Title

Towards Controllable Generative AI with Intrinsic Reasoning Capabilities

Author

Liu, Anji

Publication Date

2025

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA

Los Angeles

Towards Controllable Generative AI with Intrinsic Reasoning Capabilities

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Anji Liu

2025

ABSTRACT OF THE DISSERTATION

Towards Controllable Generative AI with Intrinsic Reasoning Capabilities

by

Anji Liu

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2025

Professor Guy Van den Broeck, Chair

Generative AI has become a transformative paradigm that enables machines to produce high-quality content such as images, language, and audio. However, beyond creating charming and coherent outputs, these systems must reason — steering their generations to satisfy specific properties. For instance, in science and engineering, this capability could ensure that synthesized molecular structures obey physical constraints or that design blueprints meet safety standards. While sound reasoning techniques from classical symbolic AI can rigorously guarantee these properties, they are often computationally prohibitive and difficult to scale. As a result, many recent approaches rely on scalable yet unsound methods, such as chain-of-thought prompting, which prioritize efficiency over rigorous correctness. In this dissertation, I will discuss how to design tractable generative AI models as drop-in replacements of existing models like autoregressive Transformers and diffusion models, with the distinguishing capability of sound reasoning. I will demonstrate how such tractable generative models enable high-fidelity yet controllable generations in various domains, and highlight the importance of building generative models with intrinsic reasoning capabilities.

The dissertation of Anji Liu is approved.

Stephan Mandt

Aditya Grover

Sriram Sankararaman

Guy Van den Broeck, Committee Chair

University of California, Los Angeles

2025

To my family.

Contents

List of Figures

List of Tables

Acknowledgements

Completing a Ph.D. still feels somewhat unreal to me. While there were certainly difficult moments along the way, when I look back on this journey, what I remember most is the joy, even during times when my research wasn't progressing as I had hoped. That sense of joy and excitement came not from the absence of struggle, but from being surrounded by a deeply supportive environment. I was incredibly fortunate to have people around me who kindly offered their guidance, encouragement, and patience. I truly could not have made it this far without them.

First and foremost, I would like to express my deepest gratitude to my advisor, Professor Guy Van den Broeck. His unwavering support and exceptional mentorship have played a central role in my growth as a researcher. Guy put an incredible amount of thought and effort into helping me learn how to think independently and eventually craft my own research agenda. He struck a rare and thoughtful balance between offering detailed guidance and allowing me the space to develop and pursue my own ideas.

I was very fortunate to have had the opportunity to visit Professor Mathias Niepert's lab at the University of Stuttgart and Professor Yitao Liang's lab at Peking University. My time in both labs was incredibly enriching, and I am deeply grateful for the warm welcome and inspiring environment I found at each place. I had the chance to meet and work alongside many amazing people.

I would also like to express my heartfelt thanks to my other committee members–Professor Stephan Mandt, Professor Aditya Grover, and Professor Sriram Sankararaman–for their valuable feedback on my research and for their generous support throughout my job application process.

I feel extremely lucky to have had such amazing labmates, collaborators, and friends throughout this journey: Steven, Tal, Yitao, YooJung, Pasha, Kareem, Zhe, Honghua, Meihua, Antonio, Poorva, Renato, Oliver, Gwen, Zoe, Benjie, Ian, Ruoyan, Wenzhe, Daniel, Vinh, Edgar, Andrei, Mario, Duy, Samir, Tanja, Julia, Arman, Yun, Aneesh, Jan, Xuejie, Zihao, Shaofei, Haowei, Xiaojian, Xue, Lifeng, Silong, Dayuan, Zhizhou, Jianshu, Ji, Zilong, Yuanjun, Hongming and so on. I hope you all the best.

Above all, I want to thank my wife, whose love, patience, and unwavering belief in me have been my greatest source of strength. I am endlessly grateful for her presence in my life and for traveling across Europe with me as my favorite companion.

I am deeply grateful to my parents and grandparents for their unconditional and lifelong support. Even from afar, they were always there for me, offering words of comfort and pride that meant more than they will ever know. This accomplishment is as much theirs as it is mine.

Vita

Education

Ph.D. (Computer Science) University of California, Los Angeles 2020 - 2025

B.Eng. (Automation Science and Electrical Engineering) Beihang University 2015 - 2019

Publications

[1] **Anji Liu**, Oliver Broadrick, Mathias Niepert, Guy Van den Broeck. Discrete Copula Diffusion. In: *The Thirteenth International Conference on Learning Representations (ICLR)*, 2025.

[2] Xuejie Liu*, **Anji Liu***, Guy Van den Broeck, Yitao Liang. A Tractable Inference Perspective of Offline RL. In: *Proceedings of the Thirty-Eighth Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2024.

[3] **Anji Liu**, Kareem Ahmed, Guy Van den Broeck. Scaling Tractable Probabilistic Circuits: A Systems Perspective. In: *The Forty-First International Conference on Machine Learning (ICML)*, 2024.

[4] **Anji Liu**, Mathias Niepert, Guy Van den Broeck. Image Inpainting via Tractable Steering of Diffusion Models. In: *The Twelfth International Conference on Learning Representations (ICLR)*, 2024.

[5] Xuejie Liu*, **Anji Liu***, Guy Van den Broeck, Yitao Liang. Understanding the Distillation Process from Deep Generative Models to Tractable Probabilistic Circuits. In: *The Fortieth International Conference on Machine Learning (ICML)*, 2023.

[6] **Anji Liu***, Honghua Zhang*, Guy Van den Broeck. Scaling Up Probabilistic Circuits by Latent Variable Distillation. In: *The Eleventh International Conference on Learning Representations (ICLR)*, 2023.

[7] Meihua Dang, **Anji Liu**, Guy Van den Broeck. Sparse Probabilistic Circuits via Pruning and Growing. In: *Proceedings of the Thirty-Sixth Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2022.

[8] **Anji Liu**, Stephan Mandt, Guy Van den Broeck. Lossless Compression with Probabilistic Circuits. In: *The Tenth International Conference on Learning Representations (ICLR)*, 2022.

[9] Antonio Vergari, YooJung Choi, **Anji Liu**, Stefano Teso, Guy Van den Broeck. A Compositional Atlas of Tractable Circuit Operations for Probabilistic Inference. In: *Proceedings of the Thirty-Fifth Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2021.

Chapter 1

Introduction

Generative AI has emerged as a powerful paradigm, allowing machines to generate high-quality content across various modalities, including images, language, and audio. However, beyond creating charming and coherent outputs, these systems must reason — steering their generations to satisfy specific properties. For instance, in science and engineering, this capability could ensure that synthesized molecular structures obey physical constraints or that design blueprints meet safety standards. Similarly, in everyday applications, such as personal assistants, it could guarantee compliance with ethical guidelines or adherence to user preferences. While sound reasoning techniques from classical symbolic AI can rigorously guarantee these properties, they are often computationally prohibitive and difficult to scale. As a result, many recent approaches rely on scalable yet unsound methods, such as chain-of-thought prompting, which prioritize efficiency over rigorous correctness.

This dissertation discusses the possibility of designing generative AI models as drop-in replacements of existing models like autoregressive Transformers and diffusion models, with the distinguishing capability of sound reasoning. This enables high-fidelity yet controllable generations that align with user requests or domain constraints. While high fidelity is ensured by accurately modeling data distributions, controllability and reasoning capability are reflected in how we extract information from the model's learned distributions, often

through probabilistic queries such as marginal probabilities and most probable explanations. Two fundamental questions are: (i) What types of probabilistic reasoning can be supported by a class of generative models? (ii) How can these reasoning capabilities be leveraged to solve downstream tasks? In addressing these questions, this dissertation focuses on two main topics.

First, we study how to advance tractable generative models that support exact and efficient probabilistic reasoning. Probabilistic Circuits (PCs) [14] are a class of generative models designed for efficient computation of probabilistic queries such as marginal probabilities. Chapter 2 introduces backgrounds about PCs and further proposes a general pipeline to derive algorithms to efficiently and exactly compute a wide range of probabilistic queries, including many information-theoretic queries. Despite their capability to perform exact reasoning, their practical adoption has been hindered by limitations in density estimation performance. To address this, Chapters 3 and 4 propose algorithmic and systems-based training techniques, respectively. These approaches collectively enhanced the expressiveness of PCs — from underfitting on hand-written digits to achieving competitive results with variational autoencoders and diffusion models on natural image datasets [100].

Do generative models with enhanced reasoning capabilities perform better on downstream tasks? Leveraging the aforementioned modeling-side improvements, Chapter 5 discusses the benefits of applying PCs to reasoning-demanding tasks such as lossless data compression, controlled image generation, and offline reinforcement learning. Specifically, in many practical applications involving deep generative models such as large language models and diffusion models, performance is strongly hindered by the need to approximate the reasoning procedure required by the task. To mitigate such problems, I show that tractable generative models like PCs can control the denoising process of diffusion models to achieve unbiased conditional generation [96]. Similarly, autoregressive Transformers (GPTs) can be controlled by tractable models such as PCs to effectively generate actions conditioned on high expected returns, achieving state-of-the-art performance on various robotics tasks [101].

In the last part of this thesis, I discuss the possibility of improving the reasoning capabilities of modern deep generative models. As an example, non-autoregressive generative models, such as diffusion models, offer significant advantages over widely adopted autoregressive models by enabling the ability to answer arbitrary conditional queries. However, non-autoregressive models fall short in modeling discrete data, such as language, where autoregressive models like GPTs currently excel. In Chapter 6, I propose a novel hybrid framework that combines autoregressive and non-autoregressive models at inference time, leveraging the strengths of both approaches. This hybrid model not only surpasses autoregressive models in density estimation but also retains the enhanced reasoning flexibility of non-autoregressive models. Additionally, preliminary findings from my ongoing research suggest that a novel non-autoregressive model architecture can achieve both high expressiveness and strong reasoning capabilities.

Chapter 2

Tractable Inference with Probabilistic Circuits

In this chapter, we first review a tractable deep representation of probability distributions termed Probabilistic Circuits (PCs). Based on this representation, we develop a systematic pipeline to perform *exact* and *efficient* probabilistic inference. Specifically, we show how complex inference scenarios for these models that commonly arise in machine learning – from computing the expectations of decision tree ensembles to information-theoretic divergences of PCs – can be represented in terms of tractable modular operations over circuits. Specifically, we characterize the tractability of simple transformations – sums, products, quotients, powers, logarithms, and exponentials – in terms of sufficient structural constraints of the circuits they operate on, and present novel hardness results for the cases in which these properties are not satisfied. Building on these operations, we derive a unified framework for reasoning about tractable models that generalizes several results in the literature and opens up novel tractable inference scenarios.

The contents of this chapter appeared in paper [177].

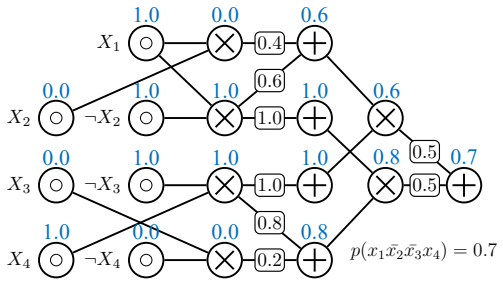


Figure 2.1: An example PC over boolean variables $X_1, \ldots, X_4$. Sum parameters are labeled on the edges. The probability of every node given input $x_1\bar{x}_2\bar{x}_3x_4$ is labeled blue on top of the corresponding node.

2.1 Background on Probabilistic Circuits

This section provides background on Probabilistic Circuits (PCs) [21], which is a tractable representation of probabilistic distributions. In particular, we will go through the syntax and semantics of PCs to demonstrate how they build complex distributions from atomic building blocks.

Probabilistic Circuits is an umbrella term for a wide variety of *tractable probabilistic models* (TPMs), including the classical Hidden Markov Model (HMM) [139] and Chow-Liu Trees [22] as well as more recent ones including Sum-Product Networks [136], Arithmetic Circuits [156], and Cutset Networks [141].

A PC $p(\mathbf{X})$ represents a distribution over $\mathbf{X}$ via a parameterized Directed Acyclic Graph (DAG) with a single root node n_{r}. There are three types of nodes: *input*, *product*, and *sum* nodes. Input nodes define primitive distributions over some variable $X \in \mathbf{X}$, while sum and product nodes merge the distributions defined by their children, denoted $\mathsf{ch}(n)$, to build more complex distributions as follows:

$$p_n(\boldsymbol{x}) := \begin{cases} f_n(\boldsymbol{x}) & n \text{ is an input node,} \\ \prod_{c \in \mathsf{ch}(n)} p_c(\boldsymbol{x}) & n \text{ is a product node,} \\ \sum_{c \in \mathsf{ch}(n)} \theta_{n,c} \cdot p_c(\boldsymbol{x}) & n \text{ is a sum node,} \end{cases} \tag{2.1}$$

where $f_n(\boldsymbol{x})$ is an univariate input distribution (e.g., Gaussian, Categorical), and $\theta_{n,c}$ denotes the parameter corresponding to edge (n, c). Intuitively, sum nodes and product nodes encode mixture and factorized distributions of their children, respectively. To ensure that a PC models a valid distribution, we assume the child parameters of every sum node n (i.e., $\{\theta_{n,c}\}_{c\in\mathsf{ch}(n)}$) sum up to 1. The size of a PC p, denoted $|p|$, is the number of edges in its DAG. The scope $\phi(n)$ of a node n is the set of variables defined by its descendent input nodes. The support $\mathsf{supp}(n)$ of a node n is the set of inputs $\boldsymbol{x} \in \mathsf{val}(\mathbf{X})$ for which the probability $p_n(\boldsymbol{x})$ is strictly greater than 0.

Figure 2.1 shows an example PC where $\odot$, $\otimes$, and $\oplus$ represent input, product, and sum nodes, respectively. The key to guaranteeing the tractability of PCs is to add proper structural constraints to their DAG structure.

2.2 Structural Properties of (Probabilistic) Circuits

In the following, we slightly generalize the notion of PCs to Circuits if the encoded function is not a valid distribution. Structural constraints on the computational graph of a circuit in terms of its scope or support provide sufficient and/or necessary conditions for certain queries to be tractably computed. We now define the structural properties needed for the query classes that this work will focus on, referring to [14] for more details.

Definition 1 (Smoothness)**.** A circuit is *smooth* if for every sum unit n, its inputs depend on the same variables: $\forall\, c_1, c_2 \in \mathsf{ch}(n), \phi(c_1) = \phi(c_2)$.

Smooth PCs generalize shallow mixture models [112] to deep and hierarchical models. For instance, a Gaussian mixture model (GMM) can be represented as a smooth PC with a single sum unit over as many input units as mixture components, each encoding a (multivariate) Gaussian density.

Definition 2 (Decomposability)**.** A circuit is *decomposable* if the inputs of every product unit n depend on disjoint sets of variables: $\mathsf{ch}(n) = \{c_1, c_2\}, \phi(c_1) \cap \phi(c_2) = \varnothing$.

Decomposable product units encode local factorizations. That is, a decomposable product unit n over variables $\mathbf{X}$ encodes $p_n(\mathbf{X}) = p_1(\mathbf{X}_1) \cdot p_2(\mathbf{X}_2)$ where $\mathbf{X}_1$ and $\mathbf{X}_2$ form a partition of $\mathbf{X}$. Taken together, decomposability and smoothness are a sufficient and necessary condition for performing tractable integration over arbitrary sets of variables in a single feedforward pass, as they enable larger integrals to be efficiently decomposed into smaller ones [14, 37]. The next Proposition formalizes it.

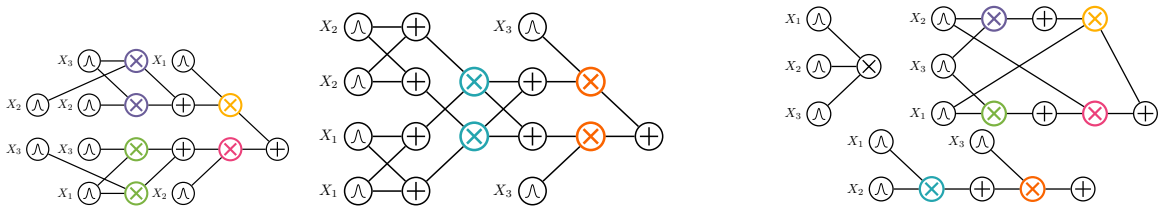
Proposition 1 (Tractable integration, [14])**.** *Let p be a smooth and decomposable circuit over $\mathbf{X}$ with input functions that can be tractably integrated. Then for any variables $\mathbf{Y} \subseteq \mathbf{X}$ and their assignment $\boldsymbol{y}$, the integral $\int_{\boldsymbol{z} \in \mathsf{val}(\mathbf{Z})} p(\boldsymbol{y}, \boldsymbol{z}) d\mathbf{Z}$ can be computed exactly in $\Theta(|p|)$ time, where $\mathbf{Z}$ denotes $\mathbf{X} \setminus \mathbf{Y}$.*

As the complex queries we focus on in this work involve integration as the last step, it is therefore needed that any intermediate operation preserves at least decomposability; smoothness is less of an issue, as it can be enforced in polytime [160]. A key additional constraint over scope decompositions is *compatibility*. Intuitively, two decomposable circuits are compatible if they can be rearranged in polynomial time[1] such that their respective product units, once matched by scope, decompose in the same way. We formalize this with the following inductive definition.

Definition 3 (Compatibility)**.** Two circuits p and q over variables $\mathbf{X}$ are *compatible* if (i) they are smooth and decomposable and (ii) any pair of product units $n \in p$ and $m \in q$ with the same scope can be rearranged into binary products that are mutually compatible and decompose in the same way: $(\phi(n) = \phi(m)) \implies (\phi(n_i) = \phi(m_i),\ n_i$ and m_i are compatible) for some rearrangement of the inputs of n (resp. m) into n_1, n_2 (resp. m_1, m_2).

We can derive from compatibility the following properties pertaining to a single circuit, which will be useful in our analysis later.

[1] By changing the order in which n-ary product units are turned into a series of binary product units.



(a) Decomposable (b) Structured-decomposable (c) Omni-compatible

Figure 2.2: Examples of circuits with different structural properties. The feedforward order is from left to right; input units are labeled by their scopes; and sum parameters are omitted for visual clarity. Product units of the rearranged omni-compatible circuits encoding $p(X_1) \cdot p(X_2) \cdot p(X_3)$ are shown in (c) and color-coded with those of matching scope in (a) and (b).

Definition 4 (Special types of compatibility)**.** A circuit is *structured-decomposable* if it is compatible with itself. A decomposable circuit p over $\mathbf{X}$ is *omni-compatible* if it is compatible with any smooth and decomposable circuit over $\mathbf{X}$.

Not all decomposable circuits are structured-decomposable (see Figs. 2.2a and 2.2b), but some can be rearranged to be compatible with any decomposable circuit. For instance, in Fig. 2.2c, the fully factorized product unit $p(\mathbf{X}) = p_1(X_1) \cdot p_2(X_2) \cdot p_3(X_3)$ can be rearranged into $p_1(X_1) \cdot (p_2(X_2) \cdot p_3(X_3))$ and $p_2(X_2) \cdot (p_1(X_1) \cdot p_3(X_3))$ to match the yellow and pink products in Fig. 2.2a. We can easily see that omni-compatible circuits must assume the form of mixtures of fully-factorized models; i.e., $\sum_i \theta_i \prod_j p_{i,j}(X_j)$. For example, an additive ensemble of decision trees over variables $\mathbf{X}$ can be represented as an omni-compatible circuit. Also note that if two circuits are compatible and neither is omni-compatible, then both must be structured decomposable.

Definition 5 (Determinism)**.** A circuit is *deterministic* if the inputs of every sum unit n have disjoint supports: $\forall\, c_1, c_2 \in \mathsf{ch}(n), c_1 \neq c_2 \implies \mathsf{supp}(c_1) \cap \mathsf{supp}(c_2) = \varnothing$.

Analogously to decomposability, determinism induces a recursive partitioning, but this time over the support of a circuit. For a deterministic sum unit n, the partitioning of its support can be made explicit by introducing an indicator function per each of its inputs, i.e., $\sum_{c \in \mathsf{ch}(n)} \theta_c p_c(\boldsymbol{x}) = \sum_{c \in \mathsf{ch}(n)} \theta_c p_c(\boldsymbol{x}) [\![\boldsymbol{x} \in \mathsf{supp}(p_c)]\!]$. Determinism allows for tractable

maximization of a circuit [14,35]. While we do not consider maximization queries in this work, determinism will still play a crucial role in the next sections. Moreover, bounded-treewidth PGMs, such as Chow-Liu trees [22] and thin junction trees [4], can efficiently be represented as a smooth, deterministic, and decomposable PC via *compilation* [32, 35]. Probabilistic sentential decision diagrams (PSDDs) [82] are deterministic and structured-decomposable PCs that can be efficiently learned from data [32].

2.3 A Compositional Atlas of Tractable Circuit Operations

Many core computational tasks in machine learning and AI involve solving complex integrals, such as expectations, that often turn out to be intractable. A fundamental question then arises: *under which conditions do these quantities admit tractable computation?* That is, when can we compute them efficiently without resorting to approximations or heuristics? If we are able to find model classes to tractably compute these quantities of interest—henceforth called *queries*—we can then design efficient algorithms with important applications in learning, approximate inference [158], model compression [91], explainable AI [174,179] and algorithmic bias detection [15, 20].

This "quest" for tracing the tractability of different queries has been carried out several times, often independently for different model classes in ML and AI and crucially, for each query in isolation. For example, the computation of the Kullback-Leibler divergence (KLD) is known to have a closed form for Gaussians, but only recently has an exact algorithm been derived for a more complex tractable model class such as probabilistic sentential decision diagrams (PSDDs) [91]. On the other hand, tractable computation of the entropy, despite being a sub-routine for the KLD, has only been derived for a different tractable model class—selective sum-product networks (SPNs) [130]—by [158]. In the current paradigm, if one were to trace the tractability of a query that has not yet been investigated but still involves the

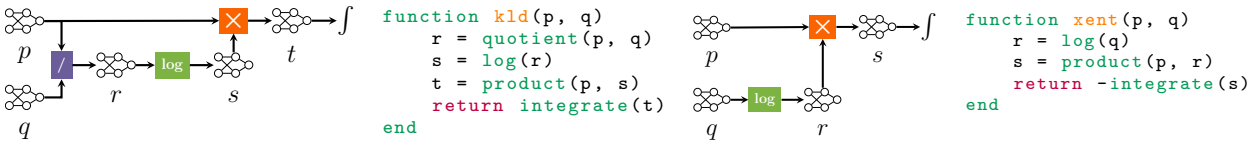


Figure 2.3: Computational pipelines of the KLD (left) and cross entropy (right) over two distributions p and q encoded as circuits, with the intermediate computations (r, s and t) also represented as circuits. Their corresponding implementations in a few lines of Julia code are shown on their right.

same "building blocks" such as logarithms, integrals and products over distributions, for instance Rényi's alpha divergence [144], they would need to derive a novel custom algorithm for each model class and prove its tractability from scratch.

We take a different path and introduce a general framework under which the tractability of complex queries can be traced in a *unified and effortless manner over model classes and query classes*. To abstract from the different model formalisms, we carry our analysis over circuit representations [14] as they subsume many tractable generative models—probabilistic circuits such as Chow-Liu trees [22], hidden Markov models (HMMs) [139], sum-product networks (SPNs) [136], and other deep mixture models—as well as discriminative ones, including decision trees [25, 77] and deep regressors [75], thus enabling a unified treatment across model classes. To generalize our analysis across queries, we propose to represent a single query as a *circuit pipeline*: a computational graph whose intermediate operations transform and combine the input circuits into other circuits. We can first build a set of simple tractable circuit transformations—sums, products, powers, logarithms, and exponentials—and then (i) analyze the tractability of a single query by propagating the sufficient conditions for tractability of the intermediate operators in the pipeline; and (ii) automatically distill a tractable inference algorithm by composing the operators used. For instance, Fig. 2.3 shows the pipeline for computing the KLD of p and q, two distributions encoded by circuits. We can identify a general class of models that supports its tractable computation: by tracing the conditions for tractable quotient, logarithm, and product over circuits such that the output circuit (i.e., t) admits tractable integration, we can derive a set of sufficient conditions for the input circuits.

Moreover, we can *reuse* the logarithm and product operations in the KLD pipeline to reason about the tractability of cross entropy, in the very same way we can reuse the corresponding subroutines we provide in Julia to quickly implement algorithms for the two queries in a couple lines of code as shown in Fig. 2.3. This compositionality greatly speeds up the design of novel tractable algorithms.

2.4 From Simple Circuit Transformations...

This section aims to build an atlas of simple operations over circuits which can then be composed into more complex queries via circuit pipelines—computational graphs whose units are tractable operators over circuits. To compose two operators, we would need that the output circuits of one satisfy the structural properties required for the inputs of the other. As such, for each of these operations we are interested in characterizing (i) its tractability in terms of the structural properties of its input circuits, and (ii) its closure w.r.t. these properties, i.e. whether they are preserved in the output circuit, in order to compose many operations together in a pipeline, while (iii) providing an efficient algorithmic implementation for it. As we are interested in pipelines for queries involving integration, we would expect the output circuits to at least retain decomposability (see Proposition 1). For a pipeline in which all operators can be computed tractably, a simple tractable algorithm can be then distilled for it. Furthermore, our analysis will highlight if one needs to resort to approximations, by tracing the hardness of representing the output of an operator as a decomposable circuit when some property of its inputs is unmet. We summarize all our main results in Table 2.1 and prove the corresponding statements in Chapter A.

Sum of Circuits. The operation of summing two circuits $p(\mathbf{Z})$ and $q(\mathbf{Y})$ is defined as $s(\mathbf{X}) = \theta_1 \cdot p(\mathbf{Z}) + \theta_2 \cdot q(\mathbf{Y})$ for $\mathbf{X} = \mathbf{Z} \cup \mathbf{Y}$ and two real parameters $\theta_1, \theta_2 \in \mathbb{R}$. This operation, which is at the core of additive ensembles of tractable representations,[2] can be realized by

[2]If p and q are PCs, then s is a PC encoding a monotonic mixture model if $\theta_1, \theta_2 > 0$ and $\theta_1 + \theta_2 = 1$.

Table 2.1: ***Tractability and hardness of simple circuit operations***. Tractable conditions on inputs translate to conditions on outputs. E.g., for the quotient p/q, if p and q are compatible (Cmp) and q is deterministic (Det), then the output is decomposable (Dec); also (+) deterministic if p is deterministic; and structured-decomposable (SD) if both p and q are. Hardness results are for representing the output as a smooth (Sm) and decomposable circuit without some input condition.

Operation		Tractability			Hardness
		Input conditions	Output conditions	Time Complexity	
Sum	$\theta_1 p + \theta_2 q$	(+Cmp)	(+SD)	$\mathcal{O}(\lvert p\rvert + \lvert q\rvert)$	NP-hard for Det out
Product	$p \cdot q$	Cmp (+Det, +SD)	Dec (+Det, +SD)	$\mathcal{O}(\lvert p\rvert\lvert q\rvert)$	#P-hard w/o Cmp
Power	$p^n, n \in \mathbb{N}$	SD (+Det)	SD (+Det)	$\mathcal{O}(\lvert p\rvert^n)$	#P-hard w/o SD
	$p^\alpha, \alpha \in \mathbb{R}$	Sm, Dec, Det (+SD)	Sm, Dec, Det (+SD)	$\mathcal{O}(\lvert p\rvert)$	#P-hard w/o Det
Quotient	p/q	Cmp; q Det (+p Det,+SD)	Dec (+Det,+SD)	$\mathcal{O}(\lvert p\rvert\lvert q\rvert)$	#P-hard w/o Det
Log	$\log(p)$	Sm, Dec, Det	Sm, Dec	$\mathcal{O}(\lvert p\rvert)$	#P-hard w/o Det
Exp	$\exp(p)$	linear	SD	$\mathcal{O}(\lvert p\rvert)$	#P-hard

introducing a single sum unit that takes as input p and q. Summation applies to any input circuits, regardless of structural assumptions, and it preserves several properties. In particular, if p and q are decomposable then s is also decomposable; moreover, if they are compatible then s is structured-decomposable as well as compatible with p and q. However, representing a sum as a deterministic circuit is known to be NP-hard [156], even for compatible and deterministic inputs.

Product of Circuits. The product of two circuits $p(\mathbf{Z})$ and $q(\mathbf{Y})$ can be expressed as $m(\mathbf{X}) = p(\mathbf{Z}) \cdot q(\mathbf{Y})$ for variables $\mathbf{X} = \mathbf{Z} \cup \mathbf{Y}$. If $\mathbf{Z}$ and $\mathbf{Y}$ are disjoint, the product m is already decomposable. Otherwise, [156] proved that representing the product of two decomposable circuits as a decomposable circuit is NP-hard, even if they are deterministic. We prove in Theorem 16 that it is #P-hard even for structured-decomposable and deterministic circuits.

Theorem 1 (Hardness of product)**.** *If p and q are two structured-decomposable and deterministic circuits, then computing their product as a decomposable circuit is #P-hard.*

[156] also introduced an efficient algorithm for the product of two structured-decomposable and deterministic PCs that are compatible (namely PSDDs). We generalize this result by proving that compatibility alone is sufficient for the tractable product computation of any two circuits.

Theorem 2 (Tractable product)**.** *If p and q are two compatible circuits, then computing their product as a decomposable circuit that is compatible with them can be done in $\mathcal{O}(|p| \cdot |q|)$ time.*

In the following, we provide a sketch of the algorithm for the case $\mathbf{X} = \mathbf{Z} = \mathbf{Y}$ and refer the readers to the detailed Algorithm 11. Intuitively, the idea is to "break down" the construction of the product circuit in a recursive manner by exploiting compatibility. The base case is where p and q are input units with simple parametric forms. Their product can be represented as a single input unit as long as we can find a simple parametric form for it, as is the case for products of exponential families such as (multivariate) Gaussians. Next, we consider the inductive steps where p and q are two sum or product units. If p and q are compatible product units, they decompose $\mathbf{X}$ the same way for some ordering of inputs; i.e., $p(\mathbf{X}) = p_1(\mathbf{X}_1)p_2(\mathbf{X}_2)$ and $q(\mathbf{X}) = q_1(\mathbf{X}_1)q_2(\mathbf{X}_2)$. Then, their product m as a decomposable circuit can be constructed recursively from the products of their inputs: $m(\mathbf{X}) = (p_1q_1)(\mathbf{X}_1) \cdot (p_2q_2)(\mathbf{X}_2)$. On the other hand, if p and q are smooth sum units, written as $p(\mathbf{X}) = \sum_i \theta_i p_i(\mathbf{X})$ and $q(\mathbf{X}) = \sum_j \theta'_j q_j(\mathbf{X})$, we can obtain their product m recursively by distributing product over sum. In other words, $m(\mathbf{X}) = \sum_{i,j} \theta_i \theta'_j (p_i q_j)(\mathbf{X})$. Note that if both input circuits are also deterministic, m is also deterministic since $\mathsf{supp}(p_i q_j) = \mathsf{supp}(p_i) \cap \mathsf{supp}(q_j)$ are disjoint for different i, j. Combining these, the algorithm will recursively compute the product of each pair of units in p and q with matching scopes. Assuming efficient products for input units, the overall complexity is $\mathcal{O}(|p||q|)$, which yields a compact circuit m of size $\mathcal{O}(|p||q|)$. This upper bound is loose and in practice product circuits will be much smaller as our experiments show (Section 2.6), especially if inputs are deterministic as products of units with disjoint supports will be "pruned" away.

Powers of a Circuit. The α-power of a PC $p(\mathbf{X})$ for an $\alpha \in \mathbb{R}$ is denoted as $p^\alpha(\mathbf{X})$ and is an operation needed to compute generalizations of the entropy of a PC and related divergences (Section 2.5). Let us first consider natural powers ($\alpha \in \mathbb{N}$) which can be computed even for

general circuits.

Theorem 3 (Natural powers)**.** *If p is a structured-decomposable circuit, then for any $\alpha \in \mathbb{N}$, its power can be represented as a structured-decomposable circuit in $\mathcal{O}(|p|^\alpha)$ time. Otherwise, if p is only smooth and decomposable, then computing $p^\alpha(\mathbf{X})$ as a decomposable circuit is #P-hard.*

The proof for tractability easily follows by directly applying the product operation repeatedly. However, we prove in Theorem 19 that the exponential dependence on α is unavoidable unless P=NP, rendering the operation intractable for large α.

Turning our attention to non-natural $\alpha \in \mathbb{R}$, and restricting our attention to PCs, structured-decomposability is not sufficient to tractably compute α-powers, which we will show in the next theorem for $\alpha = -1$. First, as zero raised to a negative power is undefined, we instead consider the *restricted α-power* of a PC, denoted as $p^\alpha(\boldsymbol{x})|_{\mathsf{supp}(p)}$ and equal to $(p(\boldsymbol{x}))^\alpha$ if $\boldsymbol{x} \in \mathsf{supp}(p)$ and 0 otherwise. Note that this is equivalent to the α-power if $\alpha \geq 0$. Abusing notation, we will also denote this by $p^\alpha(\boldsymbol{x})[\![\boldsymbol{x} \in \mathsf{supp}(p)]\!]$, where $[\![\cdot]\!]$ stands for indicator functions.

Theorem 4 (Hardness of reciprocals)**.** *If p is a structured-decomposable circuit over variables $\mathbf{X}$, then computing $p^{-1}(\mathbf{X})|_{\mathsf{supp}(p)}$ as a decomposable circuit is #P-hard.*

The key property that enables efficient computation of power circuits is determinism. More interestingly, we do not require structured-decomposability, but only smoothness and decomposability (see Theorem 21).

Theorem 5 (Tractable real powers)**.** *If p is a smooth, decomposable, and deterministic PC, then for any $\alpha \in \mathbb{R}$, its restricted power can be represented as a smooth, decomposable, and deterministic circuit that is compatible with p in $\mathcal{O}(|p|)$ time.*

Again, the proof is done by construction and detailed in Section A.2.3. The key insight is that restricted powers "break down" over a smooth and deterministic sum unit p. That

is, $\left(\sum_i \theta_i p_i(\boldsymbol{x}) [\![\boldsymbol{x} \in \mathsf{supp}(p_i)]\!]\right)^\alpha [\![\boldsymbol{x} \in \mathsf{supp}(p)]\!] = \sum_i \theta_i^\alpha p_i^\alpha(\boldsymbol{x}) [\![\boldsymbol{x} \in \mathsf{supp}(p_i)]\!]$. This follows from the fact that for any $\boldsymbol{x}$, at most one indicator $[\![\boldsymbol{x} \in \mathsf{supp}(p_i)]\!]$ evaluates to 1. As such, when multiplying a deterministic sum unit with itself, each input will only have overlapping support with itself, thus effectively matching product units only with themselves. This is why decomposability suffices. In conclusion, this recursive decomposition of the power of a circuit will result in the power circuit having the same structure as the original circuit, with input functions and sum parameters replaced by their α-powers. The space and time complexity of the algorithm is $\mathcal{O}(|p|)$ for smooth, deterministic, and decomposable PCs, even for natural powers. This will be a key insight to compactly multiply circuits with the same support structure, such as when computing logarithms and entropies (Section 2.5).

We can already see an example of how simple operators can be composed to derive other tractable ones. Consider the quotient of two circuits $p(\mathbf{X})$ and $q(\mathbf{X})$, denoted as $p(\mathbf{X})/q(\mathbf{X})$, and restricted to $\mathsf{supp}(q)$. The quotient, appearing in queries such as KLD or Itakura-Saito divergence (Section 2.5), can be computed by first taking the reciprocal circuit (i.e., the (-1)-power) of q, followed by its product with p. Thus, if q is deterministic and compatible with p, we can take its reciprocal—which will have the same structure as q—and multiply with p to obtain the quotient as a decomposable circuit. There, we prove that the quotient between p and a non-deterministic q is #P-hard even if they are compatible.

Logarithms of a PC. The logarithm of a PC $p(\mathbf{X})$, denoted $\log p(\mathbf{X})$, is fundamental in computing quantities such as entropies and divergences between distributions (Section 2.5). Since the log is undefined for 0 we will again consider the *restricted logarithm*, denoted as $\log p(\boldsymbol{x})|_{\mathsf{supp}(p)}$ and equal to $\log p(\boldsymbol{x})$ if $\boldsymbol{x} \in \mathsf{supp}(p)$ and 0 otherwise.

Theorem 6 (Logarithms)**.** *If p is a smooth, deterministic and decomposable PC, then its restricted logarithm can be represented as a decomposable circuit in $\mathcal{O}(|p|)$ time. Otherwise, if p is only smooth and decomposable, or even structured-decomposable, computing its restricted logarithm as a decomposable circuit is #P-hard.*

Note that while the input of the logarithm operator must be a PC, its output can be a

general circuit. Moreover, if p is structured decomposable, then so is its logarithm. We point out that determinism again allows the restricted log to decompose over the support of the PC, but this time the output circuit is *not* deterministic. Nevertheless, the inputs of the newly introduced sum units can be clearly partitioned into groups sharing the same support of the corresponding product units in p. This implies that whenever we have to multiply a deterministic circuit and its logarithmic circuit—for instance to compute its Shannon entropy (Section 2.5)—we can leverage the sparsifying effect of non-overlapping supports and perform only a linear number of products (cf. product and power operators).

Exponentials of a Circuit. The exponential of a circuit $p(\mathbf{X})$, denoted $\exp(p(\mathbf{X}))$, is the inverse operation of the logarithm and is a fundamental operation when representing distributions such as log-linear models [83]. Similarly to the logarithm, building a decomposable circuit that encodes an exponential of a circuit is hard in general.

Theorem 7 (Hardness of exponentials)**.** *If p is a smooth and decomposable circuit, then, computing its exponential as a decomposable circuit is #P-hard, even if p is structured-decomposable.*

Unlike the logarithm however, restricting the operation to deterministic circuits does not help with tractability, since the issue comes from product units: the exponential of a product is neither a sum nor product of exponentials. Nevertheless, it is easy to see that if p encodes a linear sum over its variables, i.e., $p(\mathbf{X}) = \sum_i \theta_i X_i$, we could easily represent its exponential as a circuit comprising a single decomposable product unit, hence tractably.

Proposition 2 (Tractable exponential of a linear circuit)**.** *If p is a linear circuit, then its exponential can be represented as an omni-compatible circuit in $\mathcal{O}(|p|)$ time.*

Note that if we were to add an additional deterministic sum unit over many omni-compatible circuits built in this way, we would retrieve a mixture of truncated exponentials [120, 190]. This is the largest class of tractable exponentials we know so far, and enlarging

Table 2.2: ***Tractability and hardness of information-theoretic queries over circuits.*** Tractability given some conditions over the input circuits; computational hardness when some of these are unmet.

Query		Tract. Conditions	Hardness
Cross Entropy	$-\int p(\boldsymbol{x}) \log q(\boldsymbol{x}) \, \mathrm{d}\mathbf{X}$	Cmp, q Det	#P-hard w/o Det
Shannon Entropy	$-\sum p(\boldsymbol{x}) \log p(\boldsymbol{x})$	Sm, Dec, Det	coNP-hard w/o Det
Rényi Entropy	$(1-\alpha)^{-1} \log \int p^{\alpha}(\boldsymbol{x}) \, d\mathbf{X}, \alpha \in \mathbb{N}$	SD	#P-hard w/o SD
	$(1-\alpha)^{-1} \log \int p^{\alpha}(\boldsymbol{x}) \, d\mathbf{X}, \alpha \in \mathbb{R}_+$	Sm, Dec, Det	#P-hard w/o Det
Mutual Information	$\int p(\boldsymbol{x}, \boldsymbol{y}) \log(p(\boldsymbol{x}, \boldsymbol{y})/(p(\boldsymbol{x})p(\boldsymbol{y})))$	Sm, SD, Det*	coNP-hard w/o SD
Kullback-Leibler Div.	$\int p(\boldsymbol{x}) \log(p(\boldsymbol{x})/q(\boldsymbol{x})) d\mathbf{X}$	Cmp, Det	#P-hard w/o Det
Rényi's Alpha Div.	$(1-\alpha)^{-1} \log \int p^{\alpha}(\boldsymbol{x}) q^{1-\alpha}(\boldsymbol{x}) \, d\mathbf{X}, \alpha \in \mathbb{N}$	Cmp, q Det	#P-hard w/o Det
	$(1-\alpha)^{-1} \log \int p^{\alpha}(\boldsymbol{x}) q^{1-\alpha}(\boldsymbol{x}) \, d\mathbf{X}, \alpha \in \mathbb{R}$	Cmp, Det	#P-hard w/o Det
Itakura-Saito Div.	$\int [p(\boldsymbol{x})/q(\boldsymbol{x}) - \log(p(\boldsymbol{x})/q(\boldsymbol{x})) - 1] d\,\mathbf{X}$	Cmp, Det	#P-hard w/o Det
Cauchy-Schwarz Div.	$-\log \frac{\int p(\boldsymbol{x}) q(\boldsymbol{x}) d\mathbf{X}}{\sqrt{\int p^2(\boldsymbol{x}) d\mathbf{X} \int q^2(\boldsymbol{x}) d\mathbf{X}}}$	Cmp	#P-hard w/o Cmp
Squared loss	$\int (p(\boldsymbol{x}) - q(\boldsymbol{x}))^2 d\,\mathbf{X}$	Cmp	#P-hard w/o Cmp

its boundaries is an open problem. Our compositional atlas is now complete: If we were to add an additional circuit operator to it, it would take the form of the already discussed powers, logarithms or exponentials.

2.5 . . . to Complex Compositional Queries

In this section, we show how our atlas of simple tractable operators can be effectively used to systematically find a tractable model class for *any advanced query that comprises these operators*. We will show its practical utility by quickly coming up with tractability proofs as well as distilling efficient algorithms for several entropy and divergence queries that are largely used in ML. We will then discuss how our discovered tractable circuit classes subsume some previously known results in the literature and prove novel hardness results for when the structural properties of these circuits are unmet. Table 2.2 summarizes our results.

We now showcase how a short tractability proof can be easily distilled, using Rényi's α-divergence[3] [144] as an example. Note that no tractable algorithm was available for it yet. A proof can be built by inferring the sufficient conditions to tractably compute each operator in the pipeline—starting from the last before the integral and proceeding backwards

[3]Several alternative formulations of α-divergences can be found in the literature such as Amari's [118] and Tsallis's [127] divergences. However, as they share the same core operations—real powers and products of circuits—our results easily extend to them as well.

according to Table 2.1.

Theorem 8 (Tractable alpha divergence)**.** *The Rényi's α-divergence between two distributions p and q, defined as $(1-\alpha)^{-1}\log\int p^{\alpha}(\boldsymbol{x})q^{1-\alpha}(\boldsymbol{x})\,d\mathbf{X}$, can be computed exactly in $\mathcal{O}(|p|^{\alpha}|q|)$ time for $\alpha\in\mathbb{N},\alpha>1$ if p and q are compatible and q is deterministic, or in $\mathcal{O}(|p||q|)$ time for $\alpha\in\mathbb{R},\alpha\neq 1$ if p and q are both deterministic and compatible.*

Proof. A circuit pipeline for Rényi's α-divergence involves first computing $r=p^{\alpha}$ and $s=q^{1-\alpha}$, then $t=r\cdot s$ and finally integrate it.[4] Therefore we require t to be a smooth and decomposable circuit (Proposition 1), which in turn requires r and s to be compatible (Theorem 2). To conclude the proof, we need to compute two compatible circuits r and s in polytime, which can be done according to Theorem 5 or Theorem 3 depending on the value of α. As these theorems state, p^{α} and $q^{1-\alpha}$ will be compatible with p and q , respectively, with sizes $\mathcal{O}(|p|^{\alpha})$ and $\mathcal{O}(|q|)$ for a natural power α or $\mathcal{O}(|p|)$ and $\mathcal{O}(|q|)$ for a real-valued α. As such, t could be computed in $\mathcal{O}(|p|^{\alpha}|q|)$ time for $\alpha\in\mathbb{N}$ or $\mathcal{O}(|p||q|)$ for $\alpha\in\mathbb{R}$ (Theorem 2). □

We leave the formal theorems and proofs for the other queries listed in Table 2.2 to Section A.3 in the Appendix for space constraints. We remark again that our technique can be used beyond this query list and *can be applied to any complex query that involves a pipeline comprising the operations we discussed in Section 2.4 and culminating in an integration.*

Shannon entropy Smooth, decomposable and deterministic PCs enable the exact computation of Shannon entropy and this tractability result translates to bounded-treewidth PGMs such as Chow-Liu trees and polytrees as they are special cases. Our framework provides a more succinct tractability proof for the computation of Shannon entropy derived by [158], which we complete by proving that it is coNP-hard for non-deterministic PCs.

Rényi entropy For non-deterministic PCs we can employ the tractable computation of Rényi entropy of order α [144], which recovers Shannon Entropy for $\alpha\to 1$. As the logarithm

[4]Note that all the operations outside integration are tractable, therefore we can skip them.

is taken after integration of the power circuit, the tractability and hardness follow directly from those of the power operation (Theorem 3 and 5).

Cross entropy As hinted by the presence of a logarithm, the cross entropy is #P-hard to compute without determinism, even for compatible PCs. Nevertheless, given our atlas the cross entropy can be tractably computed in $\mathcal{O}(|p||q|)$ if p and q are deterministic and compatible.

