Einsum Networks: Fast and Scalable Learning of Tractable Probabilistic Circuits

Robert Peharz [1] **Steven Lang** [2] **Antonio Vergari** [3] **Karl Stelzner** [2] **Alejandro Molina** [2] **Martin Trapp** [4]
Guy Van den Broeck [3] **Kristian Kersting** [2] **Zoubin Ghahramani** [5]

Abstract

Probabilistic circuits (PCs) are a promising avenue for probabilistic modeling, as they permit a wide range of exact and efficient inference routines. Recent "deep-learning-style" implementations of PCs strive for a better scalability, but are still difficult to train on real-world data, due to their sparsely connected computational graphs. In this paper, we propose Einsum Networks (EiNets), a novel implementation design for PCs, improving prior art in several regards. At their core, EiNets combine a large number of arithmetic operations in a single monolithic einsum-operation, leading to speedups and memory savings of up to two orders of magnitude, in comparison to previous implementations. As an algorithmic contribution, we show that the implementation of Expectation-Maximization (EM) can be simplified for PCs, by leveraging automatic differentiation. Furthermore, we demonstrate that EiNets scale well to datasets which were previously out of reach, such as SVHN and CelebA, and that they can be used as faithful generative image models.

1. Introduction

The central goal of probabilistic modeling is to approximate the data-generating distribution, in order to answer statistical queries by means of probabilistic inference. In recent years many novel probabilistic models based on deep neural networks have been proposed, such as Variational Autoencoders (VAEs) (Kingma & Welling, 2014) (formerly known as Density Networks (MacKay, 1995)), Normalizing Flows (Rezende & Mohamed, 2015; Papamakarios et al., 2019), Autoregressive Models (ARMs) (Larochelle & Murray, 2011; Uria et al., 2016), and Generative Adversarial Networks (GANs) (Goodfellow et al., 2014). While all these models have achieved impressive results on large-scale datasets, i.e. they have been successful in terms of **representational power** and **learning**, they unfortunately fall short in terms of **inference**, a main aspect of probabilistic modeling and reasoning (Pearl, 1988; Koller & Friedman, 2009). All of the mentioned models allow to draw unbiased samples, enabling inference via Monte Carlo estimation. This strategy, however, becomes quickly unreliable and computational expensive for all but the simplest inference queries. Also other approximate inference techniques, e.g. variational inference, are often biased and their inference quality might be hard to analyse. Besides sampling, only ARMs and Flows support efficient evaluation of the probability density for a given sample, which can be used, e.g., for model comparison and outlier detection.

However, even for ARMs and Flows the following inference task is computationally hard: Consider a density $p(X_1, \ldots, X_N)$ over N random variables, where N might be just in the order of a few dozens. For example, $X_1, \ldots, X_N$ might represent medical measurements of a particular person drawn from the general population modeled by $p(X_1, \ldots, X_N)$. Now assume that we split the variables into three disjoint sets $\mathbf{X}_q$, $\mathbf{X}_m$, and $\mathbf{X}_e$ of roughly the same size, and that we wish to compute

$$p(\mathbf{x}_q \,|\, \mathbf{x}_e) = \frac{\int p(\mathbf{x}_q, \mathbf{x}'_m, \mathbf{x}_e) \mathrm{d}\mathbf{x}'_m}{\int\int p(\mathbf{x}'_q, \mathbf{x}'_m, \mathbf{x}_e) \mathrm{d}\mathbf{x}'_q \mathrm{d}\mathbf{x}'_m}, \qquad (1)$$

for arbitrary values $\mathbf{x}_q$ and $\mathbf{x}_e$. In words, we wish to predict query variables $\mathbf{X}_q$, based on evidence $\mathbf{X}_e = \mathbf{x}_e$, while accounting for all possible values of missing variables $\mathbf{X}_m$, by marginalizing them. Conditional densities like (1) highlight the role of generative models as "multi-purpose predictors", since the choice of $\mathbf{X}_q$, $\mathbf{X}_m$, and $\mathbf{X}_e$ can be arbitrary. However, evaluating conditional densities of this form is notoriously hard for any of the models above, which represents a principal drawback of these methods.

These shortcomings in terms of inference have motivated a growing stream of research on *tractable probabilistic models*, where inference queries like (1) can be computed *ex-*

[1]Eindhoven University of Technology [2]Technical University of Darmstadt [3]University of California, Los Angeles [4]Graz University of Technology [5]University of Cambridge, Uber AI Labs. Correspondence to: Robert Peharz <r.peharz@tue.nl>, Steven Lang <steven.lang.mz@gmail.com>.

Proceedings of the 37^{th} International Conference on Machine Learning, Vienna, Austria, PMLR 119, 2020.

actly and *efficiently* (in polynomial time). One of the most prominent families of tractable models are *probabilistic circuits* (PCs),[1] which represent probability densities via computational graphs of i) *mixtures* (convex sum nodes), ii) *factorizations* (product nodes), and iii) *tractable distributions* (leaves, input nodes). Key structural properties of PCs are *smoothness* and *decomposability* (Darwiche, 2003), which ensure that *any* integral, like those appearing in (1), can be computed in *linear* time of the circuit size. These structural constraints, however, make it hard to work with PCs in practice, as they lead to highly sparse and cluttered computational graphs, which are inapt for current machine learning frameworks. Furthermore, PCs are typically implemented in the log-domain, which slows down learning and inference even further.

In this paper, we propose a novel implementation design for PCs, which ameliorates these practical difficulties, and allows to evaluate and train PCs of up to *two orders of magnitude faster* than previous "deep-learning-style" implementations (Pronobis et al., 2017; Peharz et al., 2019). The central idea is to compute all product and sum operations on the same topological layer using a single monolithic *einsum* operation.[2] In that way, the main computational work is lifted by a parallel operation for which efficient implementations, both for CPU and GPU, are readily available in most numerical frameworks. In order to ensure numerical stability, we extend the well-known log-sum-exp-trick to our setting, leading to the "log-*einsum*-exp" trick. Since our model implements PCs via a hierarchy of large einsum layers, we call our model *Einsum Network* (EiNet).

As further contributions, we present two algorithmic improvements for training PCs. First, we show that Expectation-Maximization (EM), the canonical maximum-likelihood learning algorithm for PCs (Peharz, 2015; Desana & Schnörr, 2016; Peharz et al., 2017), can be easily implemented via automatic differentiation, readily provided by most machine learning frameworks. Second, we leverage stochastic online EM (Sato, 1999) in order to further improve the learning speed of EiNets, or, enable EM for large datasets at all. In experiments we demonstrate that EiNets can rapidly learn generative models for street view house numbers (SVHN) and CelebA. To the best of our knowledge, this is the first time that PCs have been successfully trained on datasets of this size. EiNets are capable of producing high-quality image samples, while maintaining tractable inference, e.g. for conditional densities (1).

[1] We adopt the name *probabilistic circuits*, as suggested by (Vergari et al., 2019), which serves as an umbrella term for many structurally related models, like arithmetic circuits, sum-product networks, cutset networks, etc. See Section 2 for details.

[2] The *einsum* operation implements the Einstein notation of tensor-product contraction, and unifies standard linear algebra operations like dot product, outer product, and matrix multiplication.

2. Probabilistic Circuits

Probabilistic circuits (PCs) are a family of probabilistic models which allow a wide range of exact and efficient inference routines. The earliest representatives of PCs are *arithmetic circuits* (ACs) (Darwiche, 2002; 2003), which are based on *decomposable negation normal forms* (DNNFs) (Darwiche, 1999; 2001), a tractable representation for propositional logic formulas. Further members of the PC family are *sum-product networks* (SPNs) (Poon & Domingos, 2011), *cutset networks* (CNs) (Rahman et al., 2014), and *probabilistic sentential decision diagrams* (PSDDs) (Kisa et al., 2014). The main differences between different types of PCs are i) the set of *structural constraints* assumed and corresponding tractable inference routines, ii) syntax and representation, and iii) application scenarios. In order to treat these models in a unified fashion, we adopt the umbrella term *probabilistic circuits* suggested by (Vergari et al., 2019), and distinguish various instantiates of PCs mainly via their structural properties.

