Sparse Probabilistic Circuits via Pruning and Growing

Anonymous Author(s)
Affiliation
Address
email

Abstract

Probabilistic circuits (PCs) are a tractable representation of probability distributions allowing for exact and efficient computation of likelihoods and marginals. There has been significant recent progress on improving the scale and expressiveness of PCs. However, PC training performance plateaus as model size increases. We discover that most capacity in existing large PC structures is wasted: fully-connected parameter layers are only sparsely used. We propose two operations: *pruning* and *growing*, that exploit the sparsity of PC structures. Specifically, the pruning operation removes unimportant sub-networks of the PC for model compression and comes with theoretical guarantees. The growing operation increases model capacity by increasing the dimensions of latent states. By alternatingly applying pruning and growing, we increase the capacity that is meaningfully used, allowing us to significantly scale up PC learning. Empirically, our learner achieves state-of-the-art likelihoods on MNIST-family image datasets and an Penn Tree Bank language data compared to other PC learners and less tractable deep generative models such as flow-based models and variational autoencoders (VAEs).

1 Introduction

Probabilistic circuits (PCs) [33, 2] are a unifying framework to abstract from a multitude of tractable probabilistic models. The key property that separates PCs from other deep generative models such as flow-based models [24] and VAEs [13] is their *tractability*. It enables them to compute various queries, including marginal probabilities, exactly and efficiently [34]. Therefore, PCs are increasingly used in inference-demanding applications such as enforcing algorithmic fairness [1, 3], making predictions under missing data [12], and data compression [19].

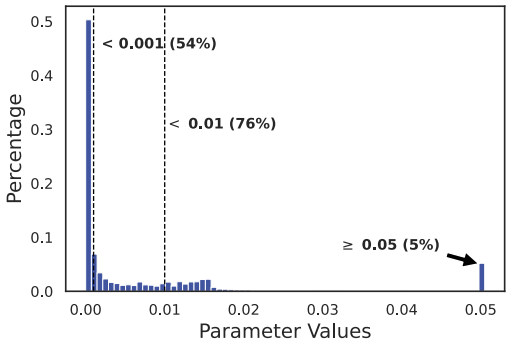


Figure 1: Histogram of parameter values for a state-of-the-art PC with 2.18M parameters on MNIST.

Recent advanced of PC learning [28], regularization [30, 18] and efficient parallelism implementation [25] have been pushing the limits of PC's expressivity and scalability such that PCs can even match the performance of less tractable deep generative models such as Flows and VAEs. This lead to a trend of building larger PCs. However, PC training performance plateaus as model size increases. This indicates that to go even further, simply scaling up might not suffice, and we need to be better at using the capacity available.

Submitted to 36th Conference on Neural Information Processing Systems (NeurIPS 2022). Do not distribute.

We discover that this might be caused by the fact that the capacity of large PCs are wasted. As shown in Figure 1, most parameters in a PC with 2.18M parameters have close-to-zero values, which have little effect to the PC distribution. Since existing PC structures usually have fully-connected parameter layers [18, 28], this indicates that the parameter values only are sparsely used.

In this work, we propose to better exploit the sparsity of large PC models by two structure learning primitives — *pruning* and *growing*. Specifically, the goal of the pruning operation is to identify and remove unimportant sub-networks of a PC. This is done by quantifying the importance of PC edges w.r.t. a dataset using *circuit flows*, a theoretically-grounded metric that upper bounds the drop of log-likelihood caused by pruning. Compared to L1 regularization, the proposed pruning operator is more informed by the PC semantics, and hence quantifies the global effects of pruning much more effectively. Empirically, the proposed pruning method achieves a compression rate of 80-98% with at most 1% drop in likelihood on various PCs.

The proposed growing operation increases the model size by copying its existing components and injecting noise. In particular, when applied to PCs compressed by the pruning operation, growing produces larger PCs that can be optimized to achieve better performance. Applying pruning and growing iteratively can greatly refine the structure and parameters of a PC. Empirically, the log-likelihoods metric can improve from 2% to 10% after a few iterations. Compared to existing PC learners as well as less tractable deep generative models such as VAEs and flow-based models, our proposed method achieves state-of-the-art density estimation results on image datasets including MNIST, EMNIST, FashionMNIST, and the Penn Tree Bank language modeling task.

2 Probabilistic Circuits

Probabilistic circuits (PCs) [33, 2] model probability distributions with a structured computation graph. They are an umbrella term for a large family of tractable probabilistic models including arithmetic circuits [7, 8], sum-product networks(SPN) [27], cutset networks [28], and-or search spaces [22], and probabilistic sentential decision diagrams [15]. The syntax and semantics of PCs are defined as follows.

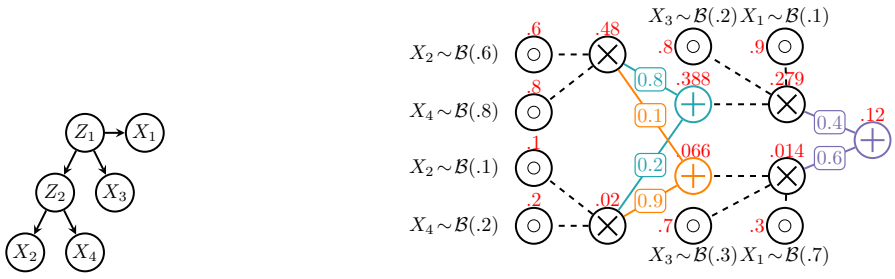
Definition 1 (Probabilistic Circuit)**.** A PC $\mathcal{C} := (\mathcal{G}, \boldsymbol{\theta})$ represents a joint probability distribution $p(\mathbf{X})$ over random variables $\mathbf{X}$ through a directed acyclic (computation) graph (DAG) $\mathcal{G}$ parameterized by $\boldsymbol{\theta}$. Similar to neural networks, each node in the DAG defines a computational unit. Specifically, the DAG $\mathcal{G}$ consists of three types of units — *input*, *sum*, and *product*. Every leaf node in $\mathcal{G}$ is an input unit; every inner unit n (i.e., sum or product) receives *inputs* from its children $\mathsf{in}(n)$, and computes *output*, which encodes a probability distribution p_n defined recursively as follows:

