# Unifying and Understanding Overparameterized Circuit Representations via Low-Rank Tensor Decompositions

Antonio Mari[1] Gennaro Vessio[1] Antonio Vergari[2]

[1]Computer Science Dept., University of Bari Aldo Moro, Italy
[2]School of Informatics, University of Edinburgh, UK

## Abstract

Tensorizing probabilistic circuits (PCs) – structured computational graphs capable of efficiently and accurately performing various probabilistic reasoning tasks – is the go-to way to represent and learn these models. This paper systematically explores the architectural options employed in modern overparameterized PCs, namely RAT-SPNs, EiNets, and HCLTs, and unifies them into a single algorithmic framework. By trying to compress the existing overparameterized layers via low-rank decompositions, we discover alternative parameterizations that possess the same expressive power but are computationally more efficient. This emphasizes the possibility of "mixing & matching" different design choices to create new PCs and helps to disentangle the few ones that really matter.

## 1 INTRODUCTION

Probabilistic circuits (PCs) [Vergari et al., 2020, Choi et al., 2020] have recently emerged as a framework to unify several, apparently different, tractable probabilistic models (TPMs). To this end, TPMs are represented as structured computational graphs that include three kinds of units: input, sum, and product units. Input units represent simple, tractable distributions defined over a subset of the variables of interest, while sum (or product) units compute a weighted average (or product) of their inputs. In this unified framework, one can abstract from the many syntactic formalisms of TPMs and focus only on the structural properties of these computational graphs that are crucial for performing specific reasoning tasks exactly and efficiently. E.g., to guarantee the tractability of marginalization, a TPM cast as a circuit only requires *smoothness* and *decomposability* [Choi et al., 2020, Darwiche, 2009] as structural properties, or *compatibility* and *structured-decomposability* to compute powers and other divergences tractably [Vergari et al., 2021].

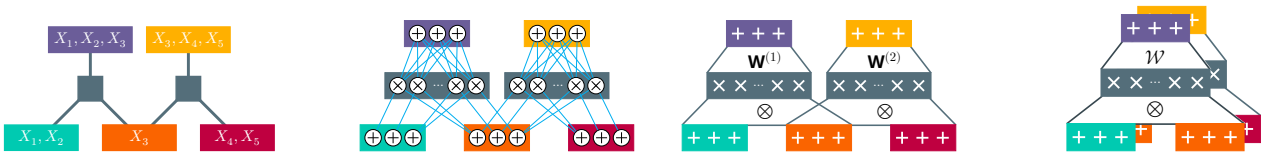
Table 1: **Destructuring modern PC architectures** into smaller design choices for building region graphs, layer parameterization, and folding (see Fig. 1) that can be mixed & matched with new ones (see Sections 2 and 3).

| PC | Region Graph | Layer | Fold |
|---|---|---|---|
| RAT-SPN | RND | Tucker | ✗ |
| EiNet | PD/RND | Tucker | ✓ |
| HCLT | CLT | CP | ✗ |
| Mix & Match | {RND, QG, PD, CLT} | × {Tucker, CP, CP-s} | ✓ |

The most effective way to learn PCs from data requires building *overparameterized* circuits, comprising millions or even billions of parameters [Liu et al., 2022], and training these parameters by SGD or EM [Peharz et al., 2016, 2020b]. This approach has been popularized by modern circuit architectures such as RAT-SPNs [Peharz et al., 2020b], Einsum networks (EiNets) [Peharz et al., 2020a], and hidden Chow-Liu trees (HCLTs) [Liu and Van den Broeck, 2021]. Algorithms for learning these circuits have been proposed from different perspectives, seemingly with different recipes, and in practice, PCs learned in these ways are treated as different models. However, a closer look reveals that they can be understood under a single framework, within which only a few algorithmic choices truly influence circuit performance and learnability.

In this paper, we systematize the process of representing and learning overparameterized circuits as a pipeline comprising three macro stages (Fig. 1): **I)** building a general skeleton that guarantees (structured-)decomposability, also called *region graph*; **II)** *overparameterizing* the skeleton with sum and product units and *vectorizing* their computations, and **III)** optionally *folding* such a computational graph into a denser tensor representation. This opens up

*Accepted for the 6th Workshop on Tractable Probabilistic Modeling at UAI* (TPM 2023).




**I.** Build a region graph **II.** Overparameterize & Vectorize **III.** Folding

Figure 1: **The general recipe for building overparameterized circuits** consists of (**I**) building a hierarchical scope partitioning or region graph (RG) to ensure decomposability, then (**II**) overparameterizing the RG, e.g., by using TUCKER (see Section 2) or (collapsed) CPlayers (see Section 3), and (**III**) optionally folding the graph to reduce time/memory consumption during execution. Finally, parameters can be learned via EM or SGD variants such as Adam.

the possibility of "mixing & matching" existing algorithmic choices to realize stages **I**-**III** and easily design novel circuit architectures (Table 1). Furthermore, we analyze how the different layer parameterizations (**II**) relate to each other via low-rank tensor decompositions. In this light, we can connect existing parameterizations (Section 2) and, at the same time, discover alternative parameterizations that can be more parameter-efficient (Section 3). Finally, we evaluate the performance of several combinations in this "architectural spectrum", highlighting the trade-offs in terms of time and space complexity and accuracy (Section 4). This helps shed light on what key algorithmic factors underlie the performance boosts recently reported in the circuit literature [Dang et al., 2022].

## 2 REPRESENTING AND LEARNING OVERPARAMETERIZED CIRCUITS

**Notation.** We follow Vergari et al. [2019] in representing tensorized PCs[1] as structured neural networks whose units can be grouped into *layers*, i.e., computational abstractions that can be better mapped to modern deep learning frameworks. We denote a vectorized layer of a PC as $\mathbf{o}, \boldsymbol{\ell}, \mathbf{r} \in \mathbb{R}^K$, consisting of $K$ (sum, product or input) units, all defined over a set of random variables (RVs) $\mathbf{X}$, also called its *scope*. We denote the $k$-th unit as, e.g., $o_k$. In the following, we will assume that PCs are smooth and decomposable. For a formal definition of circuits and these structural properties, see Choi et al. [2020] and Appendix A.

