Scaling Continuous Latent Variable Models as Probabilistic Integral Circuits

Gennaro Gala[1, *] **Cassio de Campos**[1] **Antonio Vergari**[2,ⓕ] **Erik Quaeghebeur**[1,ⓕ]

[1]Eindhoven University of Technology, NL
[2]School of Informatics, University of Edinburgh, UK

Abstract

Probabilistic integral circuits (PICs) have been recently introduced as probabilistic models enjoying the key ingredient behind expressive generative models: continuous latent variables (LVs). PICs are symbolic computational graphs defining continuous LV models as hierarchies of functions that are summed and multiplied together, or integrated over some LVs. They are tractable if LVs can be analytically integrated out, otherwise they can be approximated by tractable probabilistic circuits (PC) encoding a hierarchical numerical quadrature process, called QPCs. So far, only tree-shaped PICs have been explored, and training them via numerical quadrature requires memory-intensive processing at scale. In this paper, we address these issues, and present: (i) a pipeline for building DAG-shaped PICs out of arbitrary variable decompositions, (ii) a procedure for training PICs using tensorized circuit architectures, and (iii) neural functional sharing techniques to allow scalable training. In extensive experiments, we showcase the effectiveness of functional sharing and the superiority of QPCs over traditional PCs.

1 Introduction

Continuous latent variables (LVs) are arguably the key ingredient behind many successful generative models, from variational autoencoders [24] to generative adversarial networks [20], and more recently diffusion models [56]. While these models allow to learn expressive distributions from data, they are limited to *sampling* and require task-specific approximations when it comes to perform *probabilistic reasoning*, as even simple tasks such as computing marginals or conditionals are intractable for them. On the other hand, performing these tasks can be tractable for (hierarchical) ***discrete*** LV models [4, 3], but these prove to be more challenging to learn at scale [9, 10, 32, 33].

This inherent trade-off among tractability, ease of learning, and expressiveness can be analyzed and explored with *probabilistic integral circuits* (PICs) [18], a recently introduced class of deep generative models defining hierarchies of continuous LVs using *symbolic* functional circuits. PICs are tractable when their continuous LVs can be analytically integrated out. Intractable PICs can however be systematically approximated as (tensorized) *probabilistic circuits* (PCs) [53, 4], the representation language of discrete LV models. An instance of such PCs encodes a hierarchical numerical quadrature process of the PIC to approximate, and as such is called *quadrature PC* (QPC).

Distilling QPCs from PICs has proven to be an effective alternative way to train PCs, but it has only been explored for tree-shaped PICs, as building and scaling to richer LV structures is an open research question that requires new tools [18]. In this paper, we fill this gap by redefining the semantics of PICs and extending them to DAG-shaped hierarchies of continuous LVs. Specifically, we

*Corresponding author: g.gala@tue.nl
ⓕ Shared supervision

38th Conference on Neural Information Processing Systems (NeurIPS 2024).

design PICs as a language to represent hierarchical quasi-tensors factorizations [49], parameterized by light-weight multi-layer perceptrons.

Contributions. **(1)** We present a systematic pipeline to build DAG-shaped PICs, starting from arbitrary variable decompositions (Section 3.1). **(2)** We show how to learn and approximate PICs via a hierarchical quadrature process which we encode in tensorized QPCs that match certain circuit architectures proposed in different prior works [43, 42, 31, 36] (Section 3.2). **(3)** We present functional sharing techniques to scale the training of PICs, which lead us to parameterize them with multi-headed multi-layer perceptrons (MLPs) requiring comparable resources as PCs (Section 3.3). **(4)** In extensive experiments (Section 4), we show that (i) functional sharing proves remarkably effective for scaling and that (ii) QPCs outperform PCs commonly trained via EM or SGD, while being distilled from PICs with up to 99% less *trainable parameters*.

2 Probabilistic integral circuits

Notation. We denote input variables as $\mathbf{X}$ and latent variables (LVs) as $\mathbf{Y}$ and $\mathbf{Z}$, with $\mathbf{x}, \mathbf{y}$ and $\mathbf{z}$ as their realization respectively. We denote scalars with lower-case letters (e.g., $w \in \mathbb{R}$), vectors with boldface lower-case letters (e.g., $\mathbf{w} \in \mathbb{R}^N$), matrices with boldface upper-case letters (excluding $\mathbf{X}, \mathbf{Y}, \mathbf{Z}$, e.g., $\mathbf{W} \in \mathbb{R}^{M \times N}$), and tensors with boldface calligraphic letters (e.g., $\boldsymbol{\mathcal{W}} \in \mathbb{R}^{L \times M \times N}$).

A ***probabilistic integral circuit*** (PIC) c is a symbolic computational graph representing a non-negative function $c(\mathbf{X}) = \int c(\mathbf{X}, \mathbf{z})\, \mathrm{d}\mathbf{z}$, i.e. $c(\mathbf{x}) \geq 0$, over observed variables $\mathbf{X}$ and *continuous* latent variables $\mathbf{Z}$. Similar to probabilistic circuits (PCs) [53, 4], PICs have *input*, *sum* and *product* units.[2] Different from PCs, however, PICs operate on functions, not scalars, and make use of a new type of unit: the *integral* unit $\int$, which allows to represent continuous LVs. To satisfy non-negativity, we parameterize PICs with non-negative functions and positive sum weights, specifically:

- An input unit u (depicted as f in our figures) represents a possibly non-normalized distribution $f_u(\mathbf{X}_u, \mathbf{Z}_u) \to \mathbb{R}^+$, where $\mathbf{X}_u \subseteq \mathbf{X}$ and $\mathbf{Z}_u \subseteq \mathbf{Z}$;
- A sum unit u ($+$) outputs a weighted sum of the functions it receives from its input units, i.e. $g_u(\mathbf{X}_u, \mathbf{Z}_u) = \Sigma_{i \in \mathsf{in}(u)} w_i g_i(\mathbf{X}_i, \mathbf{Z}_i)$, where $\mathsf{in}(u)$ is the set of units u takes as input, $w_i > 0$, $\mathbf{X}_u = \cup_{i \in \mathsf{in}(u)} \mathbf{X}_i$ and $\mathbf{Z}_u = \cup_{i \in \mathsf{in}(u)} \mathbf{Z}_i$. Similarly, a product unit u ($\times$) outputs the product of its incoming functions, i.e. $g_u(\mathbf{X}_u, \mathbf{Z}_u) = \Pi_{i \in \mathsf{in}(u)} g_i(\mathbf{X}_i, \mathbf{Z}_i)$;
- Finally, an integral unit u ($\int$) encodes an "uncountable weighted sum" whose weights are compactly represented by a function $f_u(\mathbf{Z}_u, \mathbf{Y}_u) \to \mathbb{R}^+$, where $\varnothing \neq \mathbf{Y}_u \subseteq \mathbf{Z}$ are the LVs that are being integrated out by u, while $\mathbf{Z}_u$ can potentially be empty, i.e. $\mathbf{Z}_u = \varnothing$. The unit receives a function $g_i(\mathbf{X}_i, \mathbf{Y}_u)$ from its only input unit i and outputs the function $g_u(\mathbf{X}_i, \mathbf{Z}_u) = \int_\Delta f_u(\mathbf{Z}_u, \mathbf{y}_u)\, g_i(\mathbf{X}_i, \mathbf{y}_u)\, \mathrm{d}\mathbf{y}_u$, where $\Delta = \mathsf{supp}(\mathbf{Y}_u)$ is the support of $\mathbf{Y}_u$. For instance, an integral unit u with $f_u(\{Z\}, \{Y\}) = 2Z - Y^2$ and $\mathsf{supp}(Y) = [-1, 1]$ receiving function $g_i(\{X\}, \{Y\}) = X^2 - 3X + 4Y$ would output $g_u(\{Z\}, \{X\}) = {}^2\!/_3\, X(X-3)(6Z-1)$.

Fig. 1(b) and Fig. 2(b) show example PICs. Note that we use f to indicate the functions attached to input and integral units which are essentially parameters of the model, while we use g to indicate functions being outputted by all type of units. The output of a PIC c is the output function returned by its root unit u, which is only defined on $\mathbf{X}$, i.e. $c(\mathbf{X}) = g_u(\mathbf{X})$ and $\mathbf{Z}_u = \varnothing$. Similar to PCs, imposing structural constraints over PICs can unlock tractable inference [4]. As such, we assume that (i) all $+$-units receive functions defined on the same input variables (aka *smoothness*), and (ii) all $\times$-units receive functions defined over disjoint sets of input variables (aka *decomposability*).

PICs are tractable when their LVs can be analytically integrated out, meaning that we can *pass-through* integral units computing the integration problem they define, eventually outputting a function. Notably, this is possible when LVs are in linear-Gaussian relationships [27] or when functions are polynomials. Intractable PICs can however be approximated via a hierarchical numerical quadrature process that can be encoded as a PC called ***quadrature PC*** (QPC). Intuitively, each PIC integral unit can be approximated by a set of sum units in a QPC, each conditioning on some previously computed quadrature values, with a large but finite number of input units [18]. Materializing a QPC allows to train PICs by approximate maximum likelihood: Given a PIC, gradients to its parameters

[2] We refer the reader to Appendix A for an introductory overview of (probabilistic) circuits.

attached to input, sum and integral units can be backpropagated through the corresponding QPC [18]. This also provides an alternative way to train PCs that can rival traditional learners.

So far, the construction of PICs has been limited to a simple *compilation* process from probabilistic graphical models (PGMs) [27] with continuous LVs [18]. In a nutshell, the LV nodes of a PGM become integral units of a PIC model, and the PGM (conditional) distributions become the functions f_u attached to input and integral units of the PIC, as we illustrate in Fig. 1. However, the PGM structure *needs to be limited to a tree*, as to avoid that the hierarchical quadrature process would yield an exponentially large QPC, thus hindering learning. This imposes a semantics for current PICs as simple latent tree models [3], and clearly limits their expressiveness as more complex LV interactions are not possible. Building more expressive PICs requires reinterpreting this semantics and introducing new tools, which we do next.

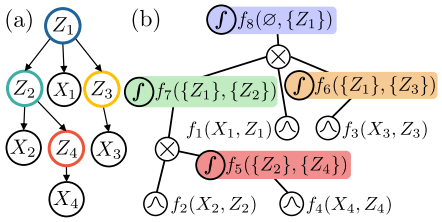


Figure 1: PGM (a) → tree PIC (b)

3 Building, learning and scaling PICs

In Section 3.1, we systematize the construction of DAG-shaped PICs, showing how to build them starting from arbitrary variable decompositions, going beyond the current state-of-the-art [18]. Then, in Section 3.2, we show how to learn and approximate such PICs with QPCs encoding a hierarchical quadrature process, retrieving PC architectures proposed in prior works [36]. Finally, in Section 3.3, we present (neural) functional sharing, a technique which we use to parameterize PICs as to make their QPC materialization fast and cheap, allowing scaling to larger models and larger datasets.

3.1 Building PICs from arbitrary variable decompositions

Standard PCs can be built according to established pipelines that allow to flexibly represent arbitrary variable decompositions, as well as rich discrete LV interactions [36, 41]. In the following, we derive an analogous pipeline for PICs that allows to take care of continuous LVs, going beyond their current tree-shaped semantics (Section 2) yet allowing to perform hierarchical quadrature without blowing up the size of the materialized QPCs. To do so, we start by formalizing the notion of hierarchical variable decomposition, or region graph, out of which we will build our PIC structures.

Definition 1 (Region Graph (RG) [15])**.** *An RG $\mathcal{R}$ over input variables $\mathbf{X}$ is a bipartite and rooted directed acyclic graph (DAG) whose nodes are either* regions*, denoting subsets of $\mathbf{X}$, or* partitions*, specifying how a region is partitioned into other regions (Fig. 2(a)).*

RGs can be (i) compiled from PGMs [5, 27, 3, 31], (ii) randomly initialized [43, 16], (iii) learned from data [15, 19, 40, 57], or (iv) built according to the data modality (e.g. images) [45, 42, 36]. If we compile from a tree PGM, as in (Fig. 1), the resulting RG will be a tree, thus yielding a tree-like PIC [18]. Our pipeline, detailed in Algorithm 1, takes an arbitrary DAG-shaped RG as input, and can deliver DAG-like PICs. Without loss of generality, we assume to have an RG $\mathcal{R}$ which only allows for (i) binary partitionings of regions (i.e. all product units will have two input units) and (ii) univariate leaves, as shown in Fig. 2(a). Our construction iteratively builds a PIC in a bottom-up fashion, associating regions to PIC units. For every leaf region $X_u \in \mathbf{X}$ in $\mathcal{R}$, we instantiate an input unit u with function $f_u(\{X_u\}, \{Z_u\})$, where $Z_u \in \mathbf{Z}$ is an arbitrary continuous LV (Line 8, Algorithm 1). Such functions can be univariate conditional densities, i.e. $p_u(X_u|Z_u)$, resembling small VAE-like decoders [24] amenable to be numerically integrated.

