by the sweep phase.

H.4. RSI

Randomized Sketching with Interpolation (RSI) (Meng et al., 2026) replaces TCI's maxvol-based right environment construction with random sketching. Like HATT, RSI sketches each input TT independently rather than evaluating exact

Algorithm 5 Left-to-right TCI sweep

Require: TTs $\boldsymbol{\mathcal{A}}, \boldsymbol{\mathcal{B}}$; left envs $\{\boldsymbol{\mathcal{L}}_i^{\boldsymbol{\mathcal{A}}}, \boldsymbol{\mathcal{L}}_i^{\boldsymbol{\mathcal{B}}}\}_{i=1}^{d-1}$; right envs $\{\boldsymbol{\mathcal{R}}_i^{\boldsymbol{\mathcal{A}}}, \boldsymbol{\mathcal{R}}_i^{\boldsymbol{\mathcal{B}}}\}_{i=1}^{d}$; max rank R; tolerance ϵ.
Ensure: Cores $\{\boldsymbol{\mathcal{T}}_{\tilde{\boldsymbol{\mathcal{X}}},i}\}_{i=1}^{d}$ and updated left envs.

$\boldsymbol{\mathcal{L}}_1^{\boldsymbol{\mathcal{A}}} \leftarrow \mathbf{1} \in \mathbb{R}^{1\times 1}$; $\boldsymbol{\mathcal{L}}_1^{\boldsymbol{\mathcal{B}}} \leftarrow \mathbf{1} \in \mathbb{R}^{1\times 1}$
for $i = 1, \ldots, d-1$ **do**
  for $\boldsymbol{\mathcal{Y}} \in \{\boldsymbol{\mathcal{A}}, \boldsymbol{\mathcal{B}}\}$ **do**
    $\boldsymbol{\Phi}_{\boldsymbol{\mathcal{Y}}} \leftarrow \textsc{einsum}\big($`"p r, r n r', r' m r'', r'' q -> p n m q",` $\boldsymbol{\mathcal{L}}_i^{\boldsymbol{\mathcal{Y}}},\ \boldsymbol{\mathcal{T}}_{\boldsymbol{\mathcal{Y}},i},\ \boldsymbol{\mathcal{T}}_{\boldsymbol{\mathcal{Y}},i+1},\ \boldsymbol{\mathcal{R}}_{i+1}^{\boldsymbol{\mathcal{Y}}}\big)$
  end for
  $\boldsymbol{\Phi} \leftarrow \boldsymbol{\Phi}_{\boldsymbol{\mathcal{A}}} * \boldsymbol{\Phi}_{\boldsymbol{\mathcal{B}}}$ ▷ element-wise Hadamard
  $\boldsymbol{\Phi} \leftarrow \textsc{rearrange}\big($`"p n m q -> (p n) (m q)",` $\boldsymbol{\Phi}\big)$
  $(\boldsymbol{\mathcal{Z}},\ \text{pivot}) \leftarrow \textsc{prrLDU}_{\text{row}}(\boldsymbol{\Phi};\ \epsilon,\ R)$
  $\boldsymbol{\mathcal{T}}_{\tilde{\boldsymbol{\mathcal{X}}},i} \leftarrow \textsc{rearrange}\big($`"(p n) r' -> p n r'",` $\boldsymbol{\mathcal{Z}}\big)$
  for $\boldsymbol{\mathcal{Y}} \in \{\boldsymbol{\mathcal{A}}, \boldsymbol{\mathcal{B}}\}$ **do**
    $\boldsymbol{\mathcal{L}}_{i+1}^{\boldsymbol{\mathcal{Y}}} \leftarrow \textsc{rearrange}\big(\textsc{einsum}\big($`"p r, r n r' -> (p n) r'",` $\boldsymbol{\mathcal{L}}_i^{\boldsymbol{\mathcal{Y}}},\ \boldsymbol{\mathcal{T}}_{\boldsymbol{\mathcal{Y}},i}\big)\big)[\text{pivot}, :]$
  end for
  if $i = d-1$ **then**
    $\boldsymbol{\mathcal{T}}_{\tilde{\boldsymbol{\mathcal{X}}},d} \leftarrow \textsc{rearrange}\big($`"r' n -> r' n 1",` $\boldsymbol{\Phi}[\text{pivot}, :]\big)$
  end if
end for
return cores $\{\boldsymbol{\mathcal{T}}_{\tilde{\boldsymbol{\mathcal{X}}},i}\}_{i=1}^{d}$, updated $\{\boldsymbol{\mathcal{L}}_i^{\boldsymbol{\mathcal{A}}}, \boldsymbol{\mathcal{L}}_i^{\boldsymbol{\mathcal{B}}}\}_{i=1}^{d}$