**Building region graphs.** Figure 1 illustrates our proposed abstract pipeline to represent and learn tensorized PCs. The first step is to build a *hierarchical scope partitioning* [Vergari et al., 2021], also called *region graph* (RG) [Dennis and Ventura, 2012]. Intuitively, building an RG is equivalent to recursively partitioning a set of RVs $\mathbf{X}$ into disjoint scopes, thus providing a skeleton for smooth and decomposable circuits. More formally, an RG over RVs $\mathbf{X}$ is a bipartite graph whose nodes are either *regions*, denoting subsets of $\mathbf{X}$, or *partitions*, denoting how to split a region into a set of disjoint sub-regions. W.l.o.g., we consider RGs that only partition a region into two, i.e., $\mathcal{R} = \mathcal{R}' \cup \mathcal{R}''$, with $\mathcal{R}' \cap \mathcal{R}'' = \varnothing$. We denote an RG by its root region $\mathcal{R} = \mathbf{X}$.

The simplest domain-agnostic way to build an RG is to randomly create partitions, as proposed by RAT-SPNs [Peharz et al., 2020b], a strategy we denote by RND. Alternatively, one can mechanically partition pixels $\mathbf{X}$ for specific domains such as images by "cutting" the image along its axis, as proposed in Poon and Domingos [2011], denoted as the PD architecture. Finally, one can learn to partition RVs, as implicitly done by HCLTs [Liu and Van den Broeck, 2021], which first learn a latent tree model [Choi et al., 2011] whose RV ordering is given heuristically by the Chow-Liu algorithm (CLT) [Chow and Liu, 1968], and then compile it into a PC. We show in Appendix B.1 that this construction induces an RG over $\mathbf{X}$, and thus that HCLTs can be cast within our 3-stage construction as well, even if they are motivated from a compilation perspective. If an RG splits each region in a single partition, as the CLT construction does, the resulting PC will be not only decomposable but also structured-decomposable (see Appendices A and B.1).

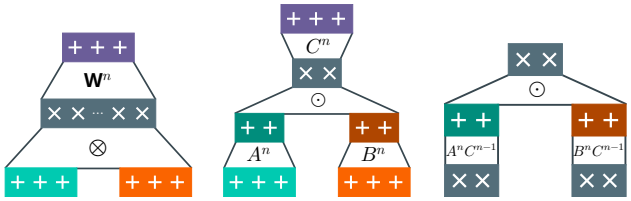
**Overparameterize & vectorize.** An RG is then "populated" by sum, product, and input units, a process that we denote as *overparameterization* since more than a single unit is associated with a region or partition. Units associated with the same region or partition, which share the same scope, can then be *vectorized* in layers (Fig. 1, **II**), thus becoming amenable to tensor-based computations. Leaf regions, i.e., regions that are not further partitioned, are associated input units, while sum and products populate inner regions and partitions.

RAT-SPNs associate $K$ sum (resp. input) units with each inner (resp. leaf) region in an RG and introduce $K^2$ products in a partition to realize a vector cross-product. Therefore every partition $\mathcal{R}_o = (\mathcal{R}_l, \mathcal{R}_r)$ identifies three vectors $\mathbf{o}, \boldsymbol{\ell}, \mathbf{r} \in \mathbb{R}^K$. The $k$-th unit of $\mathbf{o}$ computes:

$$o_k = \boldsymbol{\ell}^\top W^{(k)} \mathbf{r}, \qquad \text{(TUCKER)}$$

where $W^{(k)}$ is the $k$-th slice of the tensor $\mathbf{W} \in \mathbb{R}_+^{K \times K \times K}$ that contains the non-negative parameters of the layer. We

[1] Our analysis can be extended to non-monotonic PCs and non-probabilistic circuits [Vergari et al., 2021], i.e., circuits with non-positive parameters and non-positive outputs, relaxing the non-negativity constraint in the tensor factorization considered.




(a) TUCKER layer (b) CP factorization (c) Collapsing $C^n$

Figure 2: **From TUCKER to CP layers.** Layers in HCLTs can be interpreted as collapsed (c) low-rank approximations (b) of layers in RAT-SPNs and EiNets (a). They can also be folded and used with other RGs.

name this parameterization TUCKER layer as it takes the same form of a non-negative TUCKER tensor factorization [Tucker, 1966].[2] As such, $K$ can be thought of as the rank of an implicit factorization. Intuitively, the larger $K$, the greater the number of parameters in $\mathbf{W}$ and the greater the expressiveness of PCs whose layers are parameterized as Eq. (TUCKER). TUCKER layers also appear in overparameterized logical circuits [Ahmed et al., 2022].

**Folding.** The optional and final step in representing an overparameterized and tensorized PC is to group many layers that share the same functional form and "stack", or "fold" them, into a denser tensor representation. This is the core ingredient of the additional speed-up introduced by EiNets [Peharz et al., 2020a]. A folded layer in an EiNet stacks $F$ TUCKER layers, is parameterized by $\boldsymbol{\mathcal{W}} \in \mathbb{R}_+^{K\times K\times K\times F}$, and outputs a matrix $O \in \mathbb{R}^{K\times F}$ whose $(k, f)$-entry is defined as

$$O_{k,f} = (\boldsymbol{\ell}^{(f)})^\top W^{(k,f)} \mathbf{r}^{(f)} \qquad \text{(TUCKER-f)}$$

where $W^{(k,f)}$ is the $(k, f)$-slice of $\boldsymbol{\mathcal{W}}$ and $\boldsymbol{\ell}^{(f)}$ (resp. $\mathbf{r}^{(f)}$) are the $f$-th column of the matrix $L \in \mathbb{R}^{K\times F}$ (resp. $R \in \mathbb{R}^{K\times F}$) that stacks the input vectors for the left (resp. right) input regions. Note that this parameterization, which we denote as TUCKER-f, essentially encodes the same circuit semantics as Eq. (TUCKER), and thus the same expressiveness. Consequently, neither the folding nor the layer parameterization of RAT-SPNs and EiNets can be responsible for the difference in performance that is usually reported between these models [Liu et al., 2022]. The key factor must lie in the choice of RG used (as we will confirm in Section 4) or a discrepancy in other hyperparameters used to learn these models, e.g., the optimizer employed. We consistently use Adam in our experiments (Section 4), refuting the common belief that SGD-based optimizers are worse than EM.