Mutual information Let a joint distribution $p(\mathbf{X}, \mathbf{Y})$ and its marginals $p(\mathbf{X})$ and $p(\mathbf{Y})$ be represented as PCs. Then the mutual information (MI) over these three PCs can be computed via a pipeline involving product, quotient, and log operators and it is tractable if all circuits are compatible and deterministic. On the other hand, if the marginal distributions cannot be represented as compact deterministic PCs, we prove it to be coNP-hard.

Divergences [91] proposed an efficient algorithm to compute the KLD tailored for PSDDs.[5] This has been the only tractable divergence available for PCs so far. We greatly extend this panorama with our atlas by introducing Rényi's α-divergences which generalize several other divergences such as the KLD when $\alpha \to 1$, Hellinger's squared divergence when $\alpha = 2^{-1}$, and the $\mathcal{X}^2$-divergence when $\alpha = 2$ [54]. As Theorem 8 states, they are tractable for compatible and deterministic PCs, as is the Itakura-Saito divergence [182]. For non-deterministic PCs, we characterize the tractability of the squared loss and the Cauchy-Schwarz divergence [69]. The latter has applications in mixture models for approximate inference [171] and has been derived in closed-form only for mixtures of simple parametric forms like Gaussians [73], Weibull and Rayligh distributions [125]. Our results generalize them to deep mixture models [136].

Expectation queries Among other complex queries that can be abstracted into the general form of an expectation of a circuit f w.r.t. a PC p, i.e., $\mathbb{E}_{\boldsymbol{x} \sim p(\mathbf{X})}[f(\boldsymbol{x})]$, there are the moments of distributions, such as means and variances. They can be efficiently computed for any smooth and decomposable PC, as f is an omni-compatible circuit. This result generalizes

[5] Note that our tractability proof in Theorem 30 is only a few lines long and does not require the lengthy and ad-hoc algebraic derivations of [91].

the moment computation for simple models such as GMMs and HMMs as they can be encoded as smooth and decomposable PCs. If f is the indicator function of a logical formula, the expectation computes its probability w.r.t. the distribution p. [13] proposed an algorithm tailored to formulas f over binary variables, encoded as SDDs [36] w.r.t. distributions that are PSDDs. We generalize this result to mixed continuous-discrete distributions encoded as structured-decomposable PCs that are not necessarily deterministic and to logical formulas in the language of satisfiability modulo theories [5] over linear arithmetics with univariate literals. Lastly, if f encodes constraints over the output distribution of a deep network, we retrieve the *semantic loss* [184]. If f encodes a classifier or a regressor, then $\mathbb{E}_p[f]$ refers to computing its expected predictions w.r.t. p [76]. Our results generalize the results reported in [174] such as computing the expectations of decision trees and their ensembles [77] as well as those of *deep regression circuits* [75].[6]

2.6 Experiments

We prototyped the tractable operators defined in Section 2.4 as subroutines in Julia in the Juice.jl framework [29] to showcase how our modular atlas can practically and quickly help implement tractable algorithms for novel query classes.[7] To this extent, we distilled algorithms for the Shannon, Rényi, and cross entropies and for the KL, α, and Cauchy-Schwarz divergences. We then ran them on deterministic and structured-decomposable circuits learned as in [32] from 20 publicly available real-world benchmark datasets [106, 176].

Table 2.3 and Table 2.4 report all the intermediate circuit sizes in their respective pipelines as well as the time taken to build and execute the pipelines. First, the intermediate circuits created in the pipeline do not blow up in size: as predicted by our theoretical analysis, the size of the logarithm circuit grows by a linear factor (∼3–4x). Moreover, the size of the product circuit $p \cdot q$ is only slightly larger than $\max(|p|, |q|)$ when p and q are deterministic,

[6] Despite the name, regression circuits do not conform to our definition of circuits.
[7] Code is available at https://github.com/UCLA-StarAI/circuit-ops-atlas.

Table 2.3: Sizes of the intermediate and final circuits as processed by the operators in the pipelines of the Shannon and Rényi (for $\alpha = 1.5$) entropies and Kullback-Leibler and Alpha (for $\alpha = 1.5$) divergences when computed for two input circuits p and q learned from 20 different real-world datasets as in [32].

Dataset	p	q	p^α	$q^{1-\alpha}$	$r = \log(q)$	$s = p/q$	$t = \log(s)$	$p \times q$	$p \times r$	$p \times t$	$p^\alpha \times q^{1-\alpha}$
nltcs	2779	7174	2779	7174	26155	7202	26239	7202	26183	26239	7202
msnbc	2765	6614	2765	6614	24111	6634	24171	6634	24131	24171	6634
kdd	4963	50377	4963	50377	184575	50417	184695	50417	184615	184695	50417
plants	12909	64018	12909	64018	234661	64070	234817	64070	234713	234817	64070
audio	10278	45864	10278	45864	168062	45950	168320	45950	168148	168320	45950
jester	6475	35369	6475	35369	129579	35479	129909	35479	129689	129909	35479
netflix	5068	14636	5068	14636	53571	14706	53781	14706	53641	53781	14706
accidents	3193	8183	3193	8183	29891	8299	30239	8299	30007	30239	8299
retail	4790	14926	4790	14926	54554	14994	54758	14994	54622	54758	14994
pumsb	4277	12461	4277	12461	45500	12595	45902	12595	45634	45902	12595
dna	73828	856955	73828	856955	3141981	857029	3142203	857029	3142055	3142203	857029
kosarek	5115	12988	5115	12988	47354	13106	47708	13106	47472	47708	13106
msnweb	4859	9025	4859	9025	32675	9175	33125	9175	32825	33125	9175
book	7718	12731	7718	12731	45985	12943	46621	12943	46197	46621	12943
movie	8309	11732	8309	11732	42374	11926	42956	11926	42568	42956	11926
webkb	10598	13397	10598	13397	47859	13653	48627	13653	48115	48627	13653
cr52	10912	14348	10912	14348	51094	14546	51688	14546	51292	51688	14546
c20ng	11386	14630	11386	14630	52120	14886	52888	14886	52376	52888	14886
bbc	13884	17016	13884	17016	60857	17282	61655	17282	61123	61655	17282
ad	17744	21676	17744	21676	76870	21920	77602	21920	77114	77602	21920

much smaller than the theoretical bound of $\mathcal{O}(|p||q|)$.

In terms of execution time, our algorithms run in less than a second for most circuits and peak at slightly more than one minute to compute a pipeline of the KLD, whose output circuit has more than 3 million edges on the DNA dataset (Table 2.3). A custom and highly optimized implementation of the KLD for PSDDs by [91] runs up to ten times faster on smaller circuits but surprisingly takes ∼220 seconds for DNA, highlighting that our compositional atlas is a promising way to distill tractable algorithms. We emphasize that the aim of these experiments, however, is to demonstrate that *our compositional framework can quickly distill new tractable algorithms for queries that were not available before*, such as Rényi entropy and α and Cauchy-Schwarz divergences.

Table 2.4: Times in seconds to compute the Shannon entropy (ENT), the cross-entropy (XENT), Kullback-Leibler (KLD), Alpha (for $\alpha = 1.5$) divergence, Rényi entropy (RényiEnt), and Cauchy-Schwarz divergence (CSDiv) over the circuits learned from 20 different real-world datasets by either using the algorithm distilled by our pipelines (see Table 2.3 and Fig. 2.4) compared to the custom and highly-optimized implementations of the same ENT [158] and KLD [91] algorithms as available in Juice.jl [29].

Dataset	ENT		KLD		XENT		AlphaDiv		RényiEnt		CSDiv	
	ours	Juice	ours	Juice	ours	Juice	ours	Juice	ours	Juice	ours	Juice
NLTCS	0.143	0.001	0.830	0.207	0.422	-	0.140	-	0.013	-	0.300	-
MSNBC	0.109	0.001	0.369	0.182	0.297	-	0.105	-	0.018	-	0.227	-
KDD	0.157	0.001	3.154	0.790	2.180	-	0.885	-	0.016	-	1.136	-
PLANTS	0.679	0.005	3.983	3.909	3.739	-	1.160	-	0.088	-	1.572	-
AUDIO	0.406	0.003	2.736	1.681	1.873	-	0.537	-	0.029	-	0.771	-
JESTER	0.764	0.003	1.019	0.432	0.805	-	0.351	-	0.024	-	0.476	-
NETFLIX	0.106	0.002	0.352	0.175	0.264	-	0.100	-	0.017	-	0.201	-
ACCIDENTS	0.055	0.001	0.207	0.039	0.542	-	0.091	-	0.009	-	0.124	-
RETAIL	0.108	0.001	0.508	0.153	0.415	-	0.184	-	0.013	-	0.197	-
PUMSB	0.092	0.001	0.701	0.133	0.316	-	0.119	-	0.012	-	0.214	-
DNA	4.365	0.027	64.664	220.377	52.997	-	15.609	-	0.255	-	22.901	-
KOSAREK	0.182	0.002	0.477	0.106	0.379	-	0.139	-	0.011	-	0.735	-
MSNWEB	0.128	0.002	0.261	0.047	0.211	-	0.342	-	0.015	-	0.135	-
BOOK	0.086	0.003	0.215	0.036	0.202	-	0.075	-	0.020	-	0.115	-
MOVIE	0.272	0.002	0.443	0.063	0.373	-	0.172	-	0.015	-	0.194	-
WEBKB	0.138	0.003	0.241	0.031	0.164	-	0.079	-	0.023	-	0.098	-
CR52	0.141	0.004	0.260	0.035	0.188	-	0.087	-	0.031	-	0.143	-
C20NG	0.118	0.003	0.264	0.034	0.194	-	0.088	-	0.032	-	0.101	-
BBC	0.205	0.005	0.308	0.037	0.225	-	0.110	-	0.038	-	0.189	-
AD	0.193	0.007	0.346	0.046	0.281	-	0.151	-	0.031	-	0.207	-

```
function kld(p, q)
    r = quotient(p, q)
    s = log(r)
    t = product(p, s)
    return integrate(t)
end
```

```
function xent(p, q)
    r = log(q)
    s = product(p, r)
    return -integrate(s)
end
```

```
function csdiv(p, q)
    r = product(p, q)
    s = real_pow(p, 2.0)
    t = real_pow(q, 2.0)
    a = integrate(r)
    b = integrate(s)
    c = integrate(t)
    return -log(a / sqrt(b * c))
end
```

```
function ent(p)
    q = log(p)
    r = product(p, q)
    return -integrate(s)
end
```

```
function alphadiv(p, q, alpha=1.5)
    r = real_pow(p, alpha)
    s = real_pow(q, 1.0-alpha)
    t = product(r, s)
    return log(integrate(t)) / (1.0-alpha)
end
```

Figure 2.4: The modular operators defined in Section 2.4 can be easily composed to implement tractable algorithms for novel query classes. Here we show the code snippet for five queries: Kullback-Leibler divergence (`kld`), Cross Entropy (`xent`), Entropy (`ent`), Alpha divergence (`alphadiv`), and Cauchy-Schwarz divergence (`csdiv`).

Chapter 3

Scalable Learning of Probabilistic Circuits – Algorithmic Side

In the previous chapter, we discussed how to systematically derive efficient inference algorithms for various probabilistic queries in the context of PCs. However, such inference tools will only be helpful if we can learn PCs that accurately capture the distribution of complex and real-world data. This section focuses on three algorithmic side developments that collectively allow us to significantly improve modeling performance and lead to competitive performance against expressive deep generative models such as diffusion models and variational autoencoders.

3.1 Hidden Chow-Liu Trees – A General-Purpose Architecture

Hidden Chow-Liu Trees (HCLTs) are smooth and structured-decomposable PCs that combine the ability of Chow-Liu Trees (CLTs) [22] to capture feature correlations and the extra

The contents of this chapter appeared in papers [31, 97, 99].

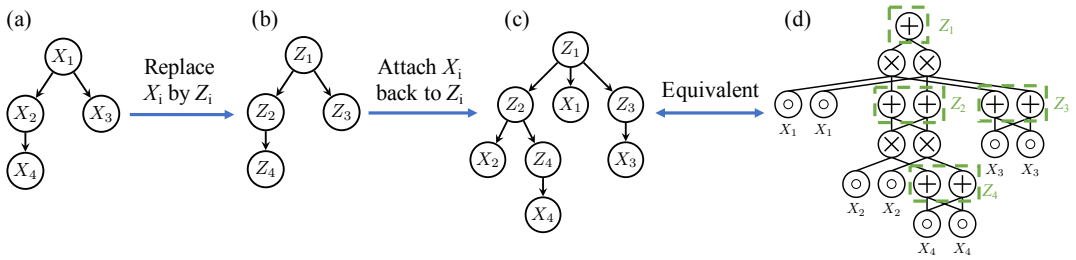


Figure 3.1: An example of constructing an HCLT PC given a dataset $\mathcal{D}$ with 4 features. (a): Construct the Chow-Liu Tree over variables $X_1, \ldots, X_4$ using $\mathcal{D}$. (b): Replace every variable X_i by its corresponding latent variable Z_i. (c): Attach all X_i back to their respective latent variables Z_i. (d): This PGM representation of HCLT is compiled into an equivalent PC.

expressive power provided by latent variable models. Every HCLT can be equivalently represented as a Probabilistic Graphical Model (PGM) [83] with latent variables. Specifically, Fig. 3.1(a)-(c) demonstrate how to construct the PGM representation of an example HCLT. Given a dataset $\mathcal{D}$ containing 4 features $\mathbf{X} = X_1, \ldots, X_4$, we first learn a CLT w.r.t. $\mathbf{X}$ (Fig. 3.1(a)). To improve expressiveness, latent variables are added to the CLT by the two following steps: (i) replace observed variables X_i by their corresponding latent variables Z_i, which are defined to be categorical variables with M (a hyperparameter) categories (Fig. 3.1(b)); (ii) connect observed variables X_i with the corresponding latent variables Z_i by directed edges $Z_i \to X_i$. This leads to the PGM representation of the HCLT shown in Fig. 3.1(c).

Finally, we are left with generating a PC that represents an equivalent distribution w.r.t. the PGM in Fig. 3.1(c). After generating the PGM representation of an HCLT model, we are left with the final step of compiling the PGM representation of the model into an equivalent PC. Recall that we define the latent variables $\{Z_i\}_{i=1}^{4}$ as categorical variables with M categories, where M is a hyperparameter. As demonstrated in Algorithm 1, we incrementally compile every PGM node into an equivalent PC unit through a bottom-up traverse (line 5) of the PGM. Specifically, leaf PGM nodes corresponding to observed variables X_i are compiled into PC input units of X_i (line 6), and inner PGM nodes corresponding to latent variables are compiled by taking products and sums (implemented by product and

Algorithm 1 Compile the PGM representation of a HCLT into an equivalent PC

Input: A PGM representation of a HCLT $\mathcal{G}$ (e.g., Fig. 3.1(c)); hyperparameter M
Output: A smooth and structured-decomposable PC p equivalent to $\mathcal{G}$
Initialize: cache $\leftarrow$ dict() a dictionary storing intermediate PC units
Sub-routines: **PC_leaf**(X_i) returns a PC input unit of variable X_i; **PC_prod**$(\{n_i\}_{i=1}^m)$ (resp. **PC_sum**$(\{n_i\}_{i=1}^m)$) returns a product (resp. sum) unit over child nodes $\{n_i\}_{i=1}^m$.
foreach node g traversed in postorder (bottom-up) of $\mathcal{G}$ **do**
    if $\text{var}(g) \in \mathbf{X}$ **then** cache$[g] \leftarrow \big[$**PC_leaf**$\big(\text{var}(g)\big)$ for $i = 1 : M\big]$
    else # That is, $\text{var}(g) \in \mathbf{Z}$
        chs_cache $\leftarrow \big[\text{cache}[c]$ for c in children$(g)\big]$ # children(g) is the set of children of g
        prod_nodes $\leftarrow \big[$**PC_prod**$\big(\big[\text{nodes}[i]$ for nodes in chs_cache$\big]\big)$ for $i = 1 : M\big]$
        cache$[g] \leftarrow \big[$**PC_sum**$\big(\text{prod_nodes}\big)$ for $i = 1 : M\big]$
return cache[root$(\mathcal{G})$][0]

sum units) of its child nodes' PC units (lines 8-10). Leaf units generated by **PC_leaf**(X) can be any simple univariate distribution of X. We used categorical leaf units in our HCLT experiments. Fig. 3.1(d) demonstrates the result PC after running Algorithm 1 with the PGM in Fig. 3.1(c) and $M = 2$. Fig. 3.1(d) illustrates an HCLT that is equivalent to the PGM shown in Fig. 3.1(c) (with $M = 2$).

3.2 Learning Sparse PCs with Pruning and Growing

Recent advancements in PC learning and regularization [159], and efficient implementations [30, 119, 132] have been pushing the limits of PC's expressiveness and scalability such that they can even match the performance of less tractable deep generative models, including flow-based models and VAEs. However, the performance of PCs plateaus as model size increases. This suggests that to further boost the performance of PCs, simply scaling up the model size does not suffice and we need to better utilize the available capacity.

We discover that this might be caused by the fact that the capacity of large PCs is wasted. As shown in Figure 3.2, most parameters in a PC with 2.18M parameters have close-to-zero values, which have little effect on the PC distribution. Since existing PC structures usually have fully-connected parameter layers [98, 141], this indicates that the parameter values are only sparsely used.

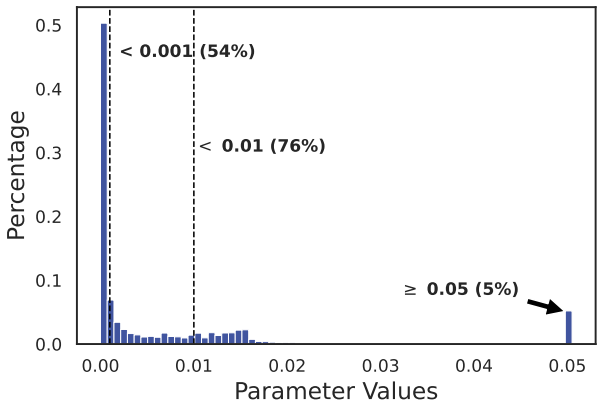
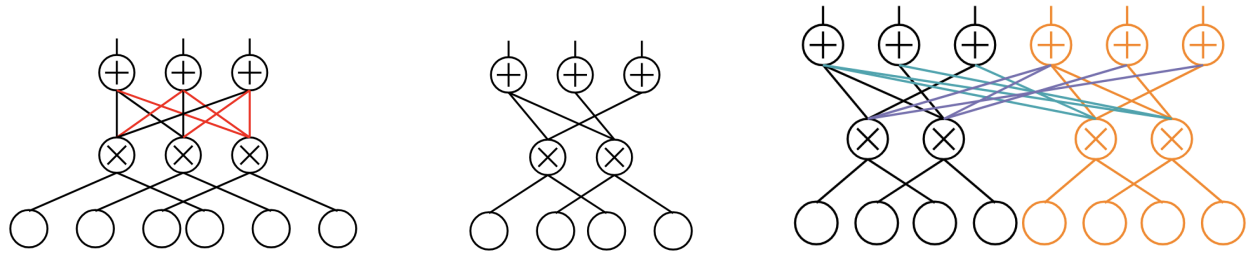


Figure 3.2: Histogram of parameter values for a state-of-the-art PC with 2.18M parameters on MNIST. 95% of the parameters have close-to-zero values.

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 parameters 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 by 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.



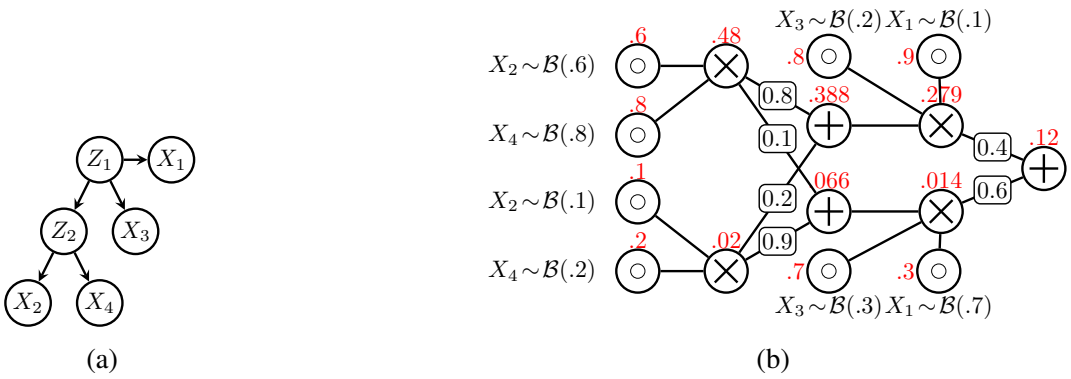
(a) PC with fully connected layers (b) PC after pruning operation (c) PC after growing operation.

Figure 3.3: A demonstration of the pruning and growing operation. From 3.3a to 3.3b, the red edges are pruned. From 3.3b to 3.3c, the nodes are doubled, and each parameter is copied 3 times.

3.2.1 Probabilistic Circuit Model Compression via Pruning

Figure 3.2 shows that most parameters in a large PC are very close to zero. Given that these parameters are weights associated with mixture (sum unit) components, the corresponding edges and sub-circuits have little impact on the sum unit output. Hence, by pruning away these unimportant components, it is possible to significantly reduce model size while retaining model expressiveness. Fig. 3.3b illustrates the result of pruning five (red) edges from the PC in Fig. 3.3a. Given a PC and a dataset, our goal is to efficiently identify a set of edges to prune, such that the log-likelihood gap between the pruned PC and the original PC on the given dataset is minimized.

Pruning by parameters. The parameter value statistics in Figure 3.2 suggest that a natural criterion is to prune edges by the magnitude of their corresponding parameter. This leads to the EPARAM (edge parameters) heuristic, which selects the set of 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 conditional distribution represented by its corresponding sum unit n, not the joint distribution of the entire PC. Figure 3.5(a) illustrates what happens when the edge with the smallest



(a) (b)

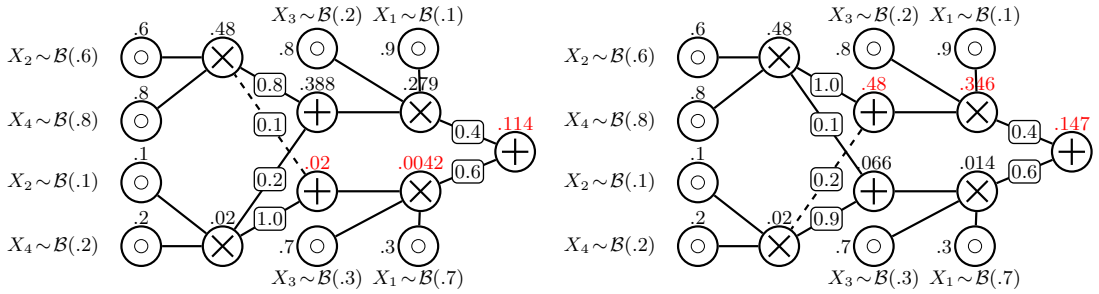
Figure 3.4: A smooth and decomposable PC (b) and an equivalent Bayesian network (a). The Bayesian network is over 4 variables $\mathbf{X} = \{X_1, X_2, X_3, X_4\}$ and 2 hidden variables $\mathbf{Z} = \{Z_1, Z_2\}$ with $h = 2$ hidden states. The feedforward computation order is from left to right; $\odot$ are input Bernoulli distributions, $\bigotimes$ are product units, and $\bigoplus$ are sum units; parameter values are annotated in the box. The probability of each unit given input assignment $\{X_1 = 0, X_2 = 1, X_3 = 0, X_4 = 1\}$ is labeled red.

parameter is pruned from the PC in Figure 3.4.

However, as shown in Figure 3.5(b), 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 the probabilistic "circuit flow" semantics of PCs. We will introduce circuit flows later in this section, along with their corresponding heuristics EFLOW. Before that, we first introduce an intermediate concept based on the notion of generative significance of PCs.

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, 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 2 shows how to draw samples from a PC distribution through a recursive implementation: (i) for an input unit n defined on variable X, the algorithm randomly samples value x according to its input univariate distribution; (ii) for a product unit, by



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

Figure 3.5: A case study comparing pruning heuristics (EPARAM and EFLOW) on the PC in Fig. 3.4 given sample $\{X_1 = 0, X_2 = 1, X_3 = 0, X_4 = 1\}$. The pruned edges are dashed and parameters are re-normalized. Compared to the likelihood of the original PC, the changed likelihoods are in red, showing that pruning by flows results in less likelihood decrease.

Algorithm 2 Sampling from a Probabilistic Circuit (PC)

Input: A PC representing joint distribution $p_{\mathcal{C}}(\mathbf{X})$
Output: An instance $\boldsymbol{x}$ sampled from $p_{\mathcal{C}}$
Sub-routine: Sample(n) recursively samples a value from the sub-PC rooted at node n
function Sample(n):
  if n is an input unit **then**
    $f_n(X) \leftarrow$ univariate distribution of n
    return sample $x \sim f_n(X)$
  else if n is a product unit **then**
    $\boldsymbol{x}_c \leftarrow$ **Sample**(c) for each $c \in \mathsf{ch}(n)$
    return Concatenate$(\{\boldsymbol{x}_c\}_{c \in \mathsf{ch}(n)})$
  else # n is a sum unit
    Sample $c^* \in \mathsf{ch}(n)$ with probability proportional to $\{\theta_{c|n}\}_{c \in \mathsf{ch}(n)}$
    return Sample(c^*)
return Sample(r), where r is the root of PC $\mathcal{C}$

decomposability its children have disjoint scope, thus we draw samples from all input units and then concatenate the samples together; (iii) for a sum unit n, by smoothness its children have identical scope, thus we first randomly sample one of its input units according to the categorical distribution defined by sum parameters $\{\theta_{n,c} : c \in \mathsf{ch}(n)\}$, and then sample from this input unit recursively. Besides actually drawing samples from the PC, we can also compute the probability that n will be visited during the sampling process. This provides a good measure of the importance of unit n to the PC distribution as a whole, which we define as the *top-down probability*.

Definition 6 (Top-down Probability)**.** The top-down probability of each unit n in a PC with parameters $\boldsymbol{\theta}$ is defined recursively as follows, assuming alternating sum and product layers:

$$q(n;\boldsymbol{\theta}) := \begin{cases} 1 & \text{if } n \text{ is the root unit,} \\ \sum_{m\in\mathsf{pa}(n)} q(m;\boldsymbol{\theta}) & \text{if } n \text{ is a sum unit,} \\ \sum_{m\in\mathsf{pa}(n)} \theta_{m,n}\cdot q(m;\boldsymbol{\theta}) & \text{if } n \text{ is a product unit,} \end{cases}$$

where $\mathsf{pa}(n)$ are the units that take n as input in the feedforward computation. Moreover, the top-down probability of a sum edge (n, c) is defined as $q(n, c;\boldsymbol{\theta}) = \theta_{n,c}\cdot q(n;\boldsymbol{\theta})$.

The top-down probability of the root is always 1; a product unit passes its top-down probability to all its inputs, and a sum unit distributes its top-down probability to its inputs proportional to the corresponding edge weights. Therefore, the top-down probability of a non-root unit is summing over all probabilities it receives from its outputs.

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. Following the intuition that the top-down probability defines the probability that units will be visited during the sampling process, pruning edges with the smallest top-down probability constitutes a reasonable pruning strategy.

Pruning by circuit flows. The top-down probability $q(n;\boldsymbol{\theta})$ represents the probability of reaching unit n in an unconditional random sampling process. Despite its ability to capture global information of PC parameters, the top-down probability is not tailored to a specific dataset. Therefore, to further utilize the dataset information, we can measure the probability of reaching certain units/edges in the sampling process *conditioning on some instance* $\boldsymbol{x}$ *being sampled*. To bridge this gap, we define circuit flow as a sample-dependent version of the top-down probability.

Definition 7 (Circuit Flow[1])**.** For a given PC with parameters $\boldsymbol{\theta}$ and example $\boldsymbol{x}$, the circuit flow of unit n on example $\boldsymbol{x}$ is the probability that n will be visited during the sampling procedure conditioned on $\boldsymbol{x}$ being sampled. This can be computed recursively as follows, assuming alternating sum and product layers:

$$\mathrm{F}_n(\boldsymbol{x}) = \begin{cases} 1 & \text{if } n \text{ is the root unit,} \\ \sum_{m \in \mathsf{pa}(n)} \mathrm{F}_m(\boldsymbol{x}) & \text{if } n \text{ is a sum unit,} \\ \sum_{m \in \mathsf{pa}(n)} \frac{\theta_{m,n} \cdot p_n(\boldsymbol{x})}{p_m(\boldsymbol{x})} \cdot \mathrm{F}_m(\boldsymbol{x}) & \text{if } n \text{ is a product unit.} \end{cases}$$

Similarly, the edge flow $\mathrm{F}_{n,c}(\boldsymbol{x})$ on sample $\boldsymbol{x}$ is defined by $\mathrm{F}_{n,c}(\boldsymbol{x}) = \theta_{n,c} \cdot p_c(\boldsymbol{x})/p_n(\boldsymbol{x}) \cdot \mathrm{F}_n(\boldsymbol{x})$. We further define $\mathrm{F}_{n,c}(\mathcal{D}) = \sum_{\boldsymbol{x} \in \mathcal{D}} \mathrm{F}_{n,c}(\boldsymbol{x})$ as the *aggregate edge flow* over dataset $\mathcal{D}$.

Effectively, we can think of $\theta^{\boldsymbol{x}}_{m,n} := \theta_{m,n} \cdot p_n(\boldsymbol{x})/p_m(\boldsymbol{x})$ as the posterior probability of component n in the mixture of sum unit m *conditioned on observing sample* $\boldsymbol{x}$. Then, circuit flow is the top-down probability under this $\boldsymbol{\theta}^{\boldsymbol{x}}$ reparameterization of the circuit: $\mathrm{F}_n(\boldsymbol{x}) = q(n; \boldsymbol{\theta}^{\boldsymbol{x}})$ and $\mathrm{F}_{n,c}(\boldsymbol{x}) = q(n, c; \boldsymbol{\theta}^{\boldsymbol{x}})$.

Circuit flow $\mathrm{F}_n(\boldsymbol{x})$ defines the probability of reaching unit n in the top-down sampling procedure of 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 $\boldsymbol{x}$.

3.2.2 Bounding and Approximating the Loss of Likelihood

In this section, we theoretically quantify the impact of edge pruning on model performance. In particular, we establish an upper bound on the log-likelihood drop $\Delta\mathtt{LL}$ on a given dataset $\mathcal{D}$ by comparing (i) the original PC $\mathcal{C}$ and (ii) the pruned PC $\mathcal{C}_{\setminus\mathcal{E}}$ caused by pruning away edges $\mathcal{E}$:

$$\Delta\mathtt{LL}(\mathcal{D}, \mathcal{C}, \mathcal{E}) = \mathtt{LL}(\mathcal{D}, \mathcal{C}) - \mathtt{LL}(\mathcal{D}, \mathcal{C}_{\setminus\mathcal{E}}). \tag{3.1}$$

[1]Earlier work defined "circuit flow" or "expected circuit flow" in the context of parameter learning [16, 97], without observing the connection to sampling. We contribute its more intuitive sampling semantics here.

We start from the case of pruning one edge (i.e., $|\mathcal{E}|=1$ in Equation 3.1). In this case, the loss of likelihood can be quantified exactly using flows and edge parameters:

Theorem 9 (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*

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

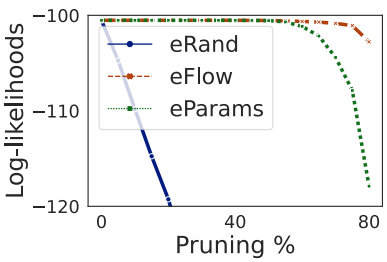
Proof is provided in Section B.1.2. By computing the second term in Theorem 9, we can pick the edge with the smallest log-likelihood drop. Additionally, the third term characterizes the log-likelihood drop without re-normalizing parameters of $\{\theta_{n,c}\}_{c\in\mathsf{ch}(n)}$. It suggests pruning the edge with the smallest edge flow. A key insight from Theorem 9 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 3.2.1 and suggests that the circuit flow heuristic proposed in the previous section is a good approximation of the derived upper bound.

Next, we bound the log-likelihood drop of pruning multiple edges.

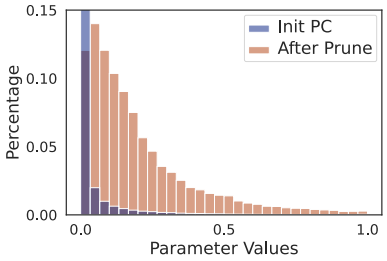
Theorem 10 (Log-likelihood drop of pruning multiple edges)**.** *Let $\mathcal{C}$ be a PC and $\mathcal{D}$ be a dataset. For any set of edges $\mathcal{E}$ in $\mathcal{C}$, if $\forall\boldsymbol{x}\in\mathcal{D}, \sum_{(n,c)\in\mathcal{E}}\mathrm{F}_{n,c}(\boldsymbol{x})<1$, the log-likelihood drop by pruning away $\mathcal{E}$ is bounded and approximated by*

$$\Delta\mathrm{LL}(\mathcal{D},\mathcal{C},\mathcal{E})\leq-\frac{1}{|\mathcal{D}|}\sum_{\boldsymbol{x}}\log(1-\sum_{(n,c)\in\mathcal{E}}\mathrm{F}_{n,c}(\boldsymbol{x}))\approx\frac{1}{|\mathcal{D}|}\sum_{(n,c)\in\mathcal{E}}\mathrm{F}_{n,c}(\mathcal{D}).\tag{3.2}$$

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. Hence, finding the set of edges with the lowest score requires evaluating the bound exponentially many times with respect to the number of pruned edges. Therefore, we do an additional approximation step of the bound via Taylor expansion, which leads to the third term of Equation 3.2. This approximation matches the EFLOW heuristic by a constant factor $1/|\mathcal{D}|$, which theoretically



(a) Comparison of heuristics ERAND, EPARAM, and EFLOW. Heuristic EFLOW can prune up to 80% of the parameters without much loglikelihoods decrease.



(b) Histogram of parameters before (the same as in Figure 1) and after pruning. The parameter values take higher significance after pruning.

Figure 3.6: Empirical evaluation of the pruning operation.

justifies the effectiveness of the heuristic. Figure 3.6 empirically compares the actual log-likelihood drop and the quantity computed from the circuit flow heuristic (that is, the approximate upper bound) for different percentages of pruned parameters. We see that the approximate bound matches closely to the actual log-likelihood drop.

3.2.3 Scalable Structure Learning

The pruning operator improves two aspects of PCs. First, as shown in Fig. 3.6(b), model parameters are more balanced after pruning. Second, pruning removes sub-circuits with negligible contributions to the model's distribution. If we treat PCs 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 impossible 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 define a scalable structure learning algorithm for PCs.

Growing. *Growing* is an operator that increases model size by copying its existing components and injecting noise. As shown in Figure 3.3, after applying the growing operation on the original PC in Figure 3.3b, we can get a new grown PC as in Figure 3.7. Specifically,

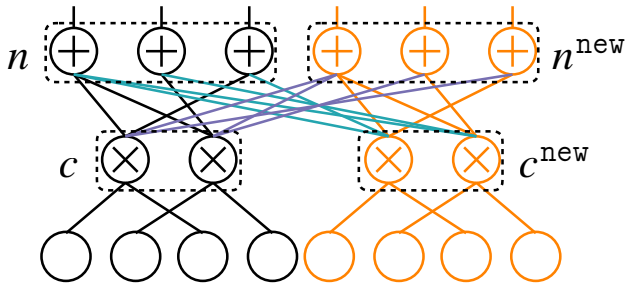


Figure 3.7: Growing operation. Each unit is doubled, and each parameterized edge is copied 3 times: $(n^{\texttt{new}}, c^{\texttt{new}})$ (orange), $(n^{\texttt{new}}, c)$ (purple), and $(n, c^{\texttt{new}})$ (green).

the growing operation is applied to units, edges, and parameters respectively: (1) for units, growing operates on every PC unit n and creates another copy $n^{\texttt{new}}$; (2) for edges, the sum edge (n, c) from the original PC (Figure 3.3b) are copied three times to the grown PC (Figure 3.7): 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 $(n^{\texttt{new}}, c)$; product edges are added to connect the copied version of a product unit and its copied inputs; (3) a new parameter $\theta^{\texttt{new}}_{c|n}$ is a noisy copy of an 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 to ensure 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 18 in appendix shows a feedforward 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 of 4, applying growing after pruning 75% of the edges from an initial PC keeps the number of parameters unchanged. We propose a joint structure and parameter learning algorithm for PCs that uses these two operations. Specifically, starting from an initial PC, we apply 75% pruning, growing, and parameter learning iteratively until convergence. We utilize HCLTs [98] 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 [15]. 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 $\theta^{\texttt{new}}_{c|n} = \mathrm{F}_{n,c}(\mathcal{D}_B)/\mathrm{F}_n(\mathcal{D}_B)$. The parameters are then updated with learning rate α: $\theta^{t+1} \leftarrow \alpha\theta^{\texttt{new}} + (1-\alpha)\theta^t$ (M-step). Empirically this approach converges faster and is better regularized compared to full-batch EM.

Parallel Computation. Existing approaches to scaling up learning and inference with PCs, such as Einsum networks [132], utilize fully connected parametrized layers (Figure 3.3a) of PC structures such as HCLT [98] and RatSPN [133]. 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 inference based on Juice.jl [30], 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 on the GPU. As a result, we can train PCs with millions of parameters in less than half an hour.

3.2.4 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 [88], EMNIST [24], and FashionMNIST [183]; (2) the character-level Penn Tree Bank language modeling task [110].

Section 3.2.5 first reports the best results we get on image datasets and language modeling tasks via the structure learning procedure proposed in Section 3.2.3. Section 3.2.6 then shows the effect of pruning and growing operations via two detailed experimental settings. It studies two different constrained optimization problems: finding the smallest PC for a

given likelihood via model compression and finding the best PC of a given size via structure learning.

Settings. For all experiments, we use hidden Chow-Liu Trees (HCLTs) [97] with the number of latent states in $\{16, 32, 64, 128\}$ as initial PC structures. We train the parameters of PCs with stochastic mini-batch EM. We perform early stopping and hyperparameter search using a validation set and report results on the test set. Please refer to Appendix B.1.3 for more details. We use mean test set bits-per-dimension (bpd) as the evaluation criteria, where $\mathsf{bpd}(\mathcal{D}, \mathcal{C}) = -\mathcal{LL}(\mathcal{D}, \mathcal{C})/(\log(2) \cdot m)$ and m is the number of features in dataset $\mathcal{D}$.

3.2.5 Density Estimation Benchmarks

Image Datasets. The MNIST-family datasets contain gray-scale pixel images of size 28×28 where each pixel takes values in $[0, 255]$. We split out 5% of training data as a validation set. We compare with two competitive PC learning algorithms: HCLT [98] and RatSPN [133], one flow-based model: IDF [64], and three VAE-based methods: BitSwap [81], BB-ANS [168], and McBits [148]. For a fair comparison, we implement RatSPN structures ourselves and use the same training pipeline and EM optimizer as our proposed method. Note that EinsumNet [132] also uses RatSPN structures but with a PyTorch implementation so its comparison is subsumed by comparison with RatSPN. All 7 methods are tested on MNIST, 4 splits of EMNIST and FashionMNIST. As shown in Table 3.1, 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 experiment details are in Appendix B.1.3.

Language Modeling Task. We use the Penn Tree Bank dataset with standard processing from [117], which contains around 5M characters and a character-level vocabulary size of 50. The data is split into sentences with a maximum sequence length of 288. We compare with three competitive normalizing-flow-based models: Bipartite flow [170] and

Table 3.1: 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

latent flows [197] including AF/SCF and IAF/SCF, since they are the only comparable work with non-autoregressive language modeling. As shown in Table 3.2, the proposed method outperforms all three baselines.

Table 3.2: Character-level language modeling results on Penn Tree Bank in test set bpd.

Dataset	Sparse PC (ours)	Bipartite flow [170]	AF/SCF [197]	IAF/SCF [197]
Penn Tree Bank	**1.35**	1.38	1.46	1.63

3.2.6 Evaluating Pruning and Growing

What is the Smallest PC for the Same Likelihood? We evaluate the ability of pruning based on circuit flows to do effective model compression by iteratively pruning a k-fraction 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 pruning percentage k from $\{0.05, 0.1, 0.3\}$. As shown in Figure 3.8, 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 numbers of latent states (legend), we discover that compressing a large PC to a get smaller PC yields better likelihoods compared to directly training an HCLT with the same number of parameters from scratch. This can be explained by the sparsity of compressed PC structures, as well as a smarter way of finding good parameters: learning a better PC with larger size and compressing it down to a smaller one.

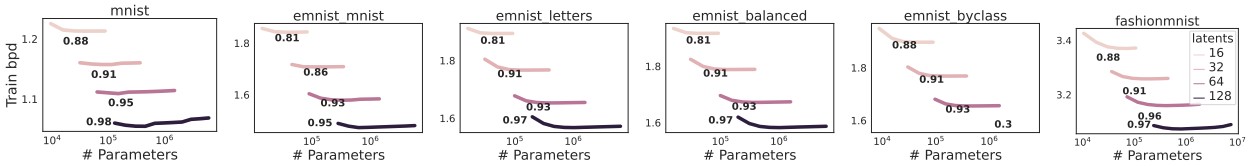


Figure 3.8: Model compression via pruning and finetuning. We report the training set bpd (y-axis) in terms of the number of parameters (x-axis) for different numbers of latent states. For each curve, compression starts from the right (initial PC #Params $|\mathcal{C}^{\texttt{init}}|$) and ends at the 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 for the Same Size? We evaluate structure learning that combines pruning and growing as proposed in Section 3.2.3. Starting from an initial HCLT, we iteratively prune 75% of the parameters, grow again, and fine-tune until meeting the stopping criteria. As shown in Figure 3.9, our method consistently improve the likelihoods of initial PCs for different numbers of latent states among all datasets.

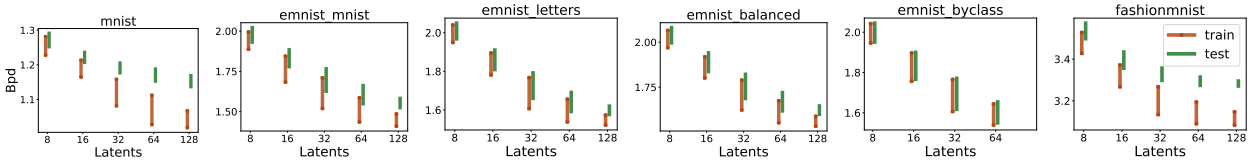


Figure 3.9: Structure learning via 75% pruning, growing and finetuning. We report bpd (y-axis) on both train (red) and test set (green) in terms of the number of latent states (x-axis). For each curve, training starts from the top (large bpd) and ends at the bottom (small bpd).

3.3 Latent Variable Distillation

By leveraging the computation power of modern GPUs, recently developed PC learning frameworks [29, 119, 132] have made it possible to train PCs with over 100M parameters (e.g., [26]). Yet these computational breakthroughs are not leading to the expected large-scale learning breakthroughs: as we scale up PCs, their performance immediately plateaus (dashed curves in Fig. 3.10), even though their actual expressive power should increase monotonically with respect to the number of parameters. Such a phenomenon suggests that the existing optimizers fail to utilize the expressive power provided by large PCs. PCs can be viewed as

latent variable models with a deep hierarchy of latent variables. As we scale them up, size of their latent space increases significantly, rendering the landscale of the marginal likelihood over observed variables highly complex.

We propose to ease this optimization bottleneck by **latent variable distillation** (LVD): we provide extra supervision to PC optimizers by leveraging less-tractable yet more expressive deep generative models to induce semantics-aware assignments to the latent variables of PCs, in addition to the observed variables.

The LVD pipeline consists of two major components: (i) inducing assignments to a subset of (or all) latent variables in a PC by information obtained from deep generative models and (ii) estimating PC parameters given the latent variable assignments. For (i), we focus on a clustering-based approach throughout this paper: we cluster training examples based on their neural embeddings and assign the same values to latent variables for examples in the same cluster; yet, we note that there is no constraint on how we should assign values to latent variables and the methodology may be engineered depending on the nature of the dataset and the architecture of PC and deep generative model. For (ii), to leverage the supervision provided by the latent variable assignments obtained in (i), instead of directly optimizing the maximum-likelihood estimation objective for PC training, we estimate PC parameters by optimizing the its lower-bound shown on the right-hand side:

$$\sum\nolimits_{i=1}^{N} \log p(\boldsymbol{x}^{(i)}) := \sum\nolimits_{i=1}^{N} \log \sum\nolimits_{\boldsymbol{z}} p(\boldsymbol{x}^{(i)}, \boldsymbol{z}) \geq \sum\nolimits_{i=1}^{N} \log p(\boldsymbol{x}^{(i)}, \boldsymbol{z}^{(i)}), \tag{3.3}$$

where $\{\boldsymbol{x}^{(i)}\}_{i=1}^{N}$ is the training set and $\boldsymbol{z}^{(i)}$ is the induced assignments to the latent variables for $\boldsymbol{x}^{(i)}$. After LVD, we continue to finetune PC on the training examples to optimize the actual MLE objective, i.e., $\sum_i \log p(\boldsymbol{x}^{(i)})$.