$$p_n(\boldsymbol{x}) := \begin{cases} f_n(\boldsymbol{x}) & \text{if } n \text{ is an input unit} \\ \prod_{c \in \mathsf{in}(n)} p_c(\boldsymbol{x}) & \text{if } n \text{ is a product unit} \\ \sum_{c \in \mathsf{in}(n)} \theta_{c|n} \cdot p_c(\boldsymbol{x}) & \text{if } n \text{ is a sum unit} \end{cases} \tag{1}$$

where $f_n(\boldsymbol{x})$ is a univariate input distribution (e.g, Gaussian, Categorical), and $\theta_{c|n}$ denotes the parameter that corresponds to edge (n, c) in the DAG. For every sum unit n, its input parameters sum up to one, i.e., $\sum_{c \in \mathsf{in}(n)} \theta_{c|n} = 1$. Intuitively, a product unit defines a factorized distribution over its inputs, and a sum unit represents a mixture over its input distributions with weights $\{\theta_{c|n} : c \in \mathsf{in}(n)\}$. Finally, the probability distribution of a PC (i.e., $p_{\mathcal{C}}$) is defined as the distribution represented by its root unit r (i.e., $p_r(\boldsymbol{x})$), that is, its output neuron. The size of a PC, denoted $|\mathcal{C}| = |\boldsymbol{\theta}|$, is the number of parameters in $\mathcal{C}$. We assume w.l.o.g that a PC alternates between layers of sum and product units before reaching its inputs. Figure 2 shows an example of PC.

As hinted by it definition, computing the (log)likelihood of PC $\mathcal{C}$ given a sample $\boldsymbol{x}$ is simply evaluating its computation units in $\mathcal{G}$ in a feedforward manner following Equation 1.

The key property that separates PCs from other deep probabilistic models such as flows [10] and VAEs [13] is their *tractability*, which is the ability to exactly and efficiently answer various probabilistic queries. This paper focuses on PCs that support linear time (w.r.t. model size) marginal probability



(a) (b)

Figure 2: A smooth and decomposable PC (2b) and equivalent Bayesian network (2a). The BN is over 4 variables $\mathbf{X} = \{X_1, X_2, X_3, X_4\}$ and 2 hidden variables $\mathbf{Z} = \{Z_1, Z_2\}$ with hidden states $h = 2$. The feedforward computation order is from left to right; $\odot$ are input Bernoulli distributions, $\bigotimes$ are products, and $\bigoplus$ are sums; solid edges are weighted with parameters, and dashed edges are non-parameterized. The probability of each unit given input assignment $\bar{x_1}x_2\bar{x_3}x_4$ is labeled in red.

computation, as they are increasingly used in downstream applications such as data compression [20] and making predictions under missing data [12], and also achieve on-par expressiveness [20, 18, 17]. To support efficient marginal inference, PCs need to be *smooth* and *decomposable*.

Definition 2 (Smoothness and Decomposability [9])**.** For a PC, the *scope* $\phi(n)$ of a PC unit n is the input variables that a unit depends on; then, (1) a product unit is *decomposable* if its children have disjoint scope; (2) a sum unit is *smooth* if its children have identical scope. A PC is decomposable if all of its produce units are decomposable; a PC is smooth if all of its sum units are smooth.

Decomposability ensures that every product unit encodes a well-defined factorized distribution over disjoint sets of variables; smoothness ensures that the mixture components of every sum units are well-defined over the same set of variables. Both structural properties will be the key to guaranteeing the effectiveness of the structure learning algorithms proposed in the following sections.

3 Probabilistic Circuit Model Compression via Pruning

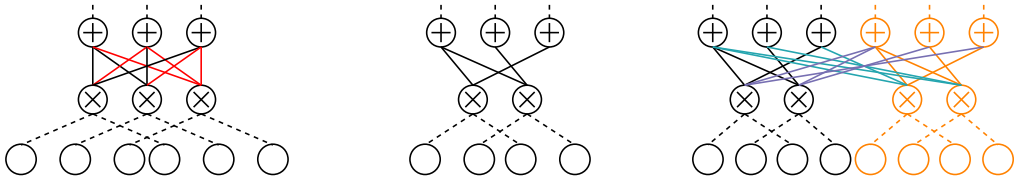
Figure 1 shows that most parameters in a large PC are very small. Given that these parameters are weights associated with mixture (sum unit) components, there are edges and sub-circuits that have little impact on the sum unit output. Hence, by pruning away these unimportant components, it is possible to significantly reduce model size while maximally retaining model expressiveness.