entries; like TCI, it then performs interpolative-decomposition sweeps to construct $\tilde{\boldsymbol{\mathcal{X}}}$. Compared to HATT, RSI differs in two ways: (i) it uses a Khatri-Rao product (KRP) sketch rather than a TT sketch, and (ii) it uses interpolative decompositions (prrLDU) rather than SVDs to truncate at each site. Because the sketching is per-factor, RSI side-steps the combinatorial maxvol search of TCI (which is specific to the Hadamard product) and immediately extends to any non-linear operation over multiple TTs.

The original RSI formulation uses a single L2R sweep after sketching. We extend it to alternating L2R/R2L sweeps, which follow naturally from the TCI connection: once a first L2R sweep produces interpolative left environments, subsequent sweeps refine the pivots in both directions exactly as in TCI.

KRP sketches. The original RSI paper uses a Khatri-Rao product sketch: a set of d matrices $\{\boldsymbol{\mathcal{S}}_i \in \mathbb{R}^{n_i \times s}\}_{i=1}^{d}$ with i.i.d. random entries (e.g., Gaussian or Rademacher), where $s = R + r_{\text{oversample}}$ is the sketch width. Unlike the TT sketch used in HATT— whose sketch cores have an internal bond structure $\boldsymbol{\mathcal{T}}_{\boldsymbol{\mathcal{S}},k} \in \mathbb{R}^{s_{k-1}\times n_k \times s_k}$ — the KRP sketch has a single "flat" sketch axis s shared across all d sites. This makes the contractions in the right-env sweep structurally simpler.

Phase 1: Sketched right environments. A right-to-left sweep (Algorithm 8) computes the per-TT right environments $\boldsymbol{\mathcal{R}}_i^{\boldsymbol{\mathcal{Y}}} \in \mathbb{R}^{r_i^Y \times s}$ for $\boldsymbol{\mathcal{Y}} \in \{\boldsymbol{\mathcal{A}}, \boldsymbol{\mathcal{B}}\}$. At each site, the previous right environment is updated by contracting the core with the per-site sketch matrix on the physical dimension and broadcasting on the shared sketch axis.

Phase 2: Alternating sweeps. The L2R and R2L sweeps of TCI (Algorithms 5 and 6) are reused unchanged: the sketched right environments play the same role in the supercore contractions as TCI's maxvol-selected ones. After the first L2R sweep, the left environments become interpolative (subset-based) just as in TCI; from then on, the algorithm is identical to TCI. The original RSI corresponds to stopping after this first L2R sweep, before any R2L.

Complexity. Forming the KRP-sketched right environments costs $\mathcal{O}(dnr^2s)$ per TT, dominated by the per-site einsum. The subsequent TCI sweeps inherit the per-sweep complexity of $\mathcal{O}(dn^2r^3)$ in time and $\mathcal{O}(n^2r^2)$ in memory. With $s \approx r$ and a constant number of sweeps, the overall cost matches TCI: $\mathcal{O}(dn^2r^3)$ time, $\mathcal{O}(n^2r^2)$ memory. The savings over TCI come not from asymptotic complexity but from avoiding the combinatorial maxvol search in the warm-up; the savings over HATT come from per-site sketches replacing TT sketches in the right-env sweep.

Algorithm 6 Right-to-left TCI sweep

Require: TTs $\boldsymbol{\mathcal{A}}, \boldsymbol{\mathcal{B}}$; left envs $\{\boldsymbol{\mathcal{L}}_i^{\boldsymbol{\mathcal{A}}}, \boldsymbol{\mathcal{L}}_i^{\boldsymbol{\mathcal{B}}}\}_{i=1}^{d}$; right envs $\{\boldsymbol{\mathcal{R}}_i^{\boldsymbol{\mathcal{A}}}, \boldsymbol{\mathcal{R}}_i^{\boldsymbol{\mathcal{B}}}\}_{i=1}^{d-1}$; max rank R; tolerance ϵ.
Ensure: Cores $\{\boldsymbol{\mathcal{T}}_{\tilde{\boldsymbol{\mathcal{X}}},i}\}_{i=1}^{d}$ and updated right envs.