As shown in Figure 3.10, with LVD, PCs successfully escape the plateau: their performance improves progressively as the number of parameters increases. We highlight two key advantages of LVD: first, it makes much better use of the extra capacity provided by large PCs; second,

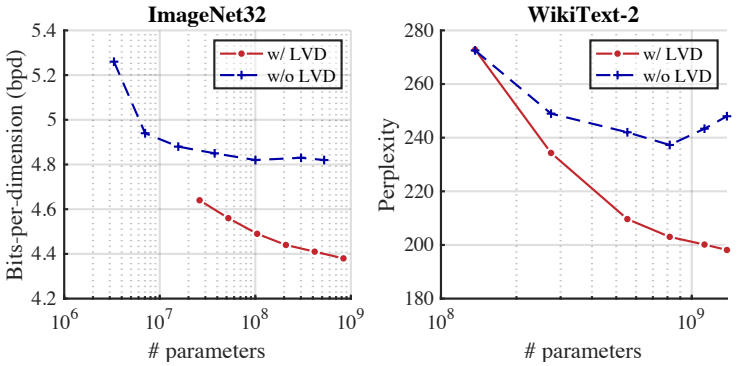


Figure 3.10: Latent variable (LV) distillation significantly boosts PC performance on challenging image (ImageNet32) and language (WikiText-2) modeling datasets. Lower is better.

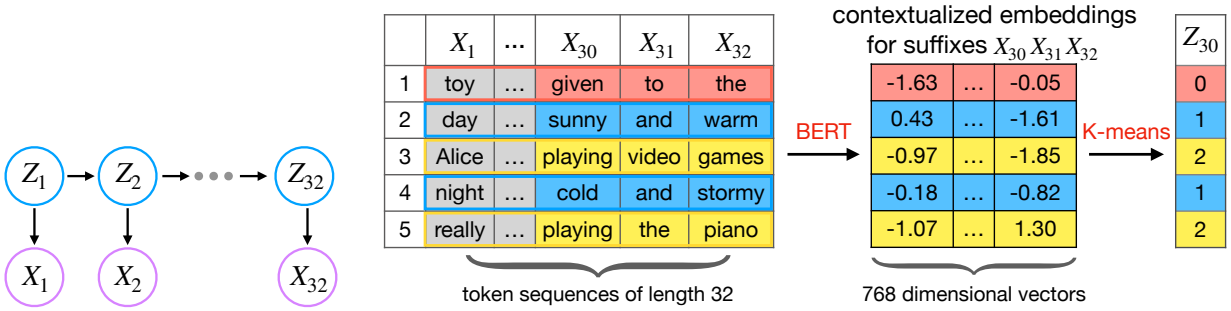
by leveraging the supervision from distilled LV assignments, we can significantly speed up the training pipeline, opening up possibilities to scale up PCs further.

We start by presenting a simple example where we apply LVD on hidden Markov models to improve their performance on language modeling benchmarks (Sec. 3.3.1). Then, we present the general framework of LVD for PCs (Sec. 3.3.2) and how to specialize the pipeline for image modeling (Sec. 3.3.3).

3.3.1 Latent Variable Distillation for Hidden Markov Model

In this section, we consider the task of language modeling by hidden Markov models (HMM) as an illustrating example for LVD. In particular, we demonstrate how we can use the BERT model [40] to induce semantics-aware assignments to the latent variables of HMMs. Experiments on the WikiText-2 [115] dataset show that our approach effectively boosts the performance of HMMs compared to their counterpart trained with only random initialization.

Dataset & Model. The WikiText-2 dataset consists of roughly 2 million tokens extracted from Wikipedia, with a vocabulary size of 33,278. Following prior works on autoregressive language modeling [140], we fix the size of the *context window* to be 32: that is, the HMM model will only be trained on subsequences of length 32 and whenever predicting the next token, the model is only conditioned on the previous 31 tokens. In particular, we adopt a *non-homogeneous* HMM model, that is, its transition and emission probabilities at each



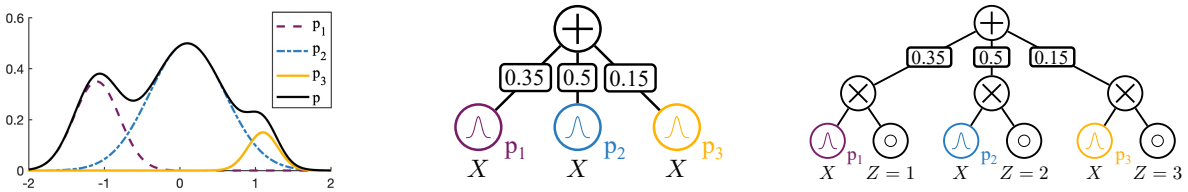
(a) Graphical model representation of an HMM modeling token sequences of length 32. X_i are the observed variables and Z_i are the latent variables.

(b) Pipeline for inferring values for one latent variable Z_{30}. We feed token sequences to the BERT model to obtain contextualized embeddings for their suffixes $X_{30}X_{31}X_{32}$; then we cluster all suffix embeddings into h clusters; here $h = 3$ is the number of hidden states and the value for Z_{30} is set to the cluster id. We repeat this procedure *independently* to infer values for all Z_is.

Figure 3.11: Latent variable distillation pipeline for hidden Markov models.

position share no parameters; Fig. 3.11a shows its representation as a graphical model, where X_is are the observed variables and Z_is are the latent variables. To facilitate training and evaluation, we pre-process the tokens from WikiText-2 by concatenating them into one giant token sequence and collect all subsequences of length 32 to construct the train, validation and test sets, respectively.

Latent Variable Distillation. Let $\mathcal{D} = \{\boldsymbol{x}^{(i)}\}_i$ be the training set; Figure 3.11 shows an example on how to induce, for each training example $\boldsymbol{x}^{(i)}$, its corresponding assignment to the latent variable Z_{30}. We first feed all training examples to the BERT model to compute the contextualized embeddings for their suffixes $X_{30}X_{31}X_{32}$. We cluster all suffix embeddings into h clusters by the K-means algorithm [104], where h is the number of hidden states; then, we set Z_{30} to be the cluster id of their corresponding suffixes, that is, suffixes in the same cluster get the same latent variable value: the intuition is that if the BERT embeddings of some suffixes are close to each other then the suffixes should be relatively similar, suggesting that they should be "generated" by the same hidden state. We repeat this procedure for 32 times to infer the values for all Z_is. Now we obtain an "augmented" training set $\mathcal{D}_{\text{aug}} = \{(\boldsymbol{x}^{(i)}, \boldsymbol{z}^{(i)})\}_i$, where $\boldsymbol{z}^{(i)}$ are the corresponding assignments to the latent variables $\mathbf{Z}$; then, as suggested by Equation 3.3, we maximize the lower-bound $\sum_i \log p(\boldsymbol{x}^{(i)}, \boldsymbol{z}^{(i)})$ for the true MLE objective



(a) A mixture of three Gaussians. (b) A PC that encodes the distribution. (c) An equivalent deterministic PC.

Figure 3.12: A mixture-of-Gaussian distribution (a) and two PCs (b-c) that encode the distribution.

$\sum_i \log p(\boldsymbol{x}^{(i)})$. The parameters of the HMM that maximize $\sum_i \log p(\boldsymbol{x}^{(i)}, \boldsymbol{z}^{(i)})$, denoted by θ^*, can be solved in closed-form. Finally, using θ^* as a starting point, we finetune the HMM model via EM to maximize the true MLE objective $\sum_i \log p(\boldsymbol{x}^{(i)})$.

Experiments. We apply LVD to HMMs with a varying number of hidden states $h = 128$, 256, 512, 750, 1024 and 1250; for comparison, we also train HMMs with random initialization. The plot on the right of Fig. 3.10 shows the test perplexity of HMMs (w/ and w/o LVD) on WikiText-2: as the number of parameters in HMM increases, the performance of the HMMs trained with random parameter initialization immediately plateaus, while the performance of the HMMs trained with LVD progressively improves, suggesting that LVD effectively exploits the express power of the larger models.

3.3.2 Latent Variable Distillation for Probabilistic Circuits

PCs can be viewed as latent variable models with discrete latent spaces [131]. Specifically, since a sum unit in a PC can be viewed as a mixture over its input distributions, it can also be interpreted as a simple latent variable model $\sum_{\boldsymbol{z}} p(\boldsymbol{x}|z)p(z)$, where z decides which input to choose from and the summation enumerates over all inputs. Fig. 3.12 shows such an example, where the sum unit in Fig. 3.12 (b) represents the mixture over Gaussians in Fig. 3.12 (a).

In general, the latent space for large PCs is hierarchical and deeply nested; as we scale them up, we are in effect scaling up the size/complexity of their latent spaces, making it difficult for optimizers to find good local optima. To overcome such bottleneck, we propose

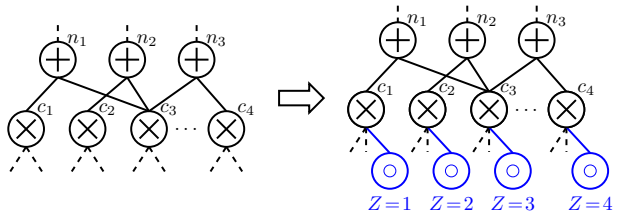


Figure 3.13: Materializing LVs in a PC.

latent variable distillation (LVD). The key intuition for LVD is to provide extra supervision on the latent variables of PCs by leveraging existing deep generative models: given a PC $p(\mathbf{X})$; we view it as a latent variable model $\sum_{\boldsymbol{z}} p(\mathbf{X}, \mathbf{Z} = \boldsymbol{z})$ over some set of latents $\mathbf{Z}$ and assume that for each training example $\boldsymbol{x}^{(i)}$, a deep generative model can always induce some semantics-aware assignment $\mathbf{Z} = \boldsymbol{z}^{(i)}$; then, instead of directly optimizing the MLE objective $\sum_i \log p(\boldsymbol{x}^{(i)})$, we can optimize its lower-bound $\sum_i \log p(\boldsymbol{x}^{(i)}, \boldsymbol{z}^{(i)})$, thus incorporating the guidance provided by the deep generative model. The LVD pipeline consists of three major steps, elaborated in the following:

Step 1: Materializing Latent Variables. The first step of LVD is to *materialize* some/all latent variables in PCs. By materializing latent variables, we can obtain a new PC representing the joint distribution $\Pr(\mathbf{X}, \mathbf{Z})$, where the latent variables $\mathbf{Z}$ are explicit and its marginal distribution $\Pr(\mathbf{X})$ corresponds to the original PC. Although we can assign every sum unit in a PC an unique LV, the semantics of such materialized LVs depend heavily on PC structure and parameters, which makes it extremely hard to obtain supervision. Instead, we choose to materialize LVs based on subsets of observed variables defined by a PC. That is, each materialized LV corresponds to all PC units with a particular variable scope. For example, we can materialize the latent variable Z, which corresponds to the scope $\phi(n_i)$ $(\forall i \in [3])$, to construct the PC in Fig. 3.12(c) that explicitly represents $p(X, Z)$, whose marginal distribution $p(X)$ corresponds to the PC in Fig. 3.12 (b). Algorithm 3 [131] provides one general way to materialize latent variables in PCs, where Fig. 3.13 shows an example where the four product units $c_1, \ldots, c_4$ are augmented with input units $Z = 1, \ldots, Z = 4$, respectively.

Algorithm 3 Materializing a LV in a PC

Input: A PC $p(\mathbf{X})$ and a variable scope $\mathbf{W}$ for some sum unit in $p(\mathbf{X})$
Output: An augmented PC p defined over $\{\mathbf{X}, Z\}$, where Z is the materialized LV corresponding to $\mathbf{W}$
$S_{\mathbf{W}} \leftarrow \{n : n \in p \text{ s.t. } n \text{ is a product unit and } \phi(n) = \mathbf{W}\}$ ▷ Created as an ordered set
for $j = 1$ **to** $|S_{\mathbf{W}}|$ **do**
    Let n_j be the jth unit in $S_{\mathbf{W}}$
    Add an input unit c over Z_i with distribution $p_c(z_i) = \begin{cases} 1 & z_i = j, \\ 0 & \text{otherwise} \end{cases}$ as a new child of n_j

Continuing with our example in Fig. 3.12, note that after materialization, the sum unit representing $p(X, Z)$ in Fig. 3.12(c) is no longer a latent variable distribution: each assignment to X, Z *uniquely* determines the input distribution to choose, where the other inputs give zero probability under this assignment; we say that this sum unit is *deterministic* [34] (as defined in Definition 5).

Determinism characterizes whether a sum unit introduces latent variables: by materializing some sum units with the scope, we enforce them to become deterministic. Intuitively, more deterministic sum units in PCs implies smaller latent spaces, which implies easier optimization; in fact, if all sum units in a PC are deterministic then the MLE solution can be computed in closed-form [82]. By materializing more latent variables, we make PCs "more deterministic", pushing the optimization procedure towards a closed-form estimation.

Step 2: Inducing Latent Variable Assignments. Latent variable materialization itself cannot provide any extra supervision to the PC training pipeline; in addition, we also need to leverage some existing deep generative models to induce semantics-aware assignments for the materialized latent variables. Though there is no general guideline on how the assignments should be induced, we focus on a clustering-based approach throughout this paper. Recall from Section 3.3.1, where we cluster the suffix embeddings generated by the BERT model and for each training example, we assign the latents the cluster id that its suffixes belong to. Similarly, for image modeling, in Section 3.3.3, we will show how to induce latent variable assignments by clustering the embeddings for patches of images. The main take-away is that the method for inducing latent variable assignments should be engineered depending on the nature of the dataset and the architecture of PC and deep generative model.

Step 3: PC Parameter Learning. Given a PC $p(\mathbf{X};\theta)$ with parameters θ and a training set $\mathcal{D} = \{\boldsymbol{x}^{(i)}\}$; in Step 1, by materializing some set of latent variables $\mathbf{Z}$, we obtain an augmented PC $p_{\text{aug}}(\mathbf{X},\mathbf{Z};\theta)$ whose marginal distribution on $\mathbf{X}$ corresponds to $p(\mathbf{X};\theta)$; in Step 2, by leveraging some deep generative model $\mathcal{G}$, we obtain an augmented training set $\mathcal{D}_{\text{aug}} = \{(\boldsymbol{x}^{(i)}, \boldsymbol{z}^{(i)})\}$. Note that since p_{aug} and p share the same parameter space, we can optimize $\sum_{i=1}^{N} \log p_{\text{aug}}(\boldsymbol{x}^{(i)}, \boldsymbol{z}^{(i)};\theta)$ as a lower-bound for $\sum_{i=1}^{N} \log p(\boldsymbol{x}^{(i)};\theta)$:

$$\sum\nolimits_{i=1}^{N} \log p(\boldsymbol{x}^{(i)};\theta) = \sum\nolimits_{i=1}^{N} \log \sum\nolimits_{\boldsymbol{z}} p_{\text{aug}}(\boldsymbol{x}^{(i)}, \boldsymbol{z};\theta) \geq \sum\nolimits_{i=1}^{N} \log p_{\text{aug}}(\boldsymbol{x}^{(i)}, \boldsymbol{z}^{(i)};\theta);$$

we denote the parameters for p_{aug} after optimization by θ^*. Finally, we initialize p with θ^* and optimize the true MLE objective with respect to the original dataset $\mathcal{D}$, $\sum_{i=1}^{N} \log p(\boldsymbol{x}^{(i)};\theta)$.

Summary. Here we summarize the general pipeline for latent variable distillation. Assume that we are given: a PC $p(\mathbf{X};\theta)$ over observed variables $\mathbf{X}$ with parameter θ, a training set $\mathcal{D} = \{\boldsymbol{x}^{(i)}\}$ and a deep generative model $\mathcal{G}$:

1. Construct a PC $p_{\text{aug}}(\mathbf{X},\mathbf{Z};\theta)$ by materializing a subset of latent variables $\mathbf{Z}$ in $p(\mathbf{X};\theta)$; note that p and p_{aug} share the same parameter space.

2. Use $\mathcal{G}$ to induce semantics-aware latent variable assignments $\boldsymbol{z}^{(i)}$ for each training example $\boldsymbol{x}^{(i)}$; denote the augmented dataset as $\mathcal{D}_{\text{aug}} = \{\boldsymbol{x}^{(i)}, \boldsymbol{z}^{(i)}\}$.

3. Optimize the log-likelihood of p_{aug} with respect to $\mathcal{D}_{\text{aug}}$, i.e., $\sum_i \log p_{\text{aug}}(\boldsymbol{x}^{(i)}, \boldsymbol{z}^{(i)};\theta)$; denote the parameters for p_{aug} after optimization as θ^*.

4. Initialize $p(\mathbf{X},\theta)$ with θ^* and then optimize the log-likelihood of p with respect to the original dataset $\mathcal{D}$, i.e., $\sum_i \log p(\boldsymbol{x}^{(i)};\theta)$.

Efficient parameter learning

Another major obstacle for scaling up PCs is training efficiency. Specifically, despite recently developed packages [29, 119] and training pipelines [132] that leverage the computation power

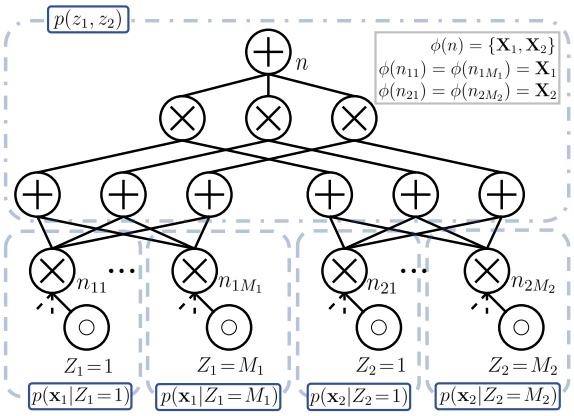


Figure 3.14: Distribution decomposition of an example PC with materialized LVs Z_1, Z_2.

of modern GPUs, training large PCs is still extremely time-consuming. For example, in our experiments, training a PC with ∼500M parameters on CIFAR (using existing optimizers) would take around one GPU day to converge. With the efficient parameter learning algorithm detailed in the following, training such a PC takes around 10 GPU hours.

The most computationally expensive part in LVD is to optimize the MLE lower bound (Eq. 3.3) with regard to the observed data and inferred LVs, which requires feeding all training samples through the whole PC. By exploiting the additional conditional independence assumptions introduced by the materialized LVs, we show that the computation cost of this optimization process can be significantly reduced. To gain some intuition, consider applying LVD to the PC in Fig. 3.12(c) with materialized LV Z. For a sample x whose latent assignment z is 1, since the Gaussian distributions p_2 and p_3 are independent with this sample, we only need to feed it to the input unit corresponds to p_1 in order to estimate its parameters. To formalize this efficient LVD algorithm, we start by introducing the conditional independence assumptions provided by the materialized LVs.

Lemma 1. *For a PC $p(\mathbf{X})$, denote $\mathbf{W}$ as the scope of some units in p. Assume the variable scope of every PC unit is either a subset of $\mathbf{W}$ or disjoint with $\mathbf{W}$. Let Z be the LV corresponds to $\mathbf{W}$ created by Algorithm 3. Then variables $\mathbf{W}$ are conditional independent of $\mathbf{X}\backslash\mathbf{W}$ given Z.*

Take Fig. 3.13 as an example. Define the scope of $\{n_i\}_{i=1}^3$ and $\{c_i\}_{i=1}^4$ as $\mathbf{W}$ and the corresponding LV as Z; denote the scope of the full PC as $\mathbf{X}$. Lemma 1 implies that variables $\mathbf{W}$ and $\mathbf{X}\backslash\mathbf{W}$ are conditional independent given Z.

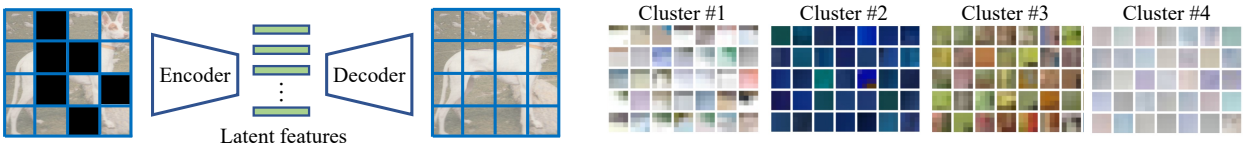
We consider a simple yet effective strategy for materializing LVs: the set of observed variables $\mathbf{X}$ is partitioned into k disjoint subsets $\{\mathbf{X}_i\}_{i=1}^k$; then for each $\mathbf{X}_i$, we use Algorithm 3 to construct a corresponding LV, termed Z_i. As a direct corollary of Lemma 1, the joint probability over $\mathbf{X}$ and $\mathbf{Z}$ can be decomposed as follows: $p(\boldsymbol{x}, \boldsymbol{z}) = p(\boldsymbol{z}) \prod_{i=1}^k p(\boldsymbol{x}_i | z_i)$.

The key to speed up LVD is the observation that the MLE lower bound objective (Eq. 3.3) can be factored into independent components following the decomposition of $p(\boldsymbol{x}, \boldsymbol{z})$:

$$\texttt{LL}(p, \mathcal{D}_{\text{aug}}) := \sum_{l=1}^{N} \log p(\boldsymbol{x}^{(l)}, \boldsymbol{z}^{(l)}) = \sum_{l=1}^{N} \sum_{i=1}^{k} \log p(\boldsymbol{x}_i^{(l)} | z_i^{(l)}) + \sum_{l=1}^{N} \log p(\boldsymbol{z}^{(l)}), \tag{3.4}$$

where $\mathcal{D}_{\text{aug}} := \{(\boldsymbol{x}^{(l)}, \boldsymbol{z}^{(l)})\}_{l=1}^N$ is the training set augmented with LV assignments. According to Eq. (3.4), optimize $\texttt{LL}(p, \mathcal{D}_{\text{aug}})$ is equivalent to performing MLE on the factorized distributions separately. Specifically, we decompose the optimization process into the following independent steps: (i) for each cluster i and category j, optimizing PC parameters w.r.t. the distribution $p(\mathbf{X}_i | Z_i = j)$ using the subset of training samples whose LV z_i is assigned to category j, and (ii) optimizing the sub-PC corresponds to $p(\boldsymbol{z})$ using the set of all LV assignments. Consider the example PC shown in Fig. 3.14. The subset of PC surrounded by every blue box encodes the distribution labeled on its edge. To maximize $\texttt{LL}(p, \mathcal{D}_{\text{aug}})$, we can separately train the sub-PCs correspond to the decomposed distributions, respectively. Compared to feeding training samples to the whole PC, the above procedure trains every latent-conditioned distribution $p(\mathbf{X}_i | Z_i = j)$ using only samples that have the corresponding LV assignment (i.e., $z_i = j$), which significantly reduces computation cost.

Recall that in the LVD pipeline, after training the PC parameters by maximizing $\texttt{LL}(p, \mathcal{D}_{\text{aug}})$, we still need to finetune the model on the original dataset $\mathcal{D}$. However, this finetuning step often suffers from slow convergence speed, which significantly slows down the



(a) Illustration of the Masked Autoencoder model.

(b) Example of image patches belonging to the same cluster.

Figure 3.15: Extracting LVs for image data. The MAE model (a) is used to extract categorical LVs $\{Z_i\}_{i=1}^k$ that correspond to image patches $\{\mathbf{X}_i\}_{i=1}^k$, respectively. (b) provides example patches from the training set that belong to four randomly chosen clusters of the LV Z_1.

learning process. To mitigate this problem, we add an additional *latent distribution training* step where we only finetune parameters corresponding to $p(\mathbf{Z})$. In this way, we only need to propagate training samples through the sub-PCs corresponding to the latent-conditioned distributions once. After this step converges, we move on to finetune the whole model, which then takes much fewer epochs to converge.

3.3.3 Extracting Latent Variables for Image Modeling

This section discusses how to induce assignments to LVs using expressive generative models. While the answer is specific to individual data types, we propose preliminary answers to the question in the context of image data. We highlight that there are many possible LV selection strategies and target generative models; the following method is only an example that shows the effectiveness of LVD.

Motivated by recent advances in image-based deep generative models [47, 102], we model images by two levels of hierarchy — the low-level models independently encode distribution of every image patch, and the top-level model represents the correlation between different patches. Formally, we define $\mathbf{X}_i$ as the variables in the ith $M \times M$ patch of an $H \times W$ image (w.l.o.g. assume H and W are both divisible by M). Therefore, the image $\mathbf{X}$ is divided into $k = H \cdot W/M^2$ subsets $\{\mathbf{X}_i\}_{i=1}^k$. Every Z_i is defined as the LV corresponds to patch $\mathbf{X}_i$.

Recall that our goal is to obtain the assignment of $\{Z_i\}_{i=1}^k$, each as a concise representation of $\{\mathbf{X}_i\}_{i=1}^k$, respectively. Despite various possible model choices, we choose to use Masked Autoencoders (MAEs) [61] as they produce good features for image patches. Specifically,

as shown in Fig. 3.15(a), MAE consists of an encoder and a decoder. During training, a randomly selected subset of patches are fed to the encoder to generate a latent representation for every patch. The features are then fed to the decoder to reconstruct the full image. The simplest way to compute latent features for every patch is to feed them into the encoder independently, and extract the corresponding features. However, we find that it is beneficial to also input other patches as context. Specifically, we first compute the latent features without context. We then compute the correlation between features of all pair of patches and construct the Maximum Spanning Tree (MST) using the pairwise correlations. Finally, to compute the feature of each patch $\mathbf{X}_i$, we additionally input patches correspond to its ancestors in the MST. Further details are given in Section B.2.2.

As shown in Section 3.3.2, LVs $\{Z_i\}_{i=1}^{k}$ are required to be categorical. To achieve this, we run the K-means algorithm on the latent features (of all training examples) and use the resultant cluster indices as the LV assignments. Fig. 3.15(b) shows some example image patches $\boldsymbol{x}_1$ belonging to four latent clusters (i.e., $Z_1 = 1, \ldots, 4$). Clearly, the LVs capture the semantics of different image patches.

To illustrate the effectiveness of LVD, we make minimum structural changes compared to Hidden Chow-Liu Trees (HCLTs) [97], a competitive PC structure. Specifically, we use the HCLT structure for all sub-PCs $\{p(\boldsymbol{x}_i|Z_i=j)\}_{i,j}$ and $p(\boldsymbol{z})$. This allows us to materialize patch-based LVs while keeping the model architecture similar to HCLTs.

3.3.4 Experiments

In this section, we evaluate the proposed latent variable distillation (LVD) technique on three natural image benchmarks, i.e., CIFAR [85] and two versions of down-sampled ImageNet (ImageNet32 and ImageNet64) [38]. On all benchmarks, we demonstrate the effectiveness of LVD from two perspectives. First, compared to PCs trained by existing EM-based optimizers, the proposed technique offers a significant performance gain especially on large PCs. Second, PCs trained by LVD achieve competitive performance against some of the less tractable deep

Table 3.3: Density estimation performance of Tractable Probabilistic Models (TPMs) and Deep Generative Models (DGMs) on three natural image datasets. Reported numbers are test set bit-per-dimension (bpd). Bold indicates best bpd (smaller is better) among all four TPMs.

Dataset	TPMs				DGMs		
	LVD (ours)	HCLT	EiNet	RAT-SPN	Glow	RealNVP	BIVA
ImageNet32	**4.39**$\pm$0.01	4.82	5.63	6.90	4.09	4.28	3.96
ImageNet64	**4.12**$\pm$0.00	4.67	5.69	6.82	3.81	3.98	-
CIFAR	**4.38**$\pm$0.02	4.61	5.81	6.95	3.35	3.49	3.08

generative models, including variational autoencoders and flow-based models.

Baselines We compare the proposed method against three TPM baselines: Hidden Chow-Liu Tree (HCLT) [97], Einsum Network (EiNet) [132], and Random Sum-Product Network (RAT-SPN) [133]. Though not exhausive, this baseline suite embodies many of the recent advancement in tractable probabilistic modeling, and can be deemed as the existing SoTA. To evaluate the performance gap with less tractable deep generative models, we additionally compare LVD with the following flow-based and VAE models: Glow [79], RealNVP [46], and BIVA [109].

To facilitate a fair comparison with the chosen TPM baselines, we implement both HCLT and RAT-SPN using the Julia package Juice.jl [29] and tune hyperparameters such as batch size, learning rate and its schedule. We use the original PyTorch implementation of EiNet and similarly tune their hyperparameters. For all TPMs, we train various models with number of parameters ranging from $\sim$1M to $\sim$100M, and report the number of the model with the best performance. For deep generative model baselines, we adopt the numbers reported in the respective original papers.

Empirical Insights We first compare the performance of the four TPM approaches. As shown in Fig. 3.16, for all three benchmarks, PCs trained by LVD are consistently better than the competitors by a large margin. In particular, on ImageNet32, a $\sim$25M PC trained by LVD is better than a HCLT with $\sim$400M parameters. Next, looking at individual curves,

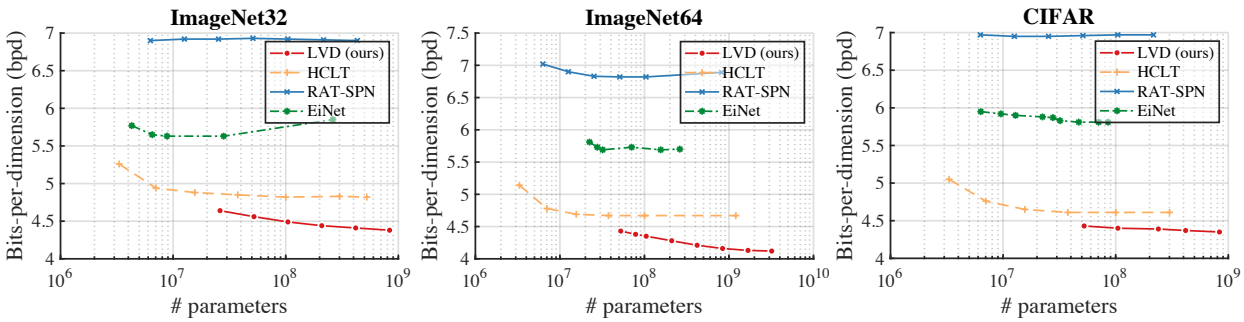


Figure 3.16: Generative modeling performance of four TPMs on three natural image datasets. For each method, we report the test set bits-per-dimension (y-axis) in terms of the number of parameters (x-axis) for different numbers of latent states.

we observe that with LVD, the test set bpd keeps decreasing as the model size increases. This indicates that LVD is able to take advantage of the extra capacity offered by large PCs. In contrast, PCs trained by EM immediately suffer from a performance bottleneck as the model size increases. Additionally, the efficient LVD learning pipeline allows us to train PCs with 500M parameters in 10 hours with a single NVIDIA A5000 GPU, while existing optimizers need over 1 day to train baseline PCs with similar sizes.

We move on to compare the performance of LVD with the three adopted DGM baselines. As shown in Table 3.3, although the performance gap is relatively large on CIFAR, the performance of LVD is highly competitive on ImageNet32 and ImageNet64, with bpd gap ranging from $\sim$0.1 to $\sim$0.3. We hypothesize that the relatively large performance gap on CIFAR is caused by insufficient training samples. Specifically, since the sub-PCs correspond to the latent-conditioned distributions $\{p(\boldsymbol{x}_i|Z_i=j)\}_{i,j}$ are constructed independently, and thus every training sample $\boldsymbol{x}_i$ can only be used to train its corresponding latent-conditioned distribution, making the model extremely data-hungry. However, we note that this is not an inherent problem of LVD. For example, by performing parameter tying of sub-PCs correspond to different image patches, we can significantly improve sample complexity of the model. This is left to future work.

Chapter 4

Scalable Learning of Probabilistic Circuits – Systems Side

In this chapter, we develop an efficient yet flexible system termed PyJuice that covers various training and inference tasks of PCs. As shown in Table 4.1, PyJuice is orders of magnitude faster than previous implementations for PCs (e.g., SPFlow [119], EiNet [132], and Juice.jl [29]) as well as Hidden Markov Models[1] (e.g., Dynamax [122]). Additionally, as we shall demonstrate in the experiments, PyJuice is more memory efficient than the baselines, enabling us to train much larger PCs with a fixed memory quota.

Unlike other deep generative models based on neural network layers that are readily amenable to efficient systems (e.g., a fully connected layer can be emulated by a single matrix multiplication and addition kernel plus an element-wise activation kernel), PCs cannot be *efficiently* computed using well-established operands due to (i) the unique connection patterns of their computation graph, and (ii) the existence of log probabilities at drastically different scales in the models, which requires to properly handle numerical underflow problems. To parallelize PCs at scale, we propose a compilation phase that converts a PC into a compact

The contents of this chapter appeared in paper [93].

[1]Every HMM has an equivalent PC representation.

Table 4.1: **Average (± stdev of 5 runs) runtime (in seconds) per epoch** of 60K samples for PyJuice and the baselines SPFlow [119], EiNet [132], Juice.jl [29], and Dynamax [122]. Using four PC structures: PD, RAT-SPN, HCLT, and HMM. All experiments ran on an RTX 4090 GPU with 24GB memory. To maximize parallelism, we always use the maximum possible batch size. "OOM" denotes out-of-memory with batch size 2. The best numbers are in boldface.

	PD [136]				
# nodes	172K	344K	688K	1.38M	2.06M
# edges	15.6M	56.3M	213M	829M	2.03B
SPFlow	>25000	>25000	>25000	>25000	>25000
EiNet	34.2$_{\pm 0.0}$	88.7$_{\pm 0.2}$	456.1$_{\pm 2.3}$	1534.7$_{\pm 0.5}$	OOM
Juice.jl	12.6$_{\pm 0.5}$	37.0$_{\pm 1.7}$	141.7$_{\pm 6.9}$	OOM	OOM
PyJuice	**2.0**$_{\pm 0.0}$	**5.3**$_{\pm 0.0}$	**15.4**$_{\pm 0.0}$	**57.1**$_{\pm 0.2}$	**203.7**$_{\pm 0.1}$

	RAT-SPN [133]				
# nodes	58K	116K	232K	465K	930K
# edges	616K	2.2M	8.6M	33.4M	132M
SPFlow	6372.1$_{\pm 4.2}$	>25000	>25000	>25000	>25000
EiNets	38.5$_{\pm 0.0}$	83.5$_{\pm 0.0}$	193.5$_{\pm 0.1}$	500.6$_{\pm 0.2}$	2445.1$_{\pm 2.6}$
Juice.jl	6.0$_{\pm 0.3}$	9.4$_{\pm 0.3}$	25.5$_{\pm 2.4}$	84.0$_{\pm 4.0}$	375.1$_{\pm 3.4}$
PyJuice	**0.6**$_{\pm 0.0}$	**0.9**$_{\pm 0.1}$	**1.6**$_{\pm 0.0}$	**5.8**$_{\pm 0.1}$	**13.8**$_{\pm 0.0}$

	HCLT [97]				
# nodes	89K	178K	355K	710K	1.42M
# edges	2.56M	10.1M	39.9M	159M	633M
SPFlow	22955.6$_{\pm 18.4}$	>25000	>25000	>25000	>25000
EiNet	52.5$_{\pm 0.3}$	77.4$_{\pm 0.4}$	233.5$_{\pm 2.8}$	1170.7$_{\pm 8.9}$	5654.3$_{\pm 17.4}$
Juice.jl	4.7$_{\pm 0.2}$	6.4$_{\pm 0.5}$	12.4$_{\pm 1.3}$	41.1$_{\pm 0.1}$	143.2$_{\pm 5.1}$
PyJuice	**0.8**$_{\pm 0.0}$	**1.3**$_{\pm 0.0}$	**2.6**$_{\pm 0.0}$	**8.8**$_{\pm 0.0}$	**24.9**$_{\pm 0.1}$

	HMM [139]				
# nodes	33K	66K	130K	259K	388K
# edges	8.16M	32.6M	130M	520M	1.17B
Dynamax	111.3$_{\pm 0.4}$	441.2$_{\pm 3.9}$	934.7$_{\pm 6.3}$	2130.5$_{\pm 19.5}$	4039.8$_{\pm 38.3}$
Juice.jl	4.6$_{\pm 0.1}$	18.8$_{\pm 0.1}$	91.6$_{\pm 0.1}$	OOM	OOM
PyJuice	**0.6**$_{\pm 0.0}$	**1.0**$_{\pm 0.0}$	**2.9**$_{\pm 0.1}$	**10.1**$_{\pm 0.2}$	**39.9**$_{\pm 0.1}$

data structure amenable to block-based parallelization on modern GPUs. Further, we improve the backpropagation process by indirectly computing the parameter updates by backpropagating a quantity called PC flow [17] that is more numerically convenient yet mathematically equivalent.

In the following, we first discuss common ways to parallelize PCs' computation in Section 4.1. Section 4.2 examines the key bottlenecks in PC parallelization. Section 4.3 and 4.4 explains our design in details. The open-sourced implementation can be found at https://github.com/Tractables/pyjuice.

4.1 Related Work on Accelerating PCs

There has been a great amount of effort put into speeding up training and inference for PCs. One of the initial attempts performs node-based computations on both CPUs [107] and GPUs [119,138], i.e., by computing the outputs for a mini-batch of inputs (data) recursively for every node. Despite its simplicity, it fails to fully exploit the parallel computation capability possessed by modern GPUs since it can only parallelize over a batch of samples. This problem

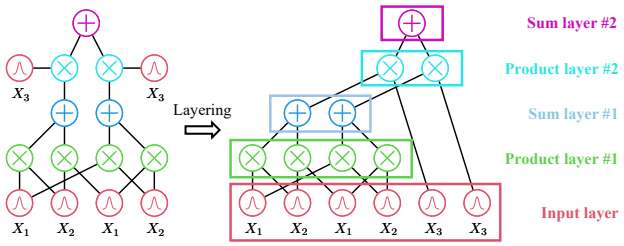


Figure 4.1: Layering a PC by grouping nodes with the same topological depth (as indicated by the colors) into disjoint subsets. Both the forward and the backward computation can be carried out independently on nodes within the same layer.

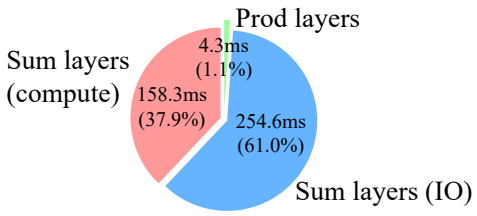
is mitigated by also parallelizing topologically independent nodes [29, 132]. Specifically, a PC is chunked into topological layers, where nodes in the same layer can be computed in parallel. This leads to 1-2 orders of magnitude speedup compared to node-based computation.

The regularity of edge connection patterns is another key factor influencing the design choices. Specifically, EiNets [132] leverage off-the-shelf Einsum operations to parallelize dense PCs where every layer contains groups of densely connected sum and product/input nodes. [111] generalize the notion of dense PCs to tensorized PCs, which greatly expands the scope of EiNets. [29] instead focus on speeding up sparse PCs, where different nodes could have drastically different numbers of edges. They use custom CUDA kernels to balance the workload of different GPU threads and achieve decent speedup on both sparse and dense PCs.

Another thread of work focuses on designing computation hardware that is more suitable for PCs. Specifically, [154] propose DAG Processing Units (DPUs) that can efficiently traverse sparse PCs, [28] introduce an indirect read reorder-buffer to improve the efficiency of data-dependent memory accesses in PCs, and [188] use addition-as-int multiplications to significantly improve the energy efficiency of PC inference algorithms.

4.2 Key Bottlenecks in PC Parallelization

This section aims to lay out the key bottlenecks to efficient PC implementations. For ease of illustration, we focus solely on the forward pass, and leave the unique challenges posed by



Summary of the PC structure

	Sum layers	Prod layers
# nodes	200K	815K
# edges	154M	1.81M
# params	154M	-

Figure 4.2: Runtime breakdown of the feedforward pass of a PC with ∼150M edges. Both the IO and the computation overhead of the sum layers are significantly larger than the total runtime of product layers. Detailed configurations of the PC are shown in the table.

the backward pass and their solution to Section 4.4.

We start by illustrating the layering procedure deployed for PCs. Starting from the input nodes, we perform a topological sort of all the nodes, clustering nodes with the same topological depth into a layer. For example, in Fig. 4.1, the PC on the left side is transformed into an equivalent layered representation on the right, where nodes of the same color belong to the same layer. The forward pass proceeds by sequentially processing each layer, and finally returns the root node's output. To avoid underflow, all probabilities are stored in the logarithm space. Therefore, product layers just need to sum up the corresponding input log-probabilities, while sum layers compute weighted sums of input log-probabilities utilizing the logsumexp trick.

Assume for now that all nodes in every layer have the same number of children. A straightforward strategy is to parallelize over every node and every sample. Specifically, given a layer of size M and batch size B, we need to compute in total $M \times B$ output values, which are evenly distributed to all processors (e.g., thread-blocks in GPUs). We apply this idea to a PC with the PD structure [136]. The PC has ∼1M nodes and ∼150M edges. Additionally, all nodes within a layer have the same number of children, making it an ideal testbed for the aforementioned algorithm.

Fig. 4.2 illustrates the runtime breakdown of the forward pass (with batch size 512). As shown in the pie chart, both the IO and the computation overhead of the sum layers are much larger than that of the product layers. We would expect sum layers to exhibit a higher

computation overhead due to (i) the number of sum edges being ∼85x more than the product edges (see the table in Fig. 4.2), and (ii) sum edges requiring more compute compared to product edges. However, we would not expect the gap in IO overhead to be as pronounced as indicated in the pie chart. Specifically, with batch size 512, the ideal memory read count of product layers should be roughly $[\text{batch size}] \times [\#\text{sum nodes}] \approx 102\text{M}$ since all children of product nodes are sum or input nodes (the number of input nodes is an order of magnitude smaller and is omitted). Similarly, the number of memory reads required by the sum layers is approximately $[\text{batch size}] \times [\#\text{prod nodes}] + [\#\text{parameters}] \approx 571\text{M}$, which is only 5.6x compared to the product layers. The ideal memory write count of product layers should be larger since there are about 4x more product nodes compared to sum nodes.

While the ideal IO overhead of the sum layers is not much larger than that of the product layers, the drastic difference in runtime (over 50x) can be explained by the significant amount of reloads of child nodes' probabilities in the sum layers. Specifically, in the adopted PD structure, every sum node has no more than 12 parents, while most product nodes have 256 parents.[2] Recall that the parents of product nodes are sum nodes and vice versa. As a result, each sum layer needs to reload the output of every product node multiple times. Although this does not lead to 256x loads from the GPU's High-Bandwidth Memory (HBM) thanks to its caching mechanism, such excessive IO access still significantly slows down the algorithm.

The fundamental principle guiding our design is to *properly group, or allocate, sum edges to different processors to minimize the reloading of product nodes' outputs*. As an added benefit, this allows us to interpret part of the core computation as matrix multiplications, allowing us to harness Tensor Cores available in modern GPUs and resulting in a significant reduction in sum layers' computational overhead.

[2]Only the children of the root sum node have 1 parent.

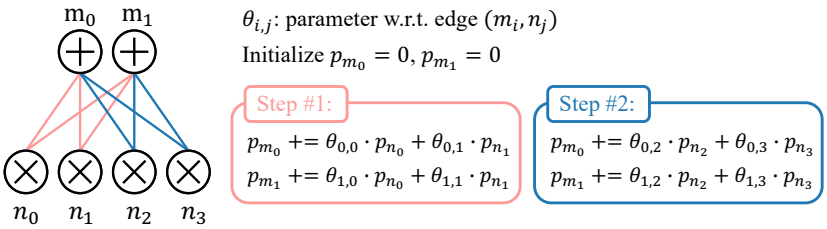


Figure 4.3: Illustration of block-based parallelization. A processor computes the output of 2 sum nodes, by iterating through blocks of 2 input product nodes and accumulating partial results.

4.3 Harnessing Block-Based PC Parallelization

This section takes gradual steps towards demonstrating how we can reduce both the IO and computation overhead using block-based parallelization. Specifically, we first utilize a fully connected sum layer to sketch the high-level idea (Sec. 4.3.1). Consequently, we move on to the general case, providing further details of the algorithm (Secs. 4.3.2, 4.3.3).

4.3.1 Fully Connected Sum Layers

Consider a fully connected sum layer comprised of M sum nodes, each connected to the same set of N product nodes as inputs. Under the parallelization strategy mentioned in Section 4.2, with a single sample, we have M processors, each computing the output of a sum node. Since the layer is fully connected, every processor loads all N input log-probabilities, which results in M reloads of every input.

The key to reducing excessive IO overhead is by parallelizing over blocks of nodes/edges. Specifically, we divide the M sum nodes into blocks of K_M nodes and the N product nodes into blocks of K_N nodes. We assume without loss of generality that M and N are divisible by K_M and K_N, respectively. Instead of independently computing the output of every sum node, we calculate the K_M outputs of a sum node block in a single processor. To achieve this, we iterate through every product node block to compute and accumulate the partial results from the $K_M \times K_N$ edges between the corresponding sum node block and product node block.