Let $\mathcal{C}_{\setminus\mathcal{E}}$ denote the PC $\mathcal{C}$ where the edges in $\mathcal{E}$ are removed from the circuit, and each sum unit is renormalized such that its parameters sum to one. Given a PC $\mathcal{C}$ and a dataset $\mathcal{D}$, our goal is to efficiently identify a set of k edges $\mathcal{E}$ such that the performance gap between the pruned PC $\mathcal{C}_{\setminus\mathcal{E}}$ and the original PC $\mathcal{C}$ is minimized:

$$\operatorname*{argmin}_{\mathcal{E}} \mathcal{LL}(\mathcal{D}, \mathcal{C}) - \mathcal{LL}(\mathcal{D}, \mathcal{C}_{\setminus\mathcal{E}}) \qquad \text{such that} \quad \mathcal{E} \subseteq \{(n, c) : \theta_{c|n} \in \boldsymbol{\theta}\} \quad \text{and} \quad |\mathcal{E}| = k, \tag{2}$$

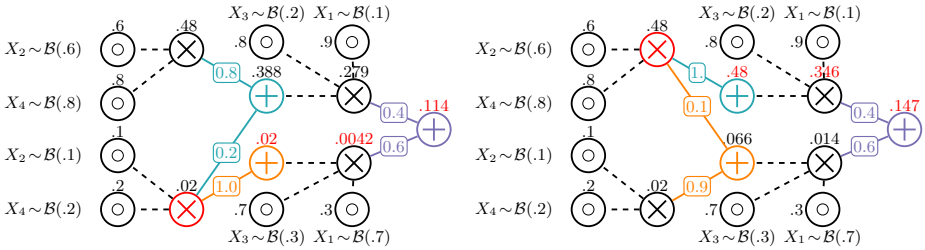
where $\mathcal{LL}(\mathcal{D}, \mathcal{C}) = \frac{1}{|\mathcal{D}|} \sum_{\boldsymbol{x} \in \mathcal{D}} \log p_{\mathcal{C}}(\boldsymbol{x})$ is the averaged log-likelihood of PC $\mathcal{C}$ given dataset $\mathcal{D}$. The edges $\mathcal{E}$ are chosen among all parameterized edges (i.e., all input edges of sum units). Figure 3b illustrates the result of pruning five (red) edges from the PC in Figure 3a.

Pruning by parameters. The parameter value statistics in Figure 1 suggest that a natural criteria for pruning edges is the magnitude of its corresponding parameter $\theta_{c|n}$. This leads to thee EPARAM heuristic, which selects k edges with the smallest parameters. However, edge parameters themselves are insufficient to quantify the importance of inputs to a sum unit in the entire PC's distribution. The parameters of a sum unit are normalized to be 1 so they only contain local information about the mixture components. Specifically, $\theta_{c|n}$ merely defines the relative importance of edge (n, c) in the joint distribution represented by its corresponding sum unit n. Figure 4a illustrates what happens when the edge with the smallest parameter is pruned from the PC in Figure 2. However, as shown



(a) PC with fully connected layers (b) PC after pruning operation (c) PC after growing operation

Figure 3: A demonstration of the pruning and growing operation. From 3a to 3b, the red edges are pruned. From 3b to 3c, the nodes are doubled, and each parameter is copied 3 times: new parent to new child (orange), new parent to old child (purple), and old parent to new child (green).



(a) EPARAM removes the orange edge $\theta = 0.1$ (b) EFLOW removes the blue edge $\theta = 0.2$

Figure 4: A case study of comparing pruning heuristics pruned from PC in Figure 2 given sample $\boldsymbol{x} = \bar{x_1} x_2 \bar{x_3} x_4$. The changed probability is colored in red. Both heuristics increase the weights of red products, since $X_2 \sim \mathcal{B}(.6), X_4 \sim \mathcal{B}(.8)$ is more consistent with sample $\boldsymbol{x}$, pruning by flows gets higher likelihoods.

in Figure 4b, pruning another edge delivers better likelihoods as it accounts more for the "global influence" of edges on the PC's output. This global influence is highly related to PC semantics and we will introduce it next along with its corresponding heuristics EFLOW.

Pruning by generative significance. A more informed pruning strategy needs to consider the global impact of edges on the distribution represented by the output of the PC. To achieve this, instead of viewing the distribution $p_{\mathcal{C}}$ in a feedforward manner following Equation 1, we quantify the significance of a unit or edge by the probability that it will be "activated" when drawing samples from the PC. Indeed, if the presence of an edge is hardly ever relevant to the generative sampling process, removing it will not significantly affect the PC's distribution.

Algorithm 1 shows that the PC sampling process proceeds in a top-down manner: a queue Q is initialized with the root unit (line 1). The algorithm then processes every unit in the queue until it is empty. For a sum unit n (lines 6-7), the sampler randomly adds one of its input units to the queue according to the categorical distribution defined by sum parameters $\{\theta_{c|n} : c \in \mathsf{in}(n)\}$; for a product unit (line 5), all its inputs are added to the queue; for an input unit n defined on variable X (line 4), the algorithm randomly samples value x according to its input distribution. Algorithm 1 is designed to sample instances following the PC distribution, therefore the probability of adding a unit n to the queue Q is the probability that n will be sampled, which provides a good measure of the importance of unit n to the PC as a whole. Following this intuition, pruning edges with the least probability of appearing in Q, which we call the *top-down probability* of n, defines a reasonable pruning strategy.

Definition 3 (Top-down probability)**.** The top-down probability of each unit in a PC $\mathcal{C}$ with parameters $\boldsymbol{\theta}$ is defined as follows. The top-down probability of the output unit r is one: $p^{\mathtt{TD}}_{r,\boldsymbol{\theta}}(r) = 1$. The top-down probability of sum and product units n is defined recursively as follows:

$$p^{\mathtt{TD}}_{r,\boldsymbol{\theta}}(n) = \sum_{m \in \mathsf{pa}_{\mathrm{sum}}(n)} \theta_{n|m} \cdot p^{\mathtt{TD}}_{r,\boldsymbol{\theta}}(m) + \sum_{m' \in \mathsf{pa}_{\mathrm{prod}}(n)} p^{\mathtt{TD}}_{r,\boldsymbol{\theta}}(m'),$$

where $\mathsf{pa}_{\mathrm{sum}}(n)$ and $\mathsf{pa}_{\mathrm{prod}}(n)$ are the sum and product units that take n as input, respectively. The top-down probability of a sum edge (n, c) is defined as $p^{\mathtt{TD}}_{r,\boldsymbol{\theta}}(n, c) = \theta_{c|n} \cdot p^{\mathtt{TD}}_{r,\boldsymbol{\theta}}(n)$.