Definition 1 (Probabilistic Circuit)**.** *Given a set of random variables* $\mathbf{X}$*, a* probabilistic circuit *(PC)* $\mathcal{P}$ *is a tuple* $(\mathcal{G}, \psi)$*, where* $\mathcal{G}$*, denoted as* computational graph*, is an acyclic directed graph* (V, E)*, and* $\psi\colon V \mapsto 2^{\mathbf{X}}$*, denoted as* scope function*, is a function assigning some* scope *(i.e., a sub-set of* $\mathbf{X}$*) to each node in* V*. For internal nodes of* $\mathcal{G}$*, i.e. any node* $\mathsf{N} \in V$ *which has children, the scope function satisfies* $\psi(\mathsf{N}) = \cup_{\mathsf{N}' \in \mathbf{ch}(\mathsf{N})} \psi(\mathsf{N}')$*, where* $\mathbf{ch}(\mathsf{N})$ *denotes the set of children of* N*. A* leaf *of* $\mathcal{G}$ *computes a probability density*[3] *over its scope* $\psi(\mathsf{L})$*. All internal nodes of* $\mathcal{G}$ *are either* sum nodes *(*S*) or* product nodes *(*P*). A sum node* S *computes a convex combination of its children, i.e.* $\mathsf{S} = \sum_{\mathsf{N} \in \mathbf{ch}(\mathsf{S})} w_{\mathsf{S},\mathsf{N}}\, \mathsf{N}$*, where* $\sum_{\mathsf{N} \in \mathbf{ch}(\mathsf{S})} w_{\mathsf{S},\mathsf{N}} = 1$*, and* $\forall \mathsf{N} \in \mathbf{ch}(\mathsf{S})\colon w_{\mathsf{S},\mathsf{N}} \geq 0$*. A product node* P *computes a product of its children, i.e.* $\mathsf{P} = \prod_{\mathsf{N} \in \mathbf{ch}(\mathsf{P})} \mathsf{N}$.

PCs can be seen as a special kind of neural network, where the first layer computes non-linear functions (probability densities) over sub-sets of $\mathbf{X}$, and all internal nodes compute either weighted sums (linear functions) or products (multiplicative interactions). Each node N in a PC sees only a sub-set of the inputs, namely variables in its scope $\psi(\mathsf{N})$. The output of a PC is typically a single root node N_{root} (i.e., a node without parents), for which we assume full scope $\psi(\mathsf{N}_{root}) = \mathbf{X}$. We define a density $\mathcal{P}(\mathbf{x})$ proportional to the output: $\mathcal{P}(\mathbf{x}) := \frac{\mathsf{N}_{root}(\mathbf{x})}{\int \mathsf{N}_{root}(\mathbf{x}) \mathrm{d}\mathbf{x}}$. PCs as defined here encode a density, but do not permit tractable inference yet. In particular, the normalization constant is hard to compute. Various tractable inference routines "can be bought" by imposing structural constraints, which we review next.

[3] This subsumes probability mass functions by assuming an adequate underlying counting measure, and also hybrid discrete-continuous densities by assuming an adequate underlying product measure of counting and Lebesgue measures.

Decomposability. A PC is *decomposable*, if for each product node $\mathsf{P} \in V$ it holds that $\psi(\mathsf{N}) \cap \psi(\mathsf{N}') = \emptyset$, for $\mathsf{N}, \mathsf{N}' \in \mathbf{ch}(\mathsf{P})$, $\mathsf{N} \neq \mathsf{N}'$, i.e. a PC is decomposable if children of product nodes have pairwise disjoint scope.

Smoothness. A PC is *smooth*, if for each sum node $\mathsf{S} \in V$ it holds that $\psi(\mathsf{N}) = \psi(\mathsf{N}')$, for $\mathsf{N}, \mathsf{N}' \in \mathbf{ch}(\mathsf{S})$, i.e. a PC is smooth if children of sum nodes have identical scope.

In smooth and decomposable PCs, any node is by induction a properly normalized density (over its scope), since i) leaves are densities by assumption, ii) sum nodes are *mixture densities*, and iii) product nodes are factorized densities. The most important consequence of decomposability and smoothness is that integrals which can be written as nested single-dimensional integrals—in particular the normalization constant—can be computed in *linear* time of the circuit size (Peharz et al., 2015). This follows from the fact that any single-dimensional integral i) commutes with a sum-node and ii) affects only a single child of a product-node. Consequently, all integrals can be distributed to the PC's leaves, i.e. we simply need to perform integration at the leaves (which is assumed to be tractable), and evaluate the internal nodes in a single feedforward pass. Smooth and decomposable PCs admit a natural interpretation as latent variable models (Zhao et al., 2015; Peharz et al., 2017), which admits a natural sampling procedure via ancestral sampling and maximum-likelihood learning via EM (Peharz, 2015; Desana & Schnörr, 2016; Peharz et al., 2017). Smooth and decomposable PCs are often referred to as sum-product networks (SPNs) (Poon & Domingos, 2011).

In this paper, we consider only smooth and decomposable PCs (aka SPNs), which allow efficient marginalization and conditioning. A further structural property, not considered in this paper, is *determinism* (Darwiche, 2003) which allows for exact probability maximization (*most-probable-explanation*), which is NP-hard in non-deterministic PCs (de Campos, 2011; Peharz et al., 2017). Moreover, PCs can be equipped with *structured decomposability* (Pipatsrisawat & Darwiche, 2008; Khosravi et al., 2019a), which allows circuit multiplication and structured expectations (Shen et al., 2016; Khosravi et al., 2019b).

3. Einsum Networks

3.1. Vectorizing Probabilistic Circuits

An immediate way to yield a denser and thus more efficient layout for PCs is to *vectorize* them. To this end, we re-define a leaf L to be a *vector* of K densities over $\psi(\mathsf{L})$, rather than a single density. For example, a leaf computing a Gaussian density is replaced by a vector $[\mathcal{N}(\cdot|\boldsymbol{\theta}_1), \dots, \mathcal{N}(\cdot|\boldsymbol{\theta}_K)]^T$, each $\mathcal{N}(\cdot|\boldsymbol{\theta}_k)$ being a Gaussian over $\psi(\mathsf{L})$, equipped with private parameters $\boldsymbol{\theta}_k$. A product node is re-defined to be an *outer product* $\otimes_{\mathsf{N} \in \mathbf{ch}(\mathsf{P})} \mathsf{N}$, containing the products of all possible combinations of densities coming from the child vectors. Since the number of elements in outer products grows exponentially, we restrict the number of children to two (this constraint is frequently imposed on PCs for simplicity). Finally, we re-define sum nodes to be a vector of K weighted sums, where each individual sum operation has its private weights and computes a convex combination over all the densities computed by its children. The vectorized version of PCs is frequently called a *region graph* (Dennis & Ventura, 2012; Trapp et al., 2019a) and has been used in previous GPU-supporting implementations (Pronobis et al., 2017; Peharz et al., 2019). It is easily verified, that our desired structural properties—smoothness and decomposability—carry over to vectorized PCs.

For the remainder of the paper, we use symbols S, P, L for the vectorized versions, and refer to them as sum nodes, product nodes, and leaf nodes, respectively, or also simply as sums, products and leaves. To any single entry in these vectors we explicitly refer to as *entry*, or *operation*. In principle, the number of entries K could be different for each leaf or sum, which would, however, lead to a less homogeneous PC design. Therefore, in this paper, we assume for simplicity the same K for all leaves and sums. Furthermore, we make some simplifying assumptions about the structure of $\mathcal{G}$. First, we assume a structure alternating between sums/leaves and products, i.e. children of sums can only be products, and children of products can only be sums or leaves. Second, we also assume the root of the PC is a sum node. These assumptions are commonly made in the PC literature and are no restriction of generality. Furthermore, we also assume that each product node has at most one parent (which must be a sum due to alternating structure). This is also not a severe structural assumption: If a product has two or more sum nodes as parents, then, by smoothness, these sum nodes have all the same scope. Consequently, they could be simply concatenated to a *single* sum vector. Of course, since in this paper we assume a constant length K for all sum and leaf vectors, this assumption requires a large enough K.