[2] Specifically it takes the form of a TUCKER2 factorization [Kolda and Bader, 2009]. A PC such as a RAT-SPN or EiNet over RVs $\mathbf{X}$ can thus be viewed as performing a sophisticated hierarchical non-negative TUCKER2 decomposition of a probability (quasi-)tensor encoding $p(\mathbf{X})$ [Grasedyck, 2010].

## 3 FROM TUCKER TO CP LAYERS

Although expressive, TUCKER layers require learning and storing $K^3$ parameters. In light of the need for more parameter efficient parameterizations, we propose to compress the parameter tensor $\mathbf{W}$ of TUCKER layers via a non-negative CANDECOMP-PARAFAC (CP) [Cichocki et al., 2007] low-rank decomposition. I.e., given a rank $R \leq K$ we assume that each entry of the tensor $\mathbf{W}^n$ that parameterizes the $n$-th layer in a PC can be computed as $\mathbf{W}^n_{i,j,k} = \sum_{r=1}^R A^n_{ir} B^n_{jr} C^n_{kr}$ where $A^n, B^n, C^n \in \mathbb{R}_+^{K\times R}$ encode the new circuit parameters. By re-arranging matrix multiplications, it is easy to show that such a layer would compute:

$$\mathbf{o} = C^n(A^n\boldsymbol{\ell}) \odot (B^n\mathbf{r}) \qquad (R\text{-rank CP})$$

where $\odot$ denotes the element-wise product. This new parameterization via low-rank tensors, illustrated in Fig. 2, requires only $3KR$ parameters and thus allows for faster inference (see Table 3). Note that we can effectively collapse the two adjacent sum layers parameterized by $C^{n-1}$ and $A^n$ (or $B^n$) since a combination of linear operations is a linear operation. In this way, the following simplified layer parameterization is obtained:

$$\mathbf{o} = (A'\boldsymbol{\ell}) \odot (B'\mathbf{r}) \qquad \text{(CP)}$$

where $A' = A^nC^{n-1}$ and $B' = B^nC^{n-1}$. In a PC composed of such layers, which we call "collapsed CP", or simply CP layers, the rank of each layer is $R$, except for the input layers that can still accommodate $K_{\text{in}}$ different input units. Through this collapsing, we retrieve the layer parameterization of HCLTs and other PCs that are the output of a compilation process, with the slight difference that they generally assume $K = R$ for all layers. All in all, by disentangling CP layers from a CLT-based RG, we can use them in non-structured decomposable PCs, e.g., by having non-tree RGs (see Table 1 and Section 4), and better understand the effect of using low-rank layer parameterizations instead of TUCKER during learning (Section 4).

By analogy, we can also apply a non-negative CP decomposition to the folded tensor $\boldsymbol{\mathcal{W}}^n$ in TUCKER-f layers, i.e., parameterize its $(i, j, k, f)$ entry as $\sum_{r=1}^R A^n_{ir} B^n_{jr} C^n_{kr} D^n_{fr}$ with the introduction of an additional matrix $D^n \in \mathbb{R}_+^{R\times F}$ per folded layer $n$. Note that this decomposition is much more parameter efficient and compact as the matrices $A^n$, $B^n$, and $C^n$ are shared across all folds $f$. Analogous to the collapse of a rank-$R$ CP layer, we can collapse adjacent sum layers and obtain a simplified layer that computes:

$$\mathbf{o}^{(f)} = (A'\boldsymbol{\ell}^{(f)}) \odot (B'\mathbf{r}^{(f)}) \qquad \text{(CP-s)}$$

Note that this parameterization, which we call CP shared, (CP-s), is different from folding a PC with CP layers (which we denote as CP-f), in that all the parameters $A'$ and $B'$ are shared across all folds.

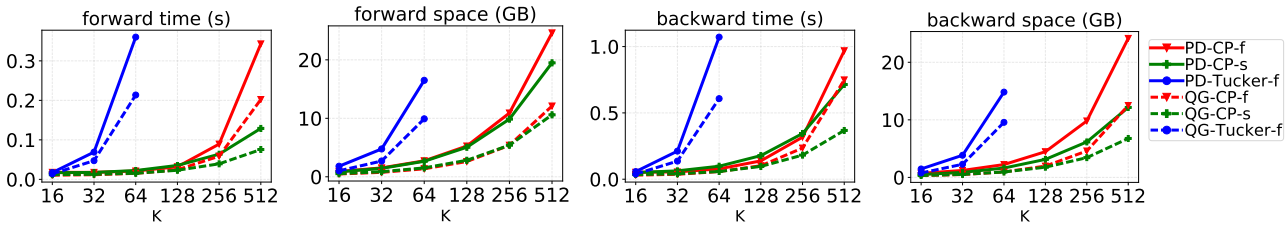



Figure 3: **Different RGs and layer parameterizations yield different time/space performance.** Time (seconds) and memory usage (GBs) for TUCKER-f, CP-s, and CP-f layers with PD and QG RGs for different values of $K$ ($x$-axis).

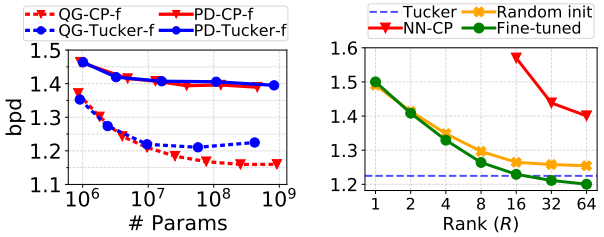



Figure 4: **The learning and compression performance** of different models depend on RG and layer parameterization. We show the bpd on the $y$-axis for models learned from scratch w.r.t. the number of parameters (left) or rank when compressed from a QG-TUCKER-f model (right).