In every step, the processor loads a block of $\boldsymbol{\theta} \in \mathbb{R}^{K_M \times K_N}$ parameters and a vector of

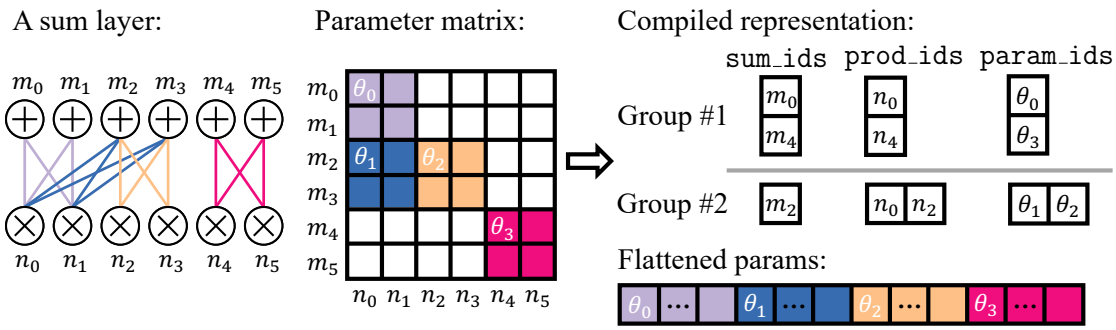


Figure 4.4: A sum layer (left) with a block-sparse parameter matrix (middle) is compiled into two kernels (right) each with a balanced workload. During execution, each kernel uses the compiled sum/prod/param indices to compute the outputs of $m_0, \ldots, m_5$.

$\boldsymbol{p}_{\text{prod}} \in \mathbb{R}^{K_N}$ input probabilities, where we (temporarily) omit the fact that all probabilities are stored in the logarithm space. The partial outputs $\boldsymbol{p}_{\text{sum}} \in \mathbb{R}^{K_M}$ are computed via a matrix-vector multiplication between $\boldsymbol{\theta}$ and $\boldsymbol{p}_{\text{prod}}$. Note that if we add a second "batch" dimension to $\boldsymbol{p}_{\text{prod}}$ and $\boldsymbol{p}_{\text{sum}}$, the computation immediately becomes a matrix-matrix multiplication, which can be computed efficiently using GPU Tensor Cores.

For example, in Fig. 4.3, define $K_M = K_N = 2$, we compute the output of m_0 and m_1 by first calculating the weighted sum w.r.t. the input probability of n_0 and n_1 in step #1, and then accumulate the probabilities coming from n_2 and n_3 in step #2. With the new parallelization strategy, every processor that computes K_M output values needs to load every input probability only once, and the number of reloads is reduced from M to M/K_M.

4.3.2 Generalizing To Practical Sum Layers

Most sum layers in practical PCs are not fully connected. However, as we shall demonstrate, they can still harness the advantages of block-based parallelization. Specifically, consider a sum layer with M sum nodes and N product nodes as inputs. Following Section 4.3.1, we partition the sum and the product nodes into blocks of K_M and K_N nodes, respectively. For every pair of sum and product node blocks, if it is either fully connected (i.e., featuring $K_M \times K_N$ edges) or unconnected (i.e., no edge between them), we call the layer block-sparse. In the following, we focus on efficiently parallelizing block-sparse PCs (whose sum layers all

exhibit block-sparsity).

As an example, the layer illustrated in Fig. 4.4(left) exhibits block sparsity with block sizes $K_M = K_N = 2$. This is evident as each pair of sum and product node blocks is either fully connected (e.g., $\{m_2, m_3\}$ and $\{n_0, n_1\}$) or disjoint (e.g., $\{m_4, m_5\}$ and $\{n_2, n_3\}$). In Fig. 4.4(middle), this pattern is more discernible in the parameter matrix, where *aligned* 2×2 blocks display either all non-zero parameters (indicated by the colors) or all zero parameters.

Similar to the procedure outlined in Section 4.3.1, computing the outputs of a block of K_M sum nodes involves iterating through all its connected product node blocks. This introduces two additional problems: (i) how to efficiently index the set of connected product node blocks, which may vary for each sum node block; (ii) different sum node blocks could connect to different numbers of product node blocks, which causes an imbalanced workload among processors. For instance, consider the layer in Fig. 4.4. The first issue is exemplified by the two sum node blocks $\{m_0, m_1\}$ and $\{m_4, m_5\}$, both of which possess a single child node block, albeit different ones. The second issue is illustrated by the node block $\{m_2, m_3\}$, which connects to two child node blocks, while the others connect to only one.

4.3.3 Efficient Implementations by Compiling PC Layers

We address both problems through a compilation process, where we assign every node an index, and precompute index tensors that enable efficient block-based parallelization. The first step is to partition the sum node blocks into groups, such that every node block within a group has a similar number of connected child node blocks. We then pad the children with pseudo-product node blocks with probability 0 such that all sum node blocks in a group have the same number of children. The partition is generated by a dynamic programming algorithm that aims to divide the layer into the smallest possible number of groups while ensuring that the fraction of added pseudo-node blocks does not exceed a pre-defined threshold.

We move on to construct the index tensors for each group. In addition to assigning every node an index, we create a vector $\boldsymbol{\theta}_{\text{flat}}$, a concatentation of all the PC parameters.

For every sum node block in a group with C_N child node blocks, we record (i) the starting index of the sum node block, (ii) the set of initial indices of its C_N child node blocks, and (iii) the corresponding set of C_N parameter indices. These parameter indices each denote the starting point for the $K_M \times K_N$ parameters of the corresponding pair of sum and product node blocks. Let C_M represent the total number of node blocks in the group. Following the indices described above, we record the following tensors: $\texttt{sum\_ids} \in \mathbb{R}^{C_M}$ containing indices of all sum node blocks; $\texttt{prod\_ids}, \texttt{param\_ids} \in \mathbb{R}^{C_M \times C_N}$, whose ith row represent the child indices and parameter indices of the ith sum node block (i.e., the node block with the start index $\texttt{sum\_ids}[i]$), respectively.

Fig. 4.4(right) illustrates the compiled index tensors of the sum layer shown on the left. Recall that we use the block sizes $K_M = K_N = 2$. The layer is then divided into two groups: the first group including two sum node blocks, $\{m_0, m_1\}$ and $\{m_4, m_5\}$, each having one child node block, and the second group including one sum node block, $\{m_2, m_3\}$, which has two child node blocks. Take, for instance, the first group. `sum_ids` stores the start indices (i.e., m_0 and m_4) of the two sum node blocks. `prod_ids` stores the initial indices of the child node blocks (i.e., n_0 and n_4) of the two sum node blocks, respectively. `param_ids` encodes the corresponding initial parameter indices θ_0 and θ_2.

Partitioning a layer into groups with the same number of children allows us to achieve different parallelization strategies for different groups, where the tradeoff between nodes, edges, and samples is different, and the GPU thread blocks need to be allocated differently in these dimensions for better performance (e.g., by ensuring high utilization).

For every group in a sum layer, the three index tensors serve as inputs to a CUDA kernel computing the log-probabilities of the sum nodes in the group. Define $\boldsymbol{l}_{\text{prod}} \in \mathbb{R}^{N \times B}$ and $\boldsymbol{l}_{\text{sum}} \in \mathbb{R}^{M \times B}$ (B is the batch size) as the set of input and output log-probabilities, respectively. Consider a group with C_M sum node blocks and C_N child node blocks per sum node block. Algorithm 4 computes the log-probabilities of the C_M sum node blocks and stores the results in the proper locations in $\boldsymbol{l}_{\text{sum}}$. Specifically, we also divide the B samples into blocks of size

Algorithm 4 Forward pass of a sum layer group

1: **Inputs:** log-probs of product nodes $\boldsymbol{l}_{\text{prod}}$, flattened parameter vector $\boldsymbol{\theta}_{\text{flat}}$, `sum_ids`, `prod_ids`, `param_ids`
2: **Inputs:** # sum nodes: M, # product nodes: N, batch size: B
3: **Inputs:** block sizes K_M, K_N, K_B for the sum node, product node, and batch dimensions, respectively
4: **Inputs:** number of sum node blocks C_M; number of product node blocks C_N; number of batch blocks C_B
5: **Outputs:** log-probs of sum nodes $\boldsymbol{l}_{\text{sum}}$
6: **Kernel launch:** schedule to launch $C_M \times C_B$ thread-blocks with $\texttt{m}=0,\dots,C_M-1$ and $\texttt{b}=0,\dots,C_B-1$
7: $\texttt{cum} \leftarrow (-\infty)_{K_M \times K_B} \in \mathbb{R}^{K_M \times K_B}$ ▷ Scratch space on SRAM
8: $\texttt{bs}, \texttt{be} \leftarrow \texttt{b} \cdot K_B, (\texttt{b}+1) \cdot K_B$ ▷ Start and end batch index
9: **for** $\texttt{n} = 0$ **to** $C_N - 1$ **do**
10: $\quad \texttt{ps}, \texttt{ns} \leftarrow \texttt{param\_ids}[\texttt{m}, \texttt{n}], \texttt{prod\_ids}[\texttt{n}, \texttt{b}]$
11: $\quad$ Load $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_{\text{flat}}[\texttt{ps}:\texttt{ps} + K_M \cdot K_N].\texttt{view}(K_M, K_N)$ to SRAM
12: $\quad$ Load $\boldsymbol{l} \leftarrow \boldsymbol{l}_{\text{prod}}[\texttt{ns}:\texttt{ns} + K_N, \texttt{bs}:\texttt{be}] \in \mathbb{R}^{K_N \times K_B}$ to SRAM
13: $\quad \boldsymbol{l}_{\max} \leftarrow \texttt{max}(\boldsymbol{l}, \texttt{dim}=0) \in \mathbb{R}^{1 \times K_B}$ ▷ Compute on chip
14: $\quad \boldsymbol{p}_{\text{p}} \leftarrow \texttt{exp}(\boldsymbol{l} - \boldsymbol{l}_{\max}) \in \mathbb{R}^{K_N \times K_B}$
15: $\quad \boldsymbol{p}_{\text{s}} \leftarrow \texttt{matmul}(\boldsymbol{\theta}, \boldsymbol{p}_{\text{p}}) \in \mathbb{R}^{K_M \times K_B}$ ▷ With Tensor Cores
16: $\quad \texttt{cum} \leftarrow \texttt{where}(\boldsymbol{l}_{\max} > \texttt{cum},$
$\quad\quad \texttt{log}(\boldsymbol{p}_{\text{s}} + \texttt{exp}(\texttt{cum} - \boldsymbol{l}_{\max}) + \boldsymbol{l}_{\max},$
$\quad\quad \texttt{log}(\texttt{exp}(\boldsymbol{l}_{\max} - \texttt{cum}) \cdot \boldsymbol{p}_{\text{s}} + 1) + \texttt{cum})$
17: $\boldsymbol{l}_{\text{sum}}[\texttt{ms}:\texttt{ms} + K_M, \texttt{bs}:\texttt{be}] \leftarrow \texttt{acc}$ (where $\texttt{ms} \leftarrow \texttt{sum\_ids}[\texttt{m}]$)

K_B, leading to $C_B := B/K_B$ blocks (assume w.l.o.g. that B is divisible by K_B). Algorithm 4 schedules to launch $C_M \times C_B$ thread-blocks, each responsible for computing $K_M \times K_B$ outputs (line 6). The main loop in line 9 iterates over all C_N child node blocks. In every step, we first load the corresponding parameter matrix $\boldsymbol{\theta} \in \mathbb{R}^{K_M \times K_N}$ (line 11) and input matrix $\boldsymbol{l} \in \mathbb{R}^{K_N \times K_B}$ (line 12). Since $\boldsymbol{l}$ contains log-probabilities, we apply a variant of the logsumexp trick: we first convert $\boldsymbol{l}$ to the arithmetic space by subtracting the per-sample maximum log-probability (lines 13-14), then compute the (partial) output probabilities from the current set of $K_M \times K_N$ edges via matrix multiplication (line 15), and in line 16 aggregate the results back to the accumulator `cum` defined in line 7. Finally, we store the log-probabilities to the target locations in $\boldsymbol{l}_{\text{sum}}$ (line 17).

4.3.4 Analysis: IO and Computation Overhead

We analyze the efficiency and IO complexity of our block-based parallelization strategy. Specifically, we benchmark on the largest sum layer in the PD structure adopted in Section 4.2. The layer consists of 29K nodes and 30M edges. In addition to the computation time, we

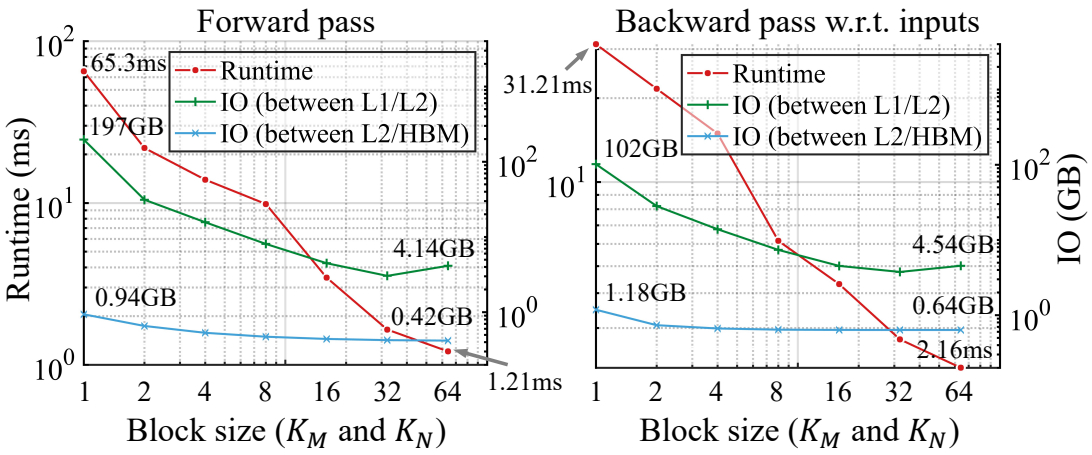


Figure 4.5: Runtime and IO overhead of a sum layer from the PD structure (with 29K nodes and 30M edges). The results demonstrate significant performance gains from our block-based parallelization, even with small block sizes.

record two types of IO overhead: (i) the IO between the L1/texture cache and the L2 cache, and (ii) the reads/writes between the L2 cache and the GPU High-Bandwidth Memory (HBM). We vary the block sizes K_M and K_N exponentially from 1 to 64. To ensure a fair comparison, we implement a dedicated kernel for $K_M = K_N = 1$, which directly parallelizes over sum node/sample pairs, allowing for better workload allocation. For other block sizes, we adjust K_B and other kernel launching hyperparameters (e.g., warps per block) and report the best runtime for every case. Results of the backward pass (w.r.t. inputs) are also reported for completeness.

Results are shown in Fig. 4.5. As the block size increases, both the forward and the backward pass become significantly faster. Notably, this is accompanied by a significant drop in IO overhead. Specifically, with a large block size, the kernel consumes 2x fewer reads/writes between the L2 cache and the HBM, and 25-50x fewer IO between the L1 and L2 cache. This corroborates the hypothesis stated in Section 4.2 that the extensive value reloads significantly slow down the computation.

Additionally, we note that even with small block sizes (e.g., 2 or 4), the speedup is quite significant compared to the baseline case ($K_M = K_N = 1$), which allows us to speed

up *sparse* PCs. Specifically, with the observation that every sparse PC can be viewed as a block-sparse PC with block size 1, we can transform a sparse PC into a block-sparse one, and pad zero parameters to edges belonging to the block-sparse PC but not the sparse PC. For PCs with relatively regular sparsity patterns, the speedup obtained by having a larger block size outpaces the overhead caused by padded edges with zero parameters, which leads to speed-ups.

4.4 Optimizing Backpropagation with PC Flows

The previous section focuses on speeding up sum layers by reducing excessive memory reloads and leveraging Tensor Cores. However, when it comes to backpropagation, directly adapting Algorithm 4 by differentiating lines 13-16 would lead to poor performance due to the following. First, we need to either store some intermediate values (e.g., $\boldsymbol{l}_{\max}$ and $\boldsymbol{p}_{\mathrm{p}}$) in the forward pass or recompute them in the backward pass. Next, since different thread-blocks could access the same product node log-probabilities in line 12, they both need to write (partial) gradients of it, which introduces inter-thread-block barriers that slow down the execution.

We overcome the problems by leveraging PC flows [17], which is only a factor of $\theta_{n,c}$ away from the desired gradients ($\partial \log p_{n_r}(\boldsymbol{x})/\partial\theta_{n,c}$). PC flows exhibit a straightforward recursive definition, facilitating a seamless transformation into an efficient implementation for the backward pass.

Definition 8 (PC flows)**.** For a PC $p_{n_r}(\mathbf{X})$ rooted at node n_r and a sample $\boldsymbol{x}$, the flow $\mathrm{F}_n(\boldsymbol{x})$ of every node n is defined recursively as follows:

$$\mathrm{F}_n(\boldsymbol{x}) := \begin{cases} 1 & n \text{ is the root node,} \\ \sum_{m \in \mathsf{pa}(n)} \mathrm{F}_m(\boldsymbol{x}) & n \text{ is input or sum,} \\ \sum_{m \in \mathsf{pa}(n)} \frac{\theta_{m,n} \cdot p_n(\boldsymbol{x})}{p_m(\boldsymbol{x})} \cdot \mathrm{F}_m(\boldsymbol{x}) & n \text{ is a product node,} \end{cases}$$

where $\mathsf{pa}(n)$ is the set of parents of n. Similarly, the edge flow $\mathrm{F}_{n,c}(\boldsymbol{x})$ w.r.t. the sample $\boldsymbol{x}$

$(c \in \mathsf{ch}(n))$ is defined as

$$\mathrm{F}_{n,c}(\boldsymbol{x}) := \theta_{n,c} \cdot p_c(\boldsymbol{x})/p_n(\boldsymbol{x}) \cdot \mathrm{F}_n(\boldsymbol{x}).$$

While similar results have been established in a slightly different context [132], we show the following equations for completeness:

$$\mathrm{F}_n(\boldsymbol{x}) = \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial \log p_n(\boldsymbol{x})} \text{ and } \mathrm{F}_{n,c}(\boldsymbol{x}) = \theta_{n,c} \cdot \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial \theta_{n,c}}.$$

Following Definition 8, we can compute $F_n(\boldsymbol{x})$ for every node n utilizing the same set of layers created for the feedforward pass. Specifically, we first set the flow of the root node to 1 following its definition. We then iterate through the layers in reverse order (i.e., parent layers before child layers). While processing a layer, all flows of the nodes in the layer are computed by the preceding layers. And our goal is to compute the (partial) flows of the child nodes of the layer. Similar to the forward pass, we compile every layer by grouping child node blocks with a similar number of parents, and use block-based parallelization to reduce reloads of parent log-probabilities.

Another important design choice that leads to a significant reduction in memory footprint is to recompute the product nodes' probabilities in the backward pass instead of storing them all in the GPU memory during the forward pass. Specifically, we maintain a scratch space on GPU HBM that can hold the results of the largest product layer. All product layers write their outputs to this same scratch space, and the required product node probabilities are re-computed when requested by a sum layer during backpropagation. Since product layers are extremely fast to evaluate compared to the sum layers (e.g., see the runtime breakdown in Fig. 4.2), this leads to significant memory savings at the cost of slightly increased computation time.

Table 4.2: Average (± standard deviation of 3 runs) runtime (in seconds) of the compilation process of four PCs.

Structure	HMM	PD	HCLT	RAT-SPN
# nodes	130K	1.38M	710K	465K
# edges	130M	829M	159M	33.4M
Compilation time (s)	$1.50_{\pm 0.02}$	$30.57_{\pm 0.86}$	$8.70_{\pm 0.32}$	$4.72_{\pm 0.16}$

4.5 Experiments

We evaluate the impact of using PyJuice to train PCs. In Section 4.5.1, we compare PyJuice against existing implementations regarding time and memory efficiency. Specifically, to demonstrate its generality and flexibility, we evaluate PyJuice on four commonly used dense PC structures as well as highly unstructured and sparse PCs. Next, we demonstrate that PyJuice can be readily used to scale up PCs for various downstream applications in Section 4.5.2. Finally, in Section 4.5.3, we benchmark existing PCs on high-resolution image datasets, hoping to incentivize future research to develop better PC structures as well as learning algorithms.

4.5.1 Faster Models with PyJuice

We first benchmark the runtime of PyJuice on four commonly used PC structures: PD [136], RAT-SPN [133], HCLT [97], and HMM [139]. For all models, we record the runtime to process 60,000 samples (including the forward pass, the backward pass, and mini-batch EM updates). We vary their structural hyperparameters and create five PCs for every structure with sizes (i.e., number of edges) ranging from 500K to 2B. We compare against four baselines: SPFlow [119], EiNet [132], Juice.jl [29], and Dynamax [122]. Dynamax is dedicated to State Space Models so it is only used to run HMMs; SPFlow and EiNet are excluded in the HMM results because we are unable to construct homogeneous HMMs with their frameworks due to the need to share the transition and emission parameters at different time steps. All experiments are carried out on an RTX 4090 GPU with 24GB memory.

Table 4.1 reports the runtime in seconds per epoch with mini-batch EMs. PyJuice is

orders of magnitude faster than all baselines in both small and large PCs. Further, we observe that most baselines exhaust 24GB of memory for larger PCs (indicated by "OOM" in the table), while PyJuice can still efficiently train these models. Additionally, in Table 4.2, we show the efficiency of the compilation process. For example, it takes only $\sim$8.7s to compile an HCLT with 159M edges. Note that we only compile the PC once and then reuse the compiled structure for training and inference.

In Fig. 4.6, we take two PCs to show the GPU memory consumption with different batch sizes. The results demonstrate that PyJuice is more memory efficient than the baselines, especially in the case of large batch sizes (note that we always need a constant-size space to store the parameters).

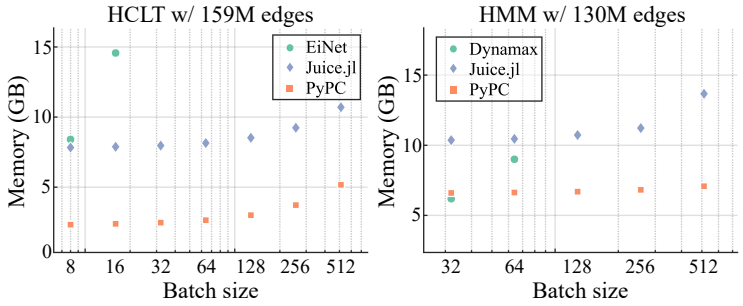


Figure 4.6: Comparison on memory efficiency. We take two PCs (i.e., an HCLT w/ 159M edges and an HMM w/ 130M edges) and record GPU memory usage under different block sizes.

We move on to benchmark PyJuice on block-sparse PCs. We create a sum layer with 209M edges (see Appx. C.3.1 for details). We partition the sum and input product nodes in the layer into blocks of 32 nodes respectively. We randomly discard blocks of 32×32 edges, resulting in block-sparse layers. As shown in Fig. 4.7, as the fraction of the removed edge block increases, the runtime of both the forward and the backward pass decreases significantly. This motivates future work on PC modeling to focus on designing effective block-sparse PCs.

Finally, we proceed to evaluate the runtime of sparse PCs. We adopt the PC pruning algorithm proposed by [31] to prune two HCLTs with 10M and 40M edges, respectively. We only compare against Juice.jl since all other implementations do not support sparse PCs.

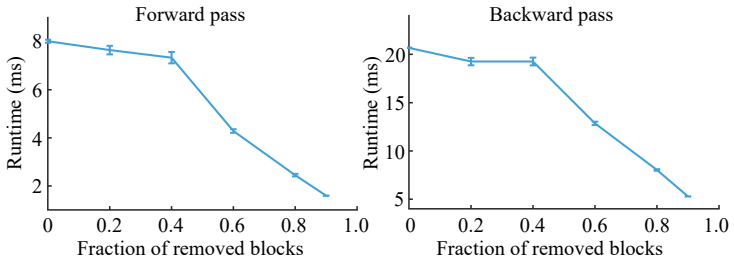


Figure 4.7: Runtime of a block-sparse sum layer as the function of the fraction of kept (non-dropped) edge blocks.

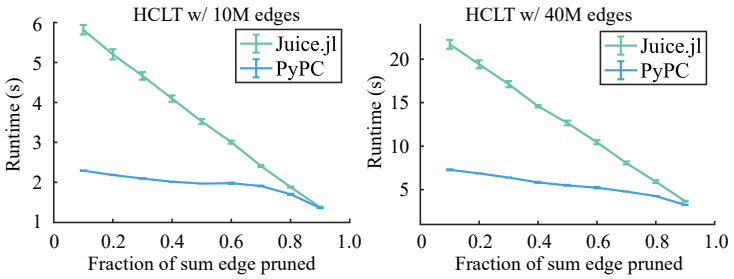


Figure 4.8: Runtime per epoch (with 60K samples) of two sparse HCLTs with different fractions of pruned edges.

As shown in Fig. 4.8, PyJuice is consistently faster than Juice.jl, despite the diminishing gap when over 90% edges are pruned. Note that with sparse PCs, PyJuice cannot fully benefit from the block-based parallelization strategy described in Section 4.3, yet it can still successfully exploit the sparsity.

4.5.2 Better PCs At Scale

This section demonstrates the ability of PyJuice to improve the state of the art by simply using larger PCs and training for more epochs thanks to its speed and memory efficiency. Specifically, we take the HMM language model proposed by [192] and the image model introduced by [100] as two examples.

HMM language models. [192] use the Latent Variable Distillation (LVD) [99] technique to train an HMM with 4096 hidden states on sequences of 32 word tokens. Specifically, LVD is used to obtain a set of "good" initial parameters for the HMM from deep generative

models. The HMM language model is then fine-tuned on the CommonGen dataset [92], and is subsequently used to control the generation process of (large) language models for constrained generation tasks. Following the same procedure, we use PyJuice to fine-tune two HMMs with hidden sizes 4096 and 8192, respectively.

Table 4.3: Perplexity of HMM language models trained on the CommonGen benchmark [92].

	Zhang et al. [192]	PyJuice	
# hidden states	4096	4096	8192
Perplexity	9.78	8.81	**8.65**

As shown in Table 4.3, by using the same HMM with 4096 hidden states, PyJuice improved the perplexity by $\sim$1.0 by running many more epochs in less time compared to the original model. We also train a larger HMM with 8192 hidden states and further improved the perplexity by a further 0.16.

Sparse Image Models. [100] design a PC learning algorithm that targets image data by separately training two sets of PCs: a set of sparse patch-level PCs (e.g., 4×4 patches) and a top-level PC that aggregates outputs of the patch-level PC. In the final training step, the PCs are supposed to be assembled and jointly fine-tuned. However, due to the huge memory consumption of the PC (with over 10M nodes), only the top-level model is fine-tuned in the original paper. With PyJuice, we can fit the entire model in 24GB of memory and fine-tune the entire model. For the PC trained on the ImageNet32 dataset [38], this fine-tuning step leads to an improvement from 4.06 to 4.04 bits-per-dimension.

4.5.3 Benchmarking Existing PCs

We use PyJuice to benchmark the performance of the PD and the HCLT structure on three natural image datasets: ImageNet [38] and its down-sampled version ImageNet32, and CelebA-HQ [103].[3] For all three datasets, we train the PCs on randomly sampled 16×16 patches, which results in a total of $16\times16\times3=768$ categorical variables each with $2^8=256$

[3]Code is available at https://github.com/liuanji/pyjuice-benchmarks

Table 4.4: Density estimation performance of PCs on three natural image datasets. Reported numbers are test set bits-per-dimension.

Dataset	PD-mid	PD-large	HCLT-mid	HCLT-large
ImageNet32	5.22	5.20	4.36	4.33
ImageNet	4.98	4.95	3.57	3.53
CelebA-HQ	4.35	4.29	2.43	2.38

possible values. As a preprocessing step, the image patches are converted losslessly into the YCoCg color space since it is observed that such color space transformations lead to improved density estimation performance.

We adopt two PD structures (i.e., PD-mid with 107M edges and PD-large with 405M edges) as well as two HCLT structures (i.e., HCLT-mid with 40M edges and HCLT-large with 174M edges). We experiment with different optimization strategies and adopt full-batch EM as it yields consistently better performance across models and datasets. Specifically, the computed PC flows are accumulated across all samples in the training set before doing one EM step.

Results are shown in Table 4.4. Notably, we achieve *better* results compared to previous papers. For example, [99] reports 4.82 bits-per-dimension (bpd) for HCLT on ImageNet32, while we achieved 4.33 bpd. The performance improvements stem from more training epochs and the ability to do more hyperparameter searches thanks to the speedup. We highlight that the goal of this section is not to set new records for tractable deep generative models, but to establish a set of baselines that can be easily reproduced to track the progress of developments in PC modeling and learning.

Chapter 5

Applications

Do generative models with enhanced reasoning capabilities perform better on downstream tasks? Leveraging the modeling-side improvements described in Chapter 4, this chapter gives three examples (i.e., controlled image generation, lossless data compression, and offline Reinforcement Learning) on how PCs can be used to achieve better performance compared to other deep generative models in reasoning-demanding tasks.

5.1 Image Inpainting via Tractable Steering of Diffusion Models

In this section, we explore the possibility of using PCs to help control the sampling process for constrained image generation tasks such as inpainting. In the following, we first provide the background of solving controlled image generation tasks with diffusion models as well as the motivation of using PCs to steer the sampling process. We then describe the proposed method in detail. Finally, we empirically evaluate the effectiveness of the proposed method.

5.1.1 Background and Motivation

Thanks to their expressiveness, diffusion models have achieved state-of-the-art results in generating photorealistic and high-resolution images [124, 142, 146]. However, steering unconditioned diffusion models toward constrained generation tasks such as image inpainting remains challenging, as diffusion models do not by design support efficient computation of the posterior sample distribution under many types of constraints [23]. This results in samples that fail to properly align with the constraints. For example, in image inpainting, the model may generate samples that are semantically incoherent with the given pixels.

Prior works approach this problem mainly by approximating the (constrained) posterior sample distribution. However, due to the intractable nature of diffusion models, such approaches introduce high bias [23, 108, 191] to the sampling process, which diminishes the benefit of using highly-expressive diffusion models.

Having observed that the lack of tractability hinders us from fully exploiting diffusion models in constrained generation tasks, we study the converse problem: *what is the benefit of models that by design support efficient constrained generation?* We present positive evidence by showing that PCs can efficiently steer the denoising process of diffusion models towards

The contents of this section appeared in paper [96].

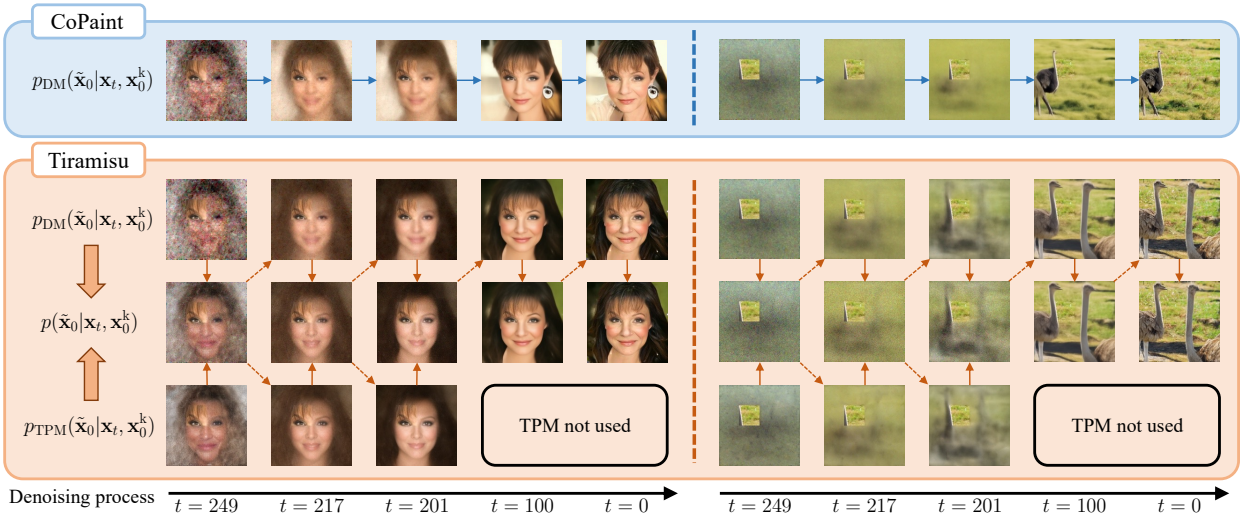


Figure 5.1: Illustration of the steering effect of the TPM on the diffusion model. The same random seed is used by the baseline (CoPaint; [191]) and our approach. At every time step, given the image at the previous noise level, Tiramisu reconstructs $\tilde{\boldsymbol{x}}_0$ with both the diffusion model and the TPM, and combines the two distributions by taking their geometric mean (solid arrows). The images then go through the noising process to generate the input for the previous time step (dashed arrows).

high-quality inpainted images. We will define a class of constraints that includes inpainting constraints for which we can provide the following guarantee. For any constraint c in this class, given a sample $\boldsymbol{x}_t$ at noise level t, we show that a PC trained on noise-free samples (i.e., $p(\mathbf{X}_0)$) can be used to efficiently compute $p(\boldsymbol{x}_0|\boldsymbol{x}_t, c)$, which is a key step in the sampling process of diffusion models. This PC-computed distribution can then be used to effectively guide the denoising process, leading to photorealistic images that adhere to the constraints. Fig. 5.1 illustrates the steering effect of PCs in the proposed algorithm **Tiramisu** (**Tra**ctable **Im**age **I**npainting via **S**teering Diff**u**sion Models). Specifically, we plot the reconstructed image by the diffusion model (the first row of Tiramisu) and the PC (the third row) at five time steps during the denoising process. Compared to the baselines, Tiramisu generates more semantically coherent images with the PC-provided guidance. In summary, there are three main contributions:

Denoising Diffusion Probabilistic Models A diffusion model [63, 163] defined on variables $\mathbf{X}_0$ is a latent variable model of the form $p_\theta(\boldsymbol{x}_0) := \int p_\theta(\boldsymbol{x}_{0:T}) d\boldsymbol{x}_{1:T}$, where $\boldsymbol{x}_{1:T}$ are the latent variables and the joint distribution $p_\theta(\boldsymbol{x}_{0:T})$ is defined as a Markov chain termed the reverse/denoise process:

$$p_\theta(\boldsymbol{x}_{0:T}) := p(\boldsymbol{x}_T) \cdot \prod_{t=1}^{T} p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t). \tag{5.1}$$

For continuous variables $\boldsymbol{x}_{0:T}$, the initial and transition probabilities of the Markov chain typically use Gaussian distributions:

$$p(\boldsymbol{x}_T) := \mathcal{N}(\boldsymbol{x}_T; \mathbf{0}, \mathbf{I}), p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t) := \mathcal{N}(\boldsymbol{x}_{t-1}; \boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t), \boldsymbol{\Sigma}_\theta(\boldsymbol{x}_t, t)),$$

where $\boldsymbol{\mu}_\theta$ and $\boldsymbol{\Sigma}_\theta$ are the mean and covariance parameters, respectively. The key property that distinguishes diffusion models from other latent variable models such as hierarchical Variational Autoencoders [173] is the fact that they have a prespecified approximate posterior $q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0) := \prod_{t=1}^{T} q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})$. This is called the forward or diffusion process. For continuous variables, the transition probabilities are also defined as Gaussians: $q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1}) := \mathcal{N}(\boldsymbol{x}_t; \sqrt{1-\beta_t}\boldsymbol{x}_{t-1}, \beta_t \mathbf{I})$, where $\{\beta_t\}_{t=1}^{T}$ is a noise schedule. Training is done by maximizing the ELBO of $p_\theta(\boldsymbol{x}_0)$ with the variational posterior $q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)$. See [78] for more details.

While it is possible to directly model $p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ with a neural network, prior works discovered that the following parameterization leads to better empirical performance [63]:

$$p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t) := \sum\nolimits_{\tilde{\boldsymbol{x}}_0} q(\boldsymbol{x}_{t-1}|\tilde{\boldsymbol{x}}_0, \boldsymbol{x}_t) \cdot p_\theta(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t), \tag{5.2}$$

where $p_\theta(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t) := \mathcal{N}(\tilde{\boldsymbol{x}}_0; \tilde{\boldsymbol{\mu}}_\theta(\boldsymbol{x}_t, t), \tilde{\boldsymbol{\Sigma}}_\theta(\boldsymbol{x}_t, t))$ is parameterized by a neural network and $q(\boldsymbol{x}_{t-1}|\tilde{\boldsymbol{x}}_0, \boldsymbol{x}_t)$ has a simple closed-form expression [63]. Following the definition of the denoising process, sampling from a diffusion model boils down to first sampling from $p(\boldsymbol{x}_T)$

and then recursively sampling $\boldsymbol{x}_{T-1}, \ldots, \boldsymbol{x}_0$ according to $p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$.

5.1.2 Guiding Diffusion Models with Tractable Probabilistic Models

Given a diffusion model trained for unconditional generation, our goal is to steer the model to generate samples given different conditions/constraints without the need for task-specific fine-tuning. In the following, we focus on the image inpainting task to demonstrate that TPMs can guide diffusion models toward more coherent samples that satisfy the constraints.

The goal of image inpainting is to predict the missing pixels given the known pixels. Define $\mathbf{X}_0^{\mathrm{k}}$ (resp. $\mathbf{X}_0^{\mathrm{u}}$) as the provided (resp. missing) pixels. We aim to enforce the inpainting constraint $\mathbf{X}_0^{\mathrm{k}} = \boldsymbol{x}_0^{\mathrm{k}}$ on every denoising step $p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ ($\forall t \in 1, \ldots, T$). Plugging in Eq. (5.2), the conditional probabilities are written as:

$$\forall t \in 1, \ldots, T \quad p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t, \boldsymbol{x}_0^{\mathrm{k}}) = \sum\nolimits_{\tilde{\boldsymbol{x}}_0} q(\boldsymbol{x}_{t-1}|\tilde{\boldsymbol{x}}_0, \boldsymbol{x}_t) \cdot p_\theta(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t, \boldsymbol{x}_0^{\mathrm{k}}),$$

where the first term on the right-hand side is independent of $\boldsymbol{x}_0^{\mathrm{k}}$. To sample coherent inpainted images, we need to draw unbiased samples from $p_\theta(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t, \boldsymbol{x}_0^{\mathrm{k}})$. Applying Bayes' rule, we get

$$p_\theta(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t, \boldsymbol{x}_0^{\mathrm{k}}) = \frac{1}{Z} \cdot p_\theta(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t) \cdot p(\boldsymbol{x}_0^{\mathrm{k}}|\tilde{\boldsymbol{x}}_0) = \frac{1}{Z} \cdot p_\theta(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t) \cdot \mathbb{1}[\tilde{\boldsymbol{x}}_0^{\mathrm{k}} = \boldsymbol{x}_0^{\mathrm{k}}], \tag{5.3}$$

where $\mathbb{1}[\cdot]$ is the indicator function and Z is a normalizing constant. One simple strategy to sample from Eq. (5.3) is by rejection sampling: sample unconditionally from $p_\theta(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t)$ (i.e., the learned denoising model) and reject samples with $\tilde{\boldsymbol{x}}_0^{\mathrm{k}} \neq \boldsymbol{x}_0^{\mathrm{k}}$. However, this is impractical since the acceptance rate could be extremely low. Existing algorithms use approximation strategies to compute or sample from Eq. (5.3). For example, [108] proposes to set $\tilde{\boldsymbol{x}}_0^{\mathrm{k}} := \boldsymbol{x}_0^{\mathrm{k}}$ after sampling $\tilde{\boldsymbol{x}}_0 \sim p_\theta(\cdot|\boldsymbol{x}_t)$, leaving $\tilde{\boldsymbol{x}}_0^{\mathrm{u}}$ untouched; [191] and [23] relax the hard inpainting constraint to $\min_{\tilde{\boldsymbol{x}}_0^{\mathrm{k}}} \|\tilde{\boldsymbol{x}}_0^{\mathrm{k}} - \boldsymbol{x}_0^{\mathrm{k}}\|_2^2$ and use gradient-based methods to gradually enforce it.

This paper explores the possibility of drawing unbiased samples from $p_\theta(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t, \boldsymbol{x}_0^{\mathrm{k}})$ given

a TPM-represented distribution $p_\theta(\boldsymbol{x}_0)$. By applying Bayes' rule from the other side, we have

$$p_\theta(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t, \boldsymbol{x}_0^{\mathrm{k}}) = \frac{1}{Z} \cdot q(\boldsymbol{x}_t|\tilde{\boldsymbol{x}}_0) \cdot p_\theta(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_0^{\mathrm{k}}) = \frac{1}{Z} \cdot \prod_i q(x_t^i|\tilde{x}_0^i) \cdot p_\theta(\tilde{\boldsymbol{x}}_0^{\mathrm{u}}|\boldsymbol{x}_0^{\mathrm{k}}) \cdot \mathbb{1}[\tilde{\boldsymbol{x}}_0^{\mathrm{k}} = \boldsymbol{x}_0^{\mathrm{k}}], \quad (5.4)$$

where Z is a normalizing constant, x_t^i is the ith variable in $\boldsymbol{x}_t$, and the factorization of $q(\boldsymbol{x}_t|\boldsymbol{x}_0)$ follows the definition of the diffusion process in Section 5.1.1. Although the right-hand side seems to be impractical to compute due to the normalizing constant, we will demonstrate in the following sections that there exists a class of expressive TPMs that can compute it efficiently and exactly.

From the diffusion model and the TPM, we have obtained two versions of the same distribution $p(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t, \boldsymbol{x}_0^{\mathrm{k}})$.[1] Thanks to the expressiveness of neural networks, the distribution approximated by the diffusion model (i.e., Eq. (5.3); termed $p_{\mathrm{DM}}(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t, \boldsymbol{x}_0^{\mathrm{k}})$) encodes high-fidelity images. However, due to the inability to compute the exact conditional probability, $p_{\mathrm{DM}}(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t, \boldsymbol{x}_0^{\mathrm{k}})$ could lead to images incoherent with the constraint [191]. In contrast, the TPM-generated distribution (i.e., Eq. (5.4); termed $p_{\mathrm{TPM}}(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t, \boldsymbol{x}_0^{\mathrm{k}})$) represents images that potentially better align with the given pixels. Therefore, $p_{\mathrm{DM}}(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t, \boldsymbol{x}_0^{\mathrm{k}})$ and $p_{\mathrm{TPM}}(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t, \boldsymbol{x}_0^{\mathrm{k}})$ can be viewed as distributions trained for the same task yet with different biases. Following prior arts [57, 192], we combine both distributions by taking the weighted average of the logits of every variable in $\tilde{\boldsymbol{x}}_0$, hoping to get images that are both semantically coherent and have high fidelity:

$$p(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t, \boldsymbol{x}_0^{\mathrm{k}}) \propto p_{\mathrm{DM}}(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t, \boldsymbol{x}_0^{\mathrm{k}})^\alpha \cdot p_{\mathrm{TPM}}(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t, \boldsymbol{x}_0^{\mathrm{k}})^{1-\alpha}, \quad (5.5)$$

where $\alpha \in (0, 1)$ is a mixing hyperparameter. In summary, as a key step of image inpainting with diffusion models, we compute $p(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t, \boldsymbol{x}_0^{\mathrm{k}})$ from both the diffusion model and a TPM, and use their weighted geometric mean in the denoising process. We note that the use of TPMs is independent of the design choices related to the diffusion model, and thus can be

[1] Note that diffusion models can only approximate this distribution.

built upon any prior approach.

5.1.3 Practical Implementation with Probabilistic Circuits

The previous section introduces how TPMs could help guide the denoising process of diffusion models toward high-quality inpainted images. While promising, a key question is *whether* $p_{\text{TPM}}(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t, \boldsymbol{x}_0^{\text{k}})$ *(Eq. 5.4) can be computed efficiently and exactly?* We answer the question in its affirmative by showing that PCs can answer the query while being expressive enough to model natural images.

Recall from Section 5.1.2 and Eq. (5.5) that at every denoising step, we need to compute $p_{\text{TPM}}(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t, \boldsymbol{x}_0^{\text{k}})$ with the PC. This section proposes an algorithm that computes $p_{\text{TPM}}(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t, \boldsymbol{x}_0^{\text{k}})$ given a PC $p(\boldsymbol{x}_0)$ in linear time w.r.t. its size. Specifically, we first demonstrate how Eq. (5.4) can be converted to a general form of queries we define as *independent soft-evidence constraints*. We then establish an efficient inference algorithm for this query class.