$\boldsymbol{\mathcal{R}}_d^{\boldsymbol{\mathcal{A}}} \leftarrow \mathbf{1}$; $\boldsymbol{\mathcal{R}}_d^{\boldsymbol{\mathcal{B}}} \leftarrow \mathbf{1}$
for $i = d, d-1, \ldots, 2$ **do**
  for $\boldsymbol{\mathcal{Y}} \in \{\boldsymbol{\mathcal{A}}, \boldsymbol{\mathcal{B}}\}$ **do**
    $\boldsymbol{\Phi}_{\boldsymbol{\mathcal{Y}}} \leftarrow$ EINSUM("p r, r n r', r' m r'', r'' q -> p n m q", $\boldsymbol{\mathcal{L}}_{i-1}^{\boldsymbol{\mathcal{Y}}}$, $\boldsymbol{\mathcal{T}}_{\boldsymbol{\mathcal{Y}},i-1}$, $\boldsymbol{\mathcal{T}}_{\boldsymbol{\mathcal{Y}},i}$, $\boldsymbol{\mathcal{R}}_i^{\boldsymbol{\mathcal{Y}}}$)
  end for
  $\boldsymbol{\Phi} \leftarrow \boldsymbol{\Phi}_{\boldsymbol{\mathcal{A}}} * \boldsymbol{\Phi}_{\boldsymbol{\mathcal{B}}}$
  $\boldsymbol{\Phi} \leftarrow$ REARRANGE("p n m q -> (p n) (m q)", $\boldsymbol{\Phi}$)
  $(\boldsymbol{\mathcal{Z}}, \text{pivot}) \leftarrow \text{PRRLDU}_{\text{col}}(\boldsymbol{\Phi}; \epsilon, R)$
  $\boldsymbol{\mathcal{T}}_{\tilde{\boldsymbol{\mathcal{X}}},i} \leftarrow$ REARRANGE("r' (m q) -> r' m q", $\boldsymbol{\mathcal{Z}}$)
  for $\boldsymbol{\mathcal{Y}} \in \{\boldsymbol{\mathcal{A}}, \boldsymbol{\mathcal{B}}\}$ **do**
    $\boldsymbol{\mathcal{R}}_{i-1}^{\boldsymbol{\mathcal{Y}}} \leftarrow$ REARRANGE(EINSUM("r m r', r' q -> r (m q)", $\boldsymbol{\mathcal{T}}_{\boldsymbol{\mathcal{Y}},i}$, $\boldsymbol{\mathcal{R}}_i^{\boldsymbol{\mathcal{Y}}}$))[:, pivot]
  end for
  if $i = 2$ **then**
    $\boldsymbol{\mathcal{T}}_{\tilde{\boldsymbol{\mathcal{X}}},1} \leftarrow$ REARRANGE("n r' -> 1 n r'", $\boldsymbol{\Phi}$[:, pivot])
  end if
end for
return cores $\{\boldsymbol{\mathcal{T}}_{\tilde{\boldsymbol{\mathcal{X}}},i}\}_{i=1}^{d}$, updated $\{\boldsymbol{\mathcal{R}}_i^{\boldsymbol{\mathcal{A}}}, \boldsymbol{\mathcal{R}}_i^{\boldsymbol{\mathcal{B}}}\}_{i=1}^{d}$

Algorithm 7 TCI for $\boldsymbol{\mathcal{X}} = \boldsymbol{\mathcal{A}} * \boldsymbol{\mathcal{B}}$

Require: TTs $\boldsymbol{\mathcal{A}}, \boldsymbol{\mathcal{B}}$; max rank R; tolerance ϵ; number of sweeps T.
Ensure: TT approximation $\tilde{\boldsymbol{\mathcal{X}}} \approx \boldsymbol{\mathcal{A}} * \boldsymbol{\mathcal{B}}$.