Once all leaf regions have been processed, we move to the inner ones. Let $\mathbb{X} \subseteq \mathbf{X}$ be an inner region partitioned in $N \geq 1$ different ways as $\{(\mathbb{X}_1^{(n)}, \mathbb{X}_2^{(n)})\}_{n=1}^{N}$, i.e. $(\mathbb{X}_1^{(n)} \cap \mathbb{X}_2^{(n)}) = \varnothing$ and $(\mathbb{X}_1^{(n)} \cup \mathbb{X}_2^{(n)}) = \mathbb{X}$ for every n. For each partition $(\mathbb{X}_1^{(n)}, \mathbb{X}_2^{(n)})$, we will merge the PIC units associated to regions $\mathbb{X}_1^{(n)}$ and $\mathbb{X}_2^{(n)}$ using consecutive applications of product and integral units—as we explain next—eventually associating a unit the partition itself. One can design such merging as desired, as long as smoothness and decomposability are not violated. Finally, in case $N = 1$, we associate to $\mathbb{X}$ the unit associated to its only partition, otherwise, in case $N > 1$, we merge the N units associated to each n-th partition using a sum unit which we then associate to $\mathbb{X}$ (Line 6, Algorithm 1).

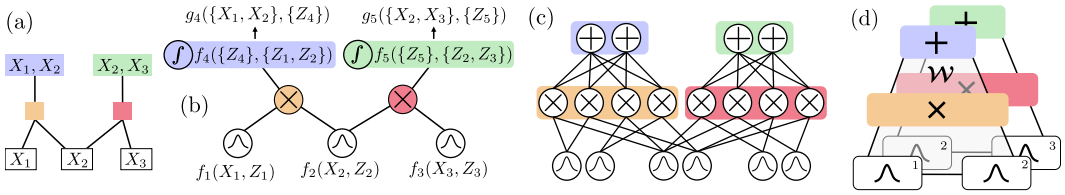


Figure 2: **The pipeline presented in this paper: RG → PIC → QPC → folded QPC**. Starting from a (fragment of) a DAG-shaped region graph (a), we build a DAG-like PIC via Algorithm 1 using Tucker-merge (b). Then, we materialize a tensorized QPC encoding a hierarchical quadrature process via Algorithm 3, using $K = 2$ quadrature points, which we fold to allow faster inference (d).

Algorithm 1 RG2PIC($\mathcal{R}$)

Input RG $\mathcal{R}$ over variables $\mathbf{X}$
Output PIC $c(\mathbf{X}) = \int c(\mathbf{X}, \mathbf{z})\mathrm{d}\mathbf{z}$

$\mathcal{U} \leftarrow$ map() ▷ from regions to PIC units
for each region $\mathbb{X} \in$ postOrder($\mathcal{R}$) **do**
  if $\mathbb{X}$ partitioned as $\{(\mathbb{X}_1^{(n)}, \mathbb{X}_2^{(n)})\}_{n=1}^N$ **then**
    $\rho \leftarrow$ True **if** $\mathbb{X} = \mathbf{X}$ **else** False
    $\mathcal{S} \leftarrow \{\text{merge}(\mathcal{U}[\mathbb{X}_1^{(n)}], \mathcal{U}[\mathbb{X}_2^{(n)}], \rho)\}_{n=1}^N$
    $\mathcal{U}[\mathbb{X}] \leftarrow$ pop($\mathcal{S}$) **if** $N{=}1$ **else** $\bigoplus([\mathcal{S}])$
  else ▷ $|\mathbb{X}| = 1$
    $\mathcal{U}[\mathbb{X}] \leftarrow \bigwedge(f_u(\mathbb{X}, \{Z_u\}))$
return A PIC with $\mathcal{U}[\mathbf{X}]$ as root unit

Algorithm 2 merge(u_1, u_2, ρ)

Input Units u_1, u_2, and boolean flag ρ
Output $\int$ or $\bigotimes$ unit u with (u_1, u_2) as descendants

procedure (Tucker-merge) ▷ $Z_{u_1} \neq Z_{u_2}$
  $\mathbf{Z}_u \leftarrow \varnothing$ **if** ρ **else** $\{Z\} \subset \mathbf{Z} \setminus \{Z_{u_1}, Z_{u_2}\}$
  return $\int(\bigotimes([u_1, u_2]), f_u(\mathbf{Z}_u, \{Z_{u_1}, Z_{u_2}\}))$
procedure (CP-merge) ▷ $Z_{u_1} = Z_{u_2}$
  $\mathbf{Z}_{u_3} \leftarrow \varnothing$ **if** ρ **else** $\{Z_{u_3}\} \subset \mathbf{Z} \setminus \{Z_{u_1}\}$
  $u_3 \leftarrow \int(u_1, f_{u_3}(\mathbf{Z}_{u_3}, \{Z_{u_1}\}))$
  $\mathbf{Z}_{u_4} \leftarrow \varnothing$ **if** ρ **else** $\{Z_{u_4}\} \subset \mathbf{Z} \setminus \{Z_{u_2}\}$
  $u_4 \leftarrow \int(u_2, f_{u_4}(\mathbf{Z}_{u_4}, \{Z_{u_2}\}))$
  return $\bigotimes([u_3, u_4])$

Merging PIC units. Let u_1 and u_2 be candidate units to merge, each outputting functions with LV Z_{u_1} and Z_{u_2} respectively. We present two ways of merging units: Tucker-merge and CP-merge, which we detail in Algorithm 2 and whose names will be clearer in the next section. If $Z_{u_1} \neq Z_{u_2}$, we use Tucker-merge: We merge u_1 and u_2 with a product, which is then input to an integral unit u with function $f_u(\{Z_u\}, \{Z_{u_1}, Z_{u_2}\})$, where $Z_u \in \mathbf{Z} \setminus \{Z_{u_1}, Z_{u_2}\}$. Otherwise, if $Z_{u_1} = Z_{u_2}$, we use CP-merge: We add two integral units, u_3 with input u_1 and u_4 with input u_2, which we finally merge with a product. We parameterize unit u_3 (resp. u_4) with $f_{u_3}(\{Z_{u_3}\}, \{Z_{u_1}\})$ (resp. $f_{u_4}(\{Z_{u_4}\}, \{Z_{u_2}\})$), where $Z_{u_3} \neq Z_{u_1}$ (resp. $Z_{u_4} \neq Z_{u_2}$). Note that whenever merging two units defined on $\mathbf{X}$, we need to marginalize out the remaining LVs, without introducing new ones. We illustrate the application of Algorithm 1 in Fig. 2(a-b). Our pipeline generalizes the PICs used in prior work [18] (Fig. 1) as we can build them by just converting latent tree structures in tree RGs and using CP-merge as merging procedure. While we now do *not* need a PGM to build a complex PIC structure, one could try to reverse-engineer our PICs to retrieve a PGM via *decompilation* [1], the result would be a very intricate hierarchy over continuous LVs [41].

3.2 Learning PICs via tensorized QPCs

Given a DAG-shaped (intractable) PIC, we now show how to approximate it with a tensorized PC encoding a hierarchical quadrature process, namely a QPC. Intuitively, we interpret PICs as to encode a set of *quasi-tensors* [49], a generalization of tensors with potentially infinite entries in each dimension corresponding to a continuous LV, which we materialize into classical tensors via quadrature. We begin with a definition of tensorized circuits and a brief refresher on numerical quadrature.

Definition 2 (Tensorized Circuit [36, 43])**.** *A **tensorized circuit** c is a parameterized computational graph encoding a function $c(\mathbf{X}) \in \mathbb{R}$, and comprising of input $\bigwedge$, product $\bigotimes$ and sum $\bigoplus$ layers. Each layer consists of many computational units defined over the same variables, and every non-input layer receives vectors as input from one or more layers. Each input layer $\boldsymbol{\ell}$ is defined on variables $\mathbf{X}_{\boldsymbol{\ell}} \subseteq \mathbf{X}$ and computes a collection of $K_{\boldsymbol{\ell}}$ parametric functions $[f_k : \mathsf{dom}(\mathbf{X}_{\boldsymbol{\ell}}) \to \mathbb{R}]_{k=1}^{K_{\boldsymbol{\ell}}}$, outputting a $K_{\boldsymbol{\ell}}$-dimensional vector. Each product layer $\boldsymbol{\ell}$ computes either an Hadamard product ($\odot$) or a Kronecker product ($\otimes$) of the vectors it receives from its inputs layers. Specifically, the Hadamard product is an element-wise product of vectors, and therefore applicable when these have same size, while the outer product of two vectors $\mathbf{u} \in \mathbb{R}^N$ and $\mathbf{v} \in \mathbb{R}^M$ is $\mathbf{w} = \mathbf{u} \otimes \mathbf{v} = ||_{i=1}^N u_i \mathbf{v} \in \mathbb{R}^{NM}$, where $||$ is the concatenation operator. Finally, a sum layer $\boldsymbol{\ell}$ with $S_{\boldsymbol{\ell}}$ sum units receives inputs from*

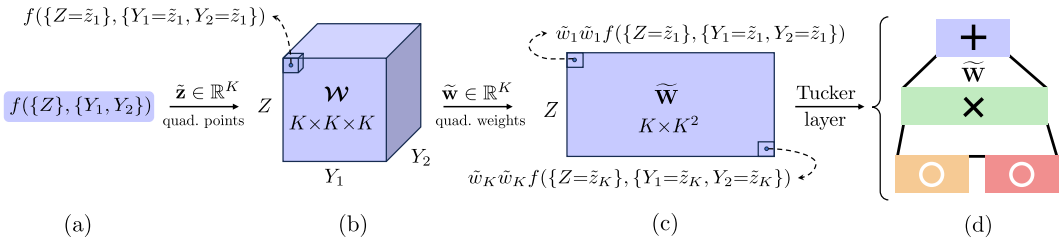


Figure 3: **From functions to sum-product layers via multivariate numerical quadrature** (Section 3.2). We illustrate how the 3-variate function $f(\{Z\},\{Y_1,Y_2\})$ (a) can be seen as an infinite (quasi) tensor that we first materialize w.r.t. integration points $\tilde{\mathbf{z}}$ as a finite tensor $\mathcal{W}$ of size $K\times K\times K$ (b, Equation (2)), then flatten as a matrix accounting for integration weights $\widetilde{\mathbf{w}}$ (c, Equation (3)), and finally use to parameterize a Tucker layer (d, Equation (Tucker-layer)).

N layers $\{\boldsymbol{\ell}_i\}_{i=1}^N$ and computes the matrix-vector product $\mathbf{W}||_{i=1}^N\boldsymbol{\ell}_i(\mathbf{X}_{\boldsymbol{\ell}_i})$, where $\mathbf{W}\in\mathbb{R}^{S_{\boldsymbol{\ell}}\times K}$, $K{=}\Sigma_{i=1}^N K_{\boldsymbol{\ell}_i}$, are the sum layer parameters. When $N=1$, then it simply computes $\mathbf{W}\boldsymbol{\ell}_1(\mathbf{X}_{\boldsymbol{\ell}_1})$.