3.2. The Basic Einsum Operation

The core computational unit in EiNets is the vectorized PC excerpt in Fig. 1, showing a sum node S with a single product child P, which itself has two children N and N' (shown here as sum nodes, but they could also be leaves). Nodes N and N' compute each a vector of K densities, the product node P computes the outer product of N and N', and the sum node S computes a matrix-vector product $\mathbf{W}\,\mathbf{vec}(\mathsf{P})$. Here, $\mathbf{W}$ is an element-wise non-negative $K \times K^2$ matrix, whose rows sum to one, and $\mathbf{vec}(\mathsf{P})$ unrolls P to a vector of K^2 elements. Previous PC implementations (Pronobis et al., 2017; Peharz et al., 2019), are also based on this core computational unit. For numerical stability, however,

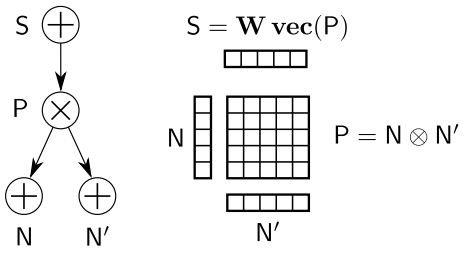


Figure 1. Basic einsum operation in EiNets: A sum node S, with a single child P, which itself has 2 children. All nodes are vectorized, as described in Section 3.1, and here illustrated for $K = 5$.

they use a computational workaround in the log-domain: The outer product is transformed into an "outer sum" of log-densities (realized with broadcasting), the matrix multiplication is implemented using a broadcasted sum of $\log \mathbf{W}$ and $\mathbf{vec}(\log \mathsf{P})$, to which then a log-sum-exp operation is applied, yielding $\log \mathsf{S}$. This workaround introduces significant overhead and needs to allocate the products explicitly.

Mathematically, the PC excerpt in Fig. 1 is a simple multilinear form, naturally expressed in *Einstein notation*:

$$\mathsf{S}_k = \mathbf{W}_{kij} \mathsf{N}_i \mathsf{N}'_j. \tag{2}$$

Here we have re-shaped $\mathbf{W}$ into a $K \times K \times K$ tensor, normalized over its last two dimensions, i.e. $W_{kij} \geq 0$, $\sum_{i,j} W_{kij} = 1$. The signature in (2) mentions three indices i, j and k labeling the axes of N, N', and $\mathbf{W}$. Axes with the same index get multiplied. Furthermore, any indices not mentioned on the left hand side get summed out. General-purpose Einstein summations are readily implemented in most numerical frameworks, and usually denoted as *einsum* operation.

However, applying (2) in a naive way would quickly lead to numerical underflow and unstable training. In order to ensure numerical stability, we develop a technique similar to the classical "log-sum-exp"-trick. We keep all probabilistic values in the log-domain, but the weight-tensor $\mathbf{W}$ is kept in the *linear* domain. Consequently, we need a numerically stable computation for

$$\log \mathsf{S}_k = \log \sum_{i,j} W_{kij} \exp(\log \mathsf{N}_i) \exp(\log \mathsf{N}'_j). \tag{3}$$

Let us define $a = \max_i \log \mathsf{N}_i$ and $a' = \max_j \log \mathsf{N}'_j$. Then, it is easy to see that (3) can also be computed as

$$a + a' + \log \sum_{i,j} W_{kij} \exp(\log \mathsf{N}_i - a) \exp(\log \mathsf{N}'_j - a'). \tag{4}$$

A sufficient condition for numerical stability of (4) is that all sum-weights $W_{kij} > 0$, since in this case the maximal values in vectors $\exp(\log \mathsf{N} - a)$ and $\exp(\log \mathsf{N}' - a')$ are guaranteed to be 1, leading to a positive argument for the $\log$. This is not a severe requirement, as positive sum-weights are commonly enforced in PCs, e.g. by using Laplace smoothing or imposing a positive lower bound on the weights.

Algorithm 1 Topological Layers

Input: PC graph $\mathcal{G} = (V, E)$
Let $\mathbf{L}$, $\mathbf{S}$, $\mathbf{P}$ be the set of all leaves, sum nodes, product nodes in V, respectively
$M \leftarrow \{\}$
$layers = [\,]$
while $M \neq \mathbf{S} \cup \mathbf{P}$ **do**
  $l_{\mathsf{S}} = \{\mathsf{S} \in \mathbf{S} \colon \mathsf{S} \notin M \wedge \forall \mathsf{P} \in \mathbf{pa}(\mathsf{S}) \colon \mathsf{P} \in M\}$
  $layers \leftarrow concatenate([l_{\mathsf{S}}], layers)$
  $M \leftarrow M \cup l_{\mathsf{S}}$
  $l_{\mathsf{P}} = \{\mathsf{P} \in \mathbf{P} \colon \mathsf{P} \notin M \wedge \forall \mathsf{S} \in \mathbf{pa}(\mathsf{P}) \colon \mathsf{S} \in M\}$
  $layers \leftarrow concatenate([l_{\mathsf{P}}], layers)$
  $M \leftarrow M \cup l_{\mathsf{P}}$
end while
$layers \leftarrow concatenate([\mathbf{L}], layers)$
Return $layers$

Given two K-dimensional vectors N, N' and the $K \times K \times K$ weight-tensor $\mathbf{W}$, our basic einsum operation (4) requires $2K$ exp-operations, K log-operations, $\mathcal{O}(K^3)$ multiplications and $\mathcal{O}(K^3)$ additions. We need to store $3K$ values for N, N', S, while the product operations are not stored explicitly. In contrast, the indirect implementations of the same operation in (Pronobis et al., 2017; Peharz et al., 2019) need $\mathcal{O}(K^3)$ additions, K^3 exp-operations and K log-operations. These implementations also store $3K$ values for N, N', S, and additional K^2 values for the explicitly computed products. While our implementation is cubic in the number of *multiplications*, the previous implementations need a cubic number of *exp-operations*. This partially explains the speedup of EiNets in our experiments in Section 4. However, the main reasons for the observed speedup are i) an optimized implementation of the einsum operation, ii) avoiding the overhead of allocating product nodes, and iii) a higher degree of parallelism, as discussed in the next section.

3.3. The Einsum Layer

Rather than computing single vectorized sums, we can do better by computing whole layers in parallel. To this end, we organize the PC graph $\mathcal{G}$ in layers using Algorithm 1, which is essentially a top-down breadth-first search. We first partition the set of nodes V into the sets $\mathbf{L}$, $\mathbf{S}$, and $\mathbf{P}$, containing all leaves, all sums and all products, respectively. We initialize an empty set M, which stores the "marked" nodes during the execution of the algorithm. We also initialize an empty list $layers$ which will contain the result of the algorithm, a topologically ordered list of sets of nodes.

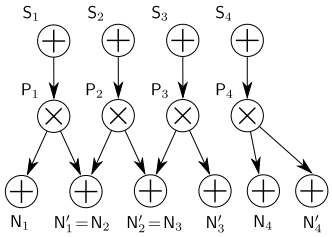


Figure 2. Example of an einsum layer, parallelizing the basic einsum operation.

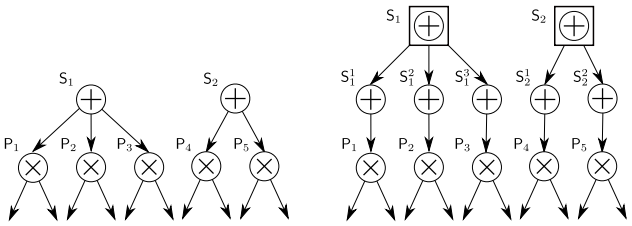


Figure 3. Decomposing a layer of sum nodes with multiple children (left) into two consecutive sum layers (right).

In line 6, we construct the set l_{S} of unmarked sum nodes (i.e. $\mathsf{S} \notin M$), but whose parents (denoted as $\mathbf{pa}(\mathsf{S})$), have already been marked. Note that in the first iteration of the while-loop, l_{S} will only contain the root of the EiNet. The set l_{S} is then inserted as head of the list $layers$ and all nodes in l_{S} are marked as visited. A similar procedure, but for product nodes, is performed in lines 9–11, yielding node set l_{P}. Note that, since we assume that each product node has only one parent (cf. Section 3.1), l_{P} will be exactly the set of children of the sum nodes in l_{S} (constructed in line 6 within the same while-iteration). If we assume for the moment that all sum nodes in l_{S} have exactly one child, then l_{P} and l_{S} form a consecutive pair of product and sum layers, as illustrated in Fig. 2. The central technique in this paper is an efficient and parallel computation of these two layers. The last step of Algorithm 1, line 13, is to insert the set of all leaf nodes as head of the list $layers$. It is easy to check that $layers$ will be a topologically ordered list of layers, ordered from leaves to root.