Algorithm 1: PC Sampling

Input : a PC $\mathcal{C}$ defined on variables $\mathbf{X}$
Output : an instance $\boldsymbol{x} \sim p_{\mathcal{C}}$

$Q \leftarrow$ a queue initialized with the root node r of $\mathcal{C}$
while Q *is not empty* **do**
  $n \leftarrow Q.\mathrm{pop}()$
  if n is an input unit **then** sample the value of $\mathrm{var}(n)$ following the distribution defined by n
  else if n is a product unit **then** $Q.\mathrm{push}(c)$ for each $c \in \mathsf{in}(n)$
  else if n *is a sum unit* **then**
    sample $i \sim \mathrm{Categorical}(\{\theta_{c|n} : c \in \mathsf{in}(n)\})$; $Q.\mathrm{push}(c_i)$, where c_i is the ith input of n

Following this inductive definition, the top-down probability of all PC units and sum edges can be computed in a single backward pass over the PC's computation graph.

Pruning by circuit flows. Despite its ability to capture global information of PC parameters, top-down probability is not tailored to a specific dataset. The top-down probability $p^{\mathtt{TD}}_{\mathcal{C},\boldsymbol{\theta}}(n)$ represents the probability of reaching n in an unconditional sampling process. However, the pruning objective of Equation 2 requires measuring the probability of reaching certain units/edges in the sampling process given that the sampled instance is some $\boldsymbol{x} \in \mathcal{D}$. To bridge this gap, we define circuit flow as a sample-dependent version of the top-down probability.

Definition 4 (Circuit Flow[1])**.** For a given $\mathcal{C} = (\mathcal{G}, \boldsymbol{\theta})$ and input $\boldsymbol{x}$, let $\boldsymbol{\theta}^{\boldsymbol{x}}$ denote a new set of parameters such that $\theta^{\boldsymbol{x}}_{c|n}$ is the probability of component c in the mixture represented by sum unit n *after observing sample* $\boldsymbol{x}$. The node flow $\mathrm{F}_n(\boldsymbol{x})$ of a unit n is then defined as the top-down probability under this reparameterization of the circuit:

$$\mathrm{F}_n(\boldsymbol{x}) = p^{\mathtt{TD}}_{r,\boldsymbol{\theta}^{\boldsymbol{x}}}(n), \qquad \text{where} \quad \theta^{\boldsymbol{x}}_{c|n} = \frac{\theta_{c|n} \cdot p_c(\boldsymbol{x})}{\sum_{c' \in \mathsf{in}(n)} \theta_{c'|n} \cdot p_{c'}(\boldsymbol{x})}.$$

Similarly, the edge flow $\mathrm{F}_{n,c}(\boldsymbol{x})$ w.r.t. PC $\mathcal{C}$ and sample $\boldsymbol{x}$ is defined by $p^{\mathtt{TD}}_{r,\boldsymbol{\theta}^{\boldsymbol{x}}}(n, c)$. We further define $\mathrm{F}_{n,c}(\mathcal{D}) = \sum_{\boldsymbol{x} \in \mathcal{D}} \mathrm{F}_{n,c}(\boldsymbol{x})$ as the *aggregate flow* over $\mathcal{D}$.

By using the sample-conditioned parameters $\boldsymbol{\theta}^{\boldsymbol{x}}$, $\mathrm{F}_n(\boldsymbol{x})$ defines the probability of reaching unit n in Algorithm 1, given that the sampled instance is $\boldsymbol{x}$. Therefore, edge flow $\mathrm{F}_{n,c}(\boldsymbol{x})$ is a natural metric of the importance of edge (n, c) given $\boldsymbol{x}$. Intuitively, the circuit flow measures how many expected samples "flow" through certain edges. We name EFLOW as the heuristic that prunes edges with the smallest aggregate flow.

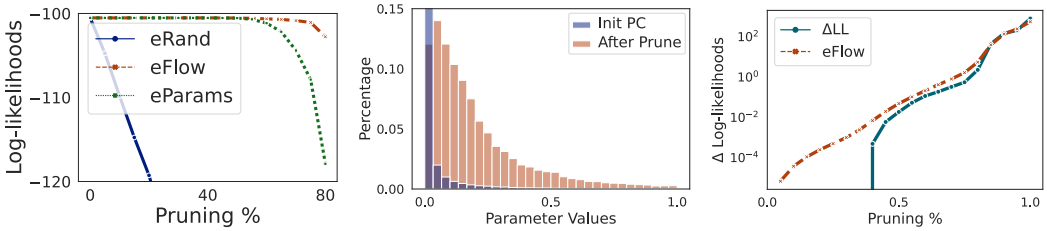
Empirical Analysis. Figure 5a compares the effect of pruning heuristics EPARAM, EFLOW, as well as an uninformed strategy, prune randomly, which we denote as ERAND. It shows that both EPARAM and EFLOW are reasonable pruning strategy, however, as we increase the percentage of pruned parameters, EFLOW has less log-likelihoods drop compared with EPARAM. Using EFLOW heuristics we can pruning up to 80% of the parameters without much log-likelihoods drop. As shown in Figure 5b, the parameter distribution is more balanced after pruning compared to Figure 1, indicating a higher significance of each edge. Section 6 will provide more empirical results. Before that, we first theoretically verify the effectiveness of the EFLOW heuristic in the next section.

4 Bounding and Approximating the Loss of Likelihood

This section establish a theoretical upper bound on the loglikelihood drop $\Delta\mathcal{LL}(\mathcal{D}, \mathcal{C}, \mathcal{E}) = \mathcal{LL}(\mathcal{D}, \mathcal{C}) - \mathcal{LL}(\mathcal{D}, \mathcal{C}_{\setminus\mathcal{E}})$ (cf. Equation 2) caused by pruning away edges $\mathcal{E}$. Interestingly, the EFLOW heuristic proposed in the previous section is a good approximation of the derived upper bound.