Numerical quadrature. A numerical quadrature rule is an approximation of the definite integral of a function as a weighted sum of function evaluations at specified points [13]. Specifically, given some integrand $f:\mathbb{R}\to\mathbb{R}$ and interval $\Delta:=[a,b]$, a quadrature rule consists of a set of K integration points $\tilde{\mathbf{z}}\in\Delta^K$ and weights $\widetilde{\mathbf{w}}\in\mathbb{R}^K$ minimizing the integration error $\varepsilon_K = |\int_\Delta f(z)\,\mathrm{d}z - \Sigma_{k=1}^K\tilde{w}_k f(\tilde{z}_k)|$, which goes to zero as $K\to\infty$. To approximate an integral of an N-dimensional function f, we can phrase the multiple integral as repeated one-dimensional integrals by applying Fubini's theorem [17], aka tensor product rule, as follows.

$$\int_{\Delta^K} f(\mathbf{z})\mathrm{d}\mathbf{z} = \int_\Delta\ldots\Big(\int_\Delta f(z_1,\ldots,z_N)\mathrm{d}z_1\Big)\ldots\mathrm{d}z_N \approx \sum_{i_1\in[K]}\tilde{w}_{i_1}\ldots\sum_{i_N\in[K]}\tilde{w}_{i_N}f(\tilde{z}_{i_1},\ldots,\tilde{z}_{i_N}). \tag{1}$$

From PICs to QPCs. Given a candidate PIC, we will explore it in post order traversal, and iteratively associate a circuit layer (Definition 2) to each PIC unit, a process that we call *materialization*. We detail such procedure in Algorithm 3, which essentially applies Eq. (1) hierarchically over the PIC units. Each unit encodes a function over potentially continuous and discrete variables, hence representing a quasi-tensor, which we approximate by evaluating it over the quadrature points only, thus materializing a classical tensor (Fig. 3 (a,b)). We facilitate quadrature by assuming that all PIC LVs have bounded domain $\Delta:=[-1,1]$. This way, we can always use the same quadrature rule $(\tilde{\mathbf{z}},\widetilde{\mathbf{w}})$ for each required (multivariate) approximation, and also simplify treatment and exposition.

We begin materializing every PIC input unit u with function $f_u(\{X_u\},\{Z_u\})$ w.r.t. integration points $\tilde{\mathbf{z}}$, effectively creating an input layer $\boldsymbol{\ell}:\mathsf{dom}(X_u)\to\mathbb{R}^K$ as $[\,f_u(X_u,Z_u{=}\tilde{z}_k)\,]_{k=1}^K$ (Line 4, Algorithm 3). The parameters of such layer can be materialized as a matrix of shape $K\times P$, where P is the number of parameters f_u requires. For example, if f_u is a univariate conditional Gaussian $p_u(X_u|Z_u)$, we use a $K\times 2$ matrix for parameterizing the layer, where each row stores the mean and standard deviation at each integration point $\tilde{z}_k$.

Next, we address the most important part of this quadrature process, i.e. the materialization of PIC integral units as sum layers. Specifically, let u be an integral unit with N-dimensional function $f_u(\mathbf{Z}_u,\mathbf{Y}_u)$, where $|\mathbf{Z}_u|=N_Z$, $|\mathbf{Y}_u|=N_Y$ and $N=N_Z+N_Y$. We materialize f_u w.r.t. integration points $\tilde{\mathbf{z}}$, effectively creating an N-dimensional tensor $\mathcal{W}^{(u)}\in\mathbb{R}^{K\times\cdots\times K}$, such that

Algorithm 3 PIC2QPC$(c,\tilde{\mathbf{z}},\widetilde{\mathbf{w}})$

Input PIC c, quadrature rule $\tilde{\mathbf{z}},\widetilde{\mathbf{w}}\in\mathbb{R}^K$
Output Tensorized QPC

1: $\mathcal{L}\leftarrow$ map() ▷ from PIC units to layers
2: **for each** unit $u\in$ postOrder(c) **do**
3: **if** u is Ⓐ **then**
4: $\mathcal{L}[u]\leftarrow[\,$Ⓐ$(f_u(X_u,Z_u{=}\tilde{z}_k))\,]_{k=1}^K$
5: **else if** u is Ⓘ(u_1,f_u) **then**
6: $\mathcal{L}[u]\leftarrow\widetilde{\mathbf{W}}^{(u)}\,\mathcal{L}[u_1]$ via Eq. (3)
7: **else if** u is $\bigoplus([u_i,w_i]_{i=1}^N)$ **then**
8: $\mathcal{L}[u]\leftarrow\mathbf{W}^{(u)}||_{i=1}^N\mathcal{L}[u_i]$ via Eq. (4)
9: **else if** u is $\bigotimes([u_1,u_2])$ **then**
10: $\bigcirc\leftarrow\odot$ **if** $\mathbf{Z}_{u_1}=\mathbf{Z}_{u_2}$ **else** $\otimes$
11: $\mathcal{L}[u]\leftarrow\mathcal{L}[u_1]\bigcirc\mathcal{L}[u_2]$
12: **return** A QPC with $\mathcal{L}[c]$ as root layer

$$w^{(u)}_{i_1,\ldots,i_N} = f_u(\{Z_1{=}\tilde{z}_{i_1},\ldots,Z_{N_Z}{=}\tilde{z}_{i_{N_Z}}\},\{Y_1{=}\tilde{z}_{i_{1+N_Z}},\ldots,Y_{N_Y}{=}\tilde{z}_{i_N}\}). \tag{2}$$

After materializing tensor $\mathcal{W}^{(u)}$, we flatten it w.r.t. variables $\mathbf{Z}_u$ and $\mathbf{Y}_u$, so as creating a matrix $\mathbf{W}^{(u)}$ of size $K^{N_Z}\times K^{N_Y}$, an operation aka *matricization*. As last step, we plug-in the quadrature

weights $\widetilde{\mathbf{w}} \in \mathbb{R}^K$ in $\mathbf{W}^{(u)}$, arriving to matrix $\widetilde{\mathbf{W}}^{(u)}$ of $K^{N_Z} \times K^{N_Y}$, whose i-th row is

$$\widetilde{\mathbf{w}}_{i:}^{(u)} = (\widetilde{\mathbf{w}} \otimes \cdots \otimes \widetilde{\mathbf{w}})\, \mathbf{w}_{i:}^{(u)} = \widetilde{\mathbf{w}}^{\otimes N_Y}\, \mathbf{w}_{i:}^{(u)}, \tag{3}$$

where $\widetilde{\mathbf{w}}^{\otimes N_Y}$ is the vector of size K^{N_Y} resulting from the N_Y-times application of the Kronecker product $\otimes$ over $\widetilde{\mathbf{w}}$ (Line 6, Algorithm 3). We illustrate this process in Fig. 3(a-c). Similarly, we also materialize every PIC sum unit u with weights $\{w_i\}_{i=1}^N$ as a sum layer, but parameterized by

$$\mathbf{W}^{(u)} = ||_{i=1}^N w_i\, \mathbf{I}_K \in \mathbb{R}^{K \times NK}, \tag{4}$$

where $\mathbf{I}_K$ is the $K \times K$ identity matrix and $||$ the concatenation operator (Line 8, Algorithm 3). Note that such sum layer can be seen as a *mixing layer* [42, 36]. Finally, consider a PIC product unit u with inputs u_1 and u_2, each outputting functions with LVs Z_{u_1} and Z_{u_2} respectively. We associate to u an Hadamard product layer if $Z_{u_1} = Z_{u_2}$, or a Kronecker product layer if $Z_{u_1} \neq Z_{u_2}$, reflecting the fact that we are marginalizing out two different LVs. We summarize our PIC materialization—illustrated in Fig. 2—in Algorithm 3, where we iteratively associate a PIC unit to a circuit layer, eventually delivering a tensorized QPC. We stress that being QPCs just standard PCs they enjoy their same properties (e.g. tractable marginalization). We will learn PICs via maximizing the likelihood of its QPC materialization.

QPCs as existing tensorized architectures. Materializing PICs built via Algorithm 1 delivers tensorized PCs with alternating sum and product layers, aka ***sum-product layers*** [36]. An instance of such layers is the Tucker layer, used in architectures like RAT-SPNs [43] and EiNets [42]. Specifically, a binary Tucker layer $\boldsymbol{\ell}$ [51] computes

$$\boldsymbol{\ell}(\mathbf{X}_{\boldsymbol{\ell}}) = \mathbf{W}\,(\boldsymbol{\ell}_1(\mathbf{X}_{\boldsymbol{\ell}_1}) \otimes \boldsymbol{\ell}_2(\mathbf{X}_{\boldsymbol{\ell}_2}))\,, \tag{Tucker-layer}$$

where $\mathbf{W} \in \mathbb{R}^{K \times K^2}$ and $\boldsymbol{\ell}_1, \boldsymbol{\ell}_2$ are input layers of $\boldsymbol{\ell}$, each outputting a K-dimensional vector. In contrast, the recent HCLT architectures [31] use the canonical polyadic (CP) layer ℓ [2], i.e.

$$\boldsymbol{\ell}(\mathbf{X}_{\boldsymbol{\ell}}) = \left(\mathbf{W}^{(1)}\boldsymbol{\ell}_1(\mathbf{X}_{\boldsymbol{\ell}_1})\right) \odot \left(\mathbf{W}^{(2)}\boldsymbol{\ell}_2(\mathbf{X}_{\boldsymbol{\ell}_2})\right), \tag{CP-layer}$$

where $\mathbf{W}^{(1)}, \mathbf{W}^{(2)} \in \mathbb{R}^{K \times K}$. We exactly recover Tucker (resp. CP) layers in our QPCs when these are materialized from PICs built via Tucker-merge (resp. CP-merge) in Algorithm 2, and hence the name of the merging procedure. Therefore, some QPCs can exactly match existent tensorized architectures, and this certainly happens when these are materialized from PICs built via Algorithm 1. This gives a new point of view on traditional tensorized architectures, and new possibilities for representation learning [54]. Figure 3 illustrates how the materialization of a 3-variate function leads to a Tucker layer. This 1-to-1 mapping between tensorized PC architectures and QPCs will allow for a fair comparison in our experiments.

Folding tensorized circuits for faster inference. The layers of a tensorized circuit that (i) share the same functional form and that (ii) can be evaluated in parallel, can be stacked together as to create a *folded layer* [36, 42] which speeds up inference and learning on GPU by orders of magnitude. For instance, let $\{\boldsymbol{\ell}_i\}_{i=1}^F$ be F parallelizable Tucker layers each parameterized by a matrix $\mathbf{W}^{(i)}$ of size $K \times K^2$. Such layers can be evaluated as a folded layer $\boldsymbol{\ell}$ parameterized by a tensor $\boldsymbol{\mathcal{W}}$ of size $F \times K \times K^2$, which computes the—otherwise sequential—F tucker layers in parallel. We illustrate folding in Fig. 2(d), and later on in Fig. 4(c). Note that (i) the input layers sharing the same function form can always be folded and that (ii) although a tensorized circuit may have many types of sum-product layers, using one type only is common in practice, and promotes depth-wise folding.

3.3 Scaling PICs with neural functional sharing

Materializing QPCs can be memory intensive and time consuming, depending on: (i) the cost of evaluating the functions we need to materialize, (ii) the degree of parallelization of the required function evaluations, and (iii) the number of integration points K. To solve these issues, we introduce *neural functional sharing* [47], i.e. we share multi-layer perceptrons as to parameterize multiple PIC units at once. This allows us to scale to larger models and datasets, as we make materialization faster and more memory-efficient than previous work [18].

PIC functional sharing. Functional sharing is to PICs as parameter-sharing is to PCs. This type of sharing can be applied over a *group* of input/integral units—grouped according to some criteria—whose functions have all the same number of input and output variables. Specifically, let $\gamma =$

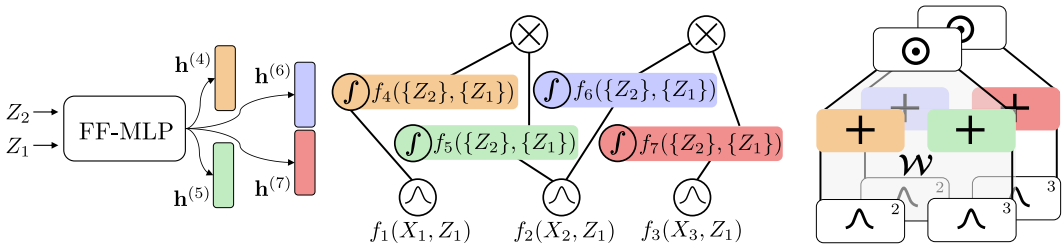


Figure 4: **From neural C-sharing to folded CP-layer** (Section 3.3). We sketch a 4-headed MLP with Fourier-Features (a) which we use to parameterize a group of 4 integral units (at the same depth) of a PIC (b), whose materialization leads to a folded CP-layer parameterized by a tensor $\mathcal{W}$ of size $2 \times 2 \times K \times K$ (c), with K being the number of integration point. Note that, during materialization, the FF-MLP block in (a) will be only evaluated K^2 times, and not $4K^2$.