$\boldsymbol{\mathcal{A}} \leftarrow$ RIGHTORTHOGONALIZE($\boldsymbol{\mathcal{A}}$); $\boldsymbol{\mathcal{B}} \leftarrow$ RIGHTORTHOGONALIZE($\boldsymbol{\mathcal{B}}$)
$\{\boldsymbol{\mathcal{R}}_i^{\boldsymbol{\mathcal{A}}}, \boldsymbol{\mathcal{R}}_i^{\boldsymbol{\mathcal{B}}}\} \leftarrow$ MAXVOLRIGHTENVS($\boldsymbol{\mathcal{A}}, \boldsymbol{\mathcal{B}}; R$) ▷ Algorithm 4
for $t = 1, \ldots, T$ **do**
  (cores, $\{\boldsymbol{\mathcal{L}}_i^{\boldsymbol{\mathcal{A}}}, \boldsymbol{\mathcal{L}}_i^{\boldsymbol{\mathcal{B}}}\}$) $\leftarrow$ L2R-TCI($\boldsymbol{\mathcal{A}}, \boldsymbol{\mathcal{B}}, \{\boldsymbol{\mathcal{L}}\}, \{\boldsymbol{\mathcal{R}}\}; R, \epsilon$) ▷ Algorithm 5
  (cores, $\{\boldsymbol{\mathcal{R}}_i^{\boldsymbol{\mathcal{A}}}, \boldsymbol{\mathcal{R}}_i^{\boldsymbol{\mathcal{B}}}\}$) $\leftarrow$ R2L-TCI($\boldsymbol{\mathcal{A}}, \boldsymbol{\mathcal{B}}, \{\boldsymbol{\mathcal{L}}\}, \{\boldsymbol{\mathcal{R}}\}; R, \epsilon$) ▷ Algorithm 6
end for
return TT $\tilde{\boldsymbol{\mathcal{X}}}$ with cores cores

Algorithm 8 KRP-sketched right environments for a TT $\boldsymbol{\mathcal{A}}$

Require: TT cores $\{\boldsymbol{\mathcal{T}}_{\boldsymbol{\mathcal{A}},i}\}_{i=1}^{d}$; KRP sketch matrices $\{\boldsymbol{\mathcal{S}}_i \in \mathbb{R}^{n_i \times s}\}_{i=1}^{d}$.
Ensure: Right environments $\{\boldsymbol{\mathcal{R}}_i^{\boldsymbol{\mathcal{A}}}\}_{i=1}^{d-1}$.

$\boldsymbol{\mathcal{R}}_d^{\boldsymbol{\mathcal{A}}} \leftarrow \mathbf{1} \in \mathbb{R}^{1 \times s}$ ▷ all-ones vector along the sketch axis
for $i = d, d-1, \ldots, 2$ **do**
  $\boldsymbol{\mathcal{R}}_{i-1}^{\boldsymbol{\mathcal{A}}} \leftarrow$ EINSUM("r n r', n s, r' s -> r s", $\boldsymbol{\mathcal{T}}_{\boldsymbol{\mathcal{A}},i}$, $\boldsymbol{\mathcal{S}}_i$, $\boldsymbol{\mathcal{R}}_i^{\boldsymbol{\mathcal{A}}}$)
end for
return $\{\boldsymbol{\mathcal{R}}_i^{\boldsymbol{\mathcal{A}}}\}_{i=1}^{d-1}$

Algorithm 9 RSI for $\boldsymbol{\mathcal{X}} = \boldsymbol{\mathcal{A}} * \boldsymbol{\mathcal{B}}$

Require: TTs $\boldsymbol{\mathcal{A}}, \boldsymbol{\mathcal{B}}$; max rank R; tolerance ϵ; sketch width $s = R + r_{\text{oversample}}$; number of sweeps $T \geq 1$.
Ensure: TT approximation $\tilde{\boldsymbol{\mathcal{X}}} \approx \boldsymbol{\mathcal{A}} * \boldsymbol{\mathcal{B}}$.

```
A ← RightOrthogonalize(A);   B ← RightOrthogonalize(B)
Draw KRP sketches {S_i ∈ R^{n_i × s}}_{i=1}^d with i.i.d. random entries
{R_i^A} ← KRP-RightEnvs(A, {S_i})                                   ▷ Algorithm 8
{R_i^B} ← KRP-RightEnvs(B, {S_i})
for t = 1, ..., T do
    (cores, {L_i^A, L_i^B}) ← L2R-TCI(A, B, {L}, {R}; R, ε)          ▷ Algorithm 5
    (cores, {R_i^A, R_i^B}) ← R2L-TCI(A, B, {L}, {R}; R, ε)          ▷ Algorithm 6; original RSI omits this and stops after the first L2R
end for
return TT X̃ with cores cores
```