Let us now consider an excerpt as in Fig. 2, consisting of L sum nodes, where each of the sum nodes has a single child. We first collect all the vectors of the "left" product children in an $L \times K$ matrix $\mathbf{N}$ and similarly all the "right" product children in an $L \times K$ matrix $\mathbf{N}'$, where "left" and "right" are arbitrary but fixed assignments. We further extend the 3-dimensional weight-tensor $\mathbf{W}$ from (2) to a 4-dimensional $L \times K \times K \times K$ tensor, where the slice $\mathbf{W}_{l:::}$ contains the weights for the l^{th} vectorized sum node. The result of all L sums—i.e. in total $L \times K$ sum operations—can be performed with a single einsum operation: $\mathbf{S}_{lk} = \mathbf{W}_{lkij}\mathbf{N}_{li}\mathbf{N}'_{lj}$. Note the similarity to (2), and the additional parallelism by simply introducing an additional index l. It is straight-forward to extend the log-einsum-exp trick (see Section (3.2)) to this operation, in order to ensure numerical stability. Constructing the two matrices $\mathbf{N}$ and $\mathbf{N}'$ requires some book-keeping and introduces some computational overhead stemming from extracting and concatenating slices from the log-probability tensors below. This overhead is essentially the symptom of the sparse and cluttered layout of PCs. The main computational work, however, is then performed by a highly parallel einsum operation. In order to allow sum nodes with multiple children we use a simple technique explained next.

3.4. The Mixing Layer

In order to compute sum nodes with multiple children, we express any sum node with several children as a cascade of 2 vectorized sum operations. In particular, for a sum node S with C children, we introduce C new sum nodes $\mathsf{S}^1, \dots, \mathsf{S}^C$, each adopting one of the children as a single child. Subsequently, the results of S^c, $c \in \{1, \dots, C\}$, get mixed in an element-wise manner, i.e. $\mathsf{S}_k = \sum_{c=1}^{C} w^c \mathsf{S}_k^c$. This decomposition is essentially an over-parameterization (Trapp et al., 2019b) of the original sum nodes, and represents the same functions. This principle is illustrated in Fig. 3, where a sum node S_1 with 3 children and a sum node S_2 with 2 children, are expressed using 5 sum nodes, each with a single child, followed by element-wise mixtures (drawn as sum nodes in boxes).

These element-wise mixtures can also be implemented with a single einsum operation, which we denote as *mixing layer*. To this end, let M be the number of sum nodes having more than one child, and D the maximal number of children (e.g. $D = 3$ in Fig. 3). We collect the K-dimensional probability vectors computed by the first sum layer in a $D \times M \times K$ tensor, where the first axis is zero-padded for sum nodes with less than D children. The mixing layer computes then a convex combination over the first dimension of this tensor. Constructing this tensor unfortunately involves some copy overhead, and wastes also some computation due to zero padding. However, allowing for sum nodes with multiple children lets us express a much wider range of structures than e.g. random binary trees, which is the only structure considered in (Peharz et al., 2019).

3.5. Exponential Families as Input Layer

For the leaves of EiNets we use log-densities of exponential families (EFs), which have the form

$$\log p = \log h(\mathbf{x}) + T(\mathbf{x})^T \theta - A(\theta), \tag{5}$$

where θ are the natural parameters, h is the so-called base measure, T is the sufficient statistic and A is the log-normalizer. Many parametric distributions can be expressed as EFs, e.g. Gaussian, Binomial, Categorical, Beta, etc. In order to facilitate learning using EM, we keep the parame-

ters in their *expectation* form ϕ (Sato, 1999). The natural parameters θ and expectation parameters ϕ are one-to-one, and connected via $\phi = \partial A(\theta)/\partial\theta$ and $\theta = \partial H(\phi)/\partial\phi$, where H denotes entropy. This dual parameterization allows us to implement EM on the abstract level of EFs, while particular instances of EFs are easily implemented by providing h, T, A, and the conversion $\theta(\phi)$. All EF densities can be computed in parallel with a handful of operations, e.g. inner product $T(\mathbf{x})^T\theta$, evaluating $A(\theta)$, etc. After having computed EFs for single variables, the leaves are modeled as fully factorized, i.e. each leaf is of the form $\log \mathsf{L}(\mathbf{x}) = \sum_{X\in\psi(\mathsf{L})} \log p_{\mathsf{L},X}(\mathbf{x})$, where $\log p_{\mathsf{L},X}(\mathbf{x})$ is the log EF density belonging to L and X, for each X in the scope of L.

3.6. Expectation-Maximization (EM)

A natural way to learn PCs is the EM algorithm, wich is known to rapidly increase the likelihood, especially in early iterations (Salakhutdinov et al., 2003). EM for PCs was derived in (Peharz et al., 2015; Desana & Schnörr, 2016; Peharz et al., 2017), leading to the following update rules for sum-weights and leaves:

$$n_{\mathsf{S},\mathsf{N}}(\mathbf{x}) = \frac{1}{\mathcal{P}(\mathbf{x})}\frac{\partial\mathcal{P}}{\partial\mathsf{S}}\mathsf{N}(\mathbf{x}), \quad p_{\mathsf{L}}(\mathbf{x}) = \frac{1}{\mathcal{P}(\mathbf{x})}\frac{\partial\mathcal{P}}{\partial\mathsf{L}}\mathsf{L}(\mathbf{x}), \tag{6}$$

$$w_{\mathsf{S},\mathsf{N}} \leftarrow \frac{w_{\mathsf{S},\mathsf{N}}\sum_{\mathbf{x}} n_{\mathsf{S},\mathsf{N}}(\mathbf{x})}{\sum_{\mathbf{x},\mathsf{N}\in\mathbf{ch}(\mathsf{S})} n_{\mathsf{S},\mathsf{N}}(\mathbf{x})}, \quad \phi_{\mathsf{L}} \leftarrow \frac{\sum_{\mathbf{x}} p_{\mathsf{L}(\mathbf{x})}T(\mathbf{x})}{\sum_{\mathbf{x}} p_{\mathsf{L}(\mathbf{x})}}, \tag{7}$$

where the sums in (7) range over all training examples $\mathbf{x}$. In (Peharz et al., 2017) and (Peharz et al., 2019), the derivatives $\frac{\partial\mathcal{P}}{\partial\mathsf{S}}$ and $\frac{\partial\mathcal{P}}{\partial\mathsf{L}}$ required for the expected statistics $n_{\mathsf{S},\mathsf{N}}$ and p_{L} were computed with an *explicitly implemented backwards-pass*, performed in the log-domain for robustness. Here we show that this implementation overhead can be avoided by leveraging automatic differentiation.

Recall that EiNets represent all probability values in the log-domain, thus the PC output is actually $\log\mathcal{P}(\mathbf{x})$, rather than $\mathcal{P}(\mathbf{x})$. Calling auto-diff on $\log\mathcal{P}(\mathbf{x})$ yields the following derivative for each sum-weight $w_{\mathsf{S},\mathsf{N}}$ (omitting argument $\mathbf{x}$): $\frac{\partial\log\mathcal{P}}{w_{\mathsf{S},\mathsf{N}}} = \frac{1}{\mathcal{P}}\frac{\partial\mathcal{P}}{\partial\mathsf{S}}\frac{\partial\mathsf{S}}{\partial w_{\mathsf{S},\mathsf{N}}} = \frac{1}{\mathcal{P}}\frac{\partial\mathcal{P}}{\partial\mathsf{S}}\mathsf{N}$, which is exactly $n_{\mathsf{S},\mathsf{N}}$ in (6), i.e. auto-diff readily provides the required expected statistics for sum nodes. In many frameworks, auto-diff readily accumulates the gradient by default, as required in (7), leading to an embarrassingly simple implementation for the E-step. For the M-step, we simply multiply the result of the accumulator with the current weights, and renormalize.