$\{u_i\}_{i=1}^N$ be a group of N input/integral units, each with function $f_i : \mathbb{R}^I \to \mathbb{R}^O$. The simplest form of functional sharing is to set all functions to be equal, i.e. $\forall i, j \in [N] : f_i = f_j$. In this way, we reduce the number of function evaluations from NK to K as long as we materialize each f_i w.r.t. the same integration points $\tilde{\mathbf{z}} \in \mathbb{R}^K$, which is the case for Algorithm 3. We call this type of sharing **F-sharing**, as per *full-sharing*. More interestingly, leveraging functional composition, we may define $f_i = h_i \circ f$, so as sharing an *inner* function f for all unit functions. Similarly as before, as long as we materialize each f_i w.r.t. the same quadrature points $\tilde{\mathbf{z}} \in \mathbb{R}^K$, we would only need K function evaluations for f instead of NK, as we can share them with all *outer* functions h_i for further evaluation. We call this type of sharing **C-sharing**, as per *composite-sharing*. The original implementation of PICs [18] used neither F-sharing nor C-sharing.

Finally, we present and apply two different ways of grouping units. The first consists of grouping all input units, a technique which is only applicable when all input variables share the same domain. With this grouping, coupled with F-sharing, we would only need to materialize $K \times P$ parameters, and use them to parameterize every QPC input layer. The second consists of grouping all integral units at the same depth of the PIC structure, which we couple with C-sharing and materialize as a folded sum-product layer. Despite grouping units that materialize into a folded layer is a natural and convenient choice, note that we can also group units that do not materialize as such. Once all units in a PIC have been grouped, materialization can be performed per-group.

PIC functional sharing with (multi-headed) MLPs. Similar to [18], we parameterize PIC input and integral units with light-weight multi-layer perceptrons (MLPs). However, instead of using a single MLP for each function, we will apply functional sharing as we strive to make the QPC materialization faster and memory efficient. Specifically, consider a group of integral units $\gamma = \{u_i\}_{i=1}^N$, each with function $f_i : \mathbb{R}^I \to \mathbb{R}$, over which we want to apply functional sharing. For every group γ, we would have an $L+1$ layered MLP of the form:

$$\phi^{(\gamma)} : \mathbb{R}^I \to \mathbb{R}^M := \phi_L^{(\gamma)} \circ \cdots \circ \phi_1^{(\gamma)} \circ \mathsf{FF}, \tag{5}$$

where $\mathsf{FF} : \mathbb{R}^I \to \mathbb{R}^M$ is a Fourier-feature layer [48], and each $\phi_i^{(\gamma)} : \mathbb{R}^M \to \mathbb{R}^M$ is a standard linear layer followed by an element-wise non linearity ψ, i.e. $\psi(\mathbf{A}\mathbf{z} + \mathbf{b})$, with $\mathbf{A} \in \mathbb{R}^{M \times M}, \mathbf{b} \in \mathbb{R}^M$, and M being the size of the MLP. Applying F-sharing over γ would simply consist of setting

$$f_i : \mathbb{R}^M \to \mathbb{R} := \mathsf{softplus}(\mathbf{h}^{(\gamma)} \cdot \phi^{(\gamma)} + b^{(\gamma)}), \qquad \text{(neural F-sharing)}$$

where $\mathbf{h}^{(\gamma)} \in \mathbb{R}^M$ and $b^{(\gamma)} \in \mathbb{R}$ are group-dependent parameters, therefore making all functions in the group equal. Instead, to implement C-sharing, we parameterize each f_i as

$$f_i : \mathbb{R}^M \to \mathbb{R} := \mathsf{softplus}(\mathbf{h}^{(i)} \cdot \phi^{(\gamma)} + b^{(i)}), \qquad \text{(neural C-sharing)}$$

where $\mathbf{h}^{(i)} \in \mathbb{R}^M$ and $b^{(i)} \in \mathbb{R}$ are function-dependent parameters, effectively creating a multi-headed MLP. As an example, consider a folded CP-layer with $F = 500$ and $K = 512$—which we actually used in practice—resulting in $2FK^2 \approx 262$M trainable parameters. Assuming no bias term, an MLP with $L = 2$ and $M = 256$ would only instead require $LM^2 + 2FM \approx 387$K trainable parameters to materialize the same tensor, resulting in more than 99% less trainable parameters. We illustrate such C-sharing in Fig. 4. In Appendix C.1 we provide more details about our MLPs.

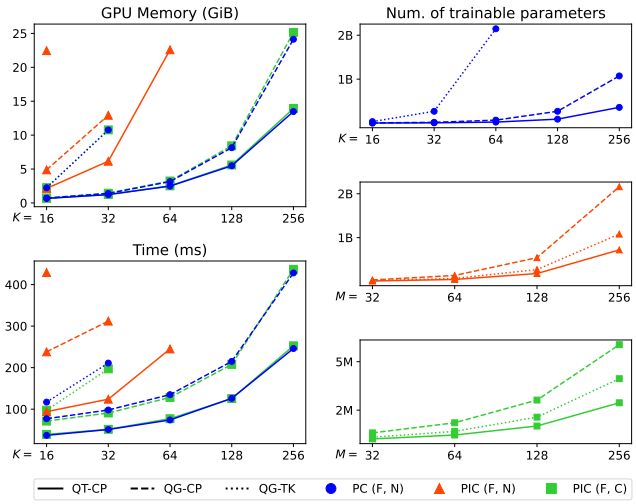


Figure 5: **Learning PICs using functional sharing requires (i) comparable resources as PCs and (ii) up to 99% less trainable parameters.** We compare the GPU memory (top-left) and time (bottom-left) required to perform an optimization step with PCs (●), PICs with functional sharing (■), and without (▲), while considering three different architectures (QT-CP, QG-CP, QG-TK). To the right, we report the number of trainable parameters for (i) PCs (●) at different K, and (ii) for PICs (▲, ■) at different MLP sizes M. The isolated ▲ nodes refer to refer to PIC (F, N) with QG-TK which we could only run at $K = 16$. The benchmark is conducted using a batch of 128 RGB images of size 64x64 and Adam [23]. Extra details in Appendix D.1.

Fast & memory-efficient QPC materialization. Combing PIC functional sharing and per-group materialization allows scaling the training of PICs via numerical quadrature, as we drastically reduce the number and the cost of function evaluations required for the QPC materialization. We can now materialize very large QPCs, matching the scale of recent over-parameterized PCs yet requiring up to 99% less trainable parameters when using a large K. This was not possible in the original formulation of PIC [18] as (i) the entire QPC was materialized in one-shot, not per-group, and (ii) no functional sharing was implemented, as each input/integral function had its own MLP.

4 Experiments

In our experiments, we first benchmark the effectiveness of functional sharing for scaling the training of PICs via numerical quadrature, comparing it with standard PCs and PICs w/o functional sharing [18]. Then, following prior work [10, 32, 33, 18], we compare QPCs and PCs as distribution estimators on several image datasets. We always train using the trapezoidal integration rule. We use an NVIDIA A100 40GB throughout our experiments. Our code is available at github.com/gengala/ten-pics.

Thanks to our pipeline, we can now use two recently introduced RGs tailored for image data which deliver architectures that scale better than those built out of classical RGs [45, 43, 31]: *quad-trees* (QTs), tree-shaped RGs, and *quad-graphs* (QGs), DAG-shaped RGs [36]. These are perfectly balanced RGs, and therefore applying Algorithm 1 over them would deliver balanced PIC structures amenable to depth-wise C-sharing of integral units. We report full details about QTs and QGs in Appendix B. We denote a tensorized architecture as [RG]-[sum-product layer]-[K], e.g. QT-CP-16, which can be trained as a standard PC or materialized as QPC from a PIC. We treat pixels as categorical variables, and, as such, our architectures model probability mass functions.

Scaling PICs. For each model type, $\{\text{PC}, \text{PIC}\}$, we specify a pair $(\cdot, \cdot)$ where the first (resp. second) argument specifies the sharing technique, $\{\text{F}, \text{C}, \text{N}\}$, for the input (resp. inner) layers/groups, where N stands for *no sharing*. In Fig. 5, we report the time and GPU memory required to perform an Adam [23] optimization step using PCs (●), and PICs with (■) and without (▲) functional C-sharing over the integral unit groups. We note that PICs using functional sharing (■) proves very effective for scaling, requiring comparable resources as standard PCs (●), while those who do not (▲)—like prior work [18]—are orders of magnitude slower and quickly go Out-Of-Memory (OOM) for $K > 64$. Remarkably, some QG-TK configurations of PICs (■), see Table D.2, require even less GPU memory than PCs, and this is because of the significant difference in the number of trainable parameters, since copies of these have to be stored by Adam during optimization. In fact, the number of parameters for PCs and PICs w/o functional sharing (●, ▲) is in the order of hundreds of millions

	QPC	PC	Sp-PC	HCLT	RAT	IDF	BitS	BBans	McB
MNIST	**1.11**	1.17	1.14	1.21	1.67	1.90	1.27	1.39	1.98
F-MNIST	**3.16**	3.32	3.27	3.34	4.29	3.47	3.28	3.66	3.72
EMN-MN	1.55	1.64	**1.52**	1.70	2.56	2.07	1.88	2.04	2.19
EMN-LE	**1.54**	1.62	1.58	1.75	2.73	1.95	1.84	2.26	3.12
EMN-BA	**1.59**	1.66	1.60	1.78	2.78	2.15	1.96	2.23	2.88
EMN-BY	1.53	**1.47**	1.54	1.73	2.72	1.98	1.87	2.23	3.14

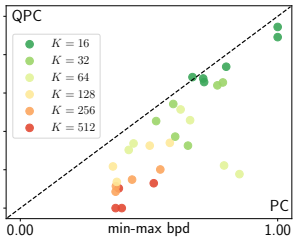


Figure 6: **QPCs improve over PCs and other DGM baselines** in terms of test-set bpd for the MNIST-family datasets. We compare against SparsePC [10], HCLT [31], RAT-SPN [43], IDF [21], BitSwap [25], BBans [50] and McBits [46]. HCLT results are taken from [18]. Columns QPC and PC report results from this paper, with QG-CP-512 being the best performing architecture for both. Scatter plot (right): bpd for QPCs (y-axis) and PCs (x-axis) paired by architecture and min-max normalized for the MNIST and F-MNIST datasets. A point below the diagonal is a win for QPCs.

	QPC*	PC*	QPC†	PC†	HCLT†	LVD†	LVD-PG†
CIFAR	**5.09**	5.50	**4.48**	4.85	4.61	4.37	3.87
ImgNet32	**5.08**	5.25	**4.46**	4.63	4.82	4.38	4.06
ImgNet64	**5.05**	5.22	**4.42**	4.59	4.67	4.12	3.80
CelebA	**4.73**	4.78	**4.11**	4.16	-	-	-

Table 1: **QPCs improve over PCs.** We mark results with * for YCoCg-R and † for YCoCg. QG-CP-512 (resp. QG-CP-256) is the best performing architecture for QPCs and PCs on CIFAR & ImgNet32 (resp. ImgNet64 & CelebA).

(hitting 2B+), while PICs with functional sharing (■) scale much more gracefully, hitting only 6M+ parameters. We emphasize that the number of trainable parameters of PICs is independent of the K at which we materialize, but only dependent on the size of the MLPs M we use to parameterize them, which can also be thought as the cost of evaluating PIC functions. We report more (tabular) details in Appendix D.1.

Distribution estimation. Following prior work [10, 32, 33, 18], we extensively test QPCs and PCs as distribution estimators on standard image datasets. Our full results are in Appendix D.2, while we only report here the bits-per-dimension (bpd) of the best performing models, which *always belong to a QG-CP architecture, reflecting the additional expressiveness of DAG-shaped RGs.* Our full results also highlights how the the more expressive yet expensive Tucker layers we introduced for PICs deliver the best performance for small K, but are hard to scale. All QPCs are materialized from PICs applying F-sharing over input units and C-sharing over groups of integral units, i.e. PIC (F, C). We begin with the MNIST-family, which includes 6 datasets of gray-scale 28x28 images: MNIST [29], FASHIONMNIST [55], and EMNIST with its 4 splits [7]. Fig. 6 shows that QPCs generally perform best, improving over standard PCs (5/6), complex heuristic-based PC learning schemes as pruning-and-growing (5/6) [10], and some deep generative models (DGMs) (6/6).