**A new nomenclature.** By decomposing the process of representing overparameterized circuits into three macro stages in the previous section and now categorizing layer parameterizations via tensor decompositions, we can precisely denote old and new PC architectures, highlighting how they differ. For example, in our nomenclature, RAT-SPNs can be denoted as RND-TUCKER and EiNets as RND-TUCKER-f if they use random RGs, making explicit the encoding of the same computational graphs, only folded. By changing RG, we could have PD-TUCKER-f as well as CLT-TUCKER-f or PD-CP-f, "mixing & matching" the ingredients behind HCLTs, which in our nomenclature would be denoted as CLT-CP.

## 4 EXPERIMENTS

In this section, we evaluate how changing the ingredients in the "overparameterized PC recipe" can lead to different accuracy and computational efficiency performance. To disentangle and measure the effect of different RGs, we employ both PD RGs as well as a simpler but still image-aware and dataset-agnostic RG, which we call *quad graph* (QG). QGs have fewer parameters than PDs but are more expressive than CLTs (see Appendix A for details). We train and evaluate our models on MNIST [LeCun et al., 2010] and Fashion-MNIST (FMNIST) [Xiao et al., 2017], using Adam as an optimizer (Appendix C.2).

**Benchmarking.** We compared the time and memory peak for PCs built with PD or QG and with TUCKER-f, CP-f, CP-s layers, varying $K$ in $\{16, 32, 64, 128, 256, 512\}$ (Fig. 3). We can observe that QGs can be more parsimonious than PDs, and that CP-f and CP-s layers scale more gracefully for larger values of $K$, which may be impractical for TUCKER-f. Non-folded versions of these parameterizations can be an order of magnitude slower in our PyTorch implementation [Peharz et al., 2020b].

**Learning.** Fig. 4 (left) shows the bits-per-dimension (bpd) on the test set for MNIST (analogous results on FMNIST can be found in Appendix C.2). First, it can be seen that QG leads to better performance than PD. Moreover, TUCKER-f and CP-f have essentially identical performance on PD and for small values of $K$ on QG, while for larger values of $K$, CP-f obtains significantly better bpd (1.16). For $K \geq 256$, CP-s performs better than TUCKER-f and comes close to CP-f (see Appendix C.2).

**Compression.** Finally, we investigate what we lose in accuracy when we compress a more expressive model, say using QG-TUCKER-f, into a lower-rank model, say $R$-ranked CP-f. First, we compress each tensor slice $\mathbf{W}^{(f)}$ of a TUCKER-f layer by performing non-negative CP via alternating least squares [Shashua and Hazan, 2005]. As expected, a data-agnostic compression scheme increases bpd (Fig. 4 (right)). However, using these compressed tensors to initialize a $R$-ranked CP-f layer and using fine-tuning can outperform not only a randomly initialized model but also the original uncompressed model.

## 5 CONCLUSION

In this work, we systematized the construction of modern overparameterized circuits in a single pipeline, highlighting how the different stages, such as RG construction and layer parameterization, are related. We proposed a precise nomenclature to denote existing and new architectures that can be created by mixing & matching these ingredients. Our analysis on layer parameterizations, based on tensor decompositions, and experiments highlight how lower-rank structures can be easier to learn and suggest possible future venues for performing structure [Dang et al., 2022] or representation learning [Vergari et al., 2018] with circuits.

## References


Kareem Ahmed, Stefano Teso, Kai-Wei Chang, Guy Van den Broeck, and Antonio Vergari. Semantic probabilistic layers for neuro-symbolic learning. *Advances in Neural Information Processing Systems*, 35:29944–29959, 2022.

Myung Jin Choi, Vincent YF Tan, Animashree Anandkumar, and Alan S Willsky. Learning latent tree graphical models. *Journal of Machine Learning Research*, 12: 1771–1812, 2011.

YooJung Choi, Antonio Vergari, and Guy Van den Broeck. Probabilistic Circuits: A Unifying Framework for Tractable Probabilistic Modeling. 2020.

CKCN Chow and Cong Liu. Approximating discrete probability distributions with dependence trees. *IEEE transactions on Information Theory*, 14(3):462–467, 1968.

Andrzej Cichocki, Rafal Zdunek, and Shun-ichi Amari. Nonnegative matrix and tensor factorization [lecture notes]. *IEEE signal processing magazine*, 25(1):142–145, 2007.

Meihua Dang, Anji Liu, and Guy Van den Broeck. Sparse probabilistic circuits via pruning and growing. *Advances in Neural Information Processing Systems*, 35:28374–28385, 2022.

Adnan Darwiche. *Modeling and reasoning with Bayesian networks*. Cambridge university press, 2009.

Aaron Dennis and Dan Ventura. Learning the architecture of sum-product networks using clustering on variables. *Advances in Neural Information Processing Systems*, 25, 2012.

Lars Grasedyck. Hierarchical singular value decomposition of tensors. *SIAM journal on matrix analysis and applications*, 31(4):2029–2054, 2010.

Tamara G Kolda and Brett W Bader. Tensor decompositions and applications. *SIAM review*, 51(3):455–500, 2009.

Yann LeCun, Corinna Cortes, and CJ Burges. MNIST handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.

Anji Liu and Guy Van den Broeck. Tractable regularization of probabilistic circuits. *Advances in Neural Information Processing Systems*, 34:3558–3570, 2021.

Anji Liu, Honghua Zhang, and Guy Van den Broeck. Scaling Up Probabilistic Circuits by Latent Variable Distillation. *arXiv preprint arXiv:2210.04398*, 2022.

Robert Peharz, Sebastian Tschiatschek, Franz Pernkopf, and Pedro M. Domingos. On Theoretical Properties of Sum-Product Networks. In *International Conference on Artificial Intelligence and Statistics*, 2015.

Robert Peharz, Robert Gens, Franz Pernkopf, and Pedro Domingos. On the latent variable interpretation in sum-product networks. *IEEE transactions on pattern analysis and machine intelligence*, 39(10):2030–2044, 2016.

Robert Peharz, Steven Lang, Antonio Vergari, Karl Stelzner, Alejandro Molina, Martin Trapp, Guy Van den Broeck, Kristian Kersting, and Zoubin Ghahramani. Einsum networks: Fast and scalable learning of tractable probabilistic circuits. In *International Conference on Machine Learning*, pages 7563–7574. PMLR, 2020a.