Furthermore, recall that the leaves are implemented as log-densities of an EF. Taking the gradient yields: $\frac{\partial\log\mathcal{P}}{\partial\log L} = \frac{1}{\mathcal{P}}\frac{\partial\mathcal{P}}{\partial\log L} = \frac{1}{\mathcal{P}}\frac{\partial\mathcal{P}}{\partial L}L$, which is p_{L} in (6). Thus, auto-diff also implements most of the EM update for the leaves. We simply need to accumulate both p_{L} and $p_{\mathsf{L}}\,T$ over the whole dataset, and use (7) to update the expectation parameters ϕ. Note, that both sum nodes and leafs can be updated using the same calls to auto-diff.

The classical EM algorithm uses a whole pass over the training set for a single update, which is computationally wasteful due to redundancies in the training data (Bottou, 1998). Similar to stochastic gradient descent (SGD), it is possible to define a stochastic version for EM (Sato, 1999). To this end, one replaces the sums over the entire dataset in (7) with sums over *mini-batches*, yielding $w^{mini}_{\mathsf{S},\mathsf{N}}$ and ϕ^{mini}_{L}. The full EM update is then replaced with gliding averages

$$w_{\mathsf{S},\mathsf{N}} \leftarrow (1-\lambda)\,w_{\mathsf{S},\mathsf{N}} + \lambda\,w^{mini}_{\mathsf{S},\mathsf{N}} \tag{8}$$

$$\phi_{\mathsf{L}} \leftarrow (1-\lambda)\,\phi_{\mathsf{L}} + \lambda\,\phi^{mini}_{\mathsf{L}}, \tag{9}$$

where $\lambda \in [0,1]$ is a step-size parameter. This *stochastic* version of EM introduces two hyper-parameters, step-size λ and the batch-size, which need to be set appropriately. Furthermore, unlike full-batch EM, stochastic EM does not guarantee that the training likelihood increases in each iteration. However, stochastic EM updates the parameters after each mini-batch and typically leads to faster learning.

Sato (1999) shows an interesting connection between stochastic EM and natural gradients (Amari, 1998). In particular, for any EF model $P(\mathbf{X},\mathbf{Z})$, where $\mathbf{X}$ are observed and $\mathbf{Z}$ latent variables, performing (8) and (9) is equivalent to SGD, where the gradient is pre-multiplied by the inverse Fisher information matrix of the *complete* model $P(\mathbf{X},\mathbf{Z})$. The difference to standard natural gradient is, that the natural gradient is defined via the inverse Fisher of the *marginal* $P(\mathbf{X})$ (Amari, 1998), rather than $P(\mathbf{X},\mathbf{Z})$. Sato's analysis also applies to EiNets, since smooth and decomposable PCs *are* EFs of the form $P(\mathbf{X},\mathbf{Z})$ (Peharz et al., 2017), where the latent variables are associated with sum nodes. Thus, (8) and (9) are in fact performing SGD with (a variant of) natural gradient.

4. Experiments

We implemented EiNets in PyTorch (Paszke et al., 2019). Code accompanying this paper can be found in a stand-alone repository;[4] moreover, the code will be incorporated and further developed in SPFlow (Molina et al., 2019), a general-purpose PC library.[5]

4.1. Efficiency Comparison

In order to demonstrate the efficiency of EiNets we compare with the two most prominent deep-learning implementations of PCs, namely LibSPN (Pronobis et al., 2017) and

[4] https://github.com/cambridge-mlg/EinsumNetworks

[5] https://github.com/SPFlow/SPFlow

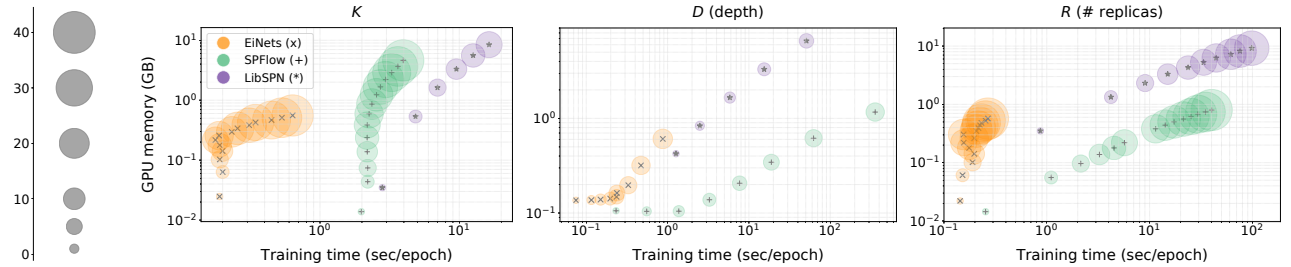


Figure 4. Illustration of training time and peak memory consumption of EiNets, SPFlow and LibSPN when training randomized binary PC trees, and varying hyper-parameters K (number of densities per sum/leaf), depth D, and number of replica R, respectively. The blob size directly corresponds to the respective hyper-parameter under change. The total number of parameters ranged within $10k - 9.4M$ (for varying K), $100k - 5.2M$ (for varying D), and $24k - 973k$ (for varying R). For LibSPN, some settings exhausted GPU memory and are therefore missing.

Table 1. Test log-likelihoods of various baselines and EiNets, on 20 binarized datasets. For each dataset, the best performing method is underlined. Results which are not statistically different from the best method (using a one sided t-test, $p = 0.05$) are in boldface. For oBMM and CCCP, no detailed results were available, so no significance testing was done with these methods.

dataset	oBMM	CCCP	Learn-SPN	ID-SPN	RAT-SPN	EiNet
nltcs	-6.070	-6.029	**-6.110**	**-6.020**	**-6.011**	**-6.015**
msnbc	-6.030	-6.045	-6.113	**-6.040**	**-6.039**	-6.119
kdd-2k	-2.140	-2.134	**-2.182**	**-2.134**	**-2.128**	**-2.183**
plants	-15.140	-12.872	-12.977	**-12.537**	-13.439	-13.676
jester	-53.860	-52.880	-53.480	**-52.858**	**-52.970**	**-52.563**
audio	-40.700	-40.020	**-40.503**	**-39.794**	**-39.958**	**-39.879**
netflix	-57.990	-56.782	-57.328	**-56.355**	-56.850	**-56.544**
accidents	-42.660	-27.700	-30.038	**-26.983**	-35.487	-35.594
retail	-11.420	-10.919	**-11.043**	**-10.847**	**-10.911**	**-10.916**
pumsb-star	-45.270	-24.229	-24.781	**-22.405**	-32.530	-31.954
dna	-99.610	-84.921	-82.523	**-81.211**	-97.232	-96.086
kosarek	-11.220	-10.880	**-10.989**	**-10.599**	**-10.888**	-11.029
msweb	-11.330	-9.970	-10.252	**-9.726**	-10.116	-10.026
book	-35.550	-35.009	**-35.886**	**-34.137**	**-34.684**	**-34.739**
each-movie	-59.500	-52.557	**-52.485**	**-51.512**	**-53.632**	**-51.705**
web-kb	-165.570	-157.492	**-158.204**	**-151.838**	**-157.530**	**-157.282**
reuters-52	-108.010	-84.628	**-85.067**	**-83.346**	**-87.367**	**-87.368**
20ng	-158.010	-153.205	-155.925	**-151.468**	**-152.062**	**-153.938**
bbc	-275.430	-248.602	**-250.687**	**-248.929**	**-252.138**	**-248.332**
ad	-63.810	-27.202	**-19.733**	**-19.053**	-48.472	-26.273

SPFlow (Molina et al., 2019). LibSPN is natively based on Tensorflow (Abadi M. et al., 2015), while SPFlow supports multiple backends. For our experiment, we used SPFlow with Tensorflow backend. We used randomized binary trees (RAT-SPNs) (Peharz et al., 2019) as a common benchmark: These PC structures are governed by two structural parameters, the *split-depth* D and *number of replica* R: Starting from the root sum node, they split the whole scope $\mathbf{X}$ into two randomized balanced parts, recursively until depth D, yielding a binary tree with 2^D leaves. This construction is repeated R times, yielding R random binary trees, which are mixed at the root.