We start from the case of pruning one edge (i.e., $k = 1$ in Equation 2). In this case, the loss of likelihood can be quantified exactly using flows and edge parameters:

[1] Earlier work defined "circuit flow" or "expected circuit flow" in the context of parameter learning [3, 18, 5], without observing the connection to sampling. We contribute its more intuitive sampling semantics here.



(a) (b) (c)

Figure 5: (5a): Comparison of pruning heuristics (1) ERAND (2) EPARAM, (3) EFLOW for different percentage (5b): Histogram of parameters before/after the pruning operation. (5c): Comparing the actual log-likelihood drop (ΔLL) and quantity computed from EFLOW heuristics (which is also the approximated upper bound in Equation 3) for different percentage of pruned parameters (x-axis).

Theorem 1 (Log-likelihood drop of pruning one edge)**.** *For a PC $\mathcal{C}$ and a dataset $\mathcal{D}$, the loss of log-likelihood by pruning away edge (n, c) is*

$$\Delta\mathcal{LL}(\mathcal{D}, \mathcal{C}, \mathcal{E}) = \frac{1}{|\mathcal{D}|} \sum_{\boldsymbol{x} \in \mathcal{D}} \log \left(\frac{1 - \theta_{c|n}}{1 - \theta_{c|n} + \theta_{c|n} \, \mathrm{F}_n(\boldsymbol{x}) - \mathrm{F}_{n,c}(\boldsymbol{x})} \right) \leq -\frac{1}{|\mathcal{D}|} \sum_{\boldsymbol{x} \in \mathcal{D}} \log(1 - \mathrm{F}_{n,c}(\boldsymbol{x})).$$

See proof in Appendix B.1. By computing the second term from Theorem 1, we can pick the edge with the smallest log-likelihood drop. Additionally, the third term characterizes the log-likelihood drop without re-normalizing parameters of $\theta_{\cdot|n}$. It suggests pruning the edge with smallest edge flow. A key insight from Theorem 1 is that the log-likelihood drop depends explicitly on the edge flow $\mathrm{F}_{n,c}(\boldsymbol{x})$ and unit flow $\mathrm{F}_n(\boldsymbol{x})$. This matches the intuition from Section 3 that the circuit flow of an edge is sufficient to quantify its importance in the PC.

Next, we bound the drop of pruning multiple edges.

Theorem 2 (Log-likelihood drop of pruning multiple edges)**.** *Let $\mathcal{C}$ be a PC and $\mathcal{D}$ be a dataset. For any set of edges $\mathcal{E}$ in $\mathcal{C}$, if $\forall \boldsymbol{x} \in \mathcal{D}, \sum_{(n,c) \in \mathcal{E}} \mathrm{F}_{n,c}(\boldsymbol{x}) < 1$, the log-likelihood drop by pruning away $\mathcal{E}$ is bounded and approximated by*

$$\Delta\mathcal{LL}(\mathcal{D}, \mathcal{C}, \mathcal{C}_{\setminus \mathcal{E}}) \leq -\frac{1}{|\mathcal{D}|} \sum_{\boldsymbol{x}} \log(1 - \sum_{(n,c) \in \mathcal{E}} \mathrm{F}_{n,c}(\boldsymbol{x})) \approx \frac{1}{|\mathcal{D}|} \sum_{(n,c) \in \mathcal{E}} \mathrm{F}_{n,c}(\mathcal{D}). \tag{3}$$

Proof of this theorem is provided in Appendix B.2. We first look at the second term of Equation 3. Although it provides an upper bound to the performance drop, it cannot be used as a pruning heuristic since the bound does not decompose over edges. And hence finding the set of edges with the lowest score requires evaluating the bound exponentially (w.r.t. k) many times. Therefore, we do an additional approximation step of the bound via Taylor expansion, which leads to the third term of Equation 3. This approximation matches the EFLOW heuristic by a constant factor $1/|\mathcal{D}|$, which theoretically justifies the effectiveness of the heuristic. As shown in Figure 5c, the approximate bound (EFLOW heuristic) matches closely to the actual log-likelihood drop.

5 Scalable Structure Learning

The pruning operator benefits two aspects of PCs. First, as shown in Figure 5b, model parameters are more balanced after pruning. Next, pruning removes sub-circuits with negligible contribution to the model distribution. If we treat PC as hierarchical mixtures of components, pruning can be regarded as an implicit structure learning step that removes the "unimportant" components for each mixture. However, since pruning only decreases model capacity, it is difficult to get a more expressive PC than the original one. To mitigate this problem, we propose a *growing* operation to increase the capacity of a PC by introducing more components for each mixture. Pruning and growing together defines an scalable structure learning algorithm for PCs.