After closer inspection of Eq. (5.4), we observe that both $q(\boldsymbol{x}_t|\tilde{\boldsymbol{x}}_0)$ and $\mathbb{1}[\tilde{\boldsymbol{x}}_0^{\text{k}} = \boldsymbol{x}_0^{\text{k}}]$ can be considered as constraints factorized over every variable. Specifically, with $w_i(\tilde{x}_0^i) := q(x_t^i|\tilde{x}_0^i)$ if $\tilde{X}_0^i \in \tilde{\mathbf{X}}_0^{\text{u}}$ and $w_i(\tilde{x}_0^i) := \mathbb{1}[\tilde{x}_0^{\text{k}} = x_0^{\text{k}}]$ otherwise, $p_{\text{TPM}}(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t, \boldsymbol{x}_0^{\text{k}})$ can be equivalently expressed as

$$p_{\text{TPM}}(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t, \boldsymbol{x}_0^{\text{k}}) = \frac{1}{Z}\prod_i w_i(\tilde{x}_0^i) \cdot p(\tilde{\boldsymbol{x}}_0), \text{ where } Z := \sum_{\boldsymbol{x}_0}\prod_i w_i(\tilde{x}_0^i) \cdot p(\tilde{\boldsymbol{x}}_0). \tag{5.6}$$

We call w_i the *soft-evidence constraint* of variable X_0^i as it defines a prior belief of its value. In the extreme case of conditioning on hard evidence, w_i becomes an indicator that puts all weight on the conditioned value. Recall from Section 5.1.1 that diffusion models parameterize $p_\theta(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t)$ as a fully-factorized distribution. In order to compute the weighted geometric mean of $p_{\text{DM}}(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t, \boldsymbol{x}_0^{\text{k}})$ and $p_{\text{TPM}}(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t, \boldsymbol{x}_0^{\text{k}})$ (cf. Eq. (5.5)), we need to also compute the univariate distributions $p_{\text{TPM}}(\tilde{x}_0^i|\boldsymbol{x}_t, \boldsymbol{x}_0^{\text{k}})$ for every $\tilde{X}_0^i \in \tilde{\mathbf{X}}_0$. While this seems to suggest the need to query the PC at least $|\tilde{\mathbf{X}}_0|$ times, we propose an algorithm that only needs a forward and a

backward pass to compute *all* target probabilities.

The forward pass Similar to the likelihood query algorithm performed by a feedforward pass of the PC, we traverse all nodes in postorder and store the output of every node n in $\texttt{fw}_n$. For sum and product nodes, the output is computed following Eq. (2.1); the output of every input node n that encodes a distribution of X_0^i is defined as $\texttt{fw}_n := \sum_{x_0^i} f_n(x_0^i) \cdot w_i(x_0^i)$, where f_n is defined in Eq. (2.1).

The backward pass The backward pass consists of two steps: (i) traversing all nodes in preorder (parents before children) to compute the backward value $\texttt{bk}_n$; (ii) computing the target probabilities using the backward value of all input nodes. For ease of presentation, we assume the PC alternates between sum and product layers, and all parents of any input node are product nodes.[2] First, we compute the backward values by setting $\texttt{bk}_{n_\text{r}}$ of the root node to 1, then recursively compute the backward value of other nodes as follows:

$$\texttt{bk}_n := \begin{cases} \sum_{m \in \texttt{pa}(n)} \left(\theta_{m,n} \cdot \texttt{fw}_n / \texttt{fw}_m\right) \cdot \texttt{bk}_m & n \text{ is a product node,} \\ \sum_{m \in \texttt{pa}(n)} \texttt{bk}_m & n \text{ is a input or sum node,} \end{cases}$$

where $\texttt{pa}(n)$ is the set of parents of node n. Next, for every i, we gather all input nodes defined on X_0^i, denoted S_i, and compute $p_\text{TPM}(\tilde{x}_0^i | \boldsymbol{x}_t, \boldsymbol{x}_0^\text{k}) := \frac{1}{Z} \sum_{n \in S_i} \texttt{bk}_n \cdot f_n(x_0^i) \cdot w_i(x_0^i)$.

Theorem 11. *For any smooth and decomposable PC $p(\mathbf{X})$ and univariate weight functions $\{w_i(X_i)\}_i$, define $p'(\boldsymbol{x}) = \frac{1}{Z} \prod_i w_i(x_i) \cdot p(\boldsymbol{x})$, where the normalizing constant $Z := \sum_{\boldsymbol{x}} \prod_i w_i(x_i) \cdot p(\boldsymbol{x})$. Assume all variables in $\mathbf{X}$ are categorical variables with C categories, the above-described algorithm computes $p'(x_i)$ for every variable X_i and its every assignment x_i in time $\mathcal{O}(|p| + |\mathbf{X}| \cdot C) = \mathcal{O}(|p|)$.*

Proof can be found in Section D.1.1.

[2]Every PC that does not satisfy such constraints can be transformed into one in linear time since (i) consecutive sum or product nodes can be merged without changing the PC's semantic, and (ii) we can add a dummy product with one child between any pair of sum and input nodes.

5.1.4 Towards High-Resolution Image Inpainting

Another key factor determining the effectiveness of the PC-guided diffusion model is the expressiveness of the PC $p(\mathbf{X}_0)$, i.e., how well it can model the target image distribution. Recent advances have significantly pushed forward the expressiveness of PCs [99, 100], leading to competitive likelihoods on datasets such as CIFAR [85] and down-sampled ImageNet [38], which allows us to directly apply the guided inpainting algorithm to them. However, there is still a gap towards directly modeling high-resolution (e.g., 256×256) image data. While it is possible that this could be achieved in the near future given the rapid development of PCs, this paper explores an alternative approach where we use a (variational) auto-encoder to transform high-resolution images to a lower-dimensional latent space. Although in this way we lose the "full tractability" over every pixel, as we shall proceed to demonstrate, a decent approximation can still be achieved. The key intuition is that the latent space concisely captures the semantic information of the image, and thus can effectively guide diffusion models toward generating semantically coherent images; fine-grained details such as color consistency of the neighboring pixels can be properly handled by the neural-network-based diffusion model. This is empirically justified in Section 5.1.5.

Define the latent space of the image $\mathbf{X}_0$ as $\mathbf{Z}_0$. We adopt Vector Quantized Generative Adversarial Networks (VQ-GANs) [50], which are equipped with an encoder $q(\boldsymbol{z}_0|\boldsymbol{x}_0)$ and a decoder $p(\boldsymbol{x}_0|\boldsymbol{z}_0)$, to transform the images between the pixel space and the latent space.[3] We approximate $p_{\text{TPM}}(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t, \boldsymbol{x}_0^k)$ (Eq. 5.6) by first estimating $p_{\text{TPM}}(\tilde{\boldsymbol{z}}_0|\boldsymbol{x}_t, \boldsymbol{x}_0^k)$ with a PC $p(\mathbf{Z}_0)$ trained on the latent space and the VQ-GAN encoder; this latent-space conditional distribution is then converted back to the pixel space. Specifically, the latent-space conditional probability is approximated via

$$p_{\text{TPM}}(\tilde{\boldsymbol{z}}_0|\boldsymbol{x}_t, \boldsymbol{x}_0^k) \approx \frac{1}{Z}\prod_i w_i^z(\tilde{z}_0^i) \cdot p(\tilde{\boldsymbol{z}}_0), \text{ where } w_i^z(\tilde{z}_0^i) := \frac{1}{Z_i}\sum_{\tilde{\boldsymbol{x}}_0}\prod_j w_j(\tilde{x}_0^j) \cdot q(\tilde{z}_0^i|\tilde{\boldsymbol{x}}_0), \quad (5.7)$$

[3] We adopt VQ-GAN since it has a discrete latent space, which makes PC training easier.

where Z and $\{Z_i\}_i$ are normalizing constants and $q(\tilde{z}_0^i|\tilde{\boldsymbol{x}}_0)$ is the VQ-GAN encoder. It is safe to assume the independence between the soft evidence for different latent variables (i.e., w_i^z) since every latent variable produced by VQ-GAN corresponds to a different image patch, which corresponds to a different set of pixel-space soft constraints (i.e., w_i). In practice, we approximate $w_i^z(\tilde{z}_0^i)$ by performing Monte Carlo sampling over $\tilde{\boldsymbol{x}}_0$ (i.e., sample $\tilde{\boldsymbol{x}}_0$ following $\prod_j w_j(\tilde{x}_0^j)$, and then feed them through the VQ-GAN encoder). Finally, $p_{\text{TPM}}(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t, \boldsymbol{x}_0^k)$ is approximated by Monte Carlo estimation of $p_{\text{TPM}}(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t, \boldsymbol{x}_0^{\text{k}}) := \mathbb{E}_{\tilde{\boldsymbol{z}}_0 \sim p_{\text{TPM}}(\cdot|\boldsymbol{x}_t, \boldsymbol{x}_0^k)}[p(\tilde{\boldsymbol{x}}_0|\tilde{\boldsymbol{z}}_0)]$, where $p(\tilde{\boldsymbol{x}}_0|\tilde{\boldsymbol{z}}_0)$ is the VQ-GAN decoder. We observe that as few as 4-8 samples lead to significant performance gains across various datasets and mask types. See Section D.1.2 for details of the design choices.

Another main contribution of this paper is to further scale up PCs based on [99, 100] to achieve likelihoods competitive with GPTs [8] on the latent image space generated by VQ-GAN. Specifically, for 256×256 images, the latent space typically consists of $16 \times 16 = 256$ categorical variables each with 2048-16384 categories. While the number of variables is similar to datasets considered by prior PC learning approaches, the variables are much more semantically complicated (e.g., patch semantic vs. pixel value). We provide the full learning details, including the model structure and the training pipeline in Section D.1.3.

In summary, similar to the pixel-space guided inpainting algorithm introduced in Section 5.1.2, its latent-space variant also computes $p_{\text{TPM}}(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t, \boldsymbol{x}_0^{\text{k}})$ to guide the diffusion model $p_{\text{DM}}(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t, \boldsymbol{x}_0^{\text{k}})$ with Eq. (5.5), except that it is approximated using a latent-space PC combined with VQ-GAN.

5.1.5 Experiments

In this section, we take gradual steps to analyze and illustrate our method **Tiramisu** (**Tra**ctable **Im**age **I**npainting via **S**teering Diff**u**sion Models). Specifically, we first qualitatively investigate the steering effect of the TPM on the denoising diffusion process (Sec. 5.1.5). Next, we perform an empirical evaluation on three high-resolution image datasets with six large-hole

masks, which significantly challenges its ability to generate semantically consistent images (Sec. 5.1.5). Finally, inspired by the fact that Tiramisu can handle arbitrary constraints that can be written as independent soft evidence, we test it on a new controlled image generation task termed *image semantic fusion*, where the goal is to fuse parts from different images and generate images with semantic coherence and high fidelity (Sec. 5.1.5).

Analysis of the TPM-Provided Guidance

Since we are largely motivated by the ability of TPMs to generate images that better match the semantics of the given pixels, it is natural to examine how the TPM-generated signal guides the diffusion model during the denoising process. Recall from Section 5.1.2 that at every denoising step t, the reconstruction distributions $p_{\mathrm{DM}}(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t, \boldsymbol{x}_0^{\mathrm{k}})$ and $p_{\mathrm{TPM}}(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t, \boldsymbol{x}_0^{\mathrm{k}})$ are computed/estimated using the diffusion model and the TPM, respectively. Both distributions are then merged into $p(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t, \boldsymbol{x}_0^{\mathrm{k}})$ (Eq. (5.5)) and are used to generate the image at the previous noise level (i.e., $\boldsymbol{x}_{t-1}$). In all experiments, we adopt CoPaint [191] to generate $p_{\mathrm{DM}}(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t, \boldsymbol{x}_0^{\mathrm{k}})$, which is independent of the design choices related to the TPM. Therefore, qualitatively comparing the denoising process of Tiramisu and CoPaint allows us to examine the steering effect provided by the TPM.

Fig. 5.1 visualizes the denoising process of Tiramisu by plotting the images corresponding to the expected values of the aforementioned distributions (i.e., $p_{\mathrm{DM}}(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t, \boldsymbol{x}_0^{\mathrm{k}})$, $p_{\mathrm{TPM}}(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t, \boldsymbol{x}_0^{\mathrm{k}})$, and $p(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t, \boldsymbol{x}_0^{\mathrm{k}})$). To minimize distraction, we first focus on image pairs of the DM- and TPM-generated image pairs in the same column. Since they are generated from the same input image $\boldsymbol{x}_t$, comparing the image pairs allows us to examine the built-in inductive biases in both distributions. For instance, in the celebrity face image, we observe that the contour of the facial features is sharper for the TPM-generated image. This is more obvious in images at larger time steps since the guidance provided by the TPM is accumulated throughout the denoising process.

Next, we look at the second row (i.e., $p(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t, \boldsymbol{x}_0^{\mathrm{k}})$) of Tiramisu. Although blurry, global

Table 5.1: Quantative results on three datasets: CelebA-HQ [103], ImageNet [38], and LSUN-Bedroom [189]. We report the average `LPIPS` value (lower is better) [193] across 100 inpainted images for all settings. Bold indicates the best result.

Tasks		Algorithms						
Dataset	Mask	Tiramisu (ours)	CoPaint	RePaint	DDNM	DDRM	DPS	Resampling
CelebA-HQ	Left	0.189	**0.185**	0.195	0.254	0.275	0.201	0.257
	Top	0.187	**0.182**	0.187	0.248	0.267	0.187	0.251
	Expand1	**0.454**	0.468	0.504	0.597	0.682	0.466	0.613
	Expand2	0.442	0.455	0.480	0.585	0.686	**0.434**	0.601
	V-strip	**0.487**	0.502	0.517	0.625	0.724	0.535	0.647
	H-strip	**0.484**	0.488	0.517	0.626	0.731	0.492	0.639
	Wide	**0.069**	0.072	0.075	0.112	0.132	0.078	0.128
ImageNet	Left	**0.286**	0.289	0.296	0.410	0.369	0.327	0.369
	Top	**0.308**	0.312	0.336	0.427	0.373	0.343	0.368
	Expand1	**0.616**	0.623	0.691	0.786	0.726	0.621	0.711
	Expand2	**0.597**	0.607	0.692	0.799	0.724	0.618	0.721
	V-strip	0.646	0.654	0.741	0.851	0.761	**0.637**	0.759
	H-strip	0.657	0.660	0.744	0.851	0.753	**0.647**	0.774
	Wide	**0.125**	0.128	0.127	0.198	0.197	0.132	0.196
LSUN-Bedroom	Left	**0.285**	0.287	0.314	0.345	0.366	0.314	0.367
	Top	**0.310**	0.323	0.347	0.376	0.368	0.355	0.372
	Expand1	**0.615**	0.637	0.676	0.716	0.695	0.641	0.699
	Expand2	**0.635**	0.641	0.666	0.720	0.691	0.638	0.690
	V-strip	**0.672**	0.676	0.711	0.760	0.721	0.674	0.725
	H-strip	0.679	0.686	0.722	0.766	0.726	**0.674**	0.724
	Wide	0.116	0.115	0.124	0.135	0.204	**0.108**	0.202
Average		**0.421**	0.427	0.459	0.532	0.531	0.434	0.514

semantics appear at the early stages of the denoising process. For example, on the right side, we can vaguely see two ostriches visible at time step 217. In contrast, the denoised image at $t = 217$ for CoPaint does not contain much semantic information. Conditioning on these blurred contents, the diffusion model can further fill in fine-grained details. Since the image semantics can be generated in a few denoising steps, we only need to query the TPM at early time steps, which also significantly reduces the computational overhead of Tiramisu. See Section 5.1.5 for quantitative analysis. As a result, compared to the baseline, Tiramisu can generate inpainted images with higher quality.

Comparison With the State of the Art

In this section, we challenge Tiramisu against state-of-the-art diffusion-based inpainting algorithms on three large-scale high-resolution image datasets: CelebA-HQ [103], ImageNet

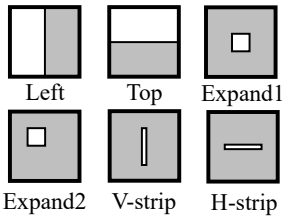


Figure 5.2: Used masks.

[38], and LSUN-Bedroom [189]. To further challenge the ability of Tiramisu to generate semantically coherent images, we use seven types of masks that reveal only 5-20% of the original image since it is very likely for inpainting algorithms to ignore the given visual cues and generate semantically inconsistent images. Details of the masks can be found in Fig. 5.2.

Methods We consider the six following diffusion-based inpainting algorithms: CoPaint [191], RePaint [108], DDNM [180], DDRM [74], DPS [23], and Resampling [172]. Although not exhaustive, this set of methods summarizes recent developments in image inpainting and can be deemed as state-of-the-art. We base our method Tiramisu on CoPaint (i.e., generate $p_{\mathrm{DM}}(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t, \boldsymbol{x}_0^{\mathrm{k}})$ with CoPaint). Please see the appendix for details on Tiramisu (Appx. D.1.2 and D.1.3) and the baselines (Appx. D.1.4).

Quantitative and qualitative results Table 5.1 shows the average LPIPS values [193] on all $3 \times 7 = 21$ dataset-mask configurations. First, Tiramisu outperforms CoPaint in 18 out of 21 settings, which demonstrates that the TPM-provided guidance consistently improves the quality of generated images. Next, compared to all baselines, Tiramisu achieves the best LPIPS value on 14 out of 21 settings, which indicates its superiority over the baselines. This conclusion is further supported by the sample inpainted images shown in Fig. 5.3, which suggests that Tiramisu generates more semantically consistent images. See Appx. D.1.5 for more samples and Appx. D.1.5 for user studies.

Computational efficiency As illustrated in Section 5.1.5, we can use PC to steer the denoising steps only in earlier stages. While engaging PCs in more denoising steps could lead to better performance, the runtime is also increased accordingly. To better understand this tradeoff, we use CelebA + the Expand1 mask as an example to analyze this tradeoff. As

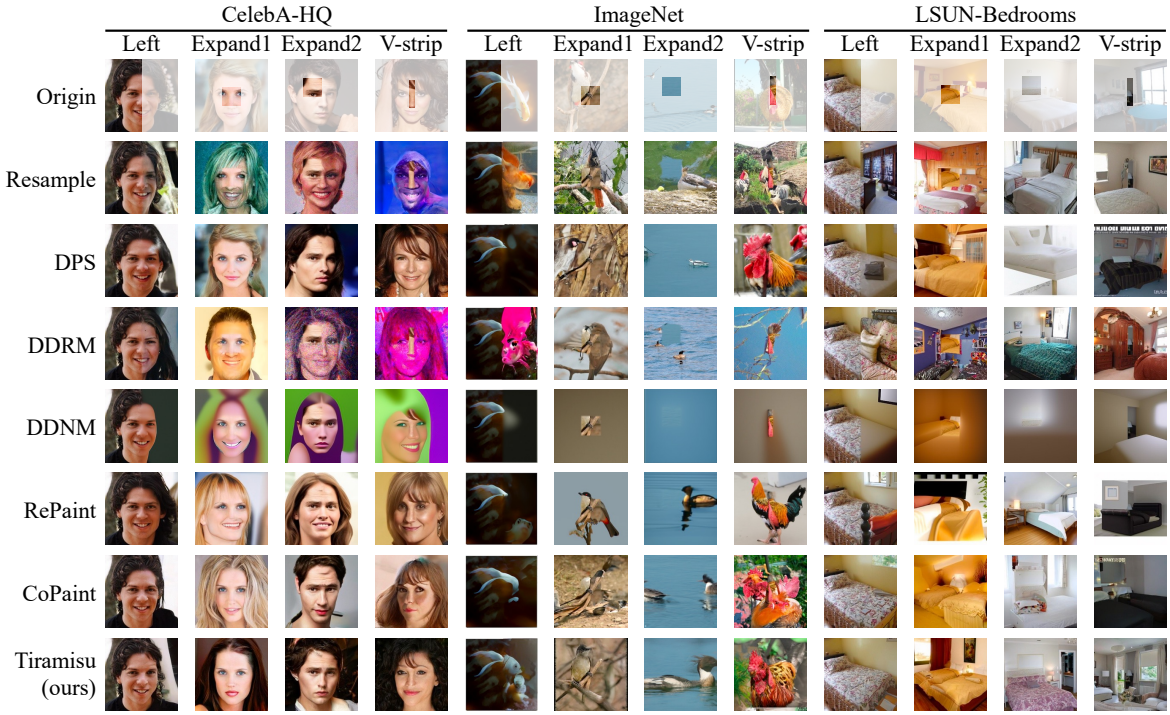


Figure 5.3: Qualitative results on all three adopted datasets. We compare Tiramisu against six diffusion-based inpainting algorithms. Please refer to Section D.1.5 for more qualitative results.

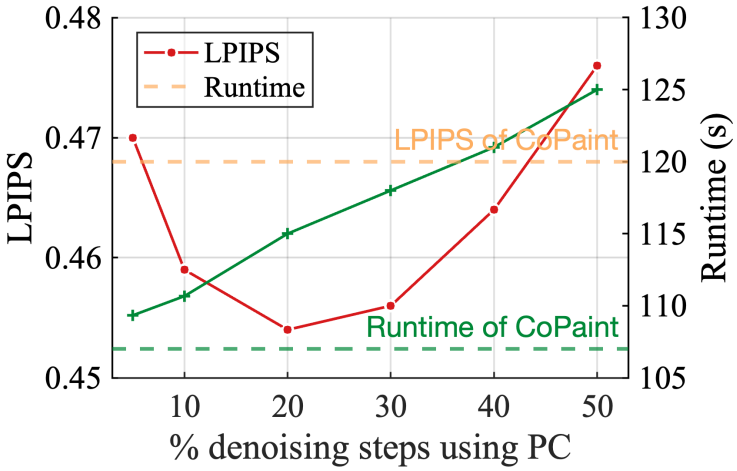


Figure 5.4: Performance and runtime.

shown in Fig. 5.4, as we use PCs in more denoising steps, the LPIPS score first decreases and then increases, suggesting that incorporating PCs in a moderate amount of steps gives the best performance (around 20% in this case). One explanation to this phenomenon is that in later denoising stages, the diffusion model mainly focuses on refining details. However, PCs are better at controlling the global semantics of images in earlier denoising stages. We then focus on the computation time. When using PCs in 20% of the denoising steps, the additional computational overhead incurred by the TPM is around 10s, which is only 10% of the total computation time.

Beyond Image Inpainting

The previous sections demonstrate the effectiveness of using TPMs on image inpainting tasks. A natural follow-up question is *whether this framework can be generalized to other controlled/constrained image generation tasks?* Although we do not have a definite answer, this section demonstrates the potential of extending Tiramisu to more complicated tasks by showing its capability to fuse the semantic information from various input patches/fragments. Specifically, consider the case of latent-space soft evidence constraints $\{w_i^{\mathbf{z}}\}_i$ (i.e., Eq. (5.7)). For various recent autoencoder models such as VQ-GAN, the latent variables of size $H_{\mathrm{l}} \times W_{\mathrm{l}}$ are encoded from images of size $H \times W$. Intuitively, every latent variable encodes the semantic

Figure 5.5: CelebA-HQ qualitative results for the semantic fusion task. In every sample, two reference images together with their masks are provided to Tiramisu. The task is to generate images that (i) semantically align with the unmasked region of both reference images, and (ii) have high fidelity. For every input, we generate five samples with different levels of semantic coherence. The left-most images are the least semantically constrained and barely match the semantic patterns of the reference images. In contrast, the right-most images strictly match the semantics of the reference images.

of an $H/H_1 \times W/W_1$ image patch. Therefore, every w_i^z can be viewed as a constraint on the semantics of the corresponding image patch.

We introduce a controlled image generation task called *semantic fusion*, where we are given several reference images each paired with a mask. The goal is to generate images that (i) semantically align with the unmasked region of every reference image, and (ii) have high quality and fidelity. Semantic fusion can be viewed as a preliminary task for more general controlled image generation since any type of visual word information (e.g., language condition) can be transferred to constraints on $\{w_i^z\}_i$.

Fig. 5.5 shows qualitative results of Tiramisu on semantic fusion tasks. For every set of reference images, we generate five samples with different semantic coherence levels by adjusting the temperature of every soft evidence function $w_i^z(z_0^i)$.

5.2 Lossless Data Compression

Despite extensive progress in image generation, common deep generative model architectures are not easily applied to lossless compression due to the challenge of using their learned distributions to apply common coding algorithms. To overcome this problem, this chapter explores the possibility of using more tractable deep generative models – PCs for lossless data compression.

5.2.1 Background and Motivation

Thanks to their expressiveness, modern Deep Generative Models (DGMs) such as Flow-based models [45], Variational Autoencoders (VAEs) [80], and Generative Adversarial Networks (GANs) [56] achieved state-of-the-art results on generative tasks such as creating high-quality samples [173] and learning low-dimensional representation of data [195]. However, these successes have not been fully transferred into neural lossless compression; see [187] for a recent survey. Specifically, GANs cannot be used for lossless compression due to their inability to assign likelihoods to observations. Latent variable models such as VAEs rely on rate estimates obtained by lower-bounding the likelihood of the data, i.e., the quantity which is theoretically optimal for lossless compression; they furthermore rely on sophisticated schemes such as bits-back coding [62] to realize these rates, oftentimes resulting in poor single-sample compression ratios [81].

Therefore, good generative performance does not imply good compression performance for lossless compression, as the model needs to support efficient algorithms to encode and decode close to the model's theoretical rate estimate. While both Flow- and VAE-based compression algorithms [64, 81] support efficient and near-optimal compression under certain assumptions (e.g., the existence of an additional source of random bits), we show that Probabilistic Circuits (PCs) [14] are also suitable for lossless compression tasks. This class of *tractable* models has

The contents of this chapter appeared in paper [95].

a particular structure that allows efficient marginalization of its random variables–a property that, as we show, enables efficient conditional entropy coding. Therefore, we introduce PCs as backbone models and develop (de)compression algorithms that achieve high compression ratios and high computational efficiency.

5.2.2 Tractability Matters in Lossless Compression

The goal of lossless compression is to map every input sample to an output codeword such that (i) the original input can be reconstructed from the codeword, and (ii) the expected length of the codewords is minimized. Practical (neural) lossless compression algorithms operate in two main phases — learning and compression [187]. In the learning phase, a generative model $p(\mathbf{X})$ is learned from a dataset $\mathcal{D} := \{\boldsymbol{x}^{(i)}\}_{i=1}^{N}$. According to Shannon's source coding theorem [155], the expected codeword length is lower-bounded by the negative cross-entropy between the data distribution $\mathcal{D}$ and the model distribution $p(\mathbf{X})$ (i.e., $-\mathbb{E}_{\boldsymbol{x}\sim\mathcal{D}}[\log p(\boldsymbol{x})]$), rendering it a natural and widely used objective to optimize the model [64, 114].

In the compression phase, compression algorithms take the learned model p and samples $\boldsymbol{x}$ as input and generate codewords whose expected length approaches the theoretical limit (i.e., the negative cross-entropy between $\mathcal{D}$ and p). Although there exist various close-to-optimal compression schemes (e.g., Huffman Coding [66] and Arithmetic Coding [145]), a natural question to ask is *what are the requirements on the model p such that compression algorithms can utilize it for encoding/decoding in a computationally efficient manner?* In this paper, we highlight the advantages of *tractable* probabilistic models for lossless compression by introducing a concrete class of models that are expressive and support efficient encoding and decoding.

To encode a sample $\boldsymbol{x}$, a standard streaming code operates by sequentially encoding every symbol x_i into a bitstream b, such that x_i occupies approximately $-\log p(x_i|x_1, \ldots, x_{i-1})$ bits in b. As a result, the length of b is approximately $-\log p(\boldsymbol{x})$. For example, Arithmetic Coding (AC) encodes the symbols $\{x_i\}_{i=1}^{D}$ (define $D := |\mathbf{X}|$ as the number of features) sequentially by

successively refining an interval that represents the sample, starting from the initial interval $[0, 1)$. To encode x_i, the algorithm partitions the current interval $[a, b)$ using the left and right side cumulative probability of x_i:

$$l_i(x_i) := p(X_i < x_i \mid x_1, \ldots, x_{i-1}), \qquad h_i(x_i) := p(X_i \leq x_i \mid x_1, \ldots, x_{i-1}). \tag{5.8}$$

Specifically, the algorithm updates $[a, b)$ to the following: $[a + (b-a) \cdot l_i(x_i), a + (b-a) \cdot h_i(x_i))$, which is a sub-interval of $[a, b)$. Finally, AC picks a number within the final interval that has the shortest binary representation. This number is encoded as a bitstream representing the codeword of $\boldsymbol{x}$. Upon decoding, the symbols $\{x_i\}_{i=1}^D$ are decoded sequentially: at iteration i, we decode variable X_i by looking up its value x such that its cumulative probability (i.e., $l_i(x)$) matches the subinterval specified by the codeword and $x_1, \ldots, x_{i-1}$ [145]; the decoded symbol x_i is then used to compute the following conditional probabilities (i.e., $l_j(x)$ for $j > i$). Despite implementation differences, computing the cumulative probabilities $l_i(x)$ and $h_i(x)$ are required for many other streaming codes (e.g., rANS). Therefore, for most streaming codes, the main computation cost of both the encoding and decoding process comes from calculating $l_i(x)$ and $h_i(x)$.

The main challenge for the above (de)compression algorithm is to balance the expressiveness of p and the computation cost of $\{l_i(x), h_i(x)\}_{i=1}^D$. On the one hand, highly expressive probability models such as energy-based models [87, 143] can potentially achieve high compression ratios at the cost of slow runtime, which is due to the requirement of estimating the model's normalizing constant. On the other hand, models that make strong independence assumptions (e.g., n-gram, fully-factorized) are cheap to evaluate but lack the expressiveness to model complex distributions over structured data such as images.

This paper explores the middle ground between the above two extremes. Specifically, we ask: *are there probabilistic models that are both expressive and permit efficient computation of the conditional probabilities in Eq.* (5.8)*?* This question can be answered in the affirmative

by establishing a new class of tractable lossless compression algorithms using Probabilistic Circuits (PCs) [14], which are neural networks that can compute various probabilistic queries efficiently. In the following, we overview the empirical and theoretical results of the proposed (de)compression algorithm.

We start with theoretical findings: the proposed encoding and decoding algorithms enjoy time complexity $\mathcal{O}(\log(D) \cdot |p|)$, where $|p| \geq D$ is the PC model size. The backbone of both algorithms, formally introduced in Section 5.2.3, is an algorithm that computes the $2 \times D$ conditional probabilities $\{l_i(x), h_i(x)\}_{i=1}^{D}$ given any $\boldsymbol{x}$ efficiently, as justified by the following theorem.

Theorem 12 (informal)**.** *Let $\boldsymbol{x}$ be a D-dimensional sample, and let p be a PC model of size $|p|$. We then have that computing all quantities $\{l_i(x_i), h_i(x_i)\}_{i=1}^{D}$ takes $\mathcal{O}(\log(D) \cdot |p|)$ time. Therefore, en- or decoding $\boldsymbol{x}$ with a streaming code (e.g., Arithmetic Coding) takes $\mathcal{O}(\log(D) \cdot |p| + D) = \mathcal{O}(\log(D) \cdot |p|)$ time.*

The properties of PCs that enable this efficient lossless compression algorithm will be described in the following, and the backbone inference algorithm with $\mathcal{O}(\log(D) \cdot |p|)$ time complexity will later be shown as Algorithm 5. Table 5.2 provides an (incomplete) summary of our empirical results. First, the PC-based lossless compression algorithm is fast and competitive. As shown in Table 5.2, the small PC model achieved a near-SoTA bitrate while being $\sim$15x faster than other neural compression algorithms with a similar bitrate. Next, PCs can be integrated with Flow-/VAE-based compression methods. As illustrated in Table 5.2(right), the integrated model significantly improved performance on sub-sampled ImageNet compared to the base IDF model.

5.2.3 Computationally Efficient (De)compression with PCs

In the previous section, we have boiled down the task of lossless compression to calculating conditional probabilities $\{l_i(x_i), h_i(x_i)\}_{i=1}^{D}$ given p and x_i. This section takes PCs into

Table 5.2: An (incomplete) summary of our empirical results. "Comp." stands for compression.

Method	MNIST (10,000 test images)		
	Theoretical bpd	Comp. bpd	En- & decoding time
PC (small)	1.26	1.30	**53**
PC (large)	**1.20**	**1.24**	168
IDF	1.90	1.96	880
BitSwap	1.27	1.31	904

Method	ImageNet32	ImageNet64
	Theoretical bpd	Theoretical bpd
PC+IDF	**3.99**	**3.71**
IDF	4.15	3.90
RealNVP	4.28	3.98
Glow	4.09	3.81

consideration and demonstrates how these queries can be computed efficiently. In the following, we first introduce the PC-based (de)compression algorithm (Section 5.2.3). We then empirically evaluate the optimality and speed of the proposed compressor and decompressor (Section 5.2.5).

Efficient (De-)compression With Structured-Decomposable PCs

As a key sub-routine in the proposed algorithm, we describe how to compute marginal queries given a smooth and (structured-)decomposable PC in $\mathcal{O}(|p|)$ time. First, we assign probabilities to every input unit: for an input unit n defined on variable X, if evidence is provided for X in the query (e.g., $X = x$ or $X < x$), we assign to n the corresponding probability (e.g., $p(X = x), p(X < x)$) according to f_n in Eq. (2.1); if evidence of X is not given, probability 1 is assigned to n. Next, we do a feedforward (children before parents) traverse of inner PC units and compute their probabilities following Eq. (2.1). The probability assigned to the root unit is the final answer of the marginal query. Concretely, consider computing $p(x_1, \overline{x_2}, x_4)$ for the PC in Fig. 2.1. This is done by (i) assigning probabilities to the input units w.r.t. the given evidence x_1, $\overline{x_2}$, and x_4 (assign 0 to the input unit labeled X_2 and $\neg X_4$ as they contradict the given evidence; all other input units are assigned probability 1), and (ii) evaluate the probabilities of sum/product units following Eq. (2.1). Evaluated probabilities are labeled next to the corresponding units, hence the marginal probability at the output is $p(x_1, \overline{x_2}, x_4) = 0.056$.

The proposed PC-based (de)compression algorithm is outlined in Fig. 5.6. Consider compressing an 2-by-2 image, whose four pixels are denoted as $X_1, \ldots, X_4$. As discussed

in Section 5.2.2, the encoder converts the image into a bitstream by encoding all variables autoregressively. For example, suppose we have encoded x_1, x_2. To encode the next variable x_3, we compute the left and right side cumulative probability of x_3 given x_1 and x_2, which are defined as $l_3(x_3)$ and $h_3(x_3)$ in Section 5.2.2, respectively. A streaming code then encodes x_3 into a bitstream using these probabilities. Decoding is also performed autoregressively. Specifically, after x_1 and x_2 are decoded, the same streaming code uses the information from the bitstream and the conditional distribution $p(x_3 \mid x_1, x_2)$ to decode x_3.

Therefore, the main computation cost of the above en- and decoding procedures comes from calculating the $2D$ conditional probabilities $\{l_i(x), h_i(x)\}_{i=1}^{D}$ w.r.t. any $\boldsymbol{x}$. Since every conditional probability can be represented as the quotient of two marginals, it is equivalent to compute the two following sets of marginals: $F(\boldsymbol{x}) := \{p(x_1, \ldots, x_i)\}_{i=1}^{D}$ and $G(\boldsymbol{x}) := \{p(x_1, \ldots, x_{i-1}, X_i < x_i)\}_{i=1}^{D}$.

As a direct application of the marginal algorithm described in the first paragraph of this section, for every $\boldsymbol{x} \in \mathsf{val}(\mathbf{X})$, computing the $2D$ marginals $\{F(\boldsymbol{x}), G(\boldsymbol{x})\}$ takes $\mathcal{O}(D \cdot |p|)$ time. However, the linear dependency on D would render compression and decompression extremely time-consuming.

We can significantly accelerate the en- and decoding times if the PC is structured-decomposable. To this end, we introduce an algorithm that computes $F(\boldsymbol{x})$ and $G(\boldsymbol{x})$ in $\mathcal{O}(\log(D) \cdot |p|)$ time (instead of $\mathcal{O}(D \cdot |p|)$), given a smooth and structured-decomposable PC p. For ease of presentation, we only discuss how to compute $F(\boldsymbol{x})$ – the values $G(\boldsymbol{x})$ can be computed analogously.[4]

Before proceeding with a formal argument, we give a high-level explanation of the acceleration. In practice, we only need to evaluate a small fraction of PC units to compute each of its D marginals. This is different from regular neural networks and the key to speeding up the computation of $F(\boldsymbol{x})$. In contrast to neural networks, changing the input only slightly will leave most activations unchanged for structured-decomposable PCs. We make use of this

[4]The only difference between the computation of the ith term of $F(\boldsymbol{x})$ and the ith term of $G(\boldsymbol{x})$ is in the value assigned to the inputs for variable X_i (i.e., probabilities $p_n(X_i = x)$ vs. $p_n(X_i < x)$).

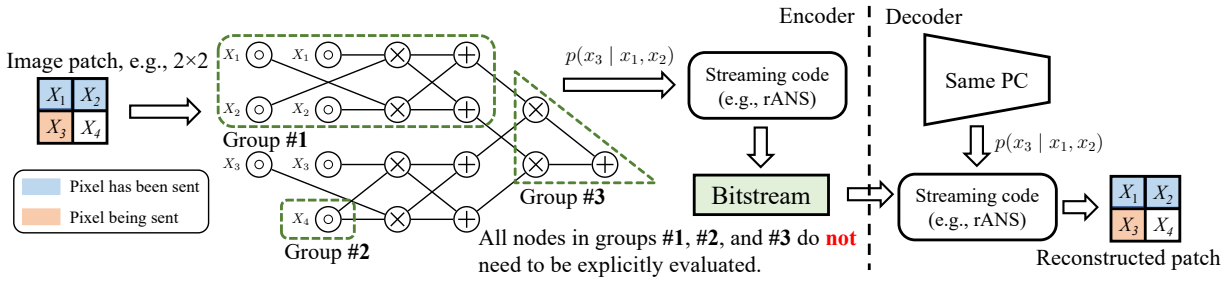


Figure 5.6: Overview of the PC-based (de)compressor. The encoder's side sequentially compresses variables one-by-one using the conditional probabilities given all sent variables. These probabilities are computed efficiently using Algorithm 5. Finally, a streaming code uses conditional probabilities to compress the variables into a bitstream. On the decoder's side, a streaming code decodes the bitstream to reconstruct the image with the conditional probabilities computed by the PC.

property by observing that adjacent marginals in $F(\boldsymbol{x})$ only differ in one variable — the ith term only adds evidence x_i compared to the $(i-1)$th term. We will show that such similarities between the marginal queries will lead to an algorithm that guarantees $\mathcal{O}(\log(D)\cdot|p|)$ overall time complexity.

An informal version of the proposed algorithm is shown in Algorithm 5.[5] In the main loop (lines 5-6), the D terms in $F(\boldsymbol{x})$ are computed one-by-one. Although the D iterations seem to suggest that the algorithm scales linearly with D, we highlight that each iteration on average re-evaluates only $\log(D)/D$ of the PC. Therefore, the computation cost of Algorithm 5 scales logarithmically w.r.t. D. The set of PC units need to be re-evaluated, eval_i, is identified in line 4, and lines 6 evaluates these units in a feedforward manner to compute the target probability (i.e., $p(x_1, \ldots, x_i)$).

Specifically, to minimize computation cost, at iteration i, we want to select a set of PC units eval_i that (i) guarantees the correctness of the target marginal, and (ii) contains the minimum number of units. We achieve this by recognizing three types of PC units that can be safely eliminated for evaluation. Take the PC shown in Fig. 5.6 as an example. Suppose we want to compute the third term in $F(\boldsymbol{x})$ (i.e., $p(x_1, x_2, x_3)$). First, all PC units in Group #1 do not need to be re-evaluated since their value only depends on x_1 and x_2 and hence

[5]See Section 5.2.4 for the formal algorithm and its detailed elaboration.

Algorithm 5 Compute $F(\boldsymbol{x})$ (see Algorithm 6 for details)

Input: A smooth and structured-decomposable PC p, variable instantiation $\boldsymbol{x}$
Output: $F_\pi(\boldsymbol{x}) = \{p(x_1, \ldots, x_i)\}_{i=1}^{D}$
Initialize: The probability $p(n)$ of every unit n is initially set to 1
        $\forall i, \text{eval}_i \leftarrow$ the set of PC units n that need to be evaluated in the ith iteration
for $i = 1$ **to** D **do**
    Evaluate PC units in eval_i in a bottom-up manner and compute $p(x_1, \ldots, x_i)$

remains unchanged. Next, PC units in Group #2 evaluate to 1. This can be justified from the two following facts: (i) input units correspond to X_4 have probability 1 while computing $p(x_1, x_2, x_3)$; (ii) for any sum or product unit, if all its children have probability 1, it also has probability 1 following Eq. (2.1). Finally, although the activations of the PC units in Group #3 will change when computing $p(x_1, x_2, x_3)$, we do not need to *explicitly* evaluate these units — the root node's probability can be equivalently computed using the weighted mixture of probabilities of units in eval_i. The correctness of this simplification step is justified in Section 5.2.4.

The idea of partially evaluating a PC originates from the Partial Propagation (PP) algorithm [9]. However, PP can only prune away units in Group #2. Thanks to the specific structure of the marginal queries, we are able to also prune away units in Groups #1 and #3.

Finally, we provide additional technical details to rigorously state the complexity of Algorithm 5. First, we need the variables $\mathbf{X}$ to have a specific order determined by the PC p. To reflect this change, we generalize $F(\boldsymbol{x})$ to $F_\pi(\boldsymbol{x}) := \{p(x_{\pi_1}, \ldots, x_{\pi_i})\}_{i=1}^{D}$, where π defines some variable order over $\mathbf{X}$, i.e., the ith variable in the order defined by π is X_{π_i}. Next, we give a technical assumption and then formally justify the *correctness* and *efficiency* of Algorithm 5 when using an optimal variable order π^*.

Definition 9. For a smooth structured-decomposable PC p over D variables, for any scope ϕ, denote $\text{nodes}(p, \phi)$ as the set of PC units in p whose scope is ϕ. We say p is *balanced* if for every scope ϕ' that is equal to the scope of any unit n in p, we have $|\text{nodes}(p, \phi')| = \mathcal{O}(|p|/D)$.

Algorithm 6 Compute $F_\pi(\boldsymbol{x})$

1: **Input:** A smooth and structured-decomposable PC p, variable order π, variable instantiation $\boldsymbol{x}$
2: **Output:** $F_\pi(\boldsymbol{x}) = \{p(x_{\pi_1}, \ldots, x_{\pi_i})\}_{i=1}^{D}$
3: **Initialize:** The probability $p(n)$ of every unit n is initially set to 1
4: $p_{\text{down}} \leftarrow$ the top-down probability of every PC unit$n$ (i.e., Algorithm 7)
5: **for** $i = 1$ **to** D **do** # Compute the ith term in $F_\pi(\boldsymbol{x})$: $p(x_{\pi_1}, \ldots, x_{\pi_i})$
6: $\quad$ $\text{eval}_i \leftarrow$ the set of PC units n with scopes $\phi(n)$ that satisfy at least one of the following conditions:
$\quad\quad$ (i) $\phi(n) = \{X_{\pi_i}\}$; $\quad$ (ii) n is a sum unit and at least one child c of n needs evaluation, i.e., $c \in \text{eval}_i$;
$\quad\quad$ (iii) n is a product unit and $X_{\pi_i} \in \phi(n)$ and $\nexists c \in \mathsf{ch}(n)$ such that $\{X_{\pi_j}\}_{j=1}^{i} \in \phi(c)$
7: $\quad$ Evaluate PC units in eval_i in a bottom-up manner to compute $\{p_n(\boldsymbol{x}) : n \in \text{eval}_i\}$
8: $\quad$ $\text{head}_i \leftarrow$ the set of PC units in eval_i such that none of their parents are in eval_i
9: $\quad$ $p(x_{\pi_1}, \ldots, x_{\pi_i}) \leftarrow \sum_{n \in \text{head}_i} p_{\text{down}}(n) \cdot p_n(\boldsymbol{x})$

Theorem 13. *For a smooth structured-decomposable balanced PC p over D variables $\mathbf{X}$ and a sample $\boldsymbol{x}$, there exists a variable order π^*, s.t. Algorithm 6 correctly computes $F_{\pi^*}(\boldsymbol{x})$ in $\mathcal{O}(\log(D) \cdot |p|)$ time.*

Proof. First, note that Algorithm 6 is a detailed version of Algorithm 5. The high-level idea of the proof is to first show how to compute the optimal variable order π^* for any smooth and structured-decomposable PC. Next, we justify the correctness of Algorithm 6 by showing (i) we only need to evaluate units that satisfy the criterion in line 6 of Algorithm 6 and (ii) weighing the PC units with the *top-down probabilities* (Section 5.2.4) always give the correct result. Finally, we use induction (on D) to demonstrate Algorithm 6 computes $\mathcal{O}(\log(D) \cdot |p|)$ PC units in total if π^* is used. Please refer to Section D.2.1 for the formal proof. □

While Definition 9 may seem restrictive at first glance, we highlight that most existing PC structures such as EiNets [132], RAT-SPNs [133] and HCLTs [97] are balanced. Once all marginal probabilities are calculated, samples $\boldsymbol{x}$ can be en- or decoded autoregressively with any streaming codes in time $\mathcal{O}(\log(D) \cdot |p|)$. Specifically, our implementation adopted the widely used streaming code rANS [48].