As a sanity check, we first reproduce the density estimation results of RAT-SPNs (Peharz et al., 2019) on 20 binary datasets (Lowd & Davis, 2010; Van Haaren & Davis, 2012; Gens & Domingos, 2013), see Table 1. We see that EiNets largely achieve the same performance as RAT-SPNs, except on 'ad' where EiNets even outperform RAT-SPNs with a large margin. We conjecture that this difference stems from stochastic EM which might help EiNets to escape from local optima of the training likelihood. In Table 1, we also report several baselines, namely online Bayesian moment matching (oBMM) (Rashwan et al., 2016), concave-convex procedure (CCCP) (Zhao et al., 2016), LearnSPN (Gens & Domingos, 2013), and ID-SPN (Rooshenas & Lowd, 2014). Note that these baseline methods are PCs learned with custom code and are not easily scalable to bigger datasets. Also, note that Learn-SPN, CCCP and ID-SPN learn the PC structure from data, while the other methods use randomized structures.

We compare EiNets with two other deep-learning PC implementations[6] LibSPN and SPFlow in terms of i) training time, ii) memory consumption, and iii) inference time. To this end, we trained and tested them on synthetic data (Gaussian noise) with $N = 2000$ samples and $D = 512$ dimensions. As leaves we used single-dimensional Gaussians. We varied each structural hyper-parameter in the following ranges: depth $D \in \{1, \dots, 9\}$, replica $R \in \{1, \dots, 40\}$, and vector length of sums/leaves $K \in \{1, \dots, 40\}$; please see (Peharz et al., 2019) for the precise meaning of these hyper-parameters. When varying one hyper-parameter, we left the others to default values $D = 4$, $R = 10$, and $K = 10$. We ran this set of experiments on a GeForce RTX 2080 Ti, using a batch size of 100 samples.

In Figure 4 we see a comparison of the three implementations in terms of train time and GPU-memory peak consumption, where the circle radii represent the magnitude of the varied hyper-parameter. We see that EiNets tend to be **one or two orders of magnitude** faster (note the log-scale)

[6]There are further deep-learning implementations of PCs, which are, however, tailored to images (Sharir et al., 2016; Butz et al., 2019), and not general-purpose.

than the competitors, especially for large models. Also in terms of memory consumption EiNets scale gracefully. In particular, for large K, memory consumption is an order of magnitude lower than for LibSPN or SPFLow. This can be easily explained by the fact that our einsum operations do not generate product nodes explicitly in memory, while the other frameworks do. Moreover, EiNets also perform superior in terms of inference time. In particular for large models, they run again one or two orders of magnitude faster than the other implementations. For space reasons, these results are deferred to the supplementary.

4.2. EiNets as Generative Image Models

While PCs are an actively researched topic, their scalability issues have restricted them so far to rather small datasets like MNIST (LeCun et al., 1998). In this paper, we use PCs as generative model for street-view house numbers (SVHN) (Netzer et al., 2011), containing 32×32 RGB images of digits, and center-cropped CelebA (Liu et al., 2015), containing 128×128 RGB face images. For SVHN, we used the first $50k$ train images and concatenated it with the extra set, yielding a train set $581k$ images. We reserved the rest of the core train set, $23k$ images, as validation set. The test set comprises $26k$ images. For CelebA, we used the standard train, validation and test splits, containing $183k$, $10k$, and $10k$ images, respectively. The data was normalized before training (division by 255), but otherwise no preprocessing was applied. To the best or our knowledge, PCs have not been successfully trained on datasets of this size before.

We first clustered both datasets into 100 clusters using the *sklearn* implementation of k-means, and learned an EiNet for each of these clusters. We then used these 100 EiNets as mixture components of a mixture model, using the cluster proportions as mixture coefficients. Note that a mixture of PCs yields again a PC, and that this step is essentially the first step of *LearnSPN* (Gens & Domingos, 2013), one of the most prominent PC structure learners.

We trained EiNets using the image-tailored structure proposed in (Poon & Domingos, 2011), to which we refer as PD structure. The PD structure recursively decomposes the image into sub-rectangles using axis-aligned splits, displaced by a certain step-size Δ. Here, Δ serves as a structural hyperparameter, which governs the number of sum nodes according to $\mathcal{O}(\frac{1}{\Delta^3})$ (Peharz, 2015). Δ can also be a vector of step-sizes; in this case the largest value in Δ which fits in the longer side of a sub-rectangle is applied. The recursive splitting process stops, when a rectangle cannot be split by any value in Δ.

For the PD structure of each EiNet component, we used a step-size $\Delta = 8$ for SVHN and $\Delta = 32$ for CelebA, i.e. we applied 4 axis-aligned splits. We only applied vertical splits, in order to reduce artifacts stemming from the PD structure. As leaves, we used factorized Gaussians, i.e. Gaussians with diagonal covariance matrix, which were further factorized over the RGB channels. The vector length for sums and leaves was set to $K = 40$. After each EM update, we projected the Gaussian variances to the interval $[10^{-6}, 10^{-2}]$, corresponding to a maximal standard deviation of 0.1. Each component was trained for 25 epochs, using a batch size of 500 and EM stepsize of 0.5. In total, training lasted 5 hours for SVHN and 3 hours for CelebA, on an NVidia P100.

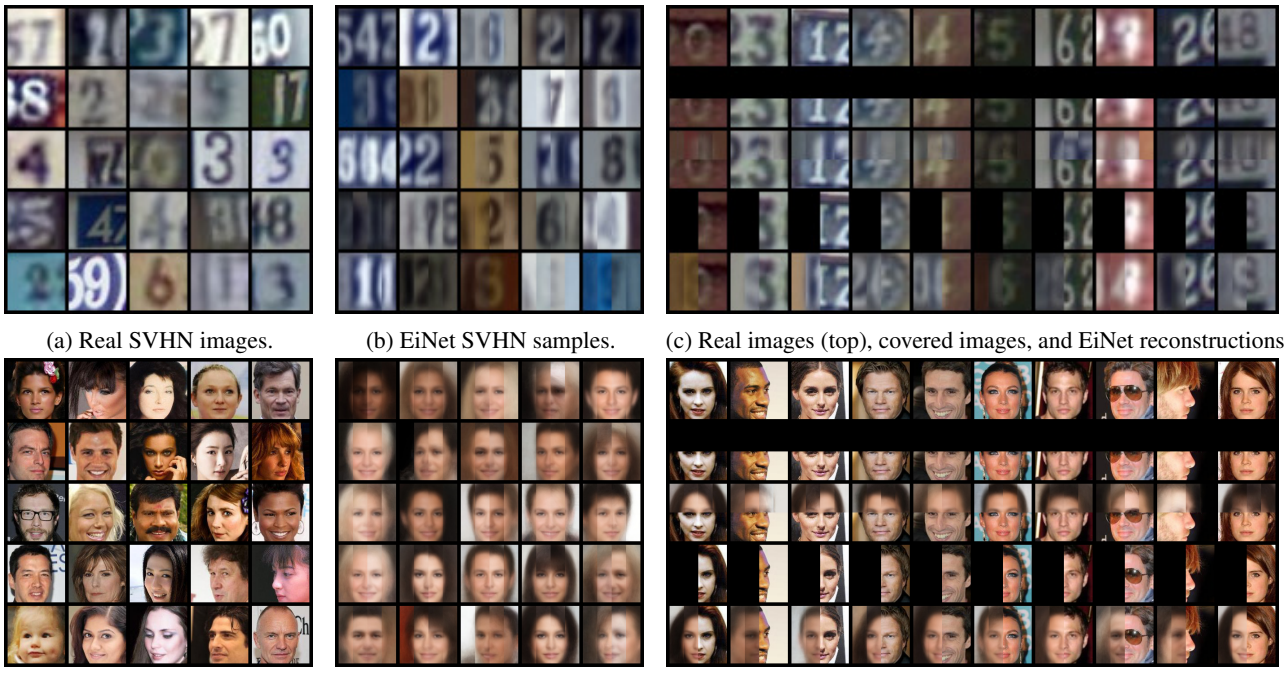
Original samples and samples from the EiNet mixture are shown in Figure 5, (a,b) for SVHN and (d,e) for CelebA, respectively. The SVHN samples are rather compelling, and some samples could be mistaken for real data samples. The CelebA samples are somewhat over-smoothed, which is a typical effect for likelihood-based approaches, but capture the main quality of faces well. In both cases, some "stripy" artifacts, typical for the PD architecture (Poon & Domingos, 2011), can be seen.