Then, we move to larger RGB image datasets as CIFAR [28], ImageNet32, ImageNet64 [14], and CelebA [34]. To compare against prior work [32, 33], we have to preprocess the datasets using the YCoCg transform, a lossy color-coding that consistently improves performance for PCs when applied to RGB images.[3] We also report results over datasets preprocessed with the lossless YCoCg-R transform [39], effectively doubling the number of datasets. We report details about these transforms in Appendix C.3. From Table 1, we see that QPCs prove again very competitive, consistently outperforming standard PCs commonly trained with Adam, and the best performing PC from the literature, HCLT, which is trained via EM schemes and patch-wise methods [32]. Furthermore, QPCs are close to PCs trained via latent variable distillation (LVD, LVD-PG in Table 1) [32, 33], a framework that requires extra supervision over their latent spaces by distilling information from existing deep generative models (DGMs). This technique requires pre-trained DGMs, several heuristics, and a final fine-tuning stage via EM or SGD, while PIC training method is instead end-to-end and self-contained.

[3]The use of the lossy YCoCg transform is undocumented in [32, 33] but confirmed via personal communication with the authors. Note that columns in Table 1 using it (†) are not directly comparable to the rest.

5 Discussion & Conclusion

With this work, we systematized the construction of PICs, extending them to DAG-like structures (Section 3.1), tensorizing (Section 3.2), and scaling their training with functional sharing (Section 3.3). In our experiments (Section 4), we showed how this pipeline is remarkably effective when the tractable approximations of PICs, QPCs, are used as distribution estimators. This in turn becomes a new and effective tool to learn PCs at scale. In fact, prior work has shown that naively training large PCs via EM or gradient-ascent is challenging, and that PC performance plateau as their size increases [10, 8, 32, 33]. Our contributions go beyond these limitations, while offering a simple, principled and fully-differentiable pipeline that delivers performance that rival more sophisticated alternatives [10, 32] (Section 4). We conjecture that this happens as training PCs via PICs drastically reduces the search space while allowing (i) smoother training dynamics and (ii) the materialization of arbitrarily large tractable models.

The development of tractable models is an important task in machine learning as they provide many inference routines, and can be used in many down-stream applications such as tabular data modeling [8], generative modeling [42], lossless compression [30], genetics [11], knowledge-graphs [37], constrained text generation [58], and more. Our work has also certain parallels with tensor networks and (quasi-)tensor decompositions [51, 26, 22, 6, 49], as [36] recently showed how hierarchical tensor decompositions can be represented using the language of tensorized circuits. Furthermore, we note that the recent non-monotonic PCs [35] (i.e., PCs with negative sum parameters) can also be thought as the result of a quadrature process from PICs whose function can return negative values.

Our work does not come without limitations. Although we showed that training PICs with function sharing requires comparable resources as standard PCs, traditional continuous LV models as VAEs, flows and diffusion models are more scalable. Also, sampling from PICs is currently not possible, as we cannot perform differentiable sampling from our (multi-headed) MLPs. Future work may include the investigation of more efficient ways of training PICs, possibly using techniques as LVD or variational inference [24] to directly maximize PIC lower-bounds, requiring numerical quadrature only as fine-tuning step to distill a performant tractable model. We believe our work will foster new research in the field of generative modeling, and specifically in the realm of tractable models.

Author Contributions

GG led the project, proposed neural functional sharing, and ran all experiments. GG and AV devised the original idea of a pipeline to build PICs, and leverage tensorized folded circuits for training them. AV and EQ equally supervised all the phases of the project. CdC supervises the project, and critically read the manuscript and provided feedback.

Acknowledgments

The Eindhoven University of Technology authors thank the support from the Eindhoven Artificial Intelligence Systems Institute and the Department of Mathematics and Computer Science of TU Eindhoven. TU/e authors also thank the support of EU European Defence Fund Project KOIOS (EDF-2021-DIGIT-R-FL-KOIOS). AV was supported by the "UNREAL: Unified Reasoning Layer for Trustworthy ML" project (EP/Y023838/1) selected by the ERC and funded by UKRI EPSRC. We thank Lorenzo Loconte for insightful discussions about (quasi-)tensor decompositions.

References

[1] Cory Butz, Jhonatan S Oliveira, and Robert Peharz. Sum-product network decompilation. In *International Conference on Probabilistic Graphical Models*, pages 53–64. PMLR, 2020.

[2] J. Douglas Carroll and Jih-Jie Chang. Analysis of individual differences in multidimensional scaling via an n-way generalization of "eckart-young" decomposition. *Psychometrika*, 35:283–319, 1970.

[3] Myung Jin Choi, Vincent Y. F. Tan, Animashree Anandkumar, and Alan S Willsky. Learning latent tree graphical models. *Journal of Machine Learning Research*, 12(49):1771–1812, 2011.

[4] YooJung Choi, Antonio Vergari, and Guy Van den Broeck. Probabilistic circuits: A unifying framework for tractable probabilistic models. Technical report, UCLA, 2020.

[5] C. Chow and C. Liu. Approximating discrete probability distributions with dependence trees. *IEEE Transactions on Information Theory*, 14(3):462–467, 1968.

[6] Andrzej Cichocki and Anh Huy Phan. Fast local algorithms for large scale nonnegative matrix and tensor factorizations. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 92-A(3):708–721, 2009.

[7] Gregory Cohen, Saeed Afshar, Jonathan Tapson, and André van Schaik. EMNIST: Extending MNIST to handwritten letters. In *IJCNN 2017*, pages 2921–2926, 2017.

[8] Alvaro Correia, Robert Peharz, and Cassio P de Campos. Joints in random forests. In *Advances in Neural Information Processing Systems*, volume 33, pages 11404–11415, 2020.

[9] Alvaro H. C. Correia, Gennaro Gala, Erik Quaeghebeur, Cassio de Campos, and Robert Peharz. Continuous mixtures of tractable probabilistic models. *Proceedings of the AAAI Conference on Artificial Intelligence*, 37(6):7244–7252, 2023.

[10] Meihua Dang, Anji Liu, and Guy Van den Broeck. Sparse probabilistic circuits via pruning and growing. In *NeurIPS 2022*, volume 35 of *Advances in Neural Information Processing Systems*, 2022.

[11] Meihua Dang, Anji Liu, Xinzhu Wei, Sriram Sankararaman, and Guy Van den Broeck. Tractable and expressive generative models of genetic variation data. In *Research in Computational Molecular Biology*, pages 356–357, 2022.

[12] Adnan Darwiche. *Modeling and reasoning with Bayesian networks*. Cambridge University Press, 2009.

[13] Philip J. Davis and Philip Rabinowitz. *Methods of numerical integration*. Academic Press, 1984.

[14] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[15] Aaron W. Dennis and Dan Ventura. Learning the architecture of sum-product networks using clustering on variables. In *Advances in Neural Information Processing Systems 25 (NeurIPS)*, pages 2033–2041. Curran Associates, Inc., 2012.

[16] Nicola Di Mauro, Gennaro Gala, Marco Iannotta, and Teresa Maria Altomare Basile. Random probabilistic circuits. In *37th Conference on Uncertainty in Artificial Intelligence (UAI)*, volume 161, pages 1682–1691. PMLR, 2021.

[17] Guido Fubini. Sugli integrali multipli. *Rend. Acc. Naz. Lincei*, 16:608–614, 1907.

[18] Gennaro Gala, Cassio de Campos, Robert Peharz, Antonio Vergari, and Erik Quaeghebeur. Probabilistic integral circuits. In *Proceedings of The 27th International Conference on Artificial Intelligence and Statistics*, volume 238 of *Proceedings of Machine Learning Research*, pages 2143–2151. PMLR, 02–04 May 2024.

[19] Robert Gens and Pedro M. Domingos. Learning the structure of sum-product networks. In *International Conference on Machine Learning*, 2013.

[20] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *NIPS 2014*, volume 27 of *Advances in Neural Information Processing Systems*, 2014.

[21] Emiel Hoogeboom, Jorn Peters, Rianne van den Berg, and Max Welling. Integer discrete flows and lossless compression. In *NeurIPS 2019*, volume 32 of *Advances in Neural Information Processing Systems*, 2019.

[22] Yong-Deok Kim and Seungjin Choi. Nonnegative tucker decomposition. In *CVPR*. IEEE Computer Society, 2007.

[23] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR 2015*, 2015.

[24] Diederik P. Kingma and Max Welling. Auto-encoding variational Bayes. In *ICLR 2014*, 2014.

[25] Friso Kingma, Pieter Abbeel, and Jonathan Ho. Bit-swap: Recursive bits-back coding for lossless compression with hierarchical latent variables. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 3408–3417, 2019.

[26] Tamara G. Kolda. Multilinear operators for higher-order decompositions. Technical report, Sandia National Laboratories, 2006.

[27] Daphne Koller and Nir Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[28] Alex Krizhevsky. Learning multiple layers of features from tiny images, 2009.

[29] Yann LeCun, Corinna Cortes, and Christopher J. C. Burges. The MNIST database of handwritten digits, 2010.

[30] Anji Liu, Stephan Mandt, and Guy Van den Broeck. Lossless compression with probabilistic circuits. In *ICLR 2022*, 2022.

[31] Anji Liu and Guy Van den Broeck. Tractable regularization of probabilistic circuits. In *NeurIPS 2021*, volume 34 of *Advances in Neural Information Processing Systems*, pages 3558–3570, 2021.

[32] Anji Liu, Honghua Zhang, and Guy Van den Broeck. Scaling up probabilistic circuits by latent variable distillation. In *ICLR 2023*, 2023.

[33] Xuejie Liu, Anji Liu, Guy Van den Broeck, and Yitao Liang. Understanding the distillation process from deep generative models to tractable probabilistic circuits. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 21825–21838, 2023.

[34] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep learning face attributes in the wild. In *Proceedings of International Conference on Computer Vision (ICCV)*, December 2015.

[35] Lorenzo Loconte, M. Sladek Aleksanteri, Stefan Mengel, Martin Trapp, Arno Solin, Nicolas Gillis, and Antonio Vergari. Subtractive mixture models via squaring: Representation and learning. In *The Twelfth International Conference on Learning Representations (ICLR)*, 2024.

[36] Lorenzo Loconte, Antonio Mari, Gennaro Gala, Robert Peharz, Cassio de Campos, Erik Quaeghebeur, Gennaro Vessio, and Antonio Vergari. What is the relationship between tensor factorizations and circuits (and how can we exploit it)? *arXiv preprint arXiv:2409.07953*, 2024.

[37] Lorenzo Loconte, Nicola Di Mauro, Robert Peharz, and Antonio Vergari. How to turn your knowledge graph embeddings into generative models via probabilistic circuits. In *Advances in Neural Information Processing Systems 37 (NeurIPS)*. Curran Associates, Inc., 2023.

[38] Ilya Loshchilov and Frank Hutter. SGDR: Stochastic gradient descent with warm restarts. In *ICLR 2017*, 2017.

[39] Henrique Malvar and Gary Sullivan. Ycocg-r: A color space with rgb reversibility and low dynamic range. *ISO/IEC JTC1/SC29/WG11 and ITU-T SG16 Q*, 6, 2003.

[40] Alejandro Molina, Antonio Vergari, Nicola Di Mauro, Sriraam Natarajan, Floriana Esposito, and Kristian Kersting. Mixed sum-product networks: A deep architecture for hybrid domains. In *AAAI Conference on Artificial Intelligence*, 2018.

[41] 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, 2017.

[42] 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 *37th International Conference on Machine Learning (ICML)*, volume 119 of *Proceedings of Machine Learning Research*, pages 7563–7574. PMLR, 2020.

[43] 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 *Proceedings of The 35th Uncertainty in Artificial Intelligence Conference*, volume 115 of *Proceedings of Machine Learning Research*, pages 334–344, 2020.

[44] Knot Pipatsrisawat and Adnan Darwiche. New compilation languages based on structured decomposability. In *Proceedings of the 23rd National Conference on Artificial Intelligence (AAAI'08)*, volume 1, pages 517–522, 2008.

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

[46] Yangjun Ruan, Karen Ullrich, Daniel S. Severo, James Townsend, Ashish Khisti, Arnaud Doucet, Alireza Makhzani, and Chris Maddison. Improving lossless compression rates via monte carlo bits-back coding. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 9136–9147, 2021.

[47] Eran Segal, Dana Pe'er, Aviv Regev, Daphne Koller, Nir Friedman, and Tommi Jaakkola. Learning module networks. *Journal of Machine Learning Research*, 6(4), 2005.

[48] Matthew Tancik, Pratul Srinivasan, Ben Mildenhall, Sara Fridovich-Keil, Nithin Raghavan, Utkarsh Singhal, Ravi Ramamoorthi, Jonathan Barron, and Ren Ng. Fourier features let networks learn high frequency functions in low dimensional domains. In *NeurIPS 2020*, volume 33 of *Advances in Neural Information Processing Systems*, pages 7537–7547, 2020.