5.2.4 Algorithm Details

This section provides additional technical details of Algorithm 5. Specifically, we demonstrate (i) how to select the set of PC units eval_i (cf. Algorithm 5 line 5) and (ii) how to compute $p(x_1, \ldots, x_i)$ as a weighted mixture of P_i (cf. Algorithm 5 line 7). Using the example in

Fig. 5.7, we aim to provide an intuitive illustration to both problems. As an extension to Algorithm 5, rigorous and executable pseudocode for the proposed algorithm can be found in Algorithm 6 and 7.

The key to speeding up the naive marginalization algorithm is the observation that we only need to evaluate a small fraction of PC units to compute each of the D marginals in $F_\pi(\boldsymbol{x})$. Suppose we want to compute $F_\pi(\boldsymbol{x})$ given the structured-decomposable PC shown in Fig. 5.7(a), where $\oplus$, $\otimes$, and $\circledcirc$ denote sum, product, and input units, respectively. Model parameters are omitted for simplicity. Consider using the variable order $\pi = (X_1, X_2, X_3)$ (Fig. 5.7(b)). We ask the following question: what is the minimum set of PC units that need to be evaluated in order to compute $p(X_1 = x_1)$ (the first term in $F_\pi(\boldsymbol{x})$)? First, every PC unit with scope $\{X_1\}$ (i.e., the two nodes colored blue) has to be evaluated. Next, every PC unit n that is not an ancestor of the two blue units (i.e., "non-ancestor units" in Fig. 5.7(b)) must have probability 1 since (i) leaf units correspond to X_2 and X_3 have probability 1 while computing $p(X_1 = x_1)$, and (ii) for any sum or product unit, if all its children have probability 1, it also has probability 1 following Eq. (2.1). Therefore, we do not need to evaluate these non-ancestor units. Another way to identify these non-ancestor units is by inspecting their variable scopes — if the variable scope of a PC unit n does not contain X_1, it must has probability 1 while computing $p(X_1 = x_1)$. Finally, following all ancestors of the two blue units (i.e., "ancestor units" in Fig. 5.7(b)), we can compute the probability of the root unit, which is the target quantity $p(X_1 = x_1)$. At a first glance, this seems to suggest that we need to evaluate these ancestor units explicitly. Fortunately, as we will proceed to show, the root unit's probability can be equivalently computed using the blue units' probabilities weighted by a set of cached *top-down probabilities*.

For ease of presentation, denote the two blue input units as n_1 and n_2, respectively. A key observation is that the probability of every ancestor unit of $\{n_1, n_2\}$ (including the root unit) can be represented as a weighted mixture over $p_{n_1}(\boldsymbol{x})$ and $p_{n_2}(\boldsymbol{x})$, the probabilities assigned to n_1 and n_2, respectively. The reason is that for each decomposable product node m, only

distributions defined on disjoint variables shall be multiplied. Since n_1 and n_2 have the same variable scope, their distributions will not be multiplied by any product node. Following the above intuition, the top-down probability $p_{\text{down}}(n)$ of PC unit n is designed to represent the "weight" of n w.r.t. the probability of the root unit. Formally, $p_{\text{down}}(n)$ is defined as the sum of the probabilities of every path from n to the root unit n_r, where the probability of a path is the product of all edge parameters traversed by it. Back to our example, using the top-down probabilities, we can compute $p(X_1 = x_1) = \sum_{i=1}^{2} p_{\text{down}}(n_i) \cdot p_{n_i}(x_1)$ without explicitly evaluating the ancestors of n_1 and n_2. The quantity $p_{\text{down}}(n)$ of all PC units n can be computed by Algorithm 7 in $\mathcal{O}(|p|)$ time. Specifically, the algorithm performs a top-down traversal over all PC units n, and updates the top-down probabilities of their children $\mathsf{ch}(n)$ along the process.

Therefore, we only need to compute the two PC units with scope $\{X_1\}$ in order to calculate $p(X_1 = x_1)$. Next, when computing the second term $p(X_1 = x_1, X_2 = x_2)$, as illustrated in Fig. 5.7(b), we can reuse the evaluated probabilities of n_1 and n_2, and similarly only need to evaluate the PC units with scope $\{X_2\}$, $\{X_2, X_3\}$, or $\{X_1, X_2, X_3\}$ (i.e., nodes colored purple). The same scheme can be used when computing the third term, and we only evaluate PC units with scope $\{X_3\}$, $\{X_2, X_3\}$, or $\{X_1, X_2, X_3\}$ (i.e., all red nodes). As a result, we only evaluate 20 PC units in total, compared to $3 \cdot |p| = 39$ units required by the naive approach.

This procedure is formalized in Algorithm 6, which adds additional technical details compared to Algorithm 5. In the main loop (lines 5-9), the D terms in $F_\pi(\boldsymbol{x})$ are computed one-by-one. While computing each term, we first find the PC units that need to be evaluated (line 6).[6]After computing their probabilities in a bottom-up manner (line 7), we additionally use the pre-computed top-down probabilities to obtain the target marginal probability (lines 8-9).

The previous example demonstrates that even without a careful choice of variable order, we can significantly lower the computation cost by only evaluating the necessary PC units.

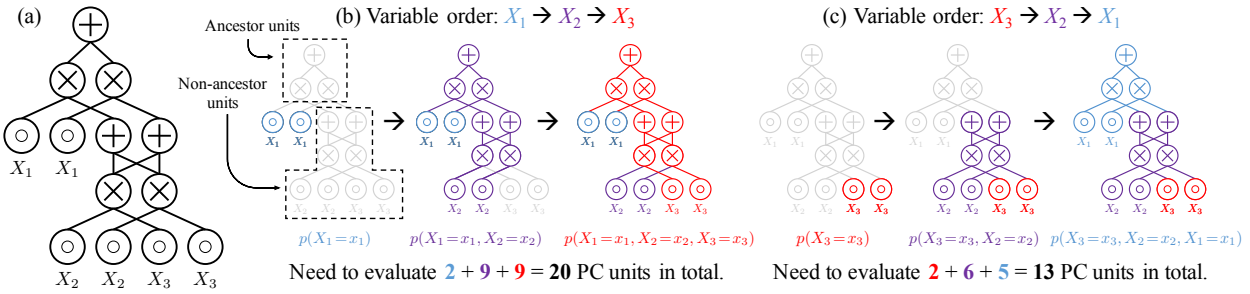


Figure 5.7: Good variable orders lead to more efficient computation of $F_\pi(\boldsymbol{x})$. Consider the PC p shown in (a). (b): If variable order X_1, X_2, X_3 is used, we need to evaluate 20 PC units in total. (c): The optimal variable order X_3, X_2, X_1 allows us to compute $F_\pi(\boldsymbol{x})$ by only evaluating 13 PC units.

Algorithm 7 PC Top-down Probabilities

Input: A smooth and structured-decomposable PC p
Output: The top-down probabilities $p_{\text{down}}(n)$ of all PC units n
For every PC unit n in p, initialize $p_{\text{down}}(n) \leftarrow 0$
foreach unit n traversed in preorder (parent before children) **do**
  if n is the root node of p **then** $p_{\text{down}}(n) \leftarrow 1$
  elif n is a sum unit **then foreach** $c \in \mathsf{ch}(n)$ **do** $p_{\text{down}}(c) \leftarrow p_{\text{down}}(c) + p_{\text{down}}(n) \cdot \theta_{n,c}$
  elif n is a product unit **then foreach** $c \in \mathsf{ch}(n)$ **do** $p_{\text{down}}(c) \leftarrow p_{\text{down}}(c) + p_{\text{down}}(n)$

We now show that with an *optimal* choice of variable order (denoted π^*), the cost can be further reduced. Consider using order $\pi^* = (X_3, X_2, X_1)$, as shown in Fig. 5.7(c), we only need to evaluate $2+6+5=13$ PC units in total when running Algorithm 6. This optimal variable order is the key to guaranteeing $\mathcal{O}(\log(D)\cdot|p|)$ computation time. In the following, we first give a technical assumption and then proceed to justify the *correctness* and *efficiency* of Algorithm 6 when using the optimal variable order π^*.

5.2.5 Experiments

We compare the proposed algorithm with competitive Flow-model-based (IDF by [64]) and VAE-based (BitSwap by [81]) neural compression algorithms using the MNIST dataset. We first evaluate bitrates. As shown in Table 5.3, the PC (de)compressor achieved compression rates close to its theoretical rate estimate — codeword bpds only have $\sim$0.04 loss w.r.t. the corresponding theoretical bpds. We note that PC and IDF have an additional advantage:

Table 5.3: Efficiency and optimality of the (de)compressor. The compression (resp. decompression) time are the total computation time used to encode (resp. decode) all 10,000 MNIST test samples on a single TITAN RTX GPU. The proposed (de)compressor for structured-decomposable PCs is 5-40x faster than IDF and BitSwap and only leads to a negligible increase in the codeword bpd compared to the theoretical bpd.

Method	# parameters	Theoretical bpd	Codeword bpd	Comp. time (s)	Decomp. time (s)
PC (HCLT, $M=16$)	3.3M	1.26	1.30	9	44
PC (HCLT, $M=24$)	5.1M	1.22	1.26	15	86
PC (HCLT, $M=32$)	7.0M	1.20	1.24	26	142
IDF	24.1M	1.90	1.96	288	592
BitSwap	2.8M	1.27	1.31	578	326

their reported bitrates were achieved while compressing one sample at a time; however, BitSwap needs to compress sequences of 100 samples to achieve 1.31 codeword bpd [81].

Next, we focus on efficiency. While achieving a better codeword bpd (i.e., 1.30) compared to IDF and BitSwap, a relatively small PC model (i.e., HCLT, $M = 16$) encodes (resp. decodes) images 30x (resp. 10x) faster than both baselines. Furthermore, a bigger PC model ($M=32$) with 7M parameters achieved codeword bpd 1.24, and is still 5x faster than BitSwap and IDF. Note that at the cost of increasing the bitrate, one can significantly improve the en- and decoding efficiency. For example, by using a small VAE model, [169] managed to compress and decompress 10,000 binarized MNIST samples in 3.26s and 2.82s, respectively.

5.3 Offline Reinforcement Learning

A popular paradigm for offline Reinforcement Learning (RL) tasks is to first fit the offline trajectories to a sequence model, and then prompt the model for actions that lead to high expected return. In addition to obtaining accurate sequence models, this paper highlights that *tractability*, the ability to exactly and efficiently answer various probabilistic queries, plays an important role in offline RL. Specifically, due to the fundamental stochasticity from the offline data-collection policies and the environment dynamics, highly non-trivial conditional/constrained generation is required to elicit rewarding actions. While it is still possible to approximate such queries, we observe that such crude estimates undermine the benefits brought by expressive sequence models. To overcome this problem, we propose **Trifle** (**Tr**actable **I**nference for Off**l**in**e** RL), which leverages modern tractable generative models to bridge the gap between good sequence models and high expected returns at evaluation time. Empirically, Trifle achieves 7 state-of-the-art scores and the highest average scores in 9 Gym-MuJoCo benchmarks against strong baselines. Further, Trifle significantly outperforms prior approaches in stochastic environments and safe RL tasks with minimum algorithmic modifications.

5.3.1 Background and Motivation

Recent advancements in deep generative models have opened up the possibility of solving offline Reinforcement Learning (RL) [89] tasks with sequence modeling techniques (termed RvS approaches). Specifically, we first fit a sequence model to the trajectories provided in an offline dataset. During evaluation, the model is tasked to sample actions with high expected returns given the current state. Leveraging modern deep generative models such as GPTs [8] and diffusion models [63], RvS algorithms have significantly boosted the performance on various RL problems [1, 11].

The contents of this chapter appeared in paper [101].

Despite its appealing simplicity, it is still unclear whether expressive modeling alone guarantees good performance of RvS algorithms, and if so, on what types of environments. This paper discovers that many common failures of RvS algorithms are not caused by modeling problems. Instead, while useful information is encoded in the model during training, the model is unable to elicit such knowledge during evaluation. Specifically, this issue is reflected in two aspects: (i) *inability to accurately estimate the expected return* of a state and a corresponding action sequence to be executed given near-perfect learned transition dynamics and reward functions; (ii) even when accurate return estimates exist in the offline dataset and are learned by the model, it could still *fail to sample rewarding actions* during evaluation.[7] At the heart of such inferior evaluation-time performance is the fact that highly non-trivial conditional generation is required to stimulate high-return actions [6, 129]. Therefore, other than expressiveness, the ability to efficiently and exactly answer various queries (e.g., computing the expected returns), termed *tractability*, plays an equally important role in RvS approaches.

Having observed that the lack of tractability is an essential cause of the underperformance of RvS algorithms, this paper studies *whether we can gain practical benefits from using Tractable Probabilistic Models (TPMs) [14, 82, 136], which by design support exact and efficient computation of certain queries?* We answer the question in its affirmative by showing that we can leverage a class of TPMs that support computing arbitrary marginal probabilities to significantly mitigate the inference-time suboptimality of RvS approaches. The proposed algorithm **Trifle** (**Tr**actable **I**nference for Of**fl**in**e** RL) has three main contributions:

Emphasizing the important role of tractable models in offline RL. This is the first paper that demonstrates the possibility of using TPMs on complex offline RL tasks. The superior empirical performance of Trifle suggests that expressive modeling is not the only aspect that determines the performance of RvS algorithms, and motivates the development of better inference-aware RvS approaches.

[7] Both observations are supported by empirical evidence as illustrated in Section 5.3.2.

Competitive empirical performance. Compared against strong offline RL baselines (including RvS, imitation learning, and offline temporal-difference algorithms), Trifle achieves the state-of-the-art result on 7 out of 9 Gym-MuJoCo benchmarks [51] and has the best average score.

Generalizability to stochastic environments and safe-RL tasks. Trifle can be extended to tackle stochastic environments as well as safe RL tasks with minimum algorithmic modifications. Specifically, we evaluate Trifle in 2 stochastic OpenAI-Gym [7] environments and action-space-constrained MuJoCo environments, and demonstrate its superior performance against all baselines.

Offline Reinforcement Learning. In Reinforcement Learning (RL), an agent interacts with an environment that is defined by a Markov Decision Process (MDP) $\langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, d_0 \rangle$ to maximize its cumulative reward. Specifically, the $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function, $\mathcal{P} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is the transition dynamics, and d_0 is the initial state distribution. Our goal is to learn a policy $\pi(a|s)$ that maximizes the expected return $\mathbb{E}[\sum_{t=0}^{T} \gamma^t r_t]$, where $\gamma \in (0, 1]$ is a discount factor and T is the maximum number of steps.

Offline RL [89] aims to solve RL problems where we cannot freely interact with the environment. Instead, we receive a dataset of trajectories collected using unknown policies. An effective learning paradigm for offline RL is to treat it as a sequence modeling problem (termed RL via Sequence Modeling or RvS methods) [11, 49, 68]. Specifically, we first learn a sequence model on the dataset, and then sample actions conditioned on past states and high future returns. Since the models typically do not encode the entire trajectory, an estimated value or return-to-go (RTG) (i.e., the Monte Carlo estimate of the sum of future rewards) is also included for every state-action pair, allowing the model to estimate the return at any time step.

5.3.2 Tractability Matters in Offline RL

Practical RvS approaches operate in two main phases – training and evaluation. In the training phase, a sequence model is adopted to learn a joint distribution over trajectories of length T: $\{(s_t, a_t, r_t, \text{RTG}_t)\}_{t=0}^{T}$.[8] During evaluation, at every time step t, the model is tasked to discover an action sequence $a_{t:T} := \{a_\tau\}_{\tau=t}^{T}$ (or just a_t) that has high expected return as well as high probability in the prior policy $p(a_{t:T}|s_t)$, which prevents it from generating out-of-distribution actions:

$$p(a_{t:T}|s_t, \mathbb{E}[V_t] \geq v) := \frac{1}{Z} \cdot \begin{cases} p(a_{t:T}|s_t) & \text{if } \mathbb{E}_{V_t \sim p(\cdot|s_t,a_t)}[V_t] \geq v, \\ 0 & \text{otherwise,} \end{cases} \tag{5.9}$$

where Z is a normalizing constant, V_t is an estimate of the value at time step t, and v is a pre-defined scalar chosen to encourage high-return policies. Depending on the problem, V_t could be the labeled RTG from the dataset (e.g., RTG_t) or the sum of future rewards capped with a value estimate (e.g., $\sum_{\tau=t}^{T-1} r_\tau + \text{RTG}_T$) [49, 68].

The above definition naturally reveals two key challenges in RvS approaches: (i) *training-time optimality* (i.e., "expressivity"): how well can we fit the offline trajectories, and (ii) *inference-time optimality*: whether actions can be unbiasedly and efficiently sampled from Eq. (5.9). While extensive breakthroughs have been achieved to improve the training-time optimality [1, 11, 68], it remains unclear whether the non-trivial constrained generation task of Eq. (5.9) hinders inference-time optimality. In the following, we present two general scenarios where existing RvS approaches underperform as a result of suboptimal inference-time performance. We attribute such failures to the fact that these models are limited to answering certain query classes (e.g., autoregressive models can only compute next token probabilities), and explore the potential of *tractable* probabilistic models for offline RL tasks in the following sections.

[8] To minimize computation cost, we only model truncated trajectories of length K ($K < T$) in practice.

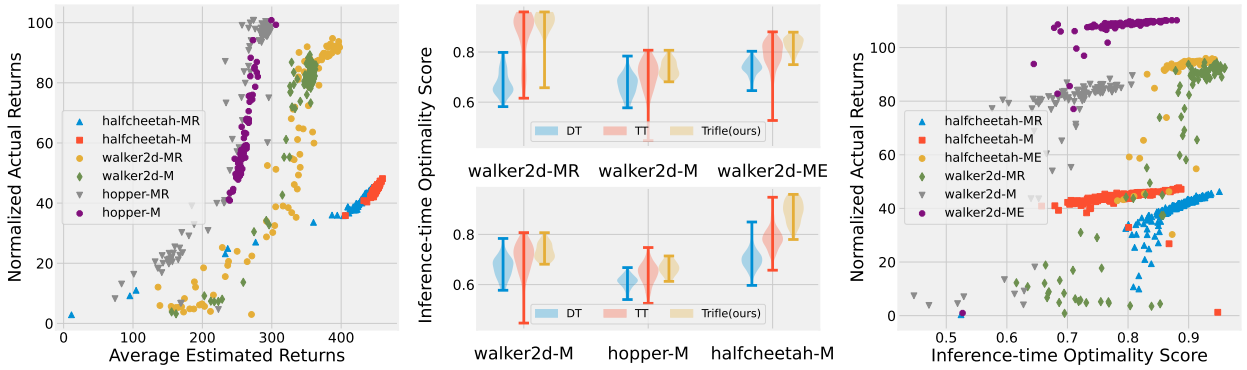


Figure 5.8: RvS approaches suffer from inference-time suboptimality. **Left:** There is a strong positive correlation between the average estimated returns by Trajectory Transformers (TT) and the actual returns in 6 Gym-MuJoCo environments (MR, M, and ME denote medium-replay, medium, and medium-expert, respectively), which suggests that the sequence model can distinguish rewarding actions from the others. **Middle:** Despite being able to recognize high-return actions, both TT and DT [11] fail to consistently sample such action, leading to bad inference-time optimality; Trifle consistently improves the inference-time optimality score. **Right:** We substantiate the relationship between low inference-time optimality scores and unfavorable environmental outcomes by showing a strong positive correlation between them.

Scenario #1 We first consider the case where the labeled RTG belongs to a (near-)optimal policy. In this case, Eq. (5.9) can be simplified to $p(a_t|s_t, \mathbb{E}[V_t] \geq v)$ (choose $V_t := \text{RTG}_t$) since one-step optimality implies multi-step optimality. In practice, although the RTGs are suboptimal, the predicted values often match well with the actual returns achieved by the agent. Take Trajectory Transformer (TT) [68] as an example, Fig. 5.8 (left) demonstrates a strong positive correlation between its predicted returns (x-axis) and the actual cumulative rewards (y-axis) on six MuJoCo [167] benchmarks, suggesting that the model has learned the "goodness" of most actions. In such cases, the performance of RvS algorithms depends mainly on their inference-time optimality, i.e., whether they can efficiently sample actions with high *predicted* returns. Specifically, let a_t be the action taken by a RvS algorithm at state s_t, and $R_t := \mathbb{E}[\text{RTG}_t]$ is the corresponding estimated expected value. We define a proxy of inference-time optimality as the quantile value of R_t in the estimated state-conditioned value distribution $p(V_t|s_t)$.[9] The higher the quantile value, the more frequent the RvS algorithm

[9]Due to the large action space, it is impractical to compute $p(V_t|s_t) := \sum_{a_t} p(V_t|s_t, a_t) \cdot p(a_t|s_t)$. Instead, in the following illustrative experiments, we train an additional GPT model $p(V_t|s_t)$ using the offline dataset.

samples actions with high estimated returns.

We evaluate the inference-time optimality of Decision Transformers (DT) [11] and Trajectory Transformers (TT) [68], two widely used RvS algorithms, on various environments and offline datasets from the Gym-MuJoCo benchmark suite [51]. As shown in Fig. 5.8 (middle), the inference-time optimality is averaged (only) around 0.7 (the maximum possible value is 1.0) for most settings. And these runs with low inference-time optimality scores receive low environment returns (Fig. 5.8 (right)).

Scenario #2 Achieving inference-time optimality becomes even harder when the labeled RTGs are suboptimal (e.g., they come from a random policy). In this case, even estimating the expected future return of an action sequence becomes highly intractable, especially when the transition dynamics of the environment are stochastic. Specifically, to evaluate a state-action pair (s_t, a_t), since RTG_t is uninformative, we need to resort to the multi-step estimate $V_t^{\text{m}} := \sum_{\tau=t}^{t'-1} r_\tau + \text{RTG}_{t'}$ $(t' > t)$, where the actions $a_{t:t'}$ are jointly chosen to maximize the expected return. Take autoregressive models as an example. Since the variables are arranged following the sequential order $\ldots, s_t, a_t, r_t, \text{RTG}_t, s_{t+1}, \ldots$, we need to explicitly sample $s_{t+1:t'}$ before proceed to compute the rewards and the RTG in V_t^{m}. In stochastic environments, estimating $\mathbb{E}[V_t^{\text{m}}]$ could suffer from high variance as the stochasticity from the intermediate states accumulates over time.

As we shall illustrate in Section 5.3.5, compared to environments with near-deterministic transition dynamics, estimating the expected returns in stochastic environments using intractable sequence models is hard, and Trifle can significantly mitigate this problem with its ability to marginalize out intermediate states and compute $\mathbb{E}[V_t^{\text{m}}]$ efficiently and exactly.

5.3.3 Exploiting Tractable Models

The previous section demonstrates that apart from modeling, inference-time suboptimality is another key factor that causes the underperformance of RvS approaches. Given such

observations, a natural follow-up question is *whether/how more tractable models can improve the evaluation-time performance in offline RL tasks?* While there are different types of tractabilities (i.e., the ability to compute different types of queries), this paper focuses on studying the additional benefit of *exactly* computing *arbitrary* marginal/condition probabilities. This strikes a proper balance between learning and inference as we can train such a tractable yet expressive model thanks to recent developments in the TPM community [27,99]. Note that in addition to proposing a competitive RvS algorithm, we aim to highlight the necessity and benefit of using more tractable models for offline RL tasks, and encourage future developments on both inference-aware RvS methods and better TPMs. As a direct response to the two failing scenarios identified in Section 5.3.2, we first demonstrate how tractability could help even when the labeled RTGs are (near-)optimal (Sec. 5.3.3). We then move on to the case where we need to use multi-step return estimates to account for biases in the labeled RTGs (Sec. 5.3.3).

From the Single-Step Case...

Consider the case where the RTGs are optimal. Recall from Section 5.3.2 that our goal is to sample actions from $p(a_t|s_t, \mathbb{E}[V_t] \geq v)$ (where $V_t := \text{RTG}_t$). Prior works use two typical ways to approximately sample from this distribution. The first approach directly trains a model to generate return-conditioned actions: $p(a_t|s_t, \text{RTG}_t)$ [11]. However, since the RTG given a state-action pair is stochastic,[10] sampling from this RTG-conditioned policy could result in actions with a small probability of getting a high return, but with a low expected return [6, 129].

An alternative approach leverages the ability of sequence models to accurately estimate the expected return (i.e., $\mathbb{E}[\text{RTG}_t]$) of state-action pairs [68]. Specifically, we first sample from a prior distribution $p(a_t|s_t)$, and then reject actions with low expected returns. Such rejection sampling-based methods typically work well when the action space is small (in

[10]This is true unless (i) the policy that generates the offline dataset is deterministic, (ii) the transition dynamics is deterministic, and (iii) the reward function is deterministic.

which we can enumerate all actions) or the dataset contains many high-rewarding trajectories (in which the rejection rate is low). However, the action could be multi-dimensional and the dataset typically contains many more low-return trajectories in practice, rendering the inference-time optimality score low (cf. Fig. 5.8).

Having examined the pros and cons of existing approaches, we are left with the question of whether a tractable model can improve sampled actions (in this single-step case). We answer it with a mixture of positive and negative results: while computing $p(a_t|s_t, \mathbb{E}[V_t] \geq v)$ is NP-hard even when $p(a_t, V_t|s_t)$ follows a simple Naive Bayes distribution, we can design an approximation algorithm that samples high-return actions with high probability in practice. We start with the negative result.

Theorem 14. *Let $a_t := \{a_t^i\}_{i=1}^k$ be a set of k boolean variables and V_t be a categorical variables with two categories 0 and 1. For some s_t, assume the joint distribution over a_t and V_t conditioned on s_t follows a Naive Bayes distribution: $p(a_t, V_t|s_t) := p(V_t|s_t) \cdot \prod_{i=1}^k p(a_t^i|V_t, s_t)$, where a_t^i denotes the i^{th} variable of a_t. Computing any marginal over the random variables is tractable yet conditioning on the expectation $p(a_t|s_t, \mathbb{E}[V_t] \geq v)$ is NP-hard.*

The proof is given in Section D.3.1. While it seems hard to directly draw samples from $p(a_t|s_t, \mathbb{E}[V_t] \geq v)$, we propose to improve the aforementioned rejection sampling-based method by adding a correction term to the original proposal distribution $p(a_t|s_t)$ to reduce the rejection rate. Specifically, the prior is often represented by an autoregressive model such as GPT: $p_{\text{GPT}}(a_t|s_t) := \prod_{i=1}^k p_{\text{GPT}}(a_t^i|s_t, a_t^{<i})$, where k is the number of action variables and a_t^i is the ith variable of a_t. We propose to sample every dimension of a_t autoregressively following:

$$\forall i \in \{1, \ldots, k\} \qquad \tilde{p}(a_t^i|s_t, a_t^{<i}; v) := \frac{1}{Z} \cdot p_{\text{GPT}}(a_t^i|s_t, a_t^{<i}) \cdot p_{\text{TPM}}(V_t \geq v|s_t, a_t^{\leq i}), \tag{5.10}$$

where Z is a normalizing constant and $p_{\text{TPM}}(V_t \geq v|s_t, a_t^{\leq i})$ is a correction term that leverages the ability of the TPM to compute the distribution of V_t given incomplete actions (i.e., evidence on a subset of action variables). Note that while Eq. (5.10) is mathematically identical to $p(a_t|s_t, V_t \geq v)$ when $p = p_{\text{TPM}} = p_{\text{GPT}}$, this formulation gives us the flexibility

to use the prior policy (i.e., $p_{\text{GPT}}(a_t^i|s_t, a_t^{<i})$) represented by more expressive autoregressive generative models.

As shown in Fig. 5.8 (middle), compared to using $p(a_t|s_t)$ (as done by TT), the inference-time optimality scores increase significantly when using the distribution specified by Eq. (5.10) (as done by Trifle) across various Gym-MuJoCo benchmarks.

...To the Multi-Step Case

Recall that when the labeled RTGs are suboptimal, our goal is to sample from $p(a_{t:t'}|s_t, \mathbb{E}[V_t^{\text{m}}] \geq v)$, where $V_t^{\text{m}} := \sum_{\tau=t}^{t'-1} r_\tau + \text{RTG}_{t'}$ is the multi-step value estimate. However, as shown in the second scenario in Section 5.3.2, it is hard even to evaluate the expected return of an action sequence due to the inability to marginalize out intermediate states $s_{t+1:t'}$. Empowered by PCs, we can solve this problem by computing the expectation efficiently as it can be broken down into computing conditional probabilities $p(r_\tau|s_t, a_{t:t'})(t \leq \tau < t')$ and $p(\text{RTG}_{t'}|s_t, a_{t:t'})$ (see Section D.3.2 for details):

$$\mathbb{E}\big[V_t^{\text{m}}\big] = \sum\nolimits_{\tau=t}^{t'-1} \mathbb{E}_{r_\tau \sim p(\cdot|s_t, a_{t:t'})}\big[r_\tau\big] + \mathbb{E}_{\text{RTG}_{t'} \sim p(\cdot|s_t, a_{t:t'})}\big[\text{RTG}_{t'}\big]. \tag{5.11}$$

We are now left with the same problem discussed in the single-step case – how to sample actions with high expected returns (i.e., $\mathbb{E}[V_t^{\text{m}}]$). Similar to Eq. (5.10), we add correction terms that bias the action (sequence) distribution towards high expected returns. Specifically, we augment the original action probability $\prod_{\tau=t}^{t'} p(a_\tau|s_t, a_{<\tau})$ with terms of the form $p(V_t^{\text{m}} \geq v|s_t, a_{\leq\tau})$ This leads to:

$$\tilde{p}(a_{t:t'}|s_t; v) := \prod\nolimits_{\tau=t}^{t'} \tilde{p}(a_\tau|s_t, a_{<\tau}; v),$$

where $\tilde{p}(a_\tau|s_t, a_{<\tau}; v) \propto p(a_\tau|s_t, a_{<\tau}) \cdot p(V_t^{\text{m}} \geq v|s_t, a_{\leq\tau})$, $a_{<\tau}$ and $a_{\leq\tau}$ represent $a_{t:\tau-1}$ and $a_{t:\tau}$, respectively.[11] In practice, while we compute $p(V_t^{\text{m}} \geq v|s_t, a_{\leq\tau})$ using the PC, $p(a_\tau|s_t, a_{<\tau}) =$

[11] We approximate $p(V_t^{\text{m}} \geq v|s_t, a_{\leq\tau})$ by assuming that the variables $\{r_t, \ldots, r_{t'-1}, \text{RTG}_{t'}\}$ are independent.

$\mathbb{E}_{s_{t+1:\tau}}[p(a_\tau|s_{\leq\tau}, a_{<\tau})]$ can either be computed exactly with the TPM or approximated (via Monte Carlo estimation over $s_{t+1:\tau}$) using an autoregressive generative model. In summary, we approximate samples from $p(a_{t:t'}|s_t, \mathbb{E}[V_t] \geq v)$ by first sampling from $\tilde{p}(a_{t:t'}|s_t; v)$, and then rejecting samples whose (predicted) expected return is smaller than v.

5.3.4 Practical Implementation

The previous section has demonstrated how to efficiently sample from the expected-value-conditioned policy (Eq. (5.9)). Based on this sampling algorithm, this section further introduces the proposed algorithm **Trifle** (**Tr**actable **I**nference for Off**fl**in**e** RL). The high-level idea of Trifle is to obtain good action (sequence) candidates from $p(a_t|s_t, \mathbb{E}[V] \geq v)$, and then use beam search to further single out the most rewarding action. Intuitively, by the definition in Eq. (5.9), the candidates are both rewarding and have relatively high likelihoods in the offline dataset, which ensures the actions are within the offline data distribution and prevents overconfident estimates during beam search.

Beam search maintains a set of N (incomplete) sequences each starting as an empty sequence. For ease of presentation, we assume the current time step is 0. At every time step t, beam search replicates each of the N actions sequences into $\lambda \in \mathbb{Z}^+$ copies and appends an action a_t to every sequence. Specifically, for every partial action sequence $a_{<t}$, we sample an action following $p(a_t|s_0, a_{<t}, \mathbb{E}[V_t] \geq v)$, where V_t can be either the single-step or the multi-step estimate depending on the task. Now that we have $\lambda \cdot N$ trajectories in total, the next step is to evaluate their expected return, which can be computed exactly using the PC (see Sec. 5.3.3). The N-best action sequences are kept and proceed to the next time step. After repeating this procedure for H time steps, we return the best action sequence. The first action in the sequence is used to interact with the environment. Please refer to Section D.3.2 for detailed descriptions of the algorithm.

Specifically, we first compute $\{p(r_\tau|s_t, a_{\leq\tau})\}_{\tau=t}^{t'-1}$ and $p(\mathrm{RTG}_{t'}|s_t, a_{\leq\tau})$, and then sum up the random variables assuming that they are independent. This introduces no error for deterministic environments and remains a decent approximation for stochastic environments.

Table 5.4: Normalized Scores on the standard Gym-MuJoCo benchmarks. The results of Trifle are averaged over 12 random seeds (For DT-base and DT-Trifle, we adopt the same number of seeds as [11]). Results of the baselines are acquired from their original papers.

Dataset	Environment	TT		TT(+Q)		DT		DD	IQL	CQL	%BC	TD3(+BC)
		base	Trifle	base	Trifle	base	Trifle					
Med-Expert	HalfCheetah	95.0±0.2	**95.1**±0.3	82.3±6.1	**89.9**±4.6	86.8±1.3	**91.9**±1.9	90.6	86.7	91.6	92.9	90.7
Med-Expert	Hopper	110.0±2.7	**113.0**±0.4	74.7±6.3	**78.5**±6.4	107.6±1.8	/	111.8	91.5	105.4	110.9	98.0
Med-Expert	Walker2d	101.9±6.8	**109.3**±0.1	109.3±2.3	**109.6**±0.2	108.1±0.2	**108.6**±0.3	108.8	109.6	108.8	109.0	110.1
Medium	HalfCheetah	46.9±0.4	**49.5**±0.2	48.7±0.3	**48.9**±0.3	42.6±0.1	**44.2**±0.7	49.1	47.4	44.0	42.5	48.3
Medium	Hopper	61.1±3.6	**67.1**±4.3	55.2±3.8	**57.8**±1.9	67.6±1.0	/	79.3	66.3	58.5	56.9	59.3
Medium	Walker2d	79.0±2.8	**83.1**±0.8	82.2±2.5	**84.7**±1.9	74±1.4	**81.3**±2.3	82.5	78.3	72.5	75.0	83.7
Med-Replay	HalfCheetah	41.9±2.5	**45.0**±0.3	48.2±0.4	**48.9**±0.3	36.6±0.8	**39.2**±0.4	39.3	44.2	45.5	40.6	44.6
Med-Replay	Hopper	91.5±3.6	**97.8**±0.3	83.4±5.6	**87.6**±6.1	82.7±7.0	/	100.0	94.7	95.0	75.9	60.9
Med-Replay	Walker2d	82.6±6.9	**88.3**±3.8	84.6±4.5	**90.6**±4.2	66.6±3.0	**73.5**±0.1	75.0	73.9	77.2	62.5	81.8
Average Score		78.9	**83.1**	74.3	77.4	74.7	/	81.8	77.0	77.6	74.0	75.3

Another design choice is the threshold value v. While it is common to use a fixed high return throughout the episode, we follow [44] and use an adaptive threshold. Specifically, at state s_t, we choose v to be the ϵ-quantile value of $p(V_t|s_t)$, which is computed using the PC.

5.3.5 Experiments

This section takes gradual steps to study whether Trifle can mitigate the inference-time suboptimality problem in different settings. First, in the case where the labeled RTGs are good performance indicators (i.e., the single-step case), we examine whether Trifle can consistently sample more rewarding actions (Sec. 5.3.5). Next, we further challenge Trifle in highly stochastic environments, where existing RvS algorithms fail catastrophically due to the failure to account for the environmental randomness (Sec. 5.3.5). Finally, we demonstrate that Trifle can be directly applied to safe RL tasks (with action constraints) by effectively conditioning on the constraints (Sec. 5.3.5). Collectively, this section highlights the potential of TPMs on offline RL tasks.

Comparison to the State of the Art

As demonstrated in Section 5.3.2 and Fig. 5.8, although the labeled RTGs in the Gym-MuJoCo [51] benchmarks are accurate enough to reflect the actual environmental return, existing RvS algorithms fail to effectively sample such actions due to their large and multi-dimensional action space. Fig. 5.8 (middle) has demonstrated that Trifle achieves better inference-time optimality. This section further examines whether higher inference-time optimality scores could consistently lead to better performance when building Trifle on top of different RvS algorithms, i.e., combining p_{TPM} (cf. Eq. (5.10)) with different prior policies p_{GPT} trained by the corresponding RvS algorithm.

Environment setup The Gym-MuJoCo benchmark suite collects trajectories in 3 locomotion environments (HalfCheetah, Hopper, Walker2D) and constructs 3 datasets (Medium-Expert, Medium, Medium-Replay) for every environment, which results in $3 \times 3 = 9$ tasks. For every environment, the main difference between the datasets is the quality of its trajectories. Specifically, the dataset "Medium" records 1 million steps collected from a Soft Actor-Critic (SAC) [59] agent. The "Medium-Replay" dataset adopts all samples in the replay buffer recorded during the training process of the SAC agent. The "Medium-Expert" dataset mixes 1 million steps of expert demonstrations and 1 million suboptimal steps generated by a partially trained SAC policy or a random policy. The results are normalized such that a well-trained SAC model hits 100 and a random policy has a 0 score.

Baselines We build Trifle on top of three effective RvS algorithms: Decision Transformer (DT) [11], Trajectory Transformer (TT) [68] as well as its variant TT(+Q) where the RTGs estimated by summing up future rewards in the trajectory are replaced by the Q-values generated by a well-trained IQL agent [84]. In addition to the above base models, we also compare Trifle against many other strong baselines: (i) Decision Diffuser (DD) [1], which is also a competitive RvS method; (ii) Offline TD learning methods IQL [84] and CQL [86]; (iii)

Imitation learning methods like the variant of BC [135] which only uses 10% of trajectories with the highest return, and TD3(+BC) [52].

Since the labeled RTGs are informative enough about the "goodness" of actions, we implement Trifle by adopting the single-step value estimate following Section 5.3.3, where we replace p_{GPT} with the policy of the three adopted base methods, i.e., $p_{\mathrm{TT}}(a_t|s_t)$, $p_{\mathrm{TT(+Q)}}(a_t|s_t)$ and $p_{\mathrm{DT}}(a_t|s_t)$.

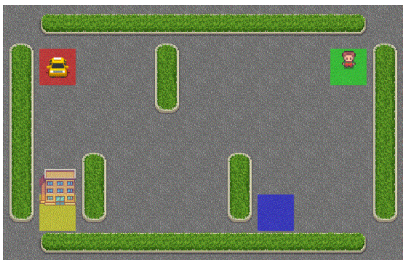
Empirical Insights Results are shown in Table 5.4.[12] First, to examine the benefit brought by TPMs, we compare Trifle with three base policies, as the main algorithmic difference is the use of the improved proposal distribution (Eq. (5.10)) for sampling actions. We can see that Trifle not only achieves a large performance gain over TT and DT in all environments, but also significantly outperforms TT(+Q) where we have access to more accurate labeled values, indicating that Trifle can enhance the inference-time optimality of base policy reliably and benefit from any improvement of the training-time optimality. See Section D.3.4 for more results and ablation studies.

Moreover, compared with all baselines, Trifle achieves the highest average score of 83.1. It also succeeds in achieving 7 state-of-the-art scores out of 9 benchmarks. We conduct further ablation studies on the rejection sampling component and the adaptive thresholding component (i.e., selecting v) in Section D.3.5.

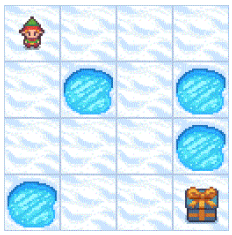
Evaluating Trifle in Stochastic Environments

This section further challenges Trifle on stochastic environments with highly suboptimal trajectories as well as labeled RTGs in the offline dataset. As demonstrated in Section 5.3.2, in this case, it is even hard to obtain accurate value estimates due to the stochasticity of transition dynamics.Section 5.3.3 demonstrates the potential of Trifle to more reliably estimate

[12]When implementing DT-Trifle, we have to modify the output layer of DT to make it combinable with TPM. Specifically, the original DT directly predicts deterministic action while the modified DT outputs categorical action distributions like TT. In the 3 unreported hopper environments, the modified DT fails to achieve the original DT scores.



(a)



(b)

Methods	Taxi	FrozenLake		
		$\epsilon = 0.3$	$\epsilon = 0.5$	$\epsilon = 0.7$
m-Trifle	**-57**	0.61	0.59	0.37
s-Trifle	-99	0.62	0.60	0.34
TT [68]	-182	0.63	0.25	0.12
DT [11]	-388	0.51	0.32	0.10
DoC [186]	-146	0.58	0.61	0.23

(c)

Figure 5.9: (a) Stochastic Taxi environment; (b) Stochastic FrozenLake Environment; (c) Average returns on the stochastic environment. All the reported numbers are averaged over 1000 trials.

and sample action sequences under suboptimal labeled RTGs and stochastic environments. This section examines this claim by comparing the five following algorithms:

(i) Trifle that adopts $V_t = \text{RTG}_t$ (termed single-step Trifle or **s-Trifle**); (ii) Trifle equipped with $V_t = \sum_{\tau=t}^{t'} r_\tau + \text{RTG}_{t'}$ (termed multi-step Trifle or **m-Trifle**; see Section D.3.4 for additional details); (iii) TT [68]; (iv) DT [11] (v) Dichotomy of Control (DoC) [186], an effective framework to deal with highly stochastic environments by designing a mutual information constraint for DT training, which is a representative baseline while orthogonal to our efforts.

We evaluate the above algorithms on two stochastic Gym environments: Taxi and FrozenLake. Here we choose the Taxi benchmark for a detailed analysis of whether and how Trifle could overcome the challenges discussed in Section 5.3.3. Among the first four algorithms, s-Trifle and DT do not compute the "more accurate" multi-step value, and TT approximates the value by Monte Carlo samples. Therefore, we expect their relative performance to be DT $\approx$ s-Trifle $<$ TT $<$ m-Trifle.

Environment setup We create a stochastic variant of the Gym-Taxi Environment [43]. As shown in Fig. 5.9a, a taxi resides in a grid world consisting of a passenger and a destination. The taxi is tasked to first navigate to the passenger's position and pick them up, and then drop them off at the destination.There are 6 discrete actions available at every step: (i) 4 navigation actions (North, South, East, or West), (ii) Pick-up, (iii) Drop-off. Whenever the

agent attempts to execute a navigation action, it has 0.3 *probability of moving toward a randomly selected unintended direction.* At the beginning of every episode, the location of the taxi, the passenger, and the destination are randomly initialized randomly. The reward function is defined as follows: (i) -1 for each action undertaken; (ii) an additional +20 for successful passenger delivery; (iii) -4 for hitting the walls; (iv) -5 for hitting the boundaries; (v) -10 for executing Pick-up or Drop-off actions unlawfully (e.g., executing Drop-off when the passenger is not in the taxi).

Following the Gym-MuJoCo benchmarks, we collect offline trajectories by running a Q-learning agent [181] in the Taxi environment and recording the first 1000 trajectories that drop off the passenger successfully, which achieves an average return of -128.

Empirical Insights We first examine the accuracy of estimated returns for s-Trifle, m-Trifle, and TT. DT is excluded since it does not explicitly estimate the value of action sequences. Fig. 5.10 illustrates the correlation between predicted and ground-truth returns of the three methods. First, s-Trifle performs the worst since it merely uses the inaccurate RTG_t to approximate the ground-truth return. Next, thanks to its ability to exactly compute the multi-step value estimates, m-Trifle outperforms TT, which approximates the multi-step value with Monte Carlo samples.

We proceed to evaluate their performance in the stochastic Taxi environment. As shown in Fig. 5.9c, the relative performance of the first four algorithms is DT $<$ TT $<$ s-Trifle $<$ m-Trifle, which largely aligns with the anticipated results. The only "surprising" result is the superior performance of s-Trifle compared to TT. One plausible explanation for this behavior is that while TT can better estimate the given actions, it fails to efficiently sample rewarding actions.

Notably, Trifle also significantly outperforms the strong baseline DoC, demonstrating its potential in handling stochastic transitions. To verify this, we further evaluate Trifle on the stochastic FrozenLake environment. Apart from fixing the stochasticity level $p = \frac{1}{3}$,[13] the

[13]When the agent takes an action, it has a probability p of moving in the intended direction and probability

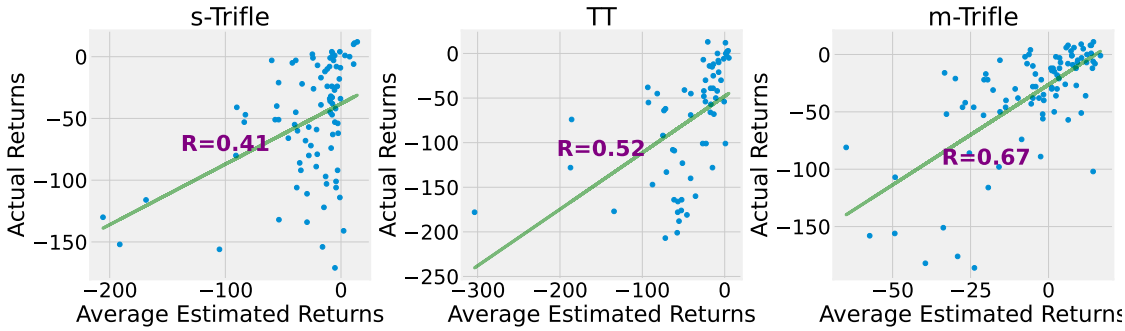


Figure 5.10: Correlation between average estimated returns and true environmental returns for s-Trifle (w/ single-step value estimates), TT, and m-Trifle (w/ multi-step value estimates) in the stochastic Taxi domain. R denotes the correlation coefficient. The results demonstrate that (i) multi-step value estimates (TT and m-Trifle) are better than single-step estimates (s-Trifle), and (ii) exactly computed multi-step estimates (m-Trifle) are better than approximated ones (TT) in stochastic environments.