We computed two common measures to evaluate image quality, namely the Fréchet Inception Score (FID) (Heusel et al., 2017) and the log-likelihood of the test set. For the FID, we sampled artificial sample sets of the same size as the respective test sets. For SVHN, EiNets achieved a FID of 86.51 and a test log-likelihood of $-4.248\,\text{nats}$, normalized over samples, pixels and channels. For CelebA, EiNets achieved a FID of 105.44 (71.47 when re-scaling images to 64×64 using Lanczos resampling) and a normalized test log-likelihood of $-3.42\,\text{nats}$.

Although the image quality is not comparable to e.g. GAN models (Goodfellow et al., 2014), we want to stress that EiNets permit tractable inference, while GANs and most other generative models are restricted to sampling. In particular, tractable inference can be immediately used for image inpainting, as demonstrated in Figure 5, (c) for SVHN and (f) for CelebA. Here, we marked parts of test samples as missing (top or left image half) and reconstructed it by drawing a sample from the conditional distribution, conditioned on the visible image part. We see that the reconstructions are plausible, in particular for SVHN.

4.3. EiNets for Outlier Detection

A natural application of generative models is *outlier detection*, by monitoring the sample probability. For example, if we have trained an EiNet over the feature space of a classifier, an atypically low value for the density $\mathcal{P}(\mathbf{x})$ might indicate outlier data. In this case, we can trigger a reject option, rather than feeding the sample to the classifier. In (Peharz et al., 2019), it was demonstrated that PCs deliver very good outlier signals, even for (semi-)discriminative models. We ran a similar experiment as in (Peharz et al., 2019), and trained an EiNet on the training set of MNIST.



(a) Real SVHN images. (b) EiNet SVHN samples. (c) Real images (top), covered images, and EiNet reconstructions

(d) Real CelebA samples. (e) EiNet CelebA samples. (f) Real images (top), covered images, and EiNet reconstructions

Figure 5. Qualitative results of EiNets trained on RGB data. Top: SVHN (32×32). Bottom: CelebA (128×128).

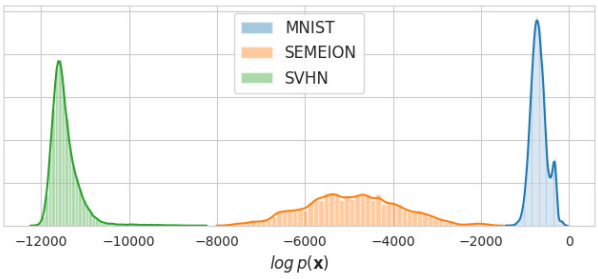


Figure 6. Histograms of sample-wise log-probabilities of the test sets of MNIST, SEMEION and SVHN, under EiNet trained on MNIST training data. Note that the histograms do not overlap.

Unlike as in (Peharz et al., 2019), who used Gaussian leaves, we trained our model on discrete pixel data and used Categorical distributions with 256 states. We used a PD structure (Poon & Domingos, 2011) with $\Delta = [7, 28]$, such that the 28×28 images are first split into 4×4 super-pixels, which are then further split into single pixels. We used distribution vectors of length $K = 40$. This structure yields are rather large model ($107M$ parameters) which we trained for 10 hours on a single NVidia P100, cross-validating over epochs (early stopping). The final test log-likelihood of this model was $-686.20\,\text{nats}$ per sample.

In Fig. 6 we see histograms over sample-wise log-probabilities for the MNIST test set, SEMEION (Buscema, 1998) and SVHN. We see that, in this case, the EiNet does a perfect job in discriminating inliers (MNIST) from outliers (SEMEION and SVHN). In fact, the smallest log-probability of an MNIST test sample was $-1362.91\,\text{nats}$, while the largest log-probabilities for SEMEION and SVHN were $-1796.84\,\text{nats}$ and $-8327.78\,\text{nats}$, respectively.

5. Conclusion

Probabilistic models form a spectrum of machine learning techniques. Most of the research is focused on representing and learning flexible and expressive models, but ignore the down-stream impact on the set of inference tasks which can be provably solved within the model. The philosophy of tractable modeling also aims to push the expressivity boundaries, but under the constraint to maintain a defined set of exact inference routines. Probabilistic circuits are certainly a central and prominent approach for this philosophy. In this paper, we addressed a major obstacle for PCs, namely their scalability in comparison to unconstrained models. Our improvements of training speed and memory-use reduction, both in the orders of one or two orders of magnitude, are compelling, and we hope that our results stimulates further developments in the area of tractable models.

Acknowledgements

Many thanks to Thomas Viehmann, for bringing einsums to PyTorch, and to Tim Rocktäschel, for his great tutorial on einsums for machine learning. RP: This project has received

funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie Grant Agreement No. 797223 — HYBSPN. GVdB: This work is partially supported by NSF grant IIS-1943641 and DARPA XAI grant #N66001-17-2-4032.

References

Abadi M. et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

Amari, S.-I. Natural gradient works efficiently in learning. *Neural computation*, 10(2):251–276, 1998.

Bottou, L. Online algorithms and stochastic approximations. In Saad, D. (ed.), *Online Learning and Neural Networks*. Cambridge University Press, 1998.

Buscema, M. *MetaNet*: The Theory of Independent Judges*, volume 33. 02 1998.

Butz, C. J., Oliveira, J. S., dos Santos, A. E., and Teixeira, A. L. Deep convolutional sum-product networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pp. 3248–3255, 2019.

Darwiche, A. Compiling knowledge into decomposable negation normal form. In *IJCAI*, volume 99, pp. 284–289, 1999.

Darwiche, A. Decomposable negation normal form. *Journal of the ACM (JACM)*, 48(4):608–647, 2001.

Darwiche, A. A logical approach to factoring belief networks. *KR*, 2:409–420, 2002.

Darwiche, A. A differential approach to inference in Bayesian networks. *Journal of the ACM*, 50(3):280–305, 2003.

de Campos, C. New complexity results for MAP in Bayesian networks. In *Proceedings of IJCAI*, pp. 2100–2106, 2011.

Dennis, A. and Ventura, D. Learning the architecture of sum-product networks using clustering on variables. In *Proceedings of NIPS*, pp. 2042–2050, 2012.

Desana, M. and Schnörr, C. Expectation maximization for sum-product networks as exponential family mixture models. *arXiv preprint arXiv:1604.07243*, 2016.

Gens, R. and Domingos, P. Learning the structure of sum-product networks. *Proceedings of ICML*, pp. 873–880, 2013.

Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. Generative adversarial nets. In *Proceedings of NIPS*, pp. 2672–2680, 2014.

Heusel, M., Ramsauer, H., Unterthiner, T., Nessler, B., and Hochreiter, S. Gans trained by a two time-scale update rule converge to a local nash equilibrium. In *Advances in neural information processing systems*, pp. 6626–6637, 2017.

Khosravi, P., Choi, Y., Liang, Y., Vergari, A., and Van den Broeck, G. On tractable computation of expected predictions. In *Advances in Neural Information Processing Systems*, pp. 11169–11180, 2019a.

Khosravi, P., Liang, Y., Choi, Y., and Van den Broeck, G. What to expect of classifiers? reasoning about logistic regression with missing features. In *Proceedings of IJCAI*, pp. 2716–2724, 2019b.

Kingma, D. P. and Welling, M. Auto-encoding variational Bayes. In *ICLR*, 2014. arXiv:1312.6114.

Kisa, D., den Broeck, G. V., Choi, A., and Darwiche, A. Probabilistic Sentential Decision Diagrams. In *KR*, 2014.

Koller, D. and Friedman, N. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009. ISBN 0-262-01319-3.

Larochelle, H. and Murray, I. The neural autoregressive distribution estimator. In *Proceedings of AISTATS*, pp. 29–37, 2011.