Algorithm 2: Grow($\mathcal{C}$, σ^2)

Input : a PC $\mathcal{C}$, Gaussian noisy variance σ^2
Output : a PC $\mathcal{C}$' after growing operation

```
old2new ← a dictionary mapping input PC units n ∈ C to units of the growed PC
foreach n ∈ C do  // visit children before parents
    if n is an input unit then old2new[n] ← (n, deepcopy(n))
    else
        chs_1, chs_2 ← [old2new[c][0] for c in in(n)], [old2new[c][1] for c in in(n)]
        if n is a product unit then old2new[n] ← (⊗(chs_1), ⊗(chs_2))
        else if n is a sum unit then
            n_1, n_2 ← ⊕([chs_1, chs_2]), ⊕([chs_1, chs_2])
            θ_{|n_i} ← normalize([θ_{|n}, θ_{|n}]) × ε    ε ~ N(1, σ²) for i in [1, 2]
            old2new[n] ← (n_1, n_2)
return old2new[r][0]  // r is the root unit of C
```

Growing Operation. *Growing* defines an operator that increases model size by coping its existing components and injecting noise. As shown in Figure 3, after applying the growing operation on the original PC in Figure 3b, we can get a new grown PC as in Figure 3c. Specifically, the growing operation is applied to units, edges, and parameters respectively: (1) for units, growing operates on every PC unit n and create another copy $n^{\texttt{new}}$; (2) for edges, the sum edge (n, c) from the original PC (Figure 3b) are copied three times to the grown PC (Figure 3c): from new parent to new child $(n^{\texttt{new}}, c^{\texttt{new}})$, from old parent to new child $(n, c^{\texttt{new}})$, and from new parent to old child $(n^{\texttt{new}}, c)$; the product edge is simply connecting new copied sum units; (3) for parameters, new parameter $\theta^{\texttt{new}}_{c|n}$ are a noisy copy from old parameter $\theta_{c|n}$, that is $\theta^{\texttt{new}}_{c|n} \leftarrow \epsilon \cdot \theta_{c|n}$ where $\epsilon \sim \mathcal{N}(1, \sigma^2)$ and σ^2 controls the Gaussian noise variance. Gaussian noise is added to the copied parameters such that after we apply the growing operation, parameter learning algorithms can find diverse parameters for different copies. After a growing operation, the PC size is 4 times the original PC size. Algorithm 2 shows a feed-forward implementation of the growing operation.

Structure Learning through Pruning and Growing. The proposed pruning and growing algorithms can be applied iteratively to refine the structure and parameters of an initial PC. Specifically, since the growing operator increases the number of PC parameters by a factor 4, applying growing after pruning 75% edges from an initial PC keeps the number of parameters unchanged. We make use of these two operations and propose a joint structure and parameter learning algorithm for PCs. Specifically, starting from an initial PC, we apply 75% pruning, growing, and parameter learning iteratively until convergence. We utilize HCLT [18] as initial PC structure as it has the state-of-the-art likelihood performance. Note that this structure learning pipeline can be applied to any PC structure.

Parameter Estimation. We use a stochastic mini-batch version of Expectation-Maximization optimization [1]. Specifically, at each iteration, we draw a mini-batch of samples $\mathcal{D}_B$, compute aggregated circuit flows $\mathrm{F}_{n,c}(\mathcal{D}_B)$ and $\mathrm{F}_n(\mathcal{D}_B)$ of these samples (E-step), and then compute new parameter $\theta^{\texttt{new}}_{c|n} = \mathrm{F}_{n,c}(\mathcal{D}_B)/\mathrm{F}_n(\mathcal{D}_B)$, and update the targeting parameter with a learning rate α: $\theta^{t+1} \leftarrow \alpha\theta^{\texttt{new}} + (1-\alpha)\theta^t$ (M-step). Empirically this approach converges faster and is more regularized compared to full-batch EM.

Parallelism Computation. Existing approaches to scale up learning and inferences of PCs such as Einsum networks [25] utilize fully connected parametrized layers (Figure 3a) of PC structures such as HCLT [18] and RatSPN [26]. These structures can be easily vectorized to utilize deep learning packages such as PyTorch. However, the sparse structure learned by pruning and growing is not easily vectorized as a dense matrix operation. We therefore implement customized GPU kernels to parallelize the computation of parameter learning and inferences based on Juice.jl [6], an open-source Julia package for learning PCs. The kernels segment PC units into layers such that the units in each layer are independent thus the computation can be fully parallelized in the GPU. As a result, we can train PCs with millions of parameters in less than half an hour.

6 Experiments

We now evaluate our proposed method pruning and growing on two different sets of density estimation benchmarks: (1) the MNIST-family image generation datasets including MNIST [16], EMNIST [4], and FashionMNIST [35]; (2) character level Penn Tree Bank language modeling task [21].

We first report the best results we get on image datasets and language modeling tasks via structure learning proposed in Section 5 (Section 6.1) and then show the effect of pruning and growing operations via two detail experimental settings as two different constrained optimization problem: find the smallest PC given the same performance via model compression and find the best PC given the same size via structure learning (Section 6.2).

Settings. For all experiments, we use hidden Chow-Liu Tree (HCLT) [18] with the number of latent states ranging between $\{16, 32, 64, 128\}$ as initial PC structures. We train the parameters of PCs with stochastic mini-batch EM algorithms (cf. Section 5). We perform early stopping and hyperparameter search using validation set and report results on the test set. Please refer to Appendix C for more details. We use mean test set bits-per-dimension (bpd) as evaluation criteria, where $\mathsf{bpd}(\mathcal{D}, \mathcal{C}) = -\mathcal{LL}(\mathcal{D}, \mathcal{C})/(\log(2) \cdot |\mathcal{D}| \cdot m)$ and m is the number of features in dataset $\mathcal{D}$.

6.1 Density Estimation Benchmarks

Image Datasets. The MNIST-family datasets contains gray scale pixel image with size $n = 28 \times 28$ and each pixel taking between $[0, 255](k = 256)$. We split out 5% from training set as validation set. We compare with 2 competitive PC learning algorithms: HCLT [18] and RatSPN [26], one flow-based model: IDF [11], and 3 VAE based methods: BitSwap [14], BB-ANS [31], and McBits [29]. As a fair comparison, we implement RatSPN structure ourselves and used the same training pipeline and EM optimizer as our method. Note that EinsumNet [25] also uses RatSPN structures but with PyTorch implementation so its comparison is subsumed by comparison of RatSPN. All 7 methods are tested on MNIST, 4 splits of EMNIST and FashionMNIST. As shown in Table 1, the best results are bold. We see that our proposed method significantly outperforms all other baselines on all datasets, and establishes new state-of-the-art results among PCs, flows, and VAE models. More experiments details are in Appendix C.