Table 5.5: Normalized Scores on the Action-Space-Constrained Gym-MuJoCo Variants. The results of Trifle and TT are both averaged over 12 random seeds, with mean and standard deviations reported.

Dataset	Environment	Trifle	TT
Med-Expert	Halfcheetah	**81.9**$\pm$4.8	77.8$\pm$5.4
Med-Expert	Hopper	**109.6**$\pm$2.4	100.0$\pm$4.2
Med-Expert	Walker2d	**105.1**$\pm$2.3	103.6$\pm$4.9

experiment design follows the DoC paper [186]. For data collection, we perturb the policy of a well-trained DQN (with an average return of 0.7) with the ϵ-greedy strategy. Here ϵ is a proxy of offline dataset quality and varies from 0.3 to 0.7. As shown in Fig. 5.9c, when the offline dataset contains many successful trials ($\epsilon = 0.3$), all methods perform closely to the optimal policy. As the rollout policy becomes more suboptimal (with the increase of ϵ), the performances of DT and TT drop quickly, while Trifle still works robustly and outperforms all baselines.

Action-Space-Constrained Gym-MuJoCo Variants

This section demonstrates that Trifle can be readily extended to safe RL tasks by leveraging TPM's ability to compute conditional probabilities. Specifically, besides achieving high

$0.5(1-p)$ of slipping to either perpendicular direction.

expected returns, safe RL tasks require additional constraints on the action or states to be satisfied. Therefore, define the constraint as c, our goal is to sample actions from $p(a_t|s_t, \mathbb{E}[V_t] \geq v, c)$, which can be achieved by conditioning on c in the candidate action sampling process.

Environment setup In MuJoCo environments, each dimension of a_t represents the torque applied on a certain rotor of the hinge joints at timestep t. We consider action space constraints in the form of "value of the torque applied to the foot rotor $\leq A$", where $A = 0.5$ is a threshold value, for three MuJoCo environments: Halfcheetah, Hopper, and Walker2d. Note that there are multiple foot joints in Halfcheetah and Walker2d, so the constraint is applied to multiple action dimensions.[14] For all settings, we adopt the "Med-Expert" offline dataset as introduced in Section 5.3.5.

Empirical Insights The key challenge in these action-constrained tasks is the need to account for the constraints applied to other action dimensions when sampling the value of some action variable. For example, autoregressive models cannot take into account constraints added to variable a_t^{i+1} when sampling a_t^i. Therefore, while enforcing the action constraint is simple, it remains hard to simultaneously guarantee good performance. As shown in Table 5.5, owing to its ability to exactly condition on the action constraints, Trifle outperforms TT significantly across all three environments.

[14]We only add constraints to the front joints in the Halfcheetah environment since the performance degrades significantly for all methods if the constraint is added to all foot joints.

Chapter 6

Tractability Matters in Diffusion Models

The previous chapters offer a roadmap to building *tractable* generative models with PCs, which are capable of performing a wide range of probabilistic inference tasks exactly and efficiently. In this chapter, using diffusion models as an example, we further demonstrate that tractability is a crucial factor determining the effectiveness of generative models. Specifically, while discrete diffusion models have recently shown significant progress in modeling complex data, they still require hundreds or even thousands of denoising steps to perform well. In this chapter, we identify a fundamental limitation that prevents discrete diffusion models from achieving strong performance with fewer steps – they fail to capture dependencies between output variables at each denoising step. To address this issue, we provide a formal explanation and introduce a general approach to supplement the missing dependency information by incorporating another deep generative model, termed the *copula* model. Our method does not require fine-tuning either the diffusion model or the copula model, yet it enables high-quality sample generation with significantly fewer denoising steps. When we apply this approach to autoregressive copula models, the combined model outperforms both models individually in unconditional and conditional text generation. Specifically, the hybrid model achieves better (un)conditional text generation using 8 to 32 times fewer denoising steps than the diffusion

The contents of this section appeared in paper [94].

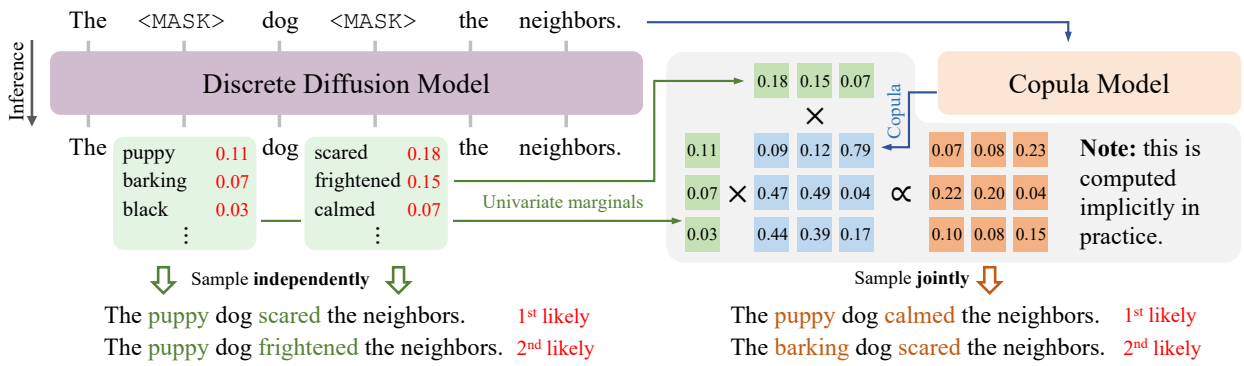


Figure 6.1: **Discrete Copula Diffusion (DCD).** At each denoising step, a partially completed sequence is given as input (top-left). The diffusion model independently predicts the univariate marginals for each masked token, which leads to the samples in the bottom left. DCD introduces an additional copula model (top-right) to capture the inter-variable dependencies, thereby supplementing the information missed by the diffusion model. By combining outputs from both models in a principled way, DCD achieves better performance than either model individually (see improved samples in the bottom-right), enabling few-step discrete diffusion generation.

model alone. In addition to presenting an effective discrete diffusion generation algorithm, this paper emphasizes the importance of modeling inter-variable dependencies in discrete diffusion.

6.1 Background and Motivation

Discrete diffusion models have recently achieved significant progress in modeling complex data such as natural languages [10, 153], protein sequences [58, 121], and graphs [65, 178]. In particular, recent discrete diffusion models for text generation [105, 153, 157] have matched or even surpassed the performance of autoregressive models at the scale of GPT-2 [140]. Additionally, discrete diffusion models offer improved inference-time controllability using guidance from auxiliary models such as classifiers [41], making them suitable for controlled generation tasks [60, 90].

Despite these promising results, discrete diffusion models still require hundreds to thousands of denoising steps to produce high-quality samples [3, 153], significantly affecting their

efficiency. In this paper, we identify a fundamental limitation in most discrete diffusion models that hinders their ability to generate high-quality samples in just a few steps.

We illustrate the problem in Fig. 6.1. At each denoising step, a partially completed sample shown in the top-left is fed into a sequence-to-sequence denoising model, which predicts the univariate marginal distributions for each masked token independently. A new output sequence is then sampled based on these univariate marginals before proceeding to the next denoising step. The key issue with this process is that when multiple "edits" (i.e., replacing masked tokens with data tokens) are made simultaneously, the model does not account for the joint probability of these changes occurring together. As a result, the generated samples often lack coherence, as shown in the bottom-left of Fig. 6.1. This problem is exacerbated in few-step generation, where many tokens must be edited simultaneously. We formally demonstrate that if the diffusion model predicts each variable independently, an irreducible term (in addition to the data entropy) remains in the negative evidence lower bound (ELBO), preventing the model from perfectly capturing the data distribution.

We propose using a generative model, which we refer to as the copula model, to compensate for the missing dependency information between output variables at each denoising step. Our method operates only at inference time and can be adapted to any discrete diffusion model and a wide range of copula models. As illustrated on the right side of Fig. 6.1, the input sequence is also fed into a copula model that (implicitly) produces information on inter-variable dependencies. This information is combined with the univariate marginals predicted by the diffusion model to produce a more accurate distribution, resulting in higher-quality samples, shown in the bottom-right corner.

We formally show that the univariate marginals from the diffusion model and the dependencies captured by the copula model can be combined in a principled way, leading to a better approximation of the true denoising distribution under mild assumptions. Further, finding this combined distribution reduces to solving a convex optimization problem that can be efficiently approximated in practice.

By instantiating the copula model as an autoregressive deep generative model such as GPT [140], we propose an algorithm that combines any pretrained discrete diffusion model with an autoregressive model to form a hybrid model called **D**iscrete **C**opula **D**iffusion (DCD). This model is capable of producing high-quality (un)conditional samples with only a few denoising steps. Empirical results on text and antibody generation show that DCD significantly outperforms both of its base models. Moreover, DCD achieves comparable or better performance using 8 to 32 times fewer denoising steps compared to the base discrete diffusion model. In addition to proposing a discrete diffusion model capable of few-step generation, we emphasize the importance of modeling inter-variable dependencies in discrete diffusion models.

6.2 Preliminaries

We aim to model the joint distribution of variables $\mathbf{X}_0$, a set of categorical variables with C categories. Discrete diffusion models [3] learn to sample from $p(\mathbf{X}_0)$ by modeling the reversal of the following noising process involving $\mathbf{X}_0$ and a set of auxiliary variables $\{\mathbf{X}_t\}_{t=1}^T$:

$$\forall t \in \{1, \dots, T\} \quad q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1}) := \mathrm{Cat}(\boldsymbol{x}_t; Q_t \cdot \boldsymbol{x}_{t-1}), \tag{6.1}$$

where $\mathrm{Cat}(\boldsymbol{x}; \mathbf{p})$ refers to the Categorical distribution over $\boldsymbol{x}$ with class probabilities $\mathbf{p}$, and Q_t is a $C \times C$ transition matrix that is applied independently to every variable x_{t-1}^i (denote x_{t-1}^i as the ith variable of $\boldsymbol{x}_{t-1}$) to get the corresponding categorical distribution of x_t^i. Specifically, each variable x_{t-1}^i is treated as a one-hot vector of size $C \times 1$, which is then multiplied by Q_t to compute the class probabilities of x_t^i. The noising process is designed such that $p(\boldsymbol{x}_T)$ follows a simple distribution regardless of the data distribution.

Instead of using a fixed number of predefined time steps, we can treat t as a continuous variable within the range $[0, T]$. The noising process is now defined by the rate of change of $p(\boldsymbol{x}_t)$ w.r.t. t: $\frac{dp(\boldsymbol{x}_t)}{dt} = Q \cdot p(\boldsymbol{x}_t)$, where $Q \in \mathbb{R}^{C \times C}$ is a transition rate matrix. For any

$0 \leq s < t \leq T$, we have

$$q(\boldsymbol{x}_t|\boldsymbol{x}_s) := \mathrm{Cat}(\boldsymbol{x}_t; \exp((t-s)\cdot Q)\cdot \boldsymbol{x}_s),$$

where $\exp(\cdot)$ denotes the matrix exponential.

Discrete diffusion models represent the reverse diffusion process as a Markov chain from $\boldsymbol{x}_T$ to $\boldsymbol{x}_0$, effectively reversing the noising process. Specifically, the reverse diffusion is modeled as:

$$p_\theta(\boldsymbol{x}_{0:T}) := p(\boldsymbol{x}_T) \prod_{t=0}^{T-1} p_\theta(\boldsymbol{x}_t|\boldsymbol{x}_{t+1}).$$

In the discrete-time framework, the model is trained by maximizing the ELBO, which is defined by the forward joint distribution ($q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)p(\boldsymbol{x}_0)$) and the reverse joint distribution ($p_\theta(\boldsymbol{x}_{0:T})$) [63]. In the continuous-time framework, we can either adopt an extended ELBO objective [194] or to learn the likelihood ratios $\{p(\boldsymbol{x}'_t)/p(\boldsymbol{x}_t)\}_{\boldsymbol{x}_t,\boldsymbol{x}'_t}$, allowing for the recovery of $p(\boldsymbol{x}_s|\boldsymbol{x}_t)$ $(s < t)$ in an indirect manner [105, 113, 165]. Following the reverse diffusion process, sampling from a diffusion model involves first sampling from the prior $p(\boldsymbol{x}_T)$ and then recursively sampling $\boldsymbol{x}_{T-1}, \ldots, \boldsymbol{x}_0$ following $\{p_\theta(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})\}_{t=0}^{T-1}$.

6.3 Challenge of Modeling Variable Dependencies

Unlike continuous diffusion models, which can produce high-quality samples with just a few steps (e.g., [164, 196]), discrete diffusion models exhibit a strong positive correlation between sample quality and the number of denoising steps. For instance, to generate 1024 text tokens, a recent discrete diffusion model SEDD [105] requires 1024 steps to reach around 35 perplexity (PPL), while with 32 denoising steps the PPL is only around 130.

We argue that the need for a large number of sampling steps in discrete diffusion models stems from their inability to capture inter-variable dependence among the outputs. Specifically,

at each time step t, discrete diffusion models independently sample each variable from $\boldsymbol{x}_t$ conditioned on $\boldsymbol{x}_{t+1}$, i.e., $p(\boldsymbol{x}_t|\boldsymbol{x}_{t+1}) := \prod_i p(x_t^i|\boldsymbol{x}_{t+1})$. As a result, when changing multiple variables from $\boldsymbol{x}_{t+1}$ to $\boldsymbol{x}_t$, the model fails to account for the joint probability of these modifications happening together. In the following, we first quantitatively analyze the performance degradation caused by this independent denoising assumption. We then discuss approaches to mitigate this issue.

Quantifying the Performance Drop. The total correlation of a distribution $p(\mathbf{X})$ is the KL-divergence between itself and the product of its univariate marginals:

$$\mathrm{D_{TC}}(p(\mathbf{X})) := \sum_{\boldsymbol{x}} p(\boldsymbol{x}) \log \Big(p(\boldsymbol{x}) / \prod_i p(x_i) \Big).$$

The following result demonstrates that, under the independent denoising assumption, there is an irreducible component in the ELBO that directly stems from ignoring inter-variable dependencies.

Proposition 3. *Assume the denoising distributions* $\{p_\theta(\boldsymbol{x}_t|\boldsymbol{x}_{t+1})\}_{t=0}^{T-1}$ *are fully factorized. Let* $\mathrm{H}(p(\mathbf{X}))$ *denote the entropy of* $p(\mathbf{X})$*. For any choice of denoising distributions (or equivalently, any parameterization* θ*), the negative ELBO of the diffusion model is lower bounded by*

$$\mathrm{H}(p(\mathbf{X}_0)) + \sum_{t=1}^{T} \mathrm{D_{TC}}(q(\mathbf{X}_{t-1}|\mathbf{X}_t)), \qquad \textit{where } \mathrm{D_{TC}}(p(\mathbf{Y}|\mathbf{X})) := \mathbb{E}_{\boldsymbol{x}\sim p}\big[\mathrm{D_{TC}}(p(\mathbf{Y}|\boldsymbol{x}))\big]. \quad (6.2)$$

Proofs of this and the following theoretical results in this chapter are provided in Section E.1. The first term represents the entropy of the data distribution and is irreducible. The second term additionally depends on the noising process and the chosen noise levels, which set an upper limit on the performance of discrete diffusion models that use the independent denoising assumption. Note that although $\mathrm{D_{TC}}(q(\mathbf{X}_t|\mathbf{X}_{t-1}))$ is zero according to the definition of the noising process, $\mathrm{D_{TC}}(q(\mathbf{X}_{t-1}|\mathbf{X}_t))$ is not unless the data distribution is fully factorized.

Closing the Performance Gap. While increasing the number of denoising steps

can improve sample quality, it also introduces significant computational overhead during inference. Our goal is to use fewer denoising steps while maintaining good sample quality. As shown in Proposition 3, given a fixed noising strategy and the number of denoising steps, the only way to reduce the negative ELBO lower bound in Eq. (6.2) is to relax the independent denoising assumption. That is, in addition to modeling the univariate marginals, we must also account for dependencies between variables.

The challenge of capturing inter-variable dependencies during each denoising step can be addressed through adjustments during either training or inference. A direct approach involves modeling both the univariate marginals and the inter-variable dependencies within the diffusion model. However, this requires improving existing sequence-to-sequence architectures (e.g., [39]) to capture dependencies between output variables directly, which is not very well studied in the literature.

Instead, we propose an inference-time solution that complements the information missed by the pretrained discrete diffusion model. Specifically, we aim to combine the univariate marginals produced by the diffusion model with the inter-variable dependencies learned by another (possibly smaller) deep generative model, which we refer to as the *copula model*. The term "copula" traditionally refers to the dependencies between random variables in statistics [123].

6.4 Modeling Variable Dependencies with Copula Models

As motivated in the previous section, our main goal is to combine the univariate marginals produced by the diffusion model with the inter-variable dependencies captured by a copula model. In this section, we first formalize the concept of "combining" two such distributions in a general context (Sec. 6.4.1). We then specialize the formulation to the case of diffusion models (Sec. 6.4.2).

6.4.1 Combining Univariate Marginals with Inter-Variable Dependencies

In this section, we discuss how to best inject inter-variable dependence using copula models given a target distribution p_{tar} over $\mathbf{X}$. Assume we have access to p_{tar} through two sources: (i) the set of all univariate marginal distributions $\{p_{\text{tar}}(X_i)\}_i$, and (ii) an estimate p_{est} of the target distribution coming from the copula model, which is also a generative model. Our goal is to combine these two estimates to construct $\hat{p}$ that is "closer" to the true distribution p_{tar} than either estimate individually.

We construct $\hat{p}$ as the distribution that (i) matches the set of univariate marginals $\{p_{\text{tar}}(X_i)\}_i$, and (ii) minimizes the KL divergence to p_{est}. The intuition is that by ensuring $\hat{p}$ has the correct univariate marginals, we can achieve a good approximation of p_{tar} even if p_{est} is biased. To formalize this, we first define information projection (I-projection).

Definition 10. The I-projection of a distribution $q(\mathbf{X})$ onto a set of distributions $\mathcal{P}$ over $\mathbf{X}$ is

$$p^* = \arg\min_{p \in \mathcal{P}} \mathrm{D}_{\mathrm{KL}}(p \parallel q).$$

Let $\mathcal{P}^p_{\text{mar}}$ denote the set of distributions over $\mathbf{X}$ that share the same univariate marginals as p. We define $\hat{p}$ as the I-projection of p_{est} onto $\mathcal{P}^{p_{\text{tar}}}_{\text{mar}}$. The following proposition shows that regardless of the initial estimate p_{est} of p_{tar}, the I-projection $\hat{p}$ will be an improved estimate of p_{tar} in KL-divergence.

Proposition 4. *If there exists i and x_i s.t. $p_{\text{tar}}(x_i) \neq p_{\text{est}}(x_i)$, then* $\mathrm{D}_{\mathrm{KL}}(p_{\text{tar}} \| \hat{p}) < \mathrm{D}_{\mathrm{KL}}(p_{\text{tar}} \| p_{\text{est}})$.

Having now seen that $\hat{p}$ is an improved estimate of p_{tar}, we next explore whether it is feasible to compute $\hat{p}$ given $\{p_{\text{tar}}(X_i)\}_i$ and p_{est}. We start by showing that $\hat{p}$ has a simple form.

Proposition 5. *Assume* $\forall \boldsymbol{x}$, $p_{\text{tar}}(\boldsymbol{x}) > 0$ *and* $p_{\text{est}}(\boldsymbol{x}) > 0$. *Then* $\hat{p}$ *exists and has the form*

$$\hat{p}(\boldsymbol{x}) = p_{\text{est}}(\boldsymbol{x}) \cdot \prod_i \sigma_i(x_i),$$

where σ_i *is a positive function that depends on* x_i.

Assume $\mathbf{X}$ consists of N categorical variables, each with C categories, we can represent the factors $\{\sigma_i\}_i$ using a matrix $\mathbf{V} \in \mathbb{R}^{N \times C}$. Under this representation, the combined distribution is

$$\hat{p}(\boldsymbol{x}) = p_{\text{est}}(\boldsymbol{x}) \cdot \prod_i \exp(\mathbf{V}[i, x_i]), \tag{6.3}$$

where $\mathbf{V}[i, j]$ denotes the element at the ith row and jth column of $\mathbf{V}$ and $\mathbf{V}[i, x_i] = \log \sigma_i(x_i)$. Determining the true matrix $\mathbf{V}^*$ corresponding to $\hat{p}$, which is the I-projection of p_{est} onto $\mathcal{P}_{\text{mar}}^{p_{\text{tar}}}$, can be reformulated as solving the following convex optimization problem.

Theorem 15. *If* $\mathbf{V}^*$ *minimizes the following convex objective function, then the corresponding* $\hat{p}$ *defined by Eq.* (6.3) *is the I-projection of* p_{est} *onto* $\mathcal{P}_{\text{mar}}^{p_{\text{tar}}}$.[1]

$$\mathcal{L}(\mathbf{V}; p_{\text{tar}}, p_{\text{est}}) := \sum_{\boldsymbol{x}} p_{\text{est}}(\boldsymbol{x}) \cdot \prod_i \exp(\mathbf{V}[i, x_i]) - \sum_{i=1}^{N} \sum_{x_i=1}^{C} \mathbf{V}[i, x_i] \cdot p_{\text{tar}}(x_i). \tag{6.4}$$

We proceed to explain why I-projecting p_{est} leads to a better estimate of p_{tar} as suggested by Proposition 4. In general, a joint distribution can be viewed as combining two independent pieces of information: (i) a set of univariate marginal distributions and (ii) a *copula* describing the association or dependence among the variables. By the classical work of [162], for continuous variables the copula can take the form of a joint distribution with uniform margins and can be combined quite simply with univariate marginal distributions to recover the full joint distribution, a fact heavily exploited in statistics [123]. While the discrete case is somewhat less straightforward, recent work of [53] has developed the fundamental notions of

[1] Eq. (6.4) closely resembles the matrix scaling problem [67]. See Section E.2 for details.

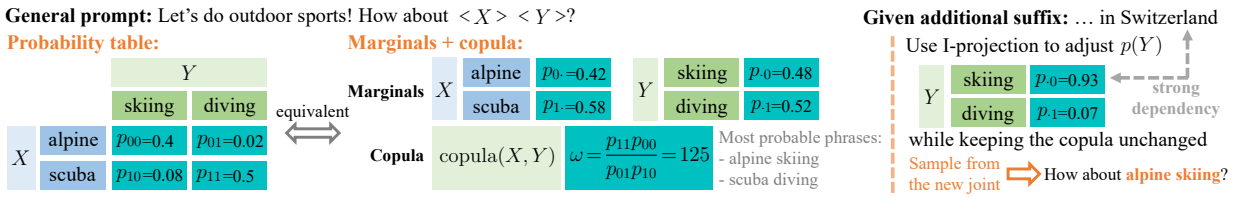


Figure 6.2: Illustration of the decomposition of a distribution into univariate marginals and a copula.

discrete copula modeling as well, where the information of a copula can be parameterized by odds ratios.

Fig. 6.2 shows an example consisting of two binary variables X and Y. The probability table on the left can be equivalently expressed using univariate marginals (i.e., $p_{0\cdot}, p_{1\cdot}$, $p_{\cdot 0}$, $p_{\cdot 1}$) and the odds ratio (i.e., copula) $\omega := \frac{p_{00}p_{11}}{p_{01}p_{10}}$ as shown in the middle of Fig. 6.2. Intuitively, $\omega = 125$ indicates that the phrases "alpine skiing" and "scuba diving" are more likely than others (e.g., "alpine diving"), and the marginals decide which of the two phrases appears more frequently. The idea of representing the copula with odds ratios generalizes to the multivariate case and is presented in Section E.3.

The following result demonstrates that, under its functional form in Eq. (6.3), I-projecting p_{est} onto $\mathcal{P}_{\text{mar}}^{p_{\text{tar}}}$ only improves the univariate marginals and leaves the copula unchanged regardless of $\mathbf{V}$.

Proposition 6. *For a positive distribution p and any $\mathbf{V} \in \mathbb{R}^{N \times C}$, the distribution $q(\boldsymbol{x}) \propto p(\boldsymbol{x}) \cdot \prod_i \exp(\mathbf{V}[i, x_i])$ has the same copula as p.*

In general, Proposition 6 holds because scaling factors (e.g., $\exp(\mathbf{V}[i, x_i])$) cancel in odds ratios. For example, in the 2×2 case in Fig. 6.2, scaling the top row of the probability table by a would result in the odds ratio $\omega = \frac{a p_{00} p_{11}}{a p_{01} p_{10}} = \frac{p_{00}p_{11}}{p_{01}p_{10}}$.

6.4.2 Modeling Dependence in Discrete Diffusion Models

Recall from Section 6.3 that our goal is to capture inter-variable dependencies between the output variables at each denoising step (e.g., sampling $\boldsymbol{x}_t$ from $q(\mathbf{X}_t | \boldsymbol{x}_{t+1})$). Similar to the

general case shown in Section 6.4.1, we first have a set of univariate marginals $\{p_{\text{dm}}(X_t^i|\boldsymbol{x}_{t+1})\}_i$ from the diffusion model. Notably, these univariate marginals are fairly accurate since for both discrete-time and continuous-time diffusion models, if their respective training losses are minimized, the model recovers the true univariate marginals. This is formally justified in Section E.4.

Alongside the univariate marginals, we assume access to a copula model that encodes a distribution over $\mathbf{X}_t$. Following Section 6.4.1, combining the copula model's distribution with the univariate marginals from the diffusion model will lead to an improved estimate of $q(\mathbf{X}_t|\boldsymbol{x}_{t+1})$ (Prop. 4).

The performance of the augmented diffusion model hinges on two key questions: (i) how well can the copula model capture the inter-variable dependencies in $q(\mathbf{X}_t|\boldsymbol{x}_{t+1})$ (defined by the data distribution and the noising process); (ii) given a good copula distribution, how to effectively combine it with the univariate marginals obtained from the diffusion model, i.e., how to solve Eq. (6.4).

6.5 Autoregressive Models as Copula Models

This section answers the two questions above tailored to the case where the copula model is an autoregressive model such as GPT [140] and State Space Models [33]. Specifically, Section 6.5.1 discusses how to approximate $q(\mathbf{X}_t|\boldsymbol{x}_{t+1})$ using an autoregressive model trained on the clean data distribution $p(\mathbf{X}_0)$ under certain noising processes. Section 6.5.2 explores the process of performing I-projection from the (autoregressive) copula distribution onto the set of distributions with univariate marginals $\{p_{\text{dm}}(X_t^i|\boldsymbol{x}_{t+1})\}_i$. Finally, Section 6.5.3 summarizes the sampling procedure with a discrete diffusion model and an autoregressive copula model.

6.5.1 Extracting Copula Distributions from Autoregressive Models

At step t, to sample $\boldsymbol{x}_t$ conditioned on $\boldsymbol{x}_{t+1}$, we need a copula distribution $p_{\text{copula}}(\mathbf{X}_t)$ that closely approximates $q(\mathbf{X}_t|\boldsymbol{x}_{t+1})$. While this might suggest that the copula model should also be trained with a diffusion model objective, which brings us back to the problem of modeling inter-variable dependencies, we show that any model trained on the clean data distribution can serve as a copula model that indirectly approximates $q(\mathbf{X}_t|\boldsymbol{x}_{t+1})$ under the absorbing mask forward noising process.

The absorbing mask noising process gradually converts data tokens in $\boldsymbol{x}_0 \sim p(\mathbf{X}_0)$ to a new category denoted `<MASK>` through the sequence $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T$. Specifically, each token in $\boldsymbol{x}_0$ is independently converted to `<MASK>` with probabilities $0 < \alpha_1 < \ldots < \alpha_T = 1$ in $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T$, respectively. This is a widely used noising strategy for discrete diffusion models. Since this process only transforms data tokens into the mask token, it preserves the dependencies between the remaining unmasked tokens. Therefore, we can decompose $q(\mathbf{X}_t|\boldsymbol{x}_{t+1})$ as $q(\boldsymbol{x}_t|\boldsymbol{x}_{t+1}) = \sum_{\tilde{\boldsymbol{x}}_t} q(\tilde{\boldsymbol{x}}_t|\boldsymbol{x}_{t+1}) q(\boldsymbol{x}_t|\tilde{\boldsymbol{x}}_t, \boldsymbol{x}_{t+1})$, where $q(\tilde{\boldsymbol{x}}_t|\boldsymbol{x}_{t+1})$ is inuitively capturing the joint distribution of generating all currently masked tokens, and $q(\boldsymbol{x}_t|\tilde{\boldsymbol{x}}_t, \boldsymbol{x}_{t+1})$ captures only the choice of which currently masked tokens will actually be generated. Formally, define I as the set of variables i such that $x^i_{t+1} = $ `<MASK>` and J as its complement. The auxiliary distributions have the following form.

Proposition 7. *Assume $p(\mathbf{X}_0)$ is the clean data distribution and $\{q(\mathbf{X}_t|\boldsymbol{x}_{t-1})\}_{t=1}^T$ follows the absorbing mask noising process. Let α_t be the probability of conversion to the mask state from X_0^i to X_t^i ($\forall i$). Define $\tilde{\mathbf{X}}_t$ as a set of auxiliary variables such that*

$$q(\tilde{\boldsymbol{x}}_t|\boldsymbol{x}_{t+1}) = p(\mathbf{X}_0^I = \tilde{\boldsymbol{x}}_t^I | \mathbf{X}_0^J = \boldsymbol{x}_{t+1}^J) \cdot \mathbb{1}[\tilde{\boldsymbol{x}}_t^J = \boldsymbol{x}_{t+1}^J]. \tag{6.5}$$

Then, the distribution $q(\mathbf{X}_t|\tilde{\boldsymbol{x}}_t, \boldsymbol{x}_{t+1})$ is the following: $q(\mathbf{X}_t|\tilde{\boldsymbol{x}}_t, \boldsymbol{x}_{t+1}) = \prod_i q(x_t^i|\tilde{x}_t^i, x_{t+1}^i)$.

– For $i \in I$, $q(x_t^i|\tilde{x}_t^i, x_{t+1}^i)$ equals α_t/α_{t+1} if $x_t^i =$ `<MASK>` and equals $1 - \alpha_t/\alpha_{t+1}$ if $x_t^i = \tilde{x}_t^i$.

– For $i \in J$, $q(x_t^i|\tilde{x}_t^i, x_{t+1}^i) = 1$ if and only if $x_t^i = x_{t+1}^i$.

Since $q(\mathbf{X}_t|\tilde{\boldsymbol{x}}_t, \boldsymbol{x}_{t+1})$ is fully factorized, the copula model only needs to account for inter-variable dependencies in $q(\tilde{\mathbf{X}}_t|\boldsymbol{x}_{t+1})$. Following Eq. (6.5), we can transform $p_{\text{copula}}(\mathbf{X}_0)$, which estimates the clean data distribution, into $p_{\text{copula}}(\tilde{\mathbf{X}}_t|\boldsymbol{x}_{t+1})$ that approximates $q(\tilde{\mathbf{X}}_t|\boldsymbol{x}_{t+1})$ by conditioning it on the unmasked tokens in $\boldsymbol{x}_{t+1}$ (i.e., $\boldsymbol{x}^J_{t+1}$). Specifically, for autoregressive copula models (i.e., $p_{\text{copula}}(\boldsymbol{x}) := \prod_i p_{\text{copula}}(x_i|\boldsymbol{x}_{<i})$), we construct $p_{\text{copula}}(\tilde{\mathbf{X}}_t|\boldsymbol{x}_{t+1})$ by conditioning each variable on the corresponding preceding tokens in $\boldsymbol{x}^J_{t+1}$ while enforcing $\tilde{x}^j_t = x^j_{t+1}$ $(\forall j \in J)$:

$$p_{\text{copula}}(\tilde{\boldsymbol{x}}_t|\boldsymbol{x}_{t+1}) := \prod_{i\in I} p_{\text{copula}}(X^i_0 = \tilde{x}^i_t|\mathbf{X}^{<i}_0 = \tilde{\boldsymbol{x}}^{<i}_t) \cdot \prod_{j\in J} \mathbb{1}[\tilde{x}^j_t = x^j_{t+1}]. \tag{6.6}$$

This copula distribution is biased even if the autoregressive model perfectly captures the data distribution since it cannot condition on subsequent unmasked tokens in $\boldsymbol{x}_{t+1}$. In contrast, while being able to condition on all unmasked tokens, diffusion models cannot capture dependence between variables. Combining the two estimates in a proper way will lead to better empirical performance.

Continuing with the example in Fig. 6.2, we assume an autoregressive copula model encodes the probability table on the left. As shown on the right, when provided with the suffix prompt "in Switzerland", the copula model alone cannot adjust its probabilities, as it can only condition on prefix prompts. However, a diffusion model that captures the strong dependence between "Switzerland" and $Y =$"skiing" can, through I-projection, set the correct marginal probabilities of Y, while keeping the copula unchanged. This allows the model to reliably generate "how about alpine skiing."

Lastly, we need the univariate marginals of $q(\tilde{\mathbf{X}}_t|\boldsymbol{x}_{t+1})$, which can be derived by renormalizing $\{q(X^i_t|\boldsymbol{x}_{t+1})\}_i$ to zero out the probability of the mask state according to the following result.

Proposition 8. *For each i and data category $c \neq$ <MASK>, $q(\tilde{X}^i_t = c|\boldsymbol{x}_{t+1}) \propto q(X^i_t = c|\boldsymbol{x}_{t+1})$.*

As a result, for each i, the distribution $p_{\text{dm}}(\tilde{X}^i_t|\boldsymbol{x}_{t+1})$ can be similarly obtained by

Algorithm 8 Draw samples from a discrete diffusion model with the help of a copula model

Inputs: a diffusion model p_{dm}, a copula model p_{copula}, number of time steps T
Outputs: a sample $\boldsymbol{x}_0$ from the discrete diffusion model augmented by the copula model
Initialize: Sample $\boldsymbol{x}_T$ from the prior noise distribution $p(\mathbf{X}_T)$
for $t = T-1$ **to** 0
  Compute $\{p_{\text{dm}}(\tilde{X}_t^i|\boldsymbol{x}_{t+1})\}_i$ and $\{p_{\text{dm}}(\tilde{X}_t^i|\boldsymbol{x}_{t+1}^{<i})\}_i$ using the diffusion model
  Compute $\mathbf{V}[i,\tilde{x}_t^i] = \log p_{\text{dm}}(\tilde{x}_t^i|\boldsymbol{x}_{t+1}) - \log p_{\text{dm}}(\tilde{x}_t^i|\boldsymbol{x}_{t+1}^{<i})$ $(\forall i, \tilde{x}_t^i)$ following Eq. (6.10)
  Sample $\tilde{\boldsymbol{x}}_t$ from $\hat{p}(\tilde{\boldsymbol{x}}_t|\boldsymbol{x}_{t+1}) \propto p_{\text{copula}}(\tilde{\boldsymbol{x}}_t|\boldsymbol{x}_{t+1}) \cdot \prod_i \exp(\mathbf{V}[i,\tilde{x}_t^i])$ (p_{copula} is defined by Eq. (6.6))
  Sample $\boldsymbol{x}_t$ from $q(\mathbf{X}_t|\tilde{\boldsymbol{x}}_t, \boldsymbol{x}_{t+1})$ (defined in Proposition 7)

renormalizing $p_{\text{dm}}(X_t^i|\boldsymbol{x}_{t+1})$, which is directly obtained from the denoising model, to exclude the mask state.

6.5.2 Approximate I-Projection with Autoregressive Models

Given univariate marginals $\{p_{\text{dm}}(\tilde{X}_t^i|\boldsymbol{x}_{t+1})\}_i$ and an autoregressive copula distribution $p_{\text{copula}}(\tilde{\mathbf{X}}_t|\boldsymbol{x}_{t+1})$, both of which estimate the target distribution $q(\tilde{\mathbf{X}}_t|\boldsymbol{x}_{t+1})$, our goal is to combine them following the I-projection procedure described in Section 6.4.1. Specifically, this involves solving the convex optimization problem in Eq. (6.4), which is specialized to the following:

$$\sum_{\tilde{\boldsymbol{x}}_t} p_{\text{copula}}(\tilde{\boldsymbol{x}}_t|\boldsymbol{x}_{t+1}) \cdot \prod_i \exp(\mathbf{V}[i,\tilde{x}_t^i]) - \sum_{i=1}^{N} \sum_{\tilde{x}_t=1}^{C} \mathbf{V}[i,\tilde{x}_t^i] \cdot p_{\text{dm}}(\tilde{x}_t^i|\boldsymbol{x}_{t+1}). \tag{6.7}$$

Following Theorem 15, if $\mathbf{V}$ minimizes Eq. (6.7), then the distribution defined by $\hat{p}(\tilde{\boldsymbol{x}}_t|\boldsymbol{x}_{t+1}) = p_{\text{copula}}(\tilde{\boldsymbol{x}}_t|\boldsymbol{x}_{t+1}) \cdot \prod_i \exp(\mathbf{V}[i,\tilde{x}_t^i])$ is the I-projection of $p_{\text{copula}}(\tilde{\boldsymbol{x}}_t|\boldsymbol{x}_{t+1})$ onto the set of distributions with the univariate marginals $\{p_{\text{dm}}(\tilde{X}_t^i|\boldsymbol{x}_{t+1})\}_i$, which is the desired combined distribution.

Consider initializing all coefficients in $\mathbf{V}$ to zero, i.e., $\hat{p}(\tilde{\boldsymbol{x}}_t|\boldsymbol{x}_{t+1}) = p_{\text{copula}}(\tilde{\boldsymbol{x}}_t|\boldsymbol{x}_{t+1})$. For each row i, if we only optimize the values $\mathbf{V}[i,:]$ and fix the rest to zero, the optimal coefficients are

$$\forall c,\ \mathbf{V}[i,c] = \log p_{\text{dm}}(\tilde{X}_t^i = c|\boldsymbol{x}_{t+1}) - \log p_{\text{copula}}(\tilde{X}_t^i = c|\boldsymbol{x}_{t+1}). \tag{6.8}$$

We approximate the solution to Eq. (6.7) by applying the above update (Eq. (6.8)) to each row in $\mathbf{V}$ independently, as it strikes a proper balance between efficiency and empirical performance.

While the first term on the right-hand side of Eq. (6.8) can be acquired from the diffusion model, the second term is not accessible through the copula model. Plug in the definition in Eq. (6.6), the required marginal probabilities can be written as (for $j \in J$, $p_{\text{copula}}(\tilde{x}_t^j | \boldsymbol{x}_{t+1}) = 1$ iff $\tilde{x}_t^j = x_{t+1}^j$)

$$\forall i \in I,\ p_{\text{copula}}(\tilde{x}_t^i | \boldsymbol{x}_{t+1}) = p_{\text{copula}}(X_i = \tilde{x}_t^i | \mathbf{X}_{K_i} = \boldsymbol{x}_{t+1}^{K_i}),\ \text{where } K_i = \{j : j \in J \text{ and } j < i\}. \quad (6.9)$$

The above probabilities cannot be computed from the autoregressive model since we need to "marginalize out" preceding tokens that are not in K_i (i.e., those not given as evidence in $\boldsymbol{x}_{t+1}$). However, these terms can be estimated using the diffusion model. Assume both the diffusion model and the autoregressive model perfectly encode the data distribution. According to Proposition 8, the diffusion model computes $p_{\text{dm}}(\tilde{X}_t^i | \boldsymbol{x}_{t+1}) = q(\tilde{X}_t^i | \boldsymbol{x}_{t+1})$. Comparing it to Eq. (6.9), which gives $p_{\text{copula}}(\tilde{X}_t^i | \boldsymbol{x}_{t+1}) = q(\tilde{X}_t^i | \boldsymbol{x}_{t+1}^{K_i})$, we only need to additionally restrict the diffusion model to only condition on preceding unmasked tokens in $\boldsymbol{x}_{t+1}$, since K_i is the intersection of J and $\{j : j < i\}$. Therefore, if both models well-approximate the data distribution, we have $p_{\text{copula}}(\tilde{x}_t^i | \boldsymbol{x}_{t+1}) \approx q(\tilde{x}_t^i | \boldsymbol{x}_{t+1}^{K_i}) = q(\tilde{x}_t^i | \boldsymbol{x}_{t+1}^{<i}) \approx p_{\text{dm}}(\tilde{x}_t^i | \boldsymbol{x}_{t+1}^{<i})$, where the equality holds since all values in $\boldsymbol{x}_{t+1}^{<i}$ but not in $\boldsymbol{x}_{t+1}^{K_i}$ are `<MASK>`, and does not "contribute to" the distribution of $\tilde{X}_t^i$ according to Proposition 7). Correspondingly, we update $\mathbf{V}$ following

$$\forall i, c,\ \mathbf{V}[i, c] = \log p_{\text{dm}}(\tilde{X}_t^i = c | \boldsymbol{x}_{t+1}) - \log p_{\text{dm}}(\tilde{X}_t^i = c | \boldsymbol{x}_{t+1}^{<i}). \quad (6.10)$$

For denoising neural networks that are implemented with bidirectional Transformers, we can simply apply causal attention masks to the self-attention layers to obtain $\{p_{\text{dm}}(\tilde{X}_t^i | \boldsymbol{x}_{t+1}^{<i})\}_i$.

6.5.3 The Overall Diffusion Sampling Process

Given a diffusion model p_{dm} and an autoregressive copula model p_{copula}, the sampling procedure is outlined in Algorithm 8. First, we sample $\boldsymbol{x}_T$ from the prior noise distribution $p(\mathbf{X}_T)$ (line 3). During each denoising step t, we compute the univariate marginals $\{p_{\text{dm}}(\tilde{X}_t^i|\boldsymbol{x}_{t+1})\}_i$ and $\{p_{\text{dm}}(\tilde{X}_t^i|\boldsymbol{x}_{t+1}^{<i})\}_i$ based on the previously obtained $\boldsymbol{x}_{t+1}$ (line 5). These marginals are then used to compute the entries in $\mathbf{V}$ (line 6), which approximates the I-projection of $p_{\text{copula}}(\tilde{\mathbf{X}}_t|\boldsymbol{x}_{t+1})$ onto the set of distributions with univariate marginals $\{p_{\text{dm}}(\tilde{X}_t^i|\boldsymbol{x}_{t+1})\}_i$ (cf. Sec. 6.5.2).

Afterwards, we sample $\tilde{\boldsymbol{x}}_t$ from the combined distribution $\hat{p}(\tilde{\mathbf{X}}_t|\boldsymbol{x}_{t+1})$ (line 7). Specifically, following Eq. (6.6), we sample autoregressively following $\hat{p}(\tilde{\boldsymbol{x}}_t|\boldsymbol{x}_{t+1}) = \prod_i \hat{p}(\tilde{x}_t^i|\boldsymbol{x}_{t+1}, \tilde{\boldsymbol{x}}_t^{<i})$, where

$$\hat{p}(\tilde{x}_t^i|\boldsymbol{x}_{t+1}, \tilde{\boldsymbol{x}}_t^{<i}) \propto p_{\text{copula}}(X_i = \tilde{x}_t^i|\mathbf{X}_{<t} = \tilde{\boldsymbol{x}}_t^{<i}) \cdot \exp(\mathbf{V}[i, \tilde{x}_t^i]) \cdot \mathbb{1}[\tilde{x}_t^i = x_{t+1}^i].$$

Finally, we sample $\boldsymbol{x}_t$ from $q(\mathbf{X}_t|\tilde{\boldsymbol{x}}_t, \boldsymbol{x}_{t+1})$ (line 8) as defined in Proposition 7. To improve the algorithm's efficiency, we introduce a variant that unmasks tokens in an autoregressive manner. Specifically, at step t, all tokens except the first $(T-t)/T$ portion of the tokens in $\boldsymbol{x}_t$ are converted to `<MASK>`. Since $\hat{p}$ is sampled autoregressively, this allows us to use techniques such as KV-caching for autoregressive Transformers [137] to significantly reduce computation cost introduced by the copula model. See Section E.5 for details and the concrete algorithm.

6.6 Experiments

We empirically validate the proposed method, **D**iscrete **C**opula **D**iffusion (DCD), on language modeling tasks (Sec. 6.6.1 and 6.6.2) and antibody sequence infilling tasks (Sec. 6.6.3). For all tasks, we evaluate whether DCD can effectively reduce the number of diffusion steps while maintaining strong performance. Specifically, since DCD combines two pretrained models: a discrete diffusion model and an autoregressive copula model, we examine whether DCD

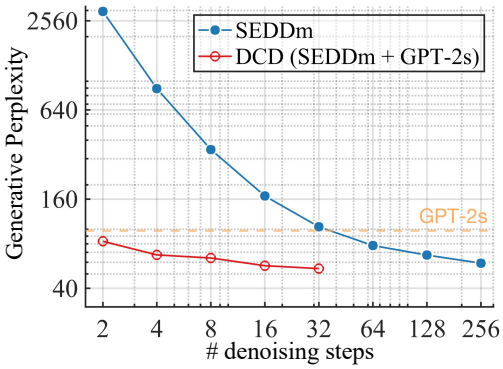


Figure 6.3: Generative perplexity (↓) with different numbers of denoising steps.