Robert Peharz, Antonio Vergari, Karl Stelzner, Alejandro Molina, Xiaoting Shao, Martin Trapp, Kristian Kersting, and Zoubin Ghahramani. Random sum-product networks: A simple and effective approach to probabilistic deep learning. In *Uncertainty in Artificial Intelligence*, pages 334–344. PMLR, 2020b.

Hoifung Poon and Pedro Domingos. Sum-product networks: A new deep architecture. In *2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*, pages 689–690. IEEE, 2011.

Amnon Shashua and Tamir Hazan. Non-negative tensor factorization with applications to statistics and computer vision. In *Proceedings of the 22nd international conference on Machine learning*, pages 792–799, 2005.

Ledyard R Tucker. Some mathematical notes on three-mode factor analysis. *Psychometrika*, 31(3):279–311, 1966.

Antonio Vergari, Robert Peharz, Nicola Di Mauro, Alejandro Molina, Kristian Kersting, and Floriana Esposito. Sum-product autoencoding: Encoding and decoding representations using sum-product networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

Antonio Vergari, Nicola Di Mauro, and Floriana Esposito. Visualizing and understanding sum-product networks. *Machine Learning*, 108:551–573, 2019.

Antonio Vergari, YooJung Choi, Robert Peharz, and Guy Van den Broeck. Probabilistic circuits: Representations, inference, learning and applications. In *Tutorial at the The 34th AAAI Conference on Artificial Intelligence*, 2020.

Antonio Vergari, YooJung Choi, Anji Liu, Stefano Teso, and Guy Van den Broeck. A Compositional Atlas of Tractable Circuit Operations: From Simple Transformations to Complex Information-Theoretic Queries. *arXiv preprint arXiv:2102.06137*, 2021.

Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. *ArXiv*, abs/1708.07747, 2017.

# A BACKGROUND ON CIRCUITS

In this section, we report formal definitions of circuits and their structural properties, examining which are met in existing architectures.

**Definition A.1** (Circuit)**.** A circuit $c$ over variables $\boldsymbol{X}$ is a parameterized computational graph encoding a function $c(\boldsymbol{X})$ and recursively defined as one of the followings:

1. A paremeterized function over $\boldsymbol{X}$ (*input unit*):
$$c(\boldsymbol{X}) = f_\theta(\boldsymbol{X})$$
2. A product of circuits over subsets of $\boldsymbol{X}$ (*product unit*):
$$c(\boldsymbol{X}) = \prod_i c_i(\phi(c_i))$$
3. A weighted sum of circuits over subsets of $\boldsymbol{X}$ (*sum unit*):
$$c(\boldsymbol{X}) = \sum_i w_i c_i(\phi(c_i))$$

In the formulas, $w_i \in \mathbb{R}$ are the sum parameters and each $\phi(c_i) \subseteq \boldsymbol{X}$ is called the *scope* of the circuit $c_i$ (consequently $\phi(c) = \boldsymbol{X} = \bigcup_i \phi(c_i)$). For a sum unit (resp. product unit) $c$, we define its *input* as $\mathsf{in}(c) = \{c_i\}_i$.

**Definition A.2** (Probabilistic Circuit)**.** A PC over variables $\boldsymbol{X}$ is a circuit $c$ encoding a (possibly non-normalized) distribution, e.g., a function that is non-negative for all values of $\boldsymbol{X}$:
$$c(\boldsymbol{x}) \geq 0, \quad \forall \boldsymbol{x} \in \mathsf{dom}(\boldsymbol{X})$$

**Definition A.3** (Smoothness)**.** A circuit is *smooth* if, for each sum unit $n$, its inputs depend on the same variables: $\forall c_1, c_2 \in \mathsf{in}(n), \phi(c_1) = \phi(c_2)$.

**Definition A.4** (Decomposability)**.** A circuit is *decomposable* if the inputs of each product unit $n$ depend on disjoint sets of variables: $\mathsf{in}(n) = \{c_1, c_2\}, \phi(c_1) \cap \phi(c_2) = \varnothing$.

**Definition A.5** (Compatibility)**.** Two circuits $\mathcal{P}$ and $\mathcal{Q}$ over variables $\boldsymbol{X}$ are *compatible* if (*i*) they are smooth and decomposable and (*ii*) any pair of product units $n \in \mathcal{P}$ and $m \in \mathcal{Q}$ with the same scope can be rearranged into binary products that are mutually compatible and decompose in the same way: $(\phi(n) = \phi(m) \implies (\phi(n_i) = \phi(m_i)$, $n_i$ and $m_i$ are compatible) for some rearrangement of the inputs of $n$ (resp. $m$) into $n_1, n_2$ (resp $m_1, m_2$).

**Definition A.6** (Structured-decomposability)**.** A circuit is *structured-decomposable* if it is compatible with itself.

Although all PC architectures mentioned in this paper are smooth and decomposable, only PCs built upon RGs in which every region node has a single child (i.e., is partitioned in a single way) are also structured-decomposable.

Table 2: **Structural properties of modern PC architectures**: they are all smooth and decomposable, but HCLTs are also structured-decomposable. RND based PCs are guaranteed to be structured-decomposable only if $\mathsf{rep} = 1$ (see Appendix B.1).

| PC | SMO. | DEC. | STR-DEC. |
|---|---|---|---|
| RAT-SPN (RND-TUCKER) | ✓ | ✓ | ✓ |
| EINET (RND-TUCKER-F) | ✓ | ✓ | ✓ |
| EINET (PD-TUCKER-F) | ✓ | ✓ | ✗ |
| HCLT (CLT-CP) | ✓ | ✓ | ✓ |

Note that such circuits are potentially less expressive because they belong to a restricted class.

We report structural properties of the examined architectures in Table 2.

# B PIPELINE INGREDIENTS

In this section, we present and discuss the algorithms involved in each step of the general recipe for building overparameterized circuits (Fig. 1). In particular, we provide an overview of the mentioned RG structures and describe our proposed RG, the *quad graph*. Then we provide pseudocodes for the overparametrization (and vectorization) and folding steps.