LeCun, Y., Cortes, C., and Burges, C. J. C. The MNIST database of handwritten digits. http://yann.lecun.com/exdb/mnist/.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Liu, Z., Luo, P., Wang, X., and Tang, X. Deep learning face attributes in the wild. In *Proceedings of International Conference on Computer Vision (ICCV)*, December 2015.

Lowd, D. and Davis, J. Learning Markov network structure with decision trees. In *Proceedings of the 10th IEEE International Conference on Data Mining*, pp. 334–343. IEEE Computer Society Press, 2010.

MacKay, D. J. C. Bayesian neural networks and density networks. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 354(1):73–80, 1995.

Molina, A., Vergari, A., Stelzner, K., Peharz, R., Subramani, P., Mauro, N. D., Poupart, P., and Kersting, K. Spflow: An easy and extensible library for deep probabilistic learning using sum-product networks, 2019.

Netzer, Y., Wang, T., Coates, A., A-Bissacco, Wu, B., and Ng, A. Y. Reading digits in natural images with unsupervised feature learning. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning 2011*, 2011.

Papamakarios, G., Nalisnick, E., Rezende, D. J., Mohamed, S., and Lakshminarayanan, B. Normalizing flows for probabilistic modeling and inference. *arXiv preprint arXiv:1912.02762*, 2019.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. 2019. URL https://pytorch.org/.

Pearl, J. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988. ISBN 1558604790.

Peharz, R. *Foundations of Sum-Product Networks for Probabilistic Modeling*. PhD thesis, Graz University of Technology, 2015.

Peharz, R., Tschiatschek, S., Pernkopf, F., and Domingos, P. On theoretical properties of sum-product networks. In *Proceedings of AISTATS*, pp. 744–752, 2015.

Peharz, R., Gens, R., Pernkopf, F., and Domingos, P. On the latent variable interpretation in sum-product networks. *IEEE transactions on pattern analysis and machine intelligence*, 39(10):2030–2044, 2017.

Peharz, R., Vergari, A., Stelzner, K., Molina, A., Trapp, M., Shao, X., Kersting, K., and Ghahramani, Z. Random sum-product networks: A simple and effective approach to probabilistic deep learning. In *Proceedings of UAI*, 2019.

Pipatsrisawat, K. and Darwiche, A. New compilation languages based on structured decomposability. In *AAAI*, volume 8, pp. 517–522, 2008.

Poon, H. and Domingos, P. Sum-product networks: A new deep architecture. In *Proceedings of UAI*, pp. 337–346, 2011.

Pronobis, A., Ranganath, A., and Rao, R. Libspn: A library for learning and inference with sum-product networks and tensorflow. In *Principled Approaches to Deep Learning Workshop*, 2017.

Rahman, T., Kothalkar, P., and Gogate, V. Cutset networks: A simple, tractable, and scalable approach for improving the accuracy of chow-liu trees. In *Joint European conference on machine learning and knowledge discovery in databases*, pp. 630–645, 2014.

Rashwan, A., Zhao, H., and Poupart, P. Online and distributed bayesian moment matching for parameter learning in sum-product networks. In *AISTATS*, pp. 1469–1477, 2016.

Rezende, D. J. and Mohamed, S. Variational inference with normalizing flows. In *Proceedings of ICML*, pp. 1530–1538, 2015.

Rooshenas, A. and Lowd, D. Learning Sum-Product Networks with Direct and Indirect Variable Interactions. In *Proceedings of ICML*, pp. 710–718, 2014.

Salakhutdinov, R., Roweis, S. T., and Ghahramani, Z. Optimization with em and expectation-conjugate-gradient. In *Proceedings of ICML*, pp. 672–679, 2003.

Sato, M.-a. Fast learning of on-line em algorithm. *Rapport Technique, ATR Human Information Processing Research Laboratories*, 1999.

Sharir, O., Tamari, R., Cohen, N., and Shashua, A. Tractable generative convolutional arithmetic circuits. *arXiv preprint arXiv:1610.04167*, 2016.

Shen, Y., Choi, A., and Darwiche, A. Tractable operations for arithmetic circuits of probabilistic models. In *Advances in Neural Information Processing Systems 29*, pp. 3936–3944. 2016.

Trapp, M., Peharz, R., Ge, H., Pernkopf, F., and Ghahramani, Z. Bayesian learning of sum-product networks. *Proceedings of NeurIPS*, 2019a.

Trapp, M., Peharz, R., and Pernkopf, F. Optimisation of overparametrized sum-product networks. *arXiv preprint arXiv:1905.08196*, 2019b.

Uria, B., Côté, M.-A., Gregor, K., Murray, I., and Larochelle, H. Neural autoregressive distribution estimation. *The Journal of Machine Learning Research*, 17 (1):7184–7220, 2016.

Van Haaren, J. and Davis, J. Markov network structure learning: A randomized feature generation approach. In *Proceedings of the 26th Conference on Artificial Intelligence*. AAAI Press, 2012.

Vergari, A., Di Mauro, N., and Van den Broeck, G. Tractable probabilistic models: Representations, algorithms, learning, and applications. http://web.cs.ucla.edu/~guyvdb/slides/TPMTutorialUAI19.pdf, 2019. Tutorial at UAI 2019.

Zhao, H., Melibari, M., and Poupart, P. On the relationship between sum-product networks and Bayesian networks. In *Proceedings of ICML*, pp. 116–124, 2015.

Zhao, H., Poupart, P., and Gordon, G. J. A unified approach for learning the parameters of sum-product networks. In *Proceedings of NIPS*, pp. 433–441. 2016.

Supplementary
Einsum Networks: Fast and Scalable Learning of Tractable Probabilistic Circuits

Anonymous Authors[1]

1. Inference Time Comparison

Section 4.1, in the main paper compared training time and memory consumption for EiNets, LibSPN (Pronobis et al., 2017) and SPFlow (Molina et al., 2019), showing that EiNets scale much more gracefully than its competitors. The same holds true for inference time. Fig. 1 shows the results corresponding to Fig. 3 in the main paper, but for inference time per sample rather than training time per epoch. Inference was done for a batch of 100 test samples for each model, i.e. the displayed inference time is $1/100$ of the evaluation time for the whole batch. Again, we see significant speedups for EiNets, of up to three orders of magnitude (for maximal depth and EiNet vs. SPFlow).

References

Molina, A., Vergari, A., Stelzner, K., Peharz, R., Subramani, P., Mauro, N. D., Poupart, P., and Kersting, K. Spflow: An easy and extensible library for deep probabilistic learning using sum-product networks, 2019.

Pronobis, A., Ranganath, A., and Rao, R. Libspn: A library for learning and inference with sum-product networks and tensorflow. In *Principled Approaches to Deep Learning Workshop*, 2017.

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

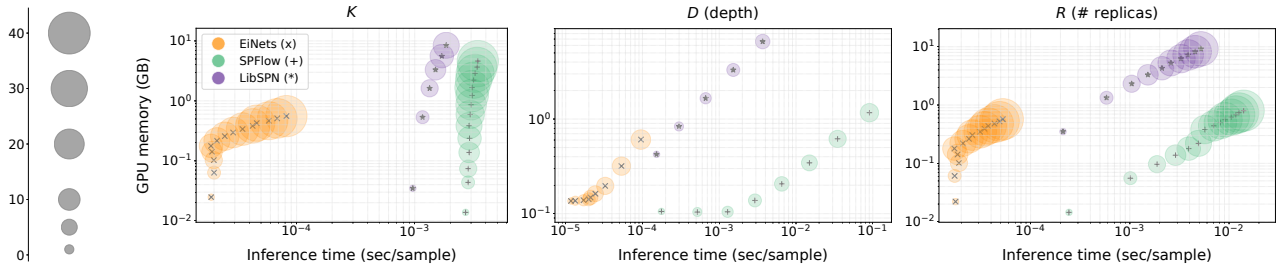


Figure 1. Illustration of inference time and peak memory consumption of EiNets, SPFlow and LibSPN when training randomized binary PC trees, and varying hyper-parameters K (number of densities per sum/leaf), depth D, and number of replica R, respectively. The blob size directly corresponds to the respective hyper-parameter under change. The total number of parameters ranged within $10k - 9.4M$ (for varying K), $100k - 5.2M$ (for varying D), and $24k - 973k$ (for varying R). For LibSPN, some settings exhausted GPU memory and are therefore missing.