Table 1: Density estimation performance on MNIST-family datasets in test set bpd.

Dataset	Sparse PC (ours)	HCLT	RatSPN	IDF	BitSwap	BB-ANS	McBits
MNIST	**1.14**	1.20	1.67	1.90	1.27	1.39	1.98
EMNIST(MNIST)	**1.52**	1.77	2.56	2.07	1.88	2.04	2.19
EMNIST(Letters)	**1.58**	1.80	2.73	1.95	1.84	2.26	3.12
EMNIST(Balanced)	**1.60**	1.82	2.78	2.15	1.96	2.23	2.88
EMNIST(ByClass)	**1.54**	1.85	2.72	1.98	1.87	2.23	3.14
FashionMNIST	**3.27**	3.34	4.29	3.47	3.28	3.66	3.72

Language Modeling Task. We use the Penn Tree Bank dataset with standard processing from Mikolov et al. [23], which contains around 5M characters and a character-level vocabulary size of $k = 50$. The data is split into sentences with a maximum sequence length of $n = 288$. We compare with 3 competitive normalizing flow based models: Bipartite flow [32] and latent flows [36] including AF/SCF and IAF/SCF, since they are the only comparable work with non-autoregressive language modeling. As shown in Tab. 2, the proposed method outperforms all 3 baselines.

Table 2: Character-level language modeling results on Penn Tree Bank in test set bpd.

Dataset	Sparse PC (ours)	Bipartite flow [32]	AF/SCF [36]	IAF/SCF [36]
Penn Tree Bank	**1.35**	1.38	1.46	1.63

6.2 Evaluating Pruning and Growing

What is the Smallest PC for the Same Likelihood? We evaluation the ability of pruning operations based on circuit flows (Section 3) to do effective model compression by iteratively pruning $k\%$ of the PC parameters and then fine-tuning them until the final training log-likelihood does not decrease by more than 1%. Specifically, we take $k\%$ ranging between $\{0.05, 0.1, 0.3\}$. As shown in Figure 6, we can achieve a compression rate of 80-98% with negligible performance loss on PCs. Besides, by fixing the number of latent parameters (x-axis) and comparing bpp across different number of latent states (legend), we discover that compressing a large PC to a get smaller PC has better likelihoods compared to directly training a HCLT with the same number of parameters from scratch, due to the sparsity of compressed PC structures, as well as the smarter way to find good parameters: finding a better PC with larger size and compress it to smaller one.

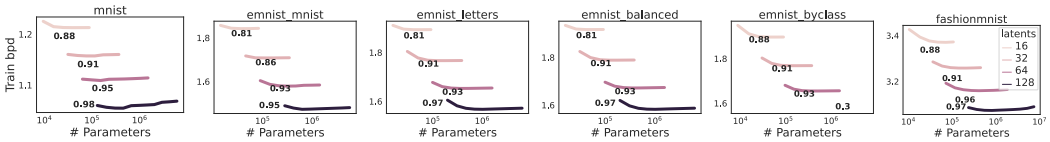


Figure 6: Model compression via pruning and finetuning. We report the training set bpd (y-axis) in terms of the number of parameters (x-axis) for different number of latent states. For each curve, compression starts from the right (initial PC #Params $|\mathcal{C}^{\texttt{init}}|$) and ends at the left (compressed PC #Params $|\mathcal{C}^{\texttt{com}}|$); compression rate (1 - $|\mathcal{C}^{\texttt{com}}|$ / $|\mathcal{C}^{\texttt{init}}|$) is annotated next to each curve.

What is the Best PC Given the Same Size? We evaluate structure learning methods combining pruning and growing proposed in Section 5. Starts from initial HCLT, we iteratively prune 75% of the parameters, growing again, and finetuning until meeting the stopping criteria. As shown in Figure 7, our method consistently improve the likelihoods of initial PCs for different number of latent states among all datasets.

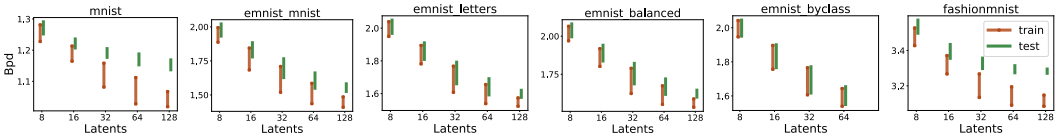


Figure 7: Structure learning via 75% pruning, growing and finetuning. We report bpd (y-axis) on both train (red) and test set (green) in terms of number of latent states (x-axis). For each curve, training starts from the top (large bpd) and ends at the bottom (small bpd).

s

7 Related Work and Conclusions

Improving the expressiveness of PCs have been a central topic in the literature. A recent trend is to construct PCs with good initial structures. For example, EiNets [25] and RAT-SPNs [26] use randomly generated structures, and HCLTs [18] define PCs dependent on the pairwise correlation on variables. Alternatively, several works propose structure learning algorithms that iteratively modify PC structures to better fit the data [5, 17]

We propose structure learning of PCs by combining pruning and growing operations to exploit the sparsity of PC structures. We show significant empirical improvements in the density estimation tasks of PCs compared to existing PC learners and competing flow-based models and VAEs. Moreover, Sparse PC provides parallelism computation of parameter learning and inferences.

References

[1] YooJung Choi, Golnoosh Farnadi, Behrouz Babaki, and Guy Van den Broeck. Learning fair naive bayes classifiers by discovering and eliminating discrimination patterns. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence*, 2020.

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

[3] YooJung Choi, Meihua Dang, and Guy Van den Broeck. Group fairness by probabilistic modeling with latent fair decisions. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence*, 2021.