## B.1 EXISTING REGION GRAPHS

**Random Region Graph.** The simplest RG structure is the Random RG (RND), proposed in [Peharz et al., 2020b]. It recursively splits regions into random, balanced 2-partitions, until a maximum depth $D$ is reached. This mechanism is repeated $\mathsf{rep}$ times, and all obtained RGs, called "repetitions", are merged into one, sharing common region nodes. Specifically, since the root is the same for each repetition, the complete RG's root will have one (partition node) child for each different 2-partition of the whole scope. Consequently, the structure will be structured-decomposable only if $\mathsf{rep} = 1$. Note that this structure is dataset-agnostic and depends only on the number of RVs. The hyper-parameters for its construction are $\mathsf{rep}$ and $D$.

**Poon Domingos structure.** A structure tailored for images was introduced in [Poon and Domingos, 2011] and later named Poon Domingos (PD) structure. It decomposes a $W \times H$ image into rectangles of size $\Delta \times \Delta$ (where $\Delta$ is a hyper-parameter) and combines them in a dynamic programming fashion, building $a\Delta \times b\Delta$ rectangles, for $a \in \{1, 2, \ldots, \frac{W}{\Delta}\}$ and $b \in \{1, 2, \ldots, \frac{H}{\Delta}\}$. Note that each rectangle constitutes an input region and that the number of regions grows according to $\mathcal{O}(\frac{1}{\Delta^3})$ [Peharz et al., 2015].

**HCLT Region Graph.** Finally, the most recently introduced RG structure, namely CLT, is dataset-aware and is based on the *Chow-Liu tree* algorithm [Chow and Liu, 1968]. The algorithm learns the tree-shaped probabilistic graphical model that best approximates the data distributions in terms of Kullback-Leibler divergence. Once learned, this tree is rooted. Then, a region graph can be constructed in the following way. Since each tree node $n$ is associated with an RV $X_n$, we define:

$$\mathcal{R}(n) = \{X_n\} \cup \bigcup_{c \in \mathsf{ch}(n)} \mathcal{R}(c)$$

where $\mathsf{ch}(n)$ is the set of $n$'s children. The region graph is easily obtained by replacing each node $n$ with the region node $\mathcal{R}(n)$ and replacing each connection with a partition node, creating additional children (input) regions $\{X_n\}$ when needed to complete the partitions.

### B.2 QUAD GRAPH

We propose the *quad graph* (QG), a novel image-tailored RG that takes a simple yet effective approach to deal with image data. Inspired by PD, the goal was to provide a more parameter-efficient solution (e.g., for $K = 128$, PD-CP-f has 128.4M parameters while QG-CP-f only 76.8M).

**Construction.** Starting from the leaves, one for each pixel, the RG is built in a bottom-up fashion, recursively merging $(2 \times 2)$ squares of contiguous regions into one (see Fig. 5a).

Initially, a region for each pixel is added to the RG (bottom of Fig. 5a). Then, looking at the whole image (at the top of the figure, which will be the root region, though not yet created), starting from the upper left corner, we can figure out which existing regions compose disjoint $2 \times 2$ blocks, forming squares (e.g. $[X_1, X_2, X_5, X_6]$ and $[X_3, X_4, X_7, X_8]$). We merge each group of four regions using the PD $(2 \times 2)$ structure, depicted in Fig. 5b: here, the generic blocks $A, B, C, D$ are merged to form either horizontal slices ($A, B$ and $C, D$) or vertical slices ($A, C$ and $B, D$), then each pair of slices is merged to reconstruct the complete block $A, B, C, D$. This process is iterated until the whole image is enclosed in a single region, requiring $\max\{\lceil log(W) \rceil, \lceil log(H) \rceil\}$ iterations.

Note that unless the image is of size $2^n \times 2^n$, sometimes it will not be possible to merge $2 \times 2$ blocks. As an example, look at Fig. 5c: starting from the upper left corner of the image and grouping adjacent pixels, some blocks of regions ($[X_3, X_6]$ and $[X_7, X_8]$) will not be able to form a square, but will instead be directly merged. Moreover, a region ($X_9$) will not be merged at all, skipping directly to the next iteration to be merged. Consequently, QG RG is not perfectly balanced in the general case.

**Pseudo-code.** Algorithm 1 specifies the construction process for image data of size $W \times H$. 2D array-shaped buffers

**Algorithm 1** $\mathsf{quadGraph}(W, H)$

1: **Input:** Image width $W$ and height $H$.
2: **Output:** The QG region graph $\mathcal{R}$ for image data with width $W$ and height $H$.
3: leaves: buffer of size $W \times H$
4: **for** $i, j$ s.t. $i \in \{1, \dots, W\}, j \in \{1, \dots, H\}$ **do**
5: $\quad \mathsf{leaves}[i, j] = \{X_{ij}\}$
6: $\mathsf{lastLayer} \leftarrow \mathsf{leaves}$
7: **while** $W > 1 \vee H > 1$ **do**
8: $\quad W \leftarrow \lceil W/2 \rceil$
9: $\quad H \leftarrow \lceil H/2 \rceil$
10: $\quad$ layer: buffer of size $W \times H$
11: $\quad$ **for** $i, j$ s.t. $i \in \{1, \dots, W\}, j \in \{1, \dots, H\}$ **do**
12: $\quad\quad \mathsf{regions} \leftarrow [\mathsf{lastLayer}[a, b] : (a, b) \in \{2i, 2i+1\} \times \{2j, 2j+1\}]$
13: $\quad\quad$ **if** $|\mathsf{regions}| = 1$ **then**
14: $\quad\quad\quad \mathsf{layer}[i, j] \leftarrow \mathsf{regions}[0]$
15: $\quad\quad$ **else if** $|\mathsf{regions}| = 2$ **then**
16: $\quad\quad\quad \mathsf{layer}[i, j] \leftarrow \mathsf{merge}(\mathsf{regions}[0], \mathsf{regions}[1])$
17: $\quad\quad$ **else if** $|\mathsf{regions}| = 4$ **then**
18: $\quad\quad\quad \mathsf{layer}[i, j] \leftarrow \mathsf{pd2by2}(\mathsf{regions})$
19: $\quad \mathsf{lastLayer} \leftarrow \mathsf{layer}$
20: $\mathcal{R} \leftarrow \mathsf{lastLayer}[0, 0]$
21: **return** $\mathcal{R}$