[49] Alex Townsend and Lloyd N Trefethen. Continuous analogues of matrix factorizations. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 471(2173):20140585, 2015.

[50] James Townsend, Thomas Bird, and David Barber. Practical lossless compression with latent variables using bits back coding. In *ICLR 2019*, 2019.

[51] L. R. Tucker. The extension of factor analysis to three-dimensional matrices. In *Contributions to mathematical psychology.*, pages 110–127. Holt, Rinehart and Winston, 1964.

[52] Antonio Vergari, YooJung Choi, Anji Liu, Stefano Teso, and Guy Van den Broeck. A compositional atlas of tractable circuit operations for probabilistic inference. In *NeurIPS 2021*, volume 36 of *Advances in Neural Information Processing Systems*, 2021.

[53] Antonio Vergari, Nicola Di Mauro, and Guy Van den Broeck. Tractable probabilistic models: Representations, algorithms, learning, and applications, 2019. Tutorial at the 35th Conference on Uncertainty in Artificial Intelligence (UAI 2019).

[54] 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. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), 2018.

[55] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms. *arXiv*, 2017.

[56] Ling Yang, Zhilong Zhang, Yang Song, Shenda Hong, Runsheng Xu, Yue Zhao, Wentao Zhang, Bin Cui, and Ming-Hsuan Yang. Diffusion models: A comprehensive survey of methods and applications. *ACM Computing Surveys*, 56(4):1–39, 2023.

[57] Yang Yang, Gennaro Gala, and Robert Peharz. Bayesian structure scores for probabilistic circuits. In *Proceedings of The 26th International Conference on Artificial Intelligence and Statistics*, volume 206 of *Proceedings of Machine Learning Research*, pages 563–575, 2023.

[58] Honghua Zhang, Meihua Dang, Nanyun Peng, and Guy Van den Broeck. Tractable control for autoregressive language generation. In *40th International Conference on Machine Learning (ICML)*, volume 202 of *Proceedings of Machine Learning Research*, pages 40932–40945. PMLR, 2023.

A Background on Circuits

Definition 3 (Circuit [4, 52]). *A circuit c over variables $\mathbf{X}$ is a parameterized computational graph encoding a function $c(\mathbf{X})$, and comprising three kinds of computational units: input, product, and sum. Each product or sum unit u outputs a scalar and receives as inputs the output scalars of other units, denoted with the set $\mathsf{in}(u)$. Each unit u computes a function f_u defined as: (i) $f_u(\mathbf{X}_u) \to \mathbb{R}$ if u is an input unit, where f_u is a function over variables $\mathbf{X}_u \subseteq \mathbf{X}$, called its scope, (ii) $\prod_{i \in \mathsf{in}(u)} f_i(\mathbf{X}_i)$ if u is a product unit, and (iii) $\sum_{i \in \mathsf{in}(u)} f_i(\mathbf{X}_i)$ if u is a sum unit, with $w_i \in \mathbb{R}$ denoting the weighted sum parameters. The scope of a product or sum unit is the union of the scopes of its input units.*

Definition 4 (Probabilistic Circuit). *A PC over variables $\mathbf{X}$ is a circuit c encoding a (possibly non-normalized) distribution, e.g., a function that is non-negative for all values of $\mathbf{X}$:*

$$c(\mathbf{x}) \geq 0, \quad \forall \mathbf{x} \in \mathsf{dom}(\mathbf{X})$$

Definition 5 (Smoothness). *A circuit is smooth if, for each sum unit u, its inputs depend on the same variables: $\forall u_1, u_2 \in \mathsf{in}(u), \mathbf{X}_{u_1} = \mathbf{X}_{u_2}$.*

Definition 6 (Decomposability). *A circuit is decomposable if the inputs of each product unit u depend on disjoint sets of variables: $\forall u_1, u_2 \in \mathsf{in}(u), \mathbf{X}_{u_1} \neq \mathbf{X}_{u_2}$.*

Definition 7 (Structured-decomposability [44, 12]). *A circuit is* structured-decomposable *if (i) it is smooth and decomposable, and (2) any pair of product units having the same scope decompose their scope at their input units in the same way.*

Although all tensorized architectures mentioned in this paper are smooth and decomposable, only PCs built from tree RGs are also structured-decomposable, and as such are potentially less expressive because they belong to a restricted class.

B Region Graphs

In Algorithm B.1 we detail the construction of the Quad-Tree (QT) and Quad-Graph (QG) region graphs [36]. Specifically, QTs (resp. QGs) are built setting the input flag isTree to True (resp. False). Intuitively, these RGs recursively split an image into patches, until reaching regions associated to exactly one pixel. The splitting is performed both horizontally and vertically, and subsequent patches can either be shared, thus yielding a RG that is not a tree (QGs), or not (QTs).

Algorithm B.1 $\mathsf{buildQuadGraph}(H, W, \mathsf{isTree})$

Input: Image height H, image width W, and whether to enforce the output RG to be a tree.
Output: A RG $\mathcal{R}$ over $H \cdot W$ variables

$\mathsf{S} \leftarrow \{\mathbb{X}_{ij} = \{X_{ij}\} \mid (i,j) \in [H] \times [W]\}$
$\mathcal{R} \leftarrow$ a RG with leaf regions S
$h \leftarrow H;\ w \leftarrow W$
while $h > 1 \vee w > 1$ **do**
  $h \leftarrow \lceil h/2 \rceil;\ w \leftarrow \lceil w/2 \rceil;\ \mathsf{S}' \leftarrow \varnothing$
  for $i, j \in [h] \times [w]$ **do**
    $\Omega \leftarrow (\{2i-1, 2i\} \times \{2j-1, 2j\}) \cap ([H] \times [W])$
    if $|\Omega| = 1$ **then**
      Let $\mathbb{X}_{pq} \in \mathsf{S}$ s.t. $(p,q) \in \Omega$
      $\mathsf{addRegion}(\mathcal{R}, \mathbb{X}_{pq})$
    else if $|\Omega| = 2$ **then**
      Let $\mathbb{X}_{pq}, \mathbb{X}_{rs} \in \mathsf{S}$ s.t.
        $(p,q), (r,s) \in \Omega, \quad p < r, q < s$
      $\mathsf{addPartition}(\mathcal{R}, \mathbb{X}_{pq} \cup \mathbb{X}_{rs}, \{\mathbb{X}_{pq}, \mathbb{X}_{rs}\})$
    else $\triangleright |\Omega| = 4$
      if isTree **then** $\mathsf{mergeTree}(\mathcal{R}, \Omega, \mathsf{S})$
      else $\mathsf{mergeDAG}(\mathcal{R}, \Omega, \mathsf{S})$
    $\mathbb{X}_{ij} \leftarrow \bigcup_{(r,s) \in \Omega} \mathbb{X}_{rs}$ s.t. $\mathbb{X}_{rs} \in \mathsf{S}$
    $\mathsf{S}' \leftarrow \mathsf{S}' \cup \{\mathbb{X}_{ij}\}$
  $\mathsf{S} \leftarrow \mathsf{S}'$
return $\mathcal{R}$

Algorithm B.2 $\mathsf{mergeTree}(\mathcal{R}, \Omega, \mathsf{S})$

Input: A RG $\mathcal{R}$, a set of four coordinates Ω, and a set of regions S
Behavior: It merges the regions indexed by Ω in $\mathcal{R}$ by forming a tree structure

Let $\mathbb{X}_{uv} = \mathbb{Y}_{p+u\ q+v} \in \mathsf{S}$ s.t.
  $(p+u, q+v) \in \Omega, \quad u, v \in \{0, 1\}$
$\mathbb{X} \leftarrow \mathbb{X}_{00} \cup \mathbb{X}_{01} \cup \mathbb{X}_{10} \cup \mathbb{X}_{11}$
$\mathsf{addPartition}(\mathcal{R}, \mathbb{X}, \{\mathbb{X}_{00}, \mathbb{X}_{01}, \mathbb{X}_{10}, \mathbb{X}_{11}\})$

Algorithm B.3 $\mathsf{mergeDAG}(\mathcal{R}, \Omega, \mathsf{S})$

Input: A RG $\mathcal{R}$, a set of four coordinates Ω, and a set of regions S
Behavior: It merges the regions indexed by Ω in $\mathcal{R}$ by forming a DAG structure

Let $\mathbb{X}_{uv} = \mathbb{Y}_{p+u\ q+v} \in \mathsf{S}$ s.t.
  $(p+u, q+v) \in \Omega, \quad u, v \in \{0, 1\}$
$\mathbb{X} \leftarrow \mathbb{X}_{00} \cup \mathbb{X}_{01} \cup \mathbb{X}_{10} \cup \mathbb{X}_{11}$
$\mathsf{addPartition}(\mathcal{R}, \mathbb{X}, \{\mathbb{X}_{00} \cup \mathbb{X}_{01}, \mathbb{X}_{10} \cup \mathbb{X}_{11}\})$
$\mathsf{addPartition}(\mathcal{R}, \mathbb{X}, \{\mathbb{X}_{00} \cup \mathbb{X}_{10}, \mathbb{X}_{01} \cup \mathbb{X}_{11}\})$
$\mathsf{addPartition}(\mathcal{R}, \mathbb{X}_{00} \cup \mathbb{X}_{01}, \{\mathbb{X}_{00}, \mathbb{X}_{01}\})$
$\mathsf{addPartition}(\mathcal{R}, \mathbb{X}_{10} \cup \mathbb{X}_{11}, \{\mathbb{X}_{10}, \mathbb{X}_{11}\})$
$\mathsf{addPartition}(\mathcal{R}, \mathbb{X}_{00} \cup \mathbb{X}_{10}, \{\mathbb{X}_{00}, \mathbb{X}_{10}\})$
$\mathsf{addPartition}(\mathcal{R}, \mathbb{X}_{01} \cup \mathbb{X}_{11}, \{\mathbb{X}_{01}, \mathbb{X}_{11}\})$

C Implementation details

C.1 Multi-headed MLP details

A multi-headed MLP of size M parameterizing a group $\gamma = \{u_i\}_{i=1}^{N}$ of N PIC units with functions of the form $\mathbb{R}^I \to \mathbb{R}^O$ consists of:

1. A Fourier-Features Layer (details below), i.e. a non-linear mapping $\mathbb{R}^I \to \mathbb{R}^M$;
2. Two linear layers followed by hyperbolic tangent as activation function, i.e. two consecutive non-linear mappings $\mathbb{R}^M \to \mathbb{R}^M$;
3. N heads with Softplus non-linearity, i.e. N different non-linear mapping $\mathbb{R}^M \to \mathbb{R}^O$.

Note that, if γ is a group of CP (resp. Tucker) integral units, then $I = 2$ (resp. $I = 3$), while the output dimension O is always equal to 1. Instead, if the group γ is a group of input units, the input dimension I is always equal to 1, while the output dimension O is equal to the number of required parameters of the specific distribution, e.g. $O = 2$ for Gaussians.

Such multi-headed MLP is implemented using grouped 1D convolutions, which allow a one-shot materialization of all the layer parameters associated to the group. We found that initializing all the heads to be equal improves convergence.

Fourier Feature Layer. Fourier Feature Layers (FFLs) are an important ingredient for the multi-headed MLPs. FFLs [48] enable MLPs to learn high-frequency functions in low-dimensional problem domains and are usually used as first layers of coordinate-based MLPs. FFLs transform input $\mathbf{z} \in \mathbb{R}^I$ to

$$\mathsf{FFL}(\mathbf{z}) : \mathbb{R}^I \to \mathbb{R}^M := [\cos(2\pi \mathbf{f}_1^\intercal \mathbf{z}), \sin(2\pi \mathbf{f}_1^\intercal \mathbf{z}), \dots, \cos(2\pi \mathbf{f}_{M/2}^\intercal \mathbf{z}), \sin(2\pi \mathbf{f}_{M/2}^\intercal \mathbf{z})],$$

where M is a hyper-parameter and vectors $\mathbf{f}_i \in \mathbb{R}^I$ are non-learnable, randomly initialized parameters. FFLs have two main benefits: (i) They allow learning more expressive functions by avoiding over-smoothing behaviors, and (ii) they reduce the total number of trainable parameters when used instead of conventional linear layers as the initial layers in MLPs.