[4] Gregory Cohen, Saeed Afshar, Jonathan Tapson, and Andre Van Schaik. Emnist: Extending mnist to handwritten letters. In *2017 international joint conference on neural networks (IJCNN)*, pages 2921–2926. IEEE, 2017.

[5] Meihua Dang, Antonio Vergari, and Guy Van den Broeck. Strudel: Learning structured-decomposable probabilistic circuits. In *Proceedings of the 10th International Conference on Probabilistic Graphical Models (PGM)*, 2020.

[6] Meihua Dang, Pasha Khosravi, Yitao Liang, Antonio Vergari, and Guy Van den Broeck. Juice: A julia package for logic and probabilistic circuits. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence (Demo Track)*, 2021.

[7] Adnan Darwiche. A logical approach to factoring belief networks. In *Proceedings of KR*, pages 409–420, 2002.

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

[9] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

[10] Laurent Dinh, David Krueger, and Yoshua Bengio. Nice: Non-linear independent components estimation. *arXiv preprint arXiv:1410.8516*, 2014.

[11] Emiel Hoogeboom, Jorn Peters, Rianne Van Den Berg, and Max Welling. Integer discrete flows and lossless compression. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

[12] Pasha Khosravi, YooJung Choi, Yitao Liang, Antonio Vergari, and Guy Van den Broeck. On tractable computation of expected predictions. In *Advances in Neural Information Processing Systems 32 (NeurIPS)*, 2019.

[13] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[14] Friso Kingma, Pieter Abbeel, and Jonathan Ho. Bit-swap: Recursive bits-back coding for lossless compression with hierarchical latent variables. In *International Conference on Machine Learning*, pages 3408–3417. PMLR, 2019.

[15] Doga Kisa, Guy Van den Broeck, Arthur Choi, and Adnan Darwiche. Probabilistic sentential decision diagrams. In *Proceedings of the 14th International Conference on Principles of Knowledge Representation and Reasoning (KR)*, 2014.

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

[17] Yitao Liang, Jessa Bekker, and Guy Van den Broeck. Learning the structure of probabilistic sentential decision diagrams. In *Proceedings of the 33rd Conference on Uncertainty in Artificial Intelligence (UAI)*, 2017.

[18] Anji Liu and Guy Van den Broeck. Tractable regularization of probabilistic circuits. In *Advances in Neural Information Processing Systems 35 (NeurIPS)*, 2021.

[19] Anji Liu, Stephan Mandt, and Guy Van den Broeck. Lossless compression with probabilistic circuits. In *International Conference on Learning Representations (ICLR)*, 2022.

[20] Anji Liu, Stephan Mandt, and Guy Van den Broeck. Lossless compression with probabilistic circuits. In *International Conference on Learning Representations (ICLR)*, 2022.

[21] Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of english: The penn treebank. *Computational Linguistics*, 19(2):313–330, 1993. ISSN 0891-2017.

[22] Radu Marinescu and Rina Dechter. And/or branch-and-bound for graphical models. In *IJCAI*, pages 224–229, 2005.

[23] Tomáš Mikolov, Ilya Sutskever, Anoop Deoras, Hai-Son Le, and Stefan Kombrink. Subword language modeling with neural networks. 2012.

[24] George Papamakarios, Eric Nalisnick, Danilo Jimenez Rezende, Shakir Mohamed, and Balaji Lakshminarayanan. Normalizing flows for probabilistic modeling and inference. *Journal of Machine Learning Research*, 22(57):1–64, 2021.

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

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

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

[28] Tahrima Rahman, Prasanna Kothalkar, and Vibhav Gogate. 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*, pages 630–645. Springer, 2014.

[29] 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 *International Conference on Machine Learning*, pages 9136–9147. PMLR, 2021.

[30] Andy Shih, Dorsa Sadigh, and Stefano Ermon. Hyperspns: Compact and expressive probabilistic circuits. *Advances in Neural Information Processing Systems*, 34, 2021.

[31] James Townsend, Thomas Bird, and David Barber. Practical lossless compression with latent variables using bits back coding. In *International Conference on Learning Representations*, 2018.

[32] Dustin Tran, Keyon Vafa, Kumar Agrawal, Laurent Dinh, and Ben Poole. Discrete flows: Invertible generative models of discrete data. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

[33] Antonio Vergari, YooJung Choi, Robert Peharz, and Guy Van den Broeck. Probabilistic circuits: Representations, inference, learning and applications. *AAAI Tutorial*, 2020.

[34] Antonio Vergari, YooJung Choi, Anji Liu, Stefano Teso, and Guy Van den Broeck. A compositional atlas of tractable circuit operations for probabilistic inference. In *Advances in Neural Information Processing Systems*, volume 34, 2021.

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

[36] Zachary Ziegler and Alexander Rush. Latent normalizing flows for discrete sequences. In *International Conference on Machine Learning*, pages 7673–7682. PMLR, 2019.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes] Contributions are clearly stated in the third and fourth paragraphs of the introduction.
   (b) Did you describe the limitations of your work? [Yes] The proposed learning algorithms are designed for smooth and decomposable PCs.
   (c) Did you discuss any potential negative societal impacts of your work? [N/A]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes] All theorems in the main text formally state all assumptions.
   (b) Did you include complete proofs of all theoretical results? [Yes] Proofs are included in the appendix.
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] Experiment setup is detailed in the main text and the appendix.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] Training details are provided in the appendix.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [N/A]
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] Details about computational resources can be found in the appendix.
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes] We cited all PC learning algorithms as well as the adopted datasets.
   (b) Did you mention the license of the assets? [Yes] We specified that the used algorithms are publicly available.
   (c) Did you include any new assets either in the supplemental material or as a URL? [Yes] We included our code in the supplementary material.
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]