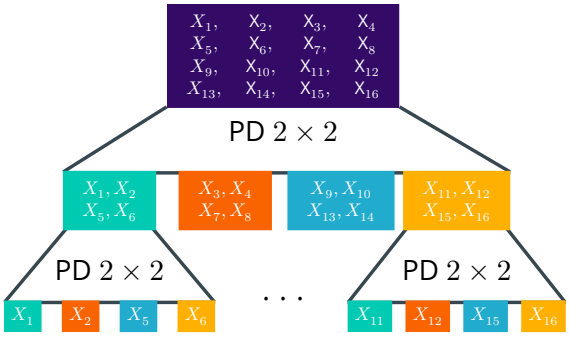
are used to store regions temporarily (lines 3, 10), and partition nodes (as well as internal region nodes) are created implicitly using the following auxiliary functions:

- merge (line 16), which takes two regions $\mathcal{R}_A, \mathcal{R}_B$, creates the partition node $\mathcal{P} = \{\mathcal{R}_A, \mathcal{R}_B\}$ and the subsequent region node $\mathcal{R} = \mathcal{R}_A \cup \mathcal{R}_B$, returning $\mathcal{R}$.
- pd2by2 (line 18), which takes a list of four regions as input and exploits merge to construct the structure depicted in Fig. 5b, returning its root region.

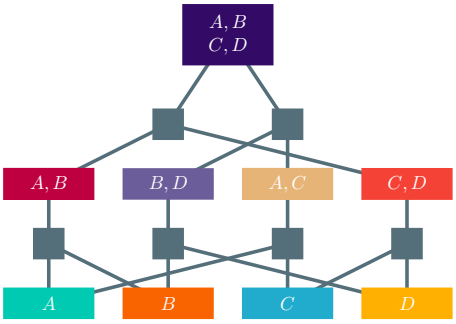
### B.3 OVERPARAMETERIZATION

The overparameterization algorithm of an RG (Algorithm 2) creates and vectorizes $K$ distributions for each input region node (lines 4-6). Consequently, the circuit is completed by iterating over each partition node $n$ (lines 8-12): a specialized makeLayer procedure builds the layers (line 11), i.e., computational graphs encoding vectorized functions $\mathbf{c} \colon \mathbb{R}_+^K \times \mathbb{R}_+^K \to \mathbb{R}_+^{K'}$. The inputs of each layer correspond to $n$'s children, while the output constitutes a vector of sum nodes and corresponds to $n$'s parent.

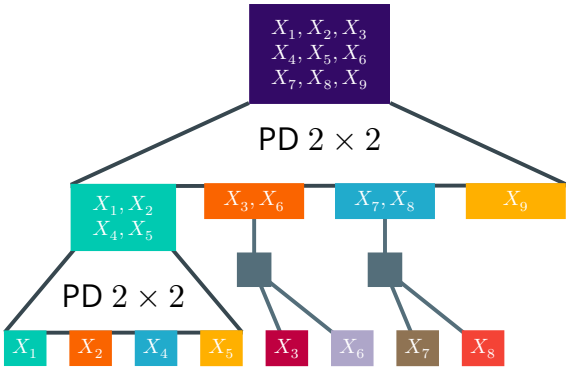
Note that the choice of the implementation for makeLayer determines the type of parameterization: Algorithm 3 contains the procedures for Eq. (TUCKER) and Eq. (CP).




(a) Quad graph (QG) (4 × 4)



(b) Poon Domingos (PD) (2 × 2)




(c) Odd values generate unbalanced graphs

Figure 5: **The quad graph structure** recursively partitions a (squared) image in four squares, with each partition structured as a 2 × 2 Poon Domingos. In the general case, the structure is unbalanced

### B.4 FOLDING

Different strategies for folding a circuit vary in the choice of layers to be included in the same fold. In [Peharz et al., 2020a], a top-down (TD) folding algorithm was proposed. Intuitively, each height level in Fig. 5a corresponds to a TD fold, so all the layers constructed from horizontally aligned regions will be folded together.

However, if, for instance, we were interested in folding all input nodes together, this strategy would no longer work, as one can see in Fig. 5c. Therefore, we propose a bottom-up (BU) alternative approach (see Algorithm 4), using the following operations:

- in (line 8), which takes as argument a layer $l$ and returns the set of layers whose output is an input for $l$;
- fold (line 10), which takes a set of layers as an argument and folds their computational graphs.

The BU procedure first folds the input layers together (as "in" returns an empty set), and, at each iteration, forms a new fold by selecting all remaining layers whose inputs have been folded. This is repeated until all layers are folded.

Note that the BU and TD procedures give the same results in the case of a balanced RG, and there is no general preference between them, except when sharing parameters across the same fold (as in Eq. (CP-s)). At this stage, the architec-

**Algorithm 2** parameterization$(\mathcal{R}, K, C)$

1: **Input:** A region graph $\mathcal{R}$, positive integers $K$ and $C$.
2: **Output:** A smooth and decomposable circuit.
3: marked $\leftarrow \varnothing$
4: **for** each input region node $r \in \mathcal{R}$ **do**
5: $\quad \mathbf{c} \in \mathbb{R}^K$: vector of $K$ input nodes $c$
6: $\quad \mathsf{vec}(r) \leftarrow \mathbf{c}$
7: $\quad$ marked $\leftarrow$ marked $\cup \{r\}$
8: **while** a partition node $n \in \mathcal{R}$ exists s.t. $\mathsf{ch}(n) \subseteq$ marked $\wedge\ \mathsf{pa}(n) \notin$ marked **do**
9: $\quad l, r \leftarrow \mathsf{ch}(n)$
10: $\quad p \leftarrow \mathsf{pa}(n)$
11: $\quad$ **if** $p = \mathsf{root}(\mathcal{R})$ **then**
12: $\quad\quad K' \leftarrow C$
13: $\quad$ **else**
14: $\quad\quad K' \leftarrow K$
15: $\quad \mathsf{vec}(p) \leftarrow \mathsf{makeLayer}(\mathsf{vec}(l), \mathsf{vec}(r), K')$
16: $\quad$ marked $\leftarrow$ marked $\cup \{p\}$
**return** $\mathsf{vec}(\mathsf{root}(\mathcal{R}))$