C.2 Training details

We train both PICs and PCs using the same training setup. Specifically, for each dataset, we perform a training cycle of T optimization steps, after which we perform a validation step and stop training if the validation log-likelihood did not improve by δ nats after 5 training cycles. Using $\delta > 0$ can avoid long trainings with negligible improvements. We report these common training hyper-parameters in Table C.1. We use Adam [23] and a batch size of 256 for all experiments.

PIC training. After some preliminary runs, we found that a learning rate of 5×10^{-3} worked best, which we annealed towards 10^{-4} using cosine annealing with warm restarts across 500 optimization steps [38]. We also apply weight decay with $\lambda = 0.01$.

PC training. After some preliminary runs, we found that a constant learning rate of 0.01 worked best for all PC models, and for all datasets. We keep the PC parameters unnormalized, and, as such, we clamp them to a small positive value (10^{-19}) after each Adam update to keep them non-negative, and subtract the log normalization constant to normalize the log-likelihoods.

Table C.1: Common training hyper-parameters for PICs and PCs.

dataset	max num epochs	T	δ
MNIST-family excl. EMNIST-BY	200	250	0
EMNIST-BY	100	1000	0
CIFAR	200	250	0
ImageNet32	50	2000	10
ImageNet64	50	2000	30
CelebA	200	750	10

C.3 YCoCg color-coding transforms

In Fig. C.1 and Fig. C.2 we provide pytorch code for the lossless and lossy versions of the YCoCg transform that we used in our experiments (Section 4). In Fig. C.3, we show how to apply them and that the lossy version is on average off less than a bit. Finally, in Fig. C.4 we show the significant visual difference of the two transforms when applied to an RGB image.

```python
def rgb2ycc_lossless(
    rgb_img: torch.Tensor
):
    assert rgb_img.size(-1) == 3

    def forward_lift(x, y):
        diff = (y - x) % 256
        average = (x + (diff >> 1)) % 256
        return average, diff

    red = rgb_img[..., 0]
    green = rgb_img[..., 1]
    blue = rgb_img[..., 2]

    temp, co = forward_lift(red, blue)
    y, cg = forward_lift(green, temp)
    ycc_img = torch.stack(
        [y, co, cg], dim=-1
    )
    return ycc_img
```

```python
def ycc2rgb_lossless(
    ycc_img: torch.Tensor
):
    assert ycc_img.size(-1) == 3

    def reverse_lift(average, diff):
        x = (average - (diff >> 1)) % 256
        y = (x + diff) % 256
        return x, y

    y = ycc_img[..., 0]
    co = ycc_img[..., 1]
    cg = ycc_img[..., 2]

    green, temp = reverse_lift(y, cg)
    red, blue = reverse_lift(temp, co)
    rgb_img = torch.stack(
        [red, green, blue], dim=-1
    )
    return rgb_img
```

Figure C.1: **Lossless YCoCg transform (aka YCoCg-R [39]).** We attach pytorch code for the RGB $\rightarrow$ YCoCg direction (left) and YCoCg $\rightarrow$ RGB direction (right), where one is the inverse of the other. The input to both functions is a tensor with discrete values in $[0, 255]$, so their output.

```python
def rgb2ycc_lossy(
    rgb_img: torch.Tensor
):
    assert rgb_img.size(-1) == 3

    deq_img = (rgb_img / 127.5) - 1
    red = (deq_img[..., 0] + 1) / 2
    green = (deq_img[..., 1] + 1) / 2
    blue = (deq_img[..., 2] + 1) / 2

    co = red - blue
    tmp = blue + co / 2
    cg = green - tmp
    y = tmp + cg / 2
    y = y * 2 - 1

    transformed_img = torch.stack(
        [y, co, cg], dim=-1
    )
    ycc_img = torch.floor(
        ((transformed_img + 1) / 2) * 256
    ).long().clip(0, 255)
    return ycc_img
```

```python
def ycc2rgb_lossy(
    ycc_img: torch.Tensor
):
    assert ycc_img.size(-1) == 3

    deq_img = (ycc_img / 127.5) - 1
    y = deq_img[..., 0]
    co = deq_img[..., 1]
    cg = deq_img[..., 2]

    y = (y + 1) / 2
    tmp = y - cg / 2
    green = cg + tmp
    blue = tmp - co / 2
    red = blue + co

    transformed_img = torch.stack(
        [red, green, blue], dim=-1
    )
    rgb_img = torch.floor(
        (transformed_img * 255)
    ).long().clip(0, 255)
    return rgb_img
```

Figure C.2: **Lossy YCoCg transform.** We attach pytorch code for the RGB $\rightarrow$ YCoCg direction (left) and YCoCg $\rightarrow$ RGB direction (right). The two functions do *not* represent a bijection. The input to both functions is a tensor with discrete values in $[0, 255]$, so their output.

```
batch_size = 100
img_size = 32
rgb_batch = torch.randint(256, (batch_size, img_size * img_size, 3))

ycc_batch_lossless = rgb2ycc_lossless(rgb_batch)
recon_rgb_batch_lossless = ycc2rgb_lossless(ycc_batch_lossless)
print((recon_rgb_batch_lossless == rgb_batch).all())  # True

ycc_batch_lossy = rgb2ycc_lossy(rgb_batch)
recon_rgb_batch_lossy = ycc2rgb_lossy(ycc_batch_lossy)
print((rgb_batch - recon_rgb_batch_lossy).abs().float().mean())  # around 0.66
```

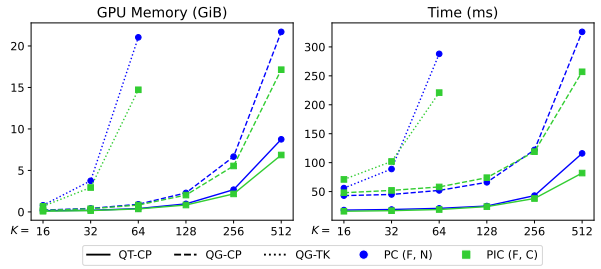
Figure C.3: **Application of the YCoCg transforms.** We show that YCoCg-R is indeed a bijection, and that the lossy YCoCg is on average off less than a bit.

Figure C.4: **Visual difference of YCoCg-R and YCoCg.** Given the RGB image to the left, we show the application of YCoCg-R in the middle and that of YCoCg to the right.

D Additional results

D.1 Scaling experiments

We report the time and GPU memory required to perform and Adamp optimization step for several model configurations in Table D.1, Fig. D.1 and Table D.2.



	QT-CP	QG-CP	QG-TK
PIC	1.1M	2.2M	1.8M
K=16	270K	800K	6M
K=32	1M	3M	51M
K=64	4M	13M	408M
K=128	17M	51.1M	-
K=256	69M	204M	-
K=512	277M	817M	-

Figure D.1: **Training PICs using functional sharing requires comparable resources as PCs.** We compare the average GPU memory (left) and time (right) required to perform an Adam optimization step with PCs (blue) and PICs (green), varying region graph type, parameterizations, and K. We pair the plots with a table reporting the number of parameters of PCs at different K and PICs, with the latter being independent of K and allowing up to 99% less trainable parameters (QG-CP-512). The benchmark is conducted using a batch size of 256 gray-scale images of size 28x28, i.e. MNIST-like. Extra (tabular) details in Table D.1.

Table D.1: **Training PICs using functional sharing requires comparable resources as PCs.** We report the time (in milliseconds, top) and GPU memory (in GiB, bottom) required to perform an Adam optimization step on 256 MNIST-like images by varying: architecture ($\{$QT-CP, QG-CP, QG-TK$\}$), size K ($\{2^i\}_{i=4}^{9}$), model ($\{\text{PC}, \text{PIC}\}$), and sharing technique ($\{\text{C}, \text{F}, \text{N}\}$). For each model we attach a pair $(\cdot, \cdot)$ where the first (resp. second) argument specifies the sharing technique for the input (resp. inner) layer(s).

RG-layer	K	PC (F, N)	PC (N, N)	PIC (F, C)	PIC (C, C)	PIC (F, N)
QT-CP	16	16	16	18	18	67
	32	17	17	19	19	63
	64	19	19	21	21	89
	128	24	24	25	26	OOM
	256	38	40	43	47	OOM
	512	82	90	116	128	OOM
QG-CP	16	48	48	43	43	193
	32	52	52	45	46	189
	64	58	59	52	52	OOM
	128	74	74	66	67	OOM
	256	119	121	122	126	OOM
	512	257	264	326	337	OOM
QG-TK	16	71	72	56	57	156
	32	102	102	89	89	OOM
	64	221	221	288	289	OOM

RG-layer	K	PC (F, N)	PC (N, N)	PIC (F, C)	PIC (C, C)	PIC (F, N)
QT-CP	16	0.08	0.14	0.15	1.00	2.68
	32	0.16	0.28	0.19	1.05	6.97
	64	0.36	0.60	0.40	1.27	20.11
	128	0.83	1.31	0.97	1.93	OOM
	256	2.17	3.12	2.68	3.83	OOM
	512	6.86	8.39	8.75	10.09	OOM
QG-CP	16	0.18	0.24	0.22	1.03	7.85
	32	0.38	0.49	0.42	1.24	13.95
	64	0.83	1.07	0.92	1.79	OOM
	128	2.03	2.50	2.29	3.25	OOM
	256	5.54	6.50	6.66	7.66	OOM
	512	17.14	19.05	21.70	23.04	OOM
QG-TK	16	0.68	0.73	0.80	1.55	18.60
	32	2.94	3.04	3.75	4.37	OOM
	64	14.72	14.92	21.04	21.71	OOM

Table D.2: **PICs using functional sharing require comparable resources as PCs to be trained, and scale much more gracefully than PICs without sharing.** We report the time (in milliseconds, top) and GPU memory (in GiB, bottom) required to perform an Adam optimization step on a batch of 128 RGB images of size 64x64 by varying: architecture ($\{$QT-CP, QG-CP, QG-TK$\}$), size K ($\{2^i\}_{i=4}^8$), model ($\{\text{PC}, \text{PIC}\}$), sharing technique ($\{\text{C}, \text{F}, \text{N}\}$), and also the MLP size M ($\{2^i\}_{i=5}^8$) of the PIC models. For each model we attach a pair $(\cdot, \cdot)$ where the first (resp. second) argument specifies the sharing technique for the input (resp. inner) layer(s).

			PIC (F, C)				PIC (F, N)			
	K	PC (F, N)	$M = 256$	128	64	32	$M = 256$	128	64	32
QT-CP	16	37	40	38	40	39	330	203	94	59
	32	51	53	49	53	51	388	161	124	72
	64	74	78	77	77	76	OOM	OOM	245	120
	128	126	126	124	126	126	OOM	OOM	OOM	OOM
	256	246	257	253	249	251	OOM	OOM	OOM	OOM
QG-CP	16	77	73	71	71	70	OOM	550	238	129
	32	98	94	92	88	89	OOM	762	312	150
	64	135	132	130	128	123	OOM	OOM	OOM	264
	128	215	213	209	208	201	OOM	OOM	OOM	OOM
	256	428	449	440	433	425	OOM	OOM	OOM	OOM
QG-TK	16	117	99	98	98	95	OOM	OOM	429	164
	32	211	204	202	199	183	OOM	OOM	OOM	OOM

			PIC (F, C)				PIC (F, N)			
	K	PC (F, N)	$M = 256$	128	64	32	$M = 256$	128	64	32
QT-CP	16	0.67	0.70	0.68	0.67	0.67	13.81	5.03	2.14	1.13
	32	1.23	1.27	1.25	1.24	1.23	31.45	12.54	6.15	2.94
	64	2.48	2.57	2.52	2.50	2.49	OOM	OOM	22.59	10.97
	128	5.48	5.74	5.61	5.54	5.51	OOM	OOM	OOM	OOM
	256	13.48	14.45	13.96	13.72	13.60	OOM	OOM	OOM	OOM
QG-CP	16	0.71	0.78	0.74	0.72	0.72	OOM	12.68	4.88	2.10
	32	1.40	1.50	1.44	1.42	1.41	OOM	27.72	12.93	6.00
	64	3.15	3.35	3.24	3.20	3.17	OOM	OOM	OOM	22.13
	128	8.15	8.74	8.44	8.30	8.22	OOM	OOM	OOM	OOM
	256	24.14	26.29	25.22	24.69	24.42	OOM	OOM	OOM	OOM
QG-TK	16	2.24	2.35	2.26	2.22	2.20	OOM	OOM	22.46	11.10
	32	10.77	11.35	10.81	10.54	10.40	OOM	OOM	OOM	OOM

D.2 Additional distribution estimation results

In this section, we report tabular results for all our experiments.