**Algorithm 3** Layer builders

**Input**: Vectorized circuits $\boldsymbol{\ell}, \mathbf{r} \in \mathbb{R}_+^K$ s.t. $\phi(\boldsymbol{\ell}) = \mathbf{X}_l$ and $\phi(\mathbf{r}) = \mathbf{X}_r$ with output size $K'$
**Output**: a vectorized circuit $\mathbf{o} \in \mathbb{R}_+^{K'}$ s.t. $\phi(\mathbf{o}) = \mathbf{X}$ and $\mathbf{X} = \mathbf{X}_l \cup \mathbf{X}_r$

**procedure** buildTucker$(\boldsymbol{\ell}, \mathbf{r}, K')$
$\quad \mathbf{W} \in \mathbb{R}^{K \times K \times K'}$ (layer weights)
$\quad$ Let $(\mathbf{o}(\mathbf{X}))_k = (\boldsymbol{\ell}(\mathbf{X}_l))_i (\mathbf{W})_{ijk} (\mathbf{r}(\mathbf{X}_r))_j$
$\quad$ **return** $\mathbf{o}$

**procedure** buildCP$(\boldsymbol{\ell}, \mathbf{r}, K')$
$\quad A \in \mathbb{R}_+^{K' \times K}, B \in \mathbb{R}_+^{K' \times K}$
$\quad$ Let $\mathbf{o}(\mathbf{X}) = (A\, \boldsymbol{\ell}(\mathbf{X}_l)) \odot (B\, \mathbf{r}(\mathbf{X}_r))$
$\quad$ **return** $\mathbf{o}$

**Algorithm 4** foldingBU($\mathbf{c}$)

**Input**: A vectorized circuit $\mathbf{c}$
**Output**: -
Let $L$ be the set of all layers
folded $\leftarrow \varnothing$
**while** folded $\neq L$ **do**
    layersToFold $\leftarrow \varnothing$
    **for** $l \in L$ **do**
        **if** in$(l) \subseteq$ folded and $l \notin$ folded **then**
            layersToFold $\leftarrow$ layersToFold $\cup \{l\}$
    fold(layersToFold)
    folded $\leftarrow$ folded $\cup$ layersToFold

Table 3: **Complexity summary** for a single TUCKER-f or CP-f layer. Note that space complexity only considers the memory used for storing the (intermediate) results of the operations, not counting parameters. Note that $B$ represents the batch size and $F$ the number of stacked layers in the folded-layer.

| Layer | # Params | Fwd time | Fwd space |
|---|---|---|---|
| TUCKER | $FK^3$ | $\mathcal{O}(BFK^3)$ | $\mathcal{O}(BFK^2)$ |
| CP ($R$-rank) | $3FKR$ | $\mathcal{O}(BFKR)$ | $\mathcal{O}(BF(K+3R))$ |

tural choices require further experimentation.

# C EXPERIMENTAL DETAILS

**Hardware specifications.** All experiments were run on a server with an NVIDIA RTX A6000 GPU with 48GB of memory. In all experiments, we used only a single GPU.

## C.1 BENCHMARKING

Each benchmarking value is averaged over 100 repetitions and measures a single forward or backward pass through the computational graph, using randomly generated data with a batch size of 128. Plots in Fig. 3 highlight the polynomial difference in complexity between TUCKER and CP parameterizations, as shown in Table 3. Note that space complexity refers to the size of all intermediate results computed, not counting parameters.

## C.2 LEARNING

For all experiments, we chose categorical input distributions for the PCs. We used a validation set (5% of the original training set) for each dataset and report the results on the test set.

**Parameter learning.** We optimize the parameters with Adam and clamp mixture parameters to a small value $\varepsilon > 0$ ($\simeq 10^{-19}$) after each update to keep the weights non-negative. Since this leads to non-normalized PCs, we optimized the following MLE objective (given the dataset $D$):

$$\mathcal{L}(\mathcal{D}, c) = \sum_{\mathbf{x} \in \mathcal{D}} (\log(c(\mathbf{x})) - \log(Z))$$

where $Z$ is the partition function of the PC.

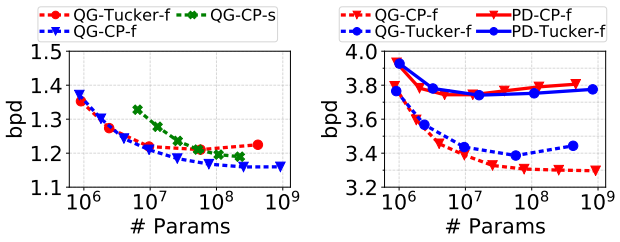



Figure 6: **CP-s performance on MNIST** are comparable with those of previously presented parameterizations (left); **results on Fashion-MNIST** demonstrate analogous properties for the different region graphs and layer parameterizations (right).

**Hyper-parameter search.** We trained for up to 500 epochs and performed early stopping (using the validation set). We grid-searched over the following hyper-parameters using the validation set:

- $K \in \{4, 8, 16, 32, 64, 128, 256, 512\}$;
- lr $\in \{0.1, 0.01, 0.001\}$;
- $\Delta \in \{4, 7\}$ for the PD region graph;
- batch_size $= 100$.

**Evaluation.** We used the test set's average bits-per-dimension (bpd) as the evalutation criterion:

$$\text{bpd}(\mathcal{D}, c) = \frac{-\mathcal{L}(\mathcal{D}, c)}{d \cdot \log 2}$$

where $d$ is the number of features in dataset $\mathcal{D}$. Fig. 6 compares QG-CP-s and the previously presented parameterizations on MNIST and demonstrates similar properties of RG and layer parameterizations on Fashion-MNIST.