Note that, every input layer of a standard PC is parameterized by a matrix $K \times P$, where P is the number of categories, which is 256 for grey-scale image datasets and $768 = 256 \cdot 3$ for RGB images. We found that sharing a single input layer among all pixels results in (slightly) worse performance for grey-scale images (as detailed in Table D.3). In contrast, we found that such sharing considerably improves performance for RGB image datasets. Besides improving performance for RGB image datasets, such sharing considerably lower the number of trainable parameters from $D \times K \times P$ to only $K \times P$ where D is the number of pixels. For instance, parameterizing all input layers of a tensorized architecture with $K = 256$ built for 64x64 images would require $805,306,368 = 256 \cdot 64 \cdot 64 \cdot 768$ parameters, while only $3,145,728$ if we apply the sharing. Therefore, without applying such sharing, we cannot even scale to big tensorized architectures (e.g. QG-CP-512) on our GPUs.

All QPCs are materialized from PICs applying F-sharing over the group of input units, and C-sharing over the groups of integral units.

We extensively compare QPCs and PCs as density estimators on several image datasets and report test-set bits-per-dimension (bpd) in Table D.3, Table D.4, and Table D.5.

Table D.3: **PCs with a shared input layer deliver comparable performance as PCs who do not on the MNIST-family datasets.** We compare the bits-per-dimension of PCs with (w/) and without (w/o) a shared input layer considering three different tensorized architectures: QT-CP-512, QG-CP-512 and QG-TK-64.

	QT-CP-512		QG-CP-512		QG-TK-64	
	w/o	w/	w/o	w/	w/o	w/
MNIST	**1.175** $\pm$ 0.001	1.213 $\pm$ 0.002	**1.177** $\pm$ 0.006	1.241 $\pm$ 0.005	**1.257** $\pm$ 0.005	1.300 $\pm$ 0.004
F-MNIST	**3.381** $\pm$ 0.001	**3.381** $\pm$ 0.002	**3.317** $\pm$ 0.005	3.375 $\pm$ 0.005	**3.499** $\pm$ 0.006	3.560 $\pm$ 0.007
EMN-MN	**1.706** $\pm$ 0.007	1.761 $\pm$ 0.005	**1.643** $\pm$ 0.007	1.711 $\pm$ 0.006	**1.756** $\pm$ 0.002	1.772 $\pm$ 0.004
EMN-LE	**1.698** $\pm$ 0.006	1.735 $\pm$ 0.007	**1.626** $\pm$ 0.004	1.656 $\pm$ 0.004	**1.725** $\pm$ 0.003	1.728 $\pm$ 0.003
EMN-BA	1.731 $\pm$ 0.007	**1.772** $\pm$ 0.007	**1.665** $\pm$ 0.004	1.696 $\pm$ 0.003	1.751 $\pm$ 0.002	**1.749** $\pm$ 0.005
EMN-BY	**1.542** $\pm$ 0.008	1.548 $\pm$ 0.007	**1.474** $\pm$ 0.009	1.481 $\pm$ 0.007	**1.665** $\pm$ 0.007	1.679 $\pm$ 0.006

Table D.4: **QPCs consistently improve over PCs on MNIST and FASHIONMNIST.** Test-set bits-per-dimension (bpd) on MNIST (top) and FASHIONMNIST (bottom) averaged over 5 runs. All QPCs are materialized from PICs applying F-sharing over the group of input units, and C-sharing over the integral units groups, i.e. QPCs are materialized from PICs (F, C). PCs do not apply any form of parameter sharing, as these deliver the best performance for these datasets, as detailed in Table D.3.

K	QT-CP		QG-CP		QG-TK	
	QPC	PC	QPC	PC	QPC	PC
16	**1.275** $\pm$ 0.009	**1.283** $\pm$ 0.004	**1.237** $\pm$ 0.009	1.248 $\pm$ 0.003	**1.215** $\pm$ 0.010	1.233 $\pm$ 0.004
32	**1.220** $\pm$ 0.003	**1.242** $\pm$ 0.004	**1.189** $\pm$ 0.008	1.212 $\pm$ 0.003	**1.168** $\pm$ 0.002	1.222 $\pm$ 0.004
64	**1.195** $\pm$ 0.002	**1.217** $\pm$ 0.002	**1.162** $\pm$ 0.002	1.185 $\pm$ 0.002	**1.144** $\pm$ 0.006	1.257 $\pm$ 0.005
128	**1.161** $\pm$ 0.003	**1.196** $\pm$ 0.004	**1.144** $\pm$ 0.006	1.171 $\pm$ 0.002	OOM	
256	**1.135** $\pm$ 0.006	**1.184** $\pm$ 0.002	**1.120** $\pm$ 0.005	1.173 $\pm$ 0.009	OOM	
512	**1.126** $\pm$ 0.004	**1.175** $\pm$ 0.001	**1.115** $\pm$ 0.005	1.177 $\pm$ 0.006	OOM	
16	**3.547** $\pm$ 0.003	3.589 $\pm$ 0.005	**3.427** $\pm$ 0.006	3.464 $\pm$ 0.005	**3.424** $\pm$ 0.009	3.446 $\pm$ 0.008
32	**3.429** $\pm$ 0.001	3.497 $\pm$ 0.003	**3.319** $\pm$ 0.001	3.385 $\pm$ 0.004	**3.323** $\pm$ 0.003	3.417 $\pm$ 0.005
64	**3.349** $\pm$ 0.005	3.442 $\pm$ 0.003	**3.258** $\pm$ 0.004	3.339 $\pm$ 0.004	**3.251** $\pm$ 0.003	3.499 $\pm$ 0.006
128	**3.289** $\pm$ 0.001	3.408 $\pm$ 0.003	**3.212** $\pm$ 0.003	3.319 $\pm$ 0.004	OOM	
256	**3.242** $\pm$ 0.001	3.392 $\pm$ 0.002	**3.174** $\pm$ 0.002	3.317 $\pm$ 0.002	OOM	
512	**3.209** $\pm$ 0.003	3.381 $\pm$ 0.001	**3.154** $\pm$ 0.004	3.317 $\pm$ 0.005	OOM	

Table D.5: **QPCs generally improve over PCs on distribution estimation.** We report the average test-set bits-per-dimensions of QT-CP-512, QG-CP-512 and QG-TK-64 for datasets up to image size 32x32, and of QT-CP-256, QG-CP-256 and QG-TK-32 for datasets of image size 64x64. All architectures are trained both as QPCs and PCs. QPCs are materialized from PICs applying F-sharing over the group of input units, and C-sharing over the groups of integral units. PCs do not apply any form of parameter sharing for MNIST-family datasets, as these delivered the best performance for such datasets Table D.3, while they apply F-sharing at the input layer for RGB datasets. We mark with * (resp. †) datasets preprocessed using YCoCg-R (resp. YCoCg). All results are averaged over 5 different runs.

	QPC	PC	QPC	PC	QPC	PC
	QT-CP-512		QG-CP-512		QG-TK-64	
MNIST	**1.126** $\pm$ 0.004	1.175 $\pm$ 0.001	**1.115** $\pm$ 0.005	1.177 $\pm$ 0.006	**1.144** $\pm$ 0.006	1.257 $\pm$ 0.005
F-MNIST	**3.209** $\pm$ 0.003	3.381 $\pm$ 0.001	**3.154** $\pm$ 0.004	3.317 $\pm$ 0.005	**3.251** $\pm$ 0.003	3.499 $\pm$ 0.006
EMN-MN	**1.592** $\pm$ 0.007	1.706 $\pm$ 0.007	**1.556** $\pm$ 0.006	1.643 $\pm$ 0.007	**1.699** $\pm$ 0.004	1.756 $\pm$ 0.002
EMN-LE	**1.622** $\pm$ 0.007	1.698 $\pm$ 0.006	**1.545** $\pm$ 0.007	1.626 $\pm$ 0.004	**1.696** $\pm$ 0.003	1.725 $\pm$ 0.003
EMN-BA	**1.638** $\pm$ 0.006	1.731 $\pm$ 0.007	**1.593** $\pm$ 0.005	1.665 $\pm$ 0.004	**1.715** $\pm$ 0.005	1.751 $\pm$ 0.002
EMN-BY	1.597 $\pm$ 0.006	**1.542** $\pm$ 0.008	1.537 $\pm$ 0.004	**1.474** $\pm$ 0.009	1.703 $\pm$ 0.004	**1.665** $\pm$ 0.007
CIFAR*	**5.198** $\pm$ 0.003	5.596 $\pm$ 0.004	**5.097** $\pm$ 0.002	5.496 $\pm$ 0.004	**5.556** $\pm$ 0.003	5.647 $\pm$ 0.004
CIFAR†	**4.577** $\pm$ 0.004	4.884 $\pm$ 0.003	**4.486** $\pm$ 0.009	4.856 $\pm$ 0.010	**4.888** $\pm$ 0.007	4.983 $\pm$ 0.004
ImgNet32*	**5.196** $\pm$ 0.007	5.286 $\pm$ 0.001	**5.085** $\pm$ 0.001	5.255 $\pm$ 0.001	**5.544** $\pm$ 0.001	5.700 $\pm$ 0.002
ImgNet32†	**4.578** $\pm$ 0.001	4.662 $\pm$ 0.002	**4.468** $\pm$ 0.001	4.632 $\pm$ 0.003	**4.893** $\pm$ 0.004	5.045 $\pm$ 0.001
	QT-CP-256		QG-CP-256		QG-TK-32	
CelebA*	**4.810** $\pm$ 0.004	4.851 $\pm$ 0.002	**4.739** $\pm$ 0.002	4.781 $\pm$ 0.002	**5.352** $\pm$ 0.002	5.364 $\pm$ 0.002
CelebA†	**4.159** $\pm$ 0.003	4.215 $\pm$ 0.003	**4.114** $\pm$ 0.003	4.155 $\pm$ 0.006	4.720 $\pm$ 0.003	**5.718** $\pm$ 0.001
ImgNet64*	**5.143** $\pm$ 0.003	5.221 $\pm$ 0.003	**5.051** $\pm$ 0.002	5.220 $\pm$ 0.005	**5.657** $\pm$ 0.004	5.764 $\pm$ 0.001
ImgNet64†	**4.523** $\pm$ 0.003	4.591 $\pm$ 0.002	**4.425** $\pm$ 0.004	4.590 $\pm$ 0.006	**5.011** $\pm$ 0.007	5.138 $\pm$ 0.001

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: Abstract and main text accurately and precisely state the actual claims of the research presented.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Limitations mentioned in the last section.

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory Assumptions and Proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [NA]

Justification: The paper does not include theoretical results.

Guidelines:

- The answer NA means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental Result Reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: All data used is publicly available. All model and experimental code to fully reproduce our own experimental results is shared.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: All data used is publicly available. All model and experimental code to fully reproduce our own experimental results is shared.

Guidelines:

- The answer NA means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (https://nips.cc/public/guides/CodeSubmissionPolicy) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (https://nips.cc/public/guides/CodeSubmissionPolicy) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental Setting/Details**

Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

Answer: [Yes]

Justification: Experimental settings overview included in paper; key details included in appendix; full details in the included code.

Guidelines:

- The answer NA means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment Statistical Significance**

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [Yes]

Justification: Statistical significance indicated for all experimental results, including the nature of the error bars.

Guidelines:

- The answer NA means that the paper does not include experiments.
- The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.

- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments Compute Resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Overview compute resources included in paper; key details discussed in appendix; full details listed in structured template-description of corresponding assets.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code Of Ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics https://neurips.cc/public/EthicsGuidelines?

   Answer: [Yes]

   Justification: Problematic aspects of generative AI mentioned, used asset licenses mentioned, produced assets (code, models) licensed and shared, reporducibility ensured.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader Impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: Positive: better interpretability of this class of generative AI models mentioned; Negative: problematic aspects of generative AI models in general mentioned.

    Guidelines:

- The answer NA means that there is no societal impact of the work performed.
- If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA] .

    Justification: The paper poses no such risks.

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: Original papers cited and/or URLs provided; license mentioned in reference.

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.

- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New Assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: Code and models included (as zip file for submission, also as URL for final); Structured templates used for details.

    Guidelines:

    - The answer NA means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and Research with Human Subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [NA]

    Justification: No crowdsourcing or research with human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional Review Board (IRB) Approvals or Equivalent for Research with Human Subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: No crowdsourcing or research with human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.

- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.