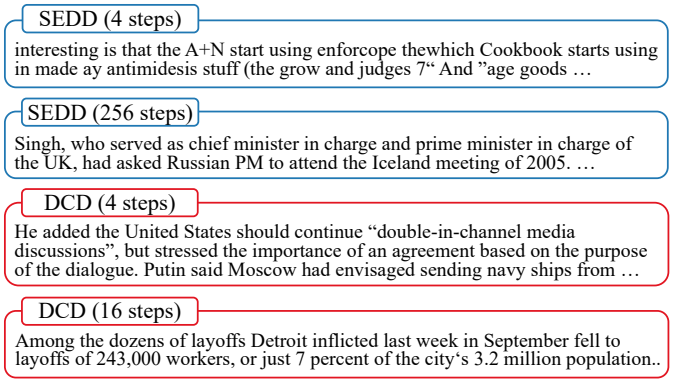


Figure 6.4: Generated text from SEDD_{M} and DCD with different number of steps. See Section E.8 for more.

outperforms each individual model.

6.6.1 Unconditional Text Generation

We first compare the quality of unconditional samples generated by models trained on either WebText [140] or OpenWebText [55], which contain web content extracted from URLs shared on Reddit with a minimum number of upvotes. We adopt the medium-sized SEDD model [105] (SEDD_{M}) since it is a SoTA discrete diffusion model for text generation. The GPT-2-small model [140] (GPT-2_{S}) serves as the copula model.

We generate samples of 128 tokens each. Following [42, 60], we evaluate sample quality using their generative perplexity, which is the perplexity of the samples when evaluated with the GPT-2-large model. Since previous studies have observed that this metric can be affected by distribution annealing methods such as nucleus sampling, we always sample directly from the models. SEDD_{M} is evaluated with 2 to 256 diffusion steps and DCD (i.e., SEDD_{M} with GPT-2_{S} as the copula model) is run with diffusion steps ranging from 2 to 32. We adopt the log-linear noise schedule suggested by the SEDD paper. See Section E.7.1 for more details.

For each configuration, we draw 10,000 samples and report the average perplexity in Fig. 6.3. First, when fixing the number of denoising steps between 2 to 32, we observe that DCD outperforms both SEDD_{M} with the same number of denoising steps and GPT-2_{S}. This

provides empirical validation of the effectiveness of the I-projection procedure for modeling inter-variable dependencies.

Additionally, DCD with just 4 denoising steps achieves performance comparable to $\text{SEDD}_{\texttt{M}}$ with 128 steps, representing a 32x reduction in the number of denoising steps. This result not only demonstrates the efficiency of DCD but also underscores the importance of modeling inter-variable dependencies in discrete diffusion models, particularly in few-step generation settings.

Finally, as shown in Fig. 6.4, SEDD fails to generate fluent and meaningful sentences given only a few diffusion steps, as too many tokens have to be generated in each step. In contrast, by modeling the inter-variable dependencies, DCD generates smooth sentences with only 4 denoising steps.

Efficiency. We compare the sample time and the generative perplexity of DCD against competitive baselines in Fig. 6.5. We additionally adopt another recent discrete diffusion baseline MDLM [153]. We adopt the autoregressive version of DCD as described in Section 6.5.3 and Section E.5. Compared to the baselines, DCD consistently achieves better generative perplexity given a fixed runtime constraint. It also requires less time to reach a desired perplexity value. We defer a comprehensive study of DCD's efficiency to Section E.6.

6.6.2 Conditional Text Generation

We now move on to conditional text generation, where certain tokens are provided in advance, and the task is to generate the remaining tokens. As shown in the first column of Table 6.1, we use five mask strategies, where tokens in specific prompt ranges are given (we use a sequence length of 128). We adopt the MAUVE score [134] with the default settings to compare the difference between the generated and original texts. See Section E.7.2 for further details.

For all methods, we use the same set of 2,000 text sequences from the validation set of WikiText-103 [116]. After applying the prompt mask, we generate 5 samples for each prompt, resulting in a total number of 10,000 samples.

Table 6.1: Evaluation of text infilling performance using the MAUVE score (↑) with 5 prompt masks. Scores of DCD are all better than (i) SEDD with the same # denoising steps, and (ii) GPT-2_S.

Prompt ranges (remainder is masked)	SSD-LM		GPT-2_S	SEDD$_M$					DCD (ours)				
	100	500	N/A	2	4	8	16	32	2	4	8	16	32
[0.1,0.2] & [0.5,0.7]	0.057	0.083	0.079	0.013	0.051	0.122	0.152	0.201	0.158	0.187	0.185	0.195	0.211
[0.25,0.75]	0.072	0.108	0.188	0.027	0.110	0.226	0.237	0.278	0.249	0.251	0.257	0.314	0.298
[0.0,0.1] & [0.4,0.6] & [0.9,1.0]	0.333	0.681	0.928	0.827	0.940	0.972	0.980	0.979	0.962	0.976	0.979	0.982	0.983
[0.4,0.5] & [0.8,1.0]	0.436	0.565	0.914	0.896	0.944	0.978	0.978	0.980	0.963	0.975	0.975	0.976	0.981
[0.2,0.3] & [0.6,0.8]	0.041	0.054	0.069	0.016	0.056	0.128	0.207	0.215	0.171	0.178	0.215	0.217	0.403

In addition to SEDD$_M$ and GPT-2_S, we compare against SSD-LM [60], which is a semi-autoregressive diffusion model designed for text infilling. We adopt the autoregressive unmasking variant of DCD described in the last paragraph of Section 6.5.3.

Results are presented in Table 6.1. First, DCD outperforms all three baselines in all five tasks. Additionally, when fixing the number of denoising steps between 2 and 32, DCD surpasses both of its base models. Notably, while both GPT-2_S and the 2-step SEDD$_M$ performs poorly on the first, the second, and the fifth tasks, combining them in a principled way allows DCD to achieve significantly better performance using only two denoising steps.

6.6.3 Antibody Sequence Infilling

We consider the task of unguided antibody infilling, where certain complementarity determining regions (CDRs) of antibodies (i.e., sequences of amino acids) are missing and to be generated by the model. We adopt NOC-D [58], which is a discrete diffusion model trained on 104K antibody sequences from the Observed Antibody Space dataset [150]. We further train a GPT model on the same dataset as the copula model. See Section E.7.3 for training details.

We follow [58] to select the same 10 antibody seed sequences from paired OAS [126]. We consider two infilling tasks: (i) three CDRs $\{\text{HCDR1}, \text{HCDR2}, \text{HCDR3}\}$ are masked, and (ii) two CDRs $\{\text{HCDR1}, \text{LCDR1}\}$ are masked. We follow the original paper and run 64 diffusion steps for NOS-D. For DCD (i.e., combining NOS-D with the trained GPT model as the

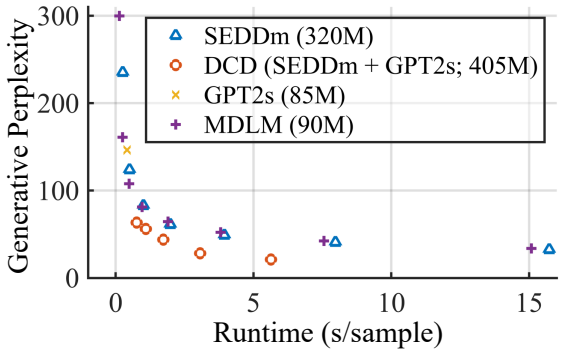


Figure 6.5: Sampling time vs. generative perplexity (the autoregressive version of DCD is used).

Figure 6.6: Antibody sequence infilling performance measured by sequence recovery rate (↑). We compare DCD against its two base models in two tasks, where amino acids at different locations are masked. DCD outperforms both baselines with only 4 denoising steps.

Method	# steps	Task	
		HCDR{1+2+3}	{H+L}CDR1
GPT	N/A	57.21	90.28
NOS-D	64	51.56	88.82
DCD	4	**58.28**	**91.58**

copula model), we use 4 denoising steps. We measure the sequence recovery rate, i.e., the accuracy of the infilled sequences given the ground truth sequence.

As shown in Fig. 6.6, by combining the univariate marginals from NOS-D and the dependencies captured by the GPT model, DCD can also perform well in antibody sequence infilling tasks.

Appendices

Appendix A

Tractable Inference with Probabilistic Circuits

A.1 Useful Sub-Routines

This section introduces the algorithmic construction of gadget circuits that will be adopted in our proofs of tractability as well as hardness. We start by introducing three primitive functions for constructing circuits—Input, Sum, and Product.

• Input$(l_p, \phi(p))$ constructs an input unit p that encodes a parameterized function l_p over variables $\phi(p)$. For example, Input$([\![X = \text{True}]\!], X)$ and Input$([\![X = \text{False}]\!], X)$ represent the positive and negative literals of a Boolean variable X, respectively. On the other hand, Input$(\mathcal{N}(\mu, \sigma), X)$ defines a Gaussian pdf with mean μ and standard deviation σ over variable X as an input function.

• Sum$(\{p_i\}_{i=1}^k, \{\theta_i\}_{i=1}^k)$ constructs a sum unit that represents the weighted combination of k circuit units $\{p_i\}_{i=1}^k$ encoded as an ordered set w.r.t. the correspondingly ordered weights $\{\theta_i\}_{i=1}^k$.

• Product$(\{p_i\}_{i=1}^k)$ builds a product unit that encodes the product of k circuit units $\{p_i\}_{i=1}^k$.

Algorithm 9 SUPPORT(p, cache)

Input: a smooth, deterministic, and decomposable circuit p over variables $\mathbf{X}$ and a cache for memorization
Output: a smooth, deterministic, and decomposable circuit s over $\mathbf{X}$ encoding $s(\boldsymbol{x}) = [\![\boldsymbol{x} \in \mathsf{supp}(p)]\!]$
if $p \in \mathsf{cache}$ **then return** $\mathsf{cache}(p)$
if p is an input unit **then** $s \leftarrow$ INPUT$([\![\boldsymbol{x} \in \mathsf{supp}(p)]\!], \phi(p))$
else if p is a sum unit **then** $s \leftarrow$ SUM$(\{\text{SUPPORT}(p_i, \mathsf{cache})\}_{i=1}^{|\mathsf{ch}(p)|}, \{1\}_{i=1}^{|\mathsf{ch}(p)|})$
else if p is a product unit **then** $s \leftarrow$ PRODUCT$(\{\text{SUPPORT}(p_i, \mathsf{cache})\}_{i=1}^{|\mathsf{ch}(p)|})$
$\mathsf{cache}(p) \leftarrow s$
return s

A.1.1 Support circuit of a deterministic circuit

Given a smooth, decomposable, and deterministic circuit $p(\mathbf{X})$, its support circuit $s(\mathbf{X})$ is a smooth, decomposable, and deterministic circuit that evaluates 1 iff the input $\boldsymbol{x}$ is in the support of p (i.e., $\boldsymbol{x} \in \mathsf{supp}(p)$) and otherwise evaluates 0, as defined below.

Definition 11 (Support circuit)**.** Let p be a smooth, decomposable, and deterministic PC over variables $\mathbf{X}$. Its support circuit is the circuit s that computes $s(\boldsymbol{x}) = [\![\boldsymbol{x} \in \mathsf{supp}(p)]\!]$, obtained by replacing every sum parameter of p by 1 and every input distribution l by the function $[\![\boldsymbol{x} \in \mathsf{supp}(l)]\!]$.

A construction algorithm for the support circuit is provided in Algorithm 9. This algorithm will later be useful in defining some circuit operations such as the logarithm.

A.1.2 Circuits encoding uniform distributions

We can build a deterministic and omni-compatible PC that encodes a (possibly unnormalized) uniform distribution over binary variables $\mathbf{X} = \{X_1, \ldots, X_n\}$: i.e., $p(\boldsymbol{x}) = c$ for a constant $c \in \mathbb{R}_+$ for all $\boldsymbol{x} \in \mathsf{val}(\mathbf{X})$. Specifically, p can be defined as a single sum unit with weight c that receives input from a product unit over n univariate input distribution units that always output 1 for all values $\mathsf{val}(X_i)$. This construction is summarized in Algorithm 10. It is a key component in the algorithms for many tractable circuit transformations/queries as well as in

Algorithm 10 uniformCircuit($\mathbf{X}, c$)

Input: a set of variables $\mathbf{X}$ and constant $c \in \mathbb{R}_+$.
Output: a deterministic and omni-compatible PC encoding an unnormalized uniform distribution over $\mathbf{X}$.
$n \leftarrow \{\}$
for $i = 1$ **to** $|\mathbf{X}|$ **do**
  $m \leftarrow \{\}$
  for x_i **in** $\mathsf{val}(X_i)$ **do**
    $m \leftarrow m \cup \{\textsc{Input}([\![X_i = x_i]\!], X_i)\}$
  $n \leftarrow n \cup \{\textsc{Sum}(m, \{1\}_{j=1}^{|\mathsf{val}(X_i)|})\}$
return $\textsc{Sum}(\{\textsc{Product}(n)\}, \{c\})$

several hardness proofs.

A.1.3 A circuit representation of the #3SAT problem

We define a circuit representation of the #3SAT problem, following the construction in [75]. Specifically, we represent each instance in the #3SAT problem as two poly-sized structured-decomposable and deterministic circuits p_β and p_γ, such that the partition function of their product equals the solution of the original #3SAT problem.

#3SAT is defined as follows: given a set of n boolean variables $\mathbf{X} = \{X_1, \ldots, X_n\}$ and a CNF that contains m clauses $\{c_1, \ldots, c_m\}$ (each clause contains exactly 3 literals), count the number of satisfiable worlds in $\mathsf{val}(\mathbf{X})$.

For every variable X_i in clause c_j, we introduce an auxiliary variable X_{ij}. Intuitively, $\{X_{ij}\}_{j=1}^m$ are copies of the variable X_i, one for each clause. Therefore, for any i, $\{X_{ij}\}_{j=1}^m$ share the same value (i.e., true or false), which can be represented by the following formula β:

$$\beta \equiv \bigwedge_{i=1}^{n} (X_{i1} \Leftrightarrow X_{i2} \Leftrightarrow \cdots \Leftrightarrow X_{im}).$$

Then we can encode the original CNF in the following formula γ by substituting X_i with

the respective X_{ij} in each clause:

$$\gamma \equiv \bigwedge_{j=1}^{m} \bigvee_{i:X_i \in \phi(c_j)} l(X_{ij}),$$

where $\phi(c)$ denotes the variable scope of clause c, and $l(X_{ij})$ denotes the literal of X_i in clause c_j. Since β restricts the variables $\{X_{ij}\}_{j=1}^{m}$ to have the same value, the model count of $\beta \wedge \gamma$ is equal to the model count of the original CNF.

We are left to show that both β and γ can be compiled into a poly-sized structured-decomposable and deterministic circuit. We start from compiling β into a circuit p_β. Note that for each i, $(X_{i1} \Leftrightarrow \cdots \Leftrightarrow X_{im})$ has exactly two satisfiable variable assignments (i.e., all true or all false), it can be compiled as a sum unit a_i over two product units b_{i1} and b_{i2} (both weights of a are set to 1), where b_{i1} takes inputs from the positive literals $\{X_{i1}, \ldots, X_{im}\}$ and b_{i2} from the negative literals $\{\neg X_{i1}, \ldots, \neg X_{im}\}$. Then p_β is represented by a product unit over $\{a_1, \ldots, a_n\}$. Note that by definition this p_β circuit is structured-decomposable and deterministic.

We proceed to compile γ into a polysized structured-decomposable and deterministic circuit p_γ. Note that in #3SAT, each clause c_j contains 3 literals. Therefore, for any $j \in \{1, \ldots, m\}$, $\bigvee_{X_i \in \phi(c_j)} l(X_{ij})$ has exactly 7 models w.r.t. the variable scope $\phi(c_j)$. Hence, we compile $\bigvee_{X_i \in \phi(c_j)} l(X_{ij})$ into a circuit d_j, which is a sum unit with 7 inputs $\{e_{j1}, \ldots, e_{j7}\}$. Each e_{jh} is constructed as a product unit over variables $\{X_{1j}, \ldots, X_{nj}\}$ that represents the h-th model of clause c_j. More formally, we have $e_{jh} \leftarrow \textsc{Product}(\{g_{ijh}\}_{i=1}^{n})$, where g_{ijh} is a sum unit over literals X_{ij} and $\neg X_{ij}$ (with both weights being 1) if $i \notin \phi(c_j)$ and otherwise g_{ijh} is the literal unit corresponds to the h-th model of clause c_j. The circuit p_γ representing the formula γ is constructed by a product unit with inputs $\{d_j\}_{j=1}^{m}$. By construction this circuit is also structured-decomposable and deterministic.

A.2 Circuit Operations

This section formally presents the tractability and hardness results w.r.t. circuit operations summarized in Table 2.1—sums, products, quotients, powers, logarithms, and exponentials. For each circuit operation, we provide both its proof of tractability by constructing a polytime algorithm given sufficient structural constraints and novel hardness results that identify necessary structural constraints for the operation to yield a decomposable circuit as output.

Throughout this paper, we will show hardness of operations to output a decomposable circuit by proving hardness of computing the partition function of the output of the operation. This follows from the fact that we can smooth and integrate a decomposable circuit in polytime (Proposition 1), thereby making the former problem at least as hard as the latter.

For the tractability theorems, we will assume that the operation referenced by the theorem is tractable over input units of circuit or pairs of compatible input units. For example, for Theorem 2 we assume tractable product of input units sharing the same scope and for Theorem 5 we assume that the powers of the input units can be tractably represented as a single new unit. Note that this is generally easy to realize for simple parametric forms e.g., multivariate Gaussians and for univariate distributions, unless specified otherwise.

Moreover, in the following results, we will adopt a more general definition of compatibility that can be applied to circuits with different variable scopes, which is often useful in practice. Formally, consider two circuits p and q with variable scope $\mathbf{Z}$ and $\mathbf{Y}$. Analogous to Definition 3, we say that p and q are compatible over variables $\mathbf{X} = \mathbf{Z} \cap \mathbf{Y}$ if (1) they are smooth and decomposable and (2) any pair of product units $n \in p$ and $m \in q$ with the same overlapping scope with $\mathbf{X}$ can be rearranged into mutually compatible binary products. Note that since our tractability results hold for this extended definition of compatibility, they are also satisfied under Definition 3.

A.2.1 Sum of Circuits

The hardness of the sum of two circuits to yield a deterministic circuit has been proven by [156] in the context of arithmetic circuits (ACs) [37]. ACs can be readily turned into circuits over binary variables according to our definition by translating their input parameters into sum parameters as done in [147].

A sum of circuits will preserve decomposability and related properties as the next proposition details.

Proposition 9 (Closure of sum of circuits)**.** *Let $p(\mathbf{Z})$ and $q(\mathbf{Y})$ be decomposable circuits. Then their sum circuit $s(\mathbf{Z} \cup \mathbf{Y}) = \theta_1 \cdot p(\mathbf{Z}) + \theta_2 \cdot q(\mathbf{Y})$ for two reals $\theta_1, \theta_2 \in \mathbb{R}$ is decomposable. If p and q are structured-decomposable and compatible, then s is structured-decomposable and compatible with both p and q. Lastly, if both inputs are also smooth, s can be smoothed in polytime.*

Proof. If p and q are decomposable, s is also decomposable by definition (no new product unit is introduced). If they are also structured-decomposable and compatible, s would be structured-decomposable and compatible with p and q as well, as summation does not affect their hierarchical scope partitioning. Note that if one input is decomposable and the other omni-compatible, then s would only be decomposable.

If $\mathbf{Z} = \mathbf{Y}$ then s is smooth; otherwise we can smooth it in polytime [35, 160], by realizing the circuit

$$s(\boldsymbol{x}) = \theta_1 \cdot p(\boldsymbol{z}) \cdot [\![q(\boldsymbol{x}|_{\mathbf{Y}\setminus\mathbf{Z}}) \neq 0]\!] + \ \theta_2 \cdot q(\boldsymbol{y}) \cdot [\![p(\boldsymbol{x}|_{\mathbf{Z}\setminus\mathbf{Y}}) \neq 0]\!]$$

where $[\![q(\boldsymbol{x}|_{\mathbf{Y}\setminus\mathbf{Z}}) \neq 0]\!]$ (resp. $[\![p(\boldsymbol{x}|_{\mathbf{Z}\setminus\mathbf{Y}}) \neq 0]\!]$) can be encoded as an input distribution over variables $\mathbf{Y} \setminus \mathbf{Z}$ (resp.$\mathbf{Z} \setminus \mathbf{Y}$). Note that if the supports of $p(\mathbf{Z} \setminus \mathbf{Y})$ and $q(\mathbf{Y} \setminus \mathbf{Z})$ are not bounded, then integrals over them would be unbounded as well. □

A.2.2 Product of Circuits

Theorem 16 (Hardness of product)**.** *Let p and q be two structured-decomposable and deterministic circuits over variables $\mathbf{X}$. Computing their product $m(\mathbf{X}) = p(\mathbf{X}) \cdot q(\mathbf{X})$ as a decomposable circuit is #P-hard.*[1]

Proof. As noted earlier, we will prove hardness of computing the product by showing hardness of computing the partition function of a product of two circuits. In particular, let p and q be two structured-decomposable and deterministic circuits over binary variables $\mathbf{X}$. Then, computing the following quantity is #P-hard:

$$\sum_{\boldsymbol{x} \in \mathsf{val}(\mathbf{X})} p(\boldsymbol{x}) \cdot q(\boldsymbol{x}). \tag{MULPC}$$

The following proof is adapted from the proof of Thm. 2 in [75]. We reduce the #3SAT problem defined in Section A.1.3, which is known to be #P-hard, to MULPC. Recall that p_β and p_γ, as constructed in Section A.1.3, are structured-decomposable and deterministic; additionally, the partition function of $p_\beta \cdot p_\gamma$ is the solution of the corresponding #3SAT problem. In other words, computing MULPC of two structured-decomposable and deterministic circuits p_β and p_γ exactly solves the original #3SAT problem. Therefore, computing the product of two structured-decomposable and deterministic circuits is #P-hard. □

Theorem 17 (Tractable product of circuits)**.** *Let $p(\mathbf{Z})$ and $q(\mathbf{Y})$ be two compatible circuits over variables $\mathbf{X} = \mathbf{Z} \cap \mathbf{Y}$. Then, computing their product $m(\mathbf{X}) = p(\mathbf{Z}) \cdot q(\mathbf{Y})$ as a decomposable circuit can be done in $\mathcal{O}(|p||q|)$ time. If both p and q are also deterministic, then so is m, moreover if p and q are structured-decomposable then m is compatible with p (and q) over $\mathbf{X}$.*

Proof. The proof proceeds by showing that computing the product of (i) two smooth and compatible sum units p and q and (ii) two smooth and compatible product units p and q given

[1] Note that this implies that product of decomposable circuits is also #P-hard, as decomposability is a weaker condition than structured-decomposability. The hardness results throughout this paper translate directly when input properties are relaxed.

the product circuits w.r.t. pairs of child units from p and q (i.e., $\forall r \in \mathsf{ch}(p)\, s \in \mathsf{ch}(q), (r{\cdot}s)(\mathbf{X})$) takes time $\mathcal{O}(|\mathsf{ch}(p)||\mathsf{ch}(q)|)$. Then, by recursion, the overall time complexity is $\mathcal{O}(|p||q|)$. Algorithm 11 illustrates the overall process in detail.

If p and q are two sum units defined as $p(\boldsymbol{x}) = \sum_{i \in \mathsf{ch}(p)} \theta_i p_i(\boldsymbol{x})$ and $q(\boldsymbol{x}) = \sum_{j \in \mathsf{ch}(q)} \theta'_j q_j(\boldsymbol{x})$, respectively. Then, their product $m(\boldsymbol{x})$ can be broken down to the weighted sum of $|\mathsf{ch}(p)| \cdot |\mathsf{ch}(q)|$ circuits that represent the products of pairs of their inputs:

$$m(\boldsymbol{x}) = \left(\sum_{i \in \mathsf{ch}(p)} \theta_i p_i(\boldsymbol{x})\right)\left(\sum_{j \in \mathsf{ch}(q)} \theta'_j q_j(\boldsymbol{x})\right) = \sum_{i \in \mathsf{ch}(p)} \sum_{j \in \mathsf{ch}(q)} \theta_i \theta'_j (p_i q_j)(\boldsymbol{x}).$$

Note that this Cartesian product of units is a deterministic sum unit if both p and q were deterministic sum units, as $\mathsf{supp}(p_i q_j) = \mathsf{supp}(p_i) \cap \mathsf{supp}(q_j)$ are disjoint for different i, j.

If p and q are two product units defined as $p(\mathbf{X}) = p_1(\mathbf{X}_1)p_2(\mathbf{X}_2)$ and $q(\mathbf{X}) = q_1(\mathbf{X}_1)q_2(\mathbf{X}_2)$, respectively. Then, their product $m(\boldsymbol{x})$ can be constructed recursively from the product of their inputs:

$$m(\boldsymbol{x}) = p_1(\boldsymbol{x}_1)p_2(\boldsymbol{x}_2) \cdot q_1(\boldsymbol{x}_1)q_2(\boldsymbol{x}_2) = p_1(\boldsymbol{x}_1)q_1(\boldsymbol{x}_1) \cdot p_2(\boldsymbol{x}_2)q_2(\boldsymbol{x}_2) = (p_1 q_1)(\boldsymbol{x}_1) \cdot (p_2 q_2)(\boldsymbol{x}_2).$$

Note that by this construction m retains the same scope partitioning of p and q, hence if they were structured-decomposable, m will be structured-decomposable and compatible with p and q. □

Possessing additional structural constrains can lead to sparser output circuits as well as efficient algorithms to construct them. First, if one among p and q is omni-compatible, it suffices that the other is just decomposable to obtain a tractable product, whose size this time is going to be linear in the size of the decomposable circuit.

Corollary 1. *Let p be a smooth and decomposable circuit over $\mathbf{X}$ and q an omni-compatible circuit over $\mathbf{X}$ comprising a sum unit with k inputs, hence its size is $k|\mathbf{X}|$. Then, $m(\mathbf{X}) = p(\mathbf{X})q(\mathbf{X})$ is a smooth and decomposable circuit constructed in $\mathcal{O}(k|p|)$ time.*

Algorithm 11 MULTIPLY(p, q, cache)

Input: two circuits $p(\mathbf{Z})$ and $q(\mathbf{Y})$ that are compatible over $\mathbf{X} = \mathbf{Z} \cap \mathbf{Y}$ and a cache for memoization
Output: their product circuit $m(\mathbf{Z} \cup \mathbf{Y}) = p(\mathbf{Z})q(\mathbf{Y})$
if $(p, q) \in \mathsf{cache}$ **then return** $\mathsf{cache}(p, q)$
if $\phi(p) \cap \phi(q) = \varnothing$ **then**
  $m \leftarrow \textsc{Product}(\{p, q\})$; $s \leftarrow$ True
else if p, q are input units **then**
  $m \leftarrow \textsc{Input}(p(\mathbf{Z}) \cdot q(\mathbf{Y}), \mathbf{Z} \cup \mathbf{Y})$
  $s \leftarrow [\![\mathsf{supp}(p(\mathbf{X})) \cap \mathsf{supp}(q(\mathbf{X})) \neq \varnothing]\!]$
else if p is an input unit **then**
  $n \leftarrow \{\}; s \leftarrow$ False $// q(\mathbf{Y}) = \sum_j \theta'_j q_j(\mathbf{Y})$
  for $j = 1$ **to** $|\mathsf{ch}(q)|$ **do**
    $n', s' \leftarrow \textsc{multiply}(p, q_j, \mathsf{cache})$
    $n \leftarrow n \cup \{n'\}$; $s \leftarrow s \vee s'$
  if s **then** $m \leftarrow \textsc{Sum}(n, \{\theta'_j\}_{j=1}^{|\mathsf{ch}(q)|})$ **else** $m \leftarrow null$
else if q is an input unit **then**
  $n \leftarrow \{\}; s \leftarrow$ False $// p(\mathbf{Z}) = \sum_i \theta_i p_i(\mathbf{Z})$
  for $i = 1$ **to** $|\mathsf{ch}(p)|$ **do**
    $n', s' \leftarrow \textsc{multiply}(p_i, q, \mathsf{cache})$
    $n \leftarrow n \cup \{n'\}$; $s \leftarrow s \vee s'$
  if s **then** $m \leftarrow \textsc{Sum}(n, \{\theta_i\}_{i=1}^{|\mathsf{ch}(p)|})$ **else** $m \leftarrow null$
else if p, q are product units **then**
  $n \leftarrow \{\}; s \leftarrow$ True
  $\{p_i, q_i\}_{i=1}^{k} \leftarrow \mathsf{sortPairsByScope}(p, q, \mathbf{X})$
  for $i = 1$ **to** k **do**
    $n', s' \leftarrow \textsc{multiply}(p_i, q_i, \mathsf{cache})$
    $n \leftarrow n \cup \{n'\}$; $s \leftarrow s \wedge s'$
  if s **then** $m \leftarrow \textsc{Product}(n)$ **else** $m \leftarrow null$
else if p, q are sum units **then**
  $n \leftarrow \{\}$; $w \leftarrow \{\}$; $s \leftarrow$ False
  for $i = 1$ **to** $|\mathsf{ch}(p)|$, $j = 1$ **to** $|\mathsf{ch}(q)|$ **do**
    $n', s' \leftarrow \textsc{multiply}(p_i, q_j, \mathsf{cache})$
    $n \leftarrow n \cup n'; w \leftarrow w \cup \{\theta_i \theta'_j\}; s \leftarrow s \vee s'$
  if s **then** $m \leftarrow \textsc{Sum}(n, w)$ **else** $m \leftarrow null$
$\mathsf{cache}(p, q) \leftarrow (m, s)$
return m, s

Second, if p and q have inputs with restricted supports, their product is going to be sparse, i.e., only a subset of their inputs is going to yield a circuit that does not constantly output zero. Note that in Algorithm 11 we can check in polytime if the supports of two units to

be multiplied are overlapping by a depth-first search (realized with a Boolean indicator s in Algorithm 11), thanks to decomposability. Therefore, for two compatible sum units p and q we will effectively build a number of units that is

$$\mathcal{O}(|\{(p_i, q_j)|p_i \in \mathsf{ch}(p), q_i \in \mathsf{ch}(q), \mathsf{supp}(p_i) \cap \mathsf{supp}(q_j) \neq \varnothing\}|).$$

In practice, this sparsifying effect will be more prominent when both p and q are deterministic. This is because having disjoint supports is required for deterministic circuits. This "decimation" of product units will be maximum if p and q partition the support in the very same way, for instance when we have $p = q$, i.e., we are multiplying one circuit with itself, or we are dealing with a logarithmic circuit (cf. Section A.2.5). In such a case, we can omit the depth-first check for overlapping supports of the product units participating in the product of a sum unit. If both p and q have an identifier for their supports, we can simply check for equality of their identifiers. This property and algorithmic insight will be key when computing powers of a deterministic circuit and its entropies (cf. Section A.3.2), as it would suffice the input circuit p to be decomposable (cf. Section 2.4) to obtain a linear time complexity.

A.2.3 Power Function of Circuits

Theorem 18 (Natural powers)**.** *If p is a structured-decomposable circuit, then for any $\alpha \in \mathbb{N}$, its power can be represented as a structured-decomposable circuit in $\mathcal{O}(|p|^\alpha)$ time. Otherwise, if p is only smooth and decomposable, then computing $p^\alpha(\mathbf{X})$ as a decomposable circuit is #P-hard.*

Proof. The proof for tractability easily follows by directly applying the product operation repeatedly.

We prove hardness for the special case of discrete variables, and by showing the hardness of computing the partition function of $p^2(\mathbf{X})$. In particular, let $\mathbf{X}$ be a collection of binary

Algorithm 12 SORTPAIRSBYSCOPE$(p, q, \mathbf{X})$

Input: two decomposable and compatible product units p and q, and a variable scope $\mathbf{X}$.

Output: Pairs of compatible sum units $\{(p_i, q_i)\}_{i=1}^{k}$.
children_$p \leftarrow \{p_i\}_{i=1}^{|\mathsf{ch}(p)|}$, children_$q \leftarrow \{q_i\}_{i=1}^{|\mathsf{ch}(q)|}$
pairs $\leftarrow \{\}$. // "pairs" stores circuit pairs with matched scope.
cmp_$p \leftarrow \{\{\}\}_{i=1}^{|\mathsf{ch}(p)|}$, cmp_$q \leftarrow \{\{\}\}_{j=1}^{|\mathsf{ch}(q)|}$.
// cmp_$p[i]$ (resp. cmp_$q[j]$) stores the children of q (resp. p) whose scopes are subsets of p_i's (resp. q_j's) scope.
for i $= 1$ **to** $|\mathsf{ch}(p)|$ **do**
  for j $= 1$ **to** $|\mathsf{ch}(q)|$ **do**
    if $\phi(p_i) \cap \mathbf{X} = \phi(q_j) \cap \mathbf{X}$ **then**
      pairs.*append*$((p_i, q_j))$
      children_p.*pop*(p_i), children_q.*pop*(q_j)
    else if $\phi(p_i) \cap \mathbf{X} \subset \phi(q_j) \cap \mathbf{X}$ **then**
      cmp_$q[j]$.*append*(p_i)
      children_p.*pop*(p_i), children_q.*pop*(q_j)
    else if $\phi(q_j) \cap \mathbf{X} \subset \phi(p_i) \cap \mathbf{X}$ **then**
      cmp_$p[i]$.*append*(q_j)
      children_p.*pop*(p_i), children_q.*pop*(q_j)
for $i = 1$ **to** $|\mathsf{ch}(p)|$ **do**
  if *len*(cmp_$p[i]$) $\neq 0$ **then**
    $s \leftarrow$ SUM({PRODUCT(cmp_$p[i]$)}, {1})
    pairs.*append*$((p_i, s))$
for $j = 1$ **to** $|\mathsf{ch}(q)|$ **do**
  if *len*(cmp_$q[j]$) $\neq 0$ **then**
    $r \leftarrow$ SUM({PRODUCT(cmp_$q[j]$)}, {1})
    pairs.*append*$((r, q_j))$
for r, s **in** *zip*(children_p, children_q) **do**
  pairs.*append*$((r, s))$
if *len*(children_p) $>$ *len*(children_q) **then**
  for $i =$ *len*(children_q) $+ 1$ **to** *len*(children_p) **do**
    pairs.*append*((children_$p[i]$, children_$q[1]$))
else if *len*(children_p) $<$ *len*(children_q) **then**
  for $j =$ *len*(children_p) $+ 1$ **to** *len*(children_q) **do**
    pairs.*append*((children_$p[1]$, children_$q[j]$))
return *pairs*

variables and let p be a smooth and decomposable circuit over $\mathbf{X}$, then computing the quantity

$$\sum_{\boldsymbol{x} \in \mathsf{val}(\mathbf{X})} p^2(\boldsymbol{x}) \qquad \text{(POW2PC)}$$

is #P-hard.

The proof builds a reduction from the #3SAT problem, which is known to be #P-hard. We employ the same setting of Section A.1.3, where a CNF over n Boolean variables $\mathbf{X} = \{X_1, \ldots, X_n\}$ and containing m clauses $\{c_1, \ldots, c_m\}$, each with exactly 3 literals, is encoded into two structured-decomposable and deterministic circuits p_β and p_γ over variables $\hat{\mathbf{X}} = \{X_{11}, \ldots, X_{1m}, \ldots, X_{n1}, \ldots, X_{nm}\}$.

Then, we construct circuit p_α as the sum of p_β and p_γ, i.e., $p_\alpha(\hat{\boldsymbol{x}}) := p_\beta(\hat{\boldsymbol{x}}) + p_\gamma(\hat{\boldsymbol{x}})$. By definition p_α is smooth and decomposable, but not structured-decomposable. We proceed to show that if we can represent $p_\alpha^2(\hat{\boldsymbol{x}})$ as a smooth and decomposable circuit in polytime, we could solve POW2PC and hence #3SAT. That would mean that computing POW2PC is #P-hard.

By definition, $p_\alpha^2(\hat{\boldsymbol{x}}) = (p_\beta(\hat{\boldsymbol{x}}) + p_\gamma(\hat{\boldsymbol{x}}))^2 = p_\beta^2(\hat{\boldsymbol{x}}) + p_\gamma^2(\hat{\boldsymbol{x}}) + 2p_\beta(\hat{\boldsymbol{x}}) \cdot p_\gamma(\hat{\boldsymbol{x}})$, and hence

$$\sum_{\hat{\boldsymbol{x}} \in \mathsf{val}(\hat{\mathbf{X}})} p_\alpha^2(\hat{\boldsymbol{x}}) = \sum_{\hat{\boldsymbol{x}} \in \mathsf{val}(\hat{\mathbf{X}})} p_\beta^2(\hat{\boldsymbol{x}}) + \sum_{\hat{\boldsymbol{x}} \in \mathsf{val}(\hat{\mathbf{X}})} p_\gamma^2(\hat{\boldsymbol{x}}) + \sum_{\hat{\boldsymbol{x}} \in \mathsf{val}(\hat{\mathbf{X}})} p_\beta(\hat{\boldsymbol{x}}) \cdot p_\gamma(\hat{\boldsymbol{x}}).$$

Since p_β and p_γ are both structured-decomposable and deterministic the first two summations over the squared circuits can be computed in time $\mathcal{O}(|p_\beta| + |p_\gamma|)$ (see Theorem 5). It follows that if we could efficiently solve POW2PC we could then solve the that third summation, i.e., $\sum_{\hat{\boldsymbol{x}} \in \mathsf{val}(\hat{\mathbf{X}})} p_\beta(\hat{\boldsymbol{x}}) \cdot p_\gamma(\hat{\boldsymbol{x}})$. However, since such a summation is the instance of MULPC between p_β and p_γ reduced from #3SAT (see Theorem 1), it would mean that we could solve #3SAT. We can conclude that computing POW2PC is #P-hard. □

Theorem 19 (Hardness of natural power of a structured-decomposable circuit)**.** *Let p be a structured-decomposable circuit over variables $\mathbf{X}$. Let k be a natural number. Then there is no polynomial $f(x, y)$ such that the power p^k can be computed in $\mathcal{O}(f(|p|, k))$ time unless P=NP.*

Proof. We construct the proof by showing that for a structured-decomposable circuit p, if we

could compute

$$\sum_{\boldsymbol{x} \in \mathsf{val}(\mathbf{X})} p^k(\boldsymbol{x}). \tag{POWkPC}$$

in $\mathcal{O}(f(|p|, k))$ time, then we could solve the 3SAT problem in polytime, which is known to be NP-hard.

The 3SAT problem is defined as follows: given a set of n Boolean variables $\mathbf{X} = \{X_1, \ldots, X_n\}$ and a CNF that contains m clauses $\{c_1, \ldots, c_m\}$, each one containing exactly 3 literals, determine whether there exists a satisfiable configuration in $\mathsf{val}(\mathbf{X})$.

We start by constructing m gadget circuits $\{d_j\}_{j=1}^m$ for the m clauses such that $d_j(\boldsymbol{x})$ evaluates to $\frac{1}{m}$ iff $\boldsymbol{x}$ satisfies c_j and otherwise evaluates to 0, respectively.

Since each clause c_j contains exactly 3 literals, it comprises exactly 7 models w.r.t. the variables appearing in it, i.e., its scope $\phi(c_j)$. Therefore, following a similar construction in Section A.1.3, we can compile d_j as a weighted sum of 7 circuits that represent the 7 models of c_j, respectively. By choosing all weights of d_j as $\frac{1}{m}$, the circuit d_j outputs $\frac{1}{m}$ iff c_j is satisfied; otherwise it outputs 0.

The gadget circuits $\{d_j\}_{j=1}^m$ are then summed together to represent a circuit p. That is, $p = \textsc{Sum}(\{d_j\}_{j=1}^m, \{1\}_{j=1}^m)$. In the following, we complete the proof by showing that if the power circuit p^k (we will pick later $k = \lceil \max(m, n)^2 \cdot \log 2 \rceil$) can be computed in $\mathcal{O}(f(|p|, k))$ time, then the corresponding 3SAT problem can be solved in $\mathcal{O}(f(|p|, k))$ time.

If the original CNF is satisfiable, then there exists at least 1 world such that all clauses are satisfied. In this case, all circuits in $\{d_j\}_{j=1}^m$ will evaluate $\frac{1}{m}$. Since p is the sum of the circuits $\{d_j\}_{j=1}^m$, it will evaluate 1 for any world that satisfies the CNF. We obtain the bound

$$\sum_{\boldsymbol{x} \in \mathsf{val}(\mathbf{X})} p^k(\boldsymbol{x}) > m \cdot \frac{1}{m} = 1.$$

In contrast, if the CNF is unsatisfiable, each variable assignment $\boldsymbol{x} \in \mathsf{val}(\mathbf{X})$ satisfies at

most $m-1$ clauses, so the circuit p will output at most $\frac{m-1}{m}$. Therefore , we retrieve the following bound

$$\sum_{\boldsymbol{x}\in\mathsf{val}(\mathbf{X})} p^k(\boldsymbol{x}) \leq 2^n \left(\frac{m-1}{m}\right)^k.$$

Then, we can retrieve a value for k to separate the two bounds as follows.

$$2^n \left(\frac{m-1}{m}\right)^k < 1 \;\Leftrightarrow k > \frac{\log(2^{-n})}{\log\frac{m-1}{m}} \Leftrightarrow\; k > \frac{n\log 2}{\log(m)-\log(m-1)} \overset{(a)}{\Leftrightarrow}\; k > m\cdot n\cdot \log 2,$$

where (a) follows the fact that $\log\left(\frac{m}{m-1}\right) \leq \frac{1}{m-1}$. Let $l = \max(m,n)$. If we choose $k = \lceil l^2 \cdot \log 2\rceil$, then we can separate the two bounds above.

Therefore, if there exists a polynomial $f(x,y)$ such that the power p^k ($k = \lceil l^2\cdot\log 2\rceil$) can be computed in $\mathcal{O}(f(|p|,k))$ time, then we can solve 3SAT in $\mathcal{O}(f(|p|,k))$ time since the CNF is satisfiable iff $\sum_{\boldsymbol{x}\in\mathsf{val}(\mathbf{X})} p^k(\boldsymbol{x}) > 1$, which is impossible unless P=NP. □

Theorem 20 (Hardness of reciprocal of a circuit)**.** *Let p be a smooth and decomposable circuit over variables $\mathbf{X}$. Then computing $p^{-1}(\mathbf{X})|_{\mathsf{supp}(p)}$ as a decomposable circuit is #P-hard, even if p is structured-decomposable.*

Proof. We prove it for the case of PCs over discrete variables. We will prove hardness of computing the reciprocal by showing hardness of computing the partition of the reciprocal of a circuit. In particular, let $\mathbf{X} = \{X_1,\ldots,X_n\}$ be a collection of binary variables and let p be a smooth and decomposable PC over $\mathbf{X}$, then computing the quantity

$$\sum_{\boldsymbol{x}\in\mathsf{val}(\mathbf{X})} \frac{1}{p(\boldsymbol{x})} \tag{INVPC}$$

is #P-hard.

Proof is by reduction from the EXPLR problem as defined in Theorem 22. Similarly to Theorem 22, the reduction is built by constructing a smooth and decomposable unnormalized

circuit $p(x) = 2^n \cdot 1 + 2^n e^{-(w_0 + \sum_i w_i x_i)}$. The circuit p comprises a sum unit over two sub-circuits. The first is a uniform (unnormalized) distribution over $\mathbf{X}$ defined as a product unit over n univariate input distribution units that always output 1 for all values $\mathsf{val}(X_i)$ (see Section A.1.2 for a construction algorithm). The second is an exponential of a linear circuit (Algorithm 15) and encodes $e^{-(w_0 + \sum_i w_i x_i)}$ via a product unit over n univariate input distributions, where one of them encodes $e^{-w_0 - w_1 x_1}$ and the rest $e^{-w_j x_j}$ for $j = 2, \ldots, n$. Both sub-circuits participates in the sum with parameters 2^n.

The size of the constructed circuit is linear in n, and INVPC of this circuit corresponds to the solution of the EXPLR problem. If you can represent the reciprocal of this circuit as a decomposable circuit, you can compute its marginals (including the partition function) which solves INVPC and hence EXPLR. Furthermore, the circuit is also omni-compatible because mixture of fully-factorized distributions. □

Theorem 21 (Tractable real power of a deterministic circuit)**.** *Let p be a smooth, decomposable, and deterministic circuit over variables $\mathbf{X}$. Then, for any real number $\alpha \in \mathbb{R}$, its restricted power, defined as $a(\boldsymbol{x})|_{\mathsf{supp}(p)} = p^\alpha(\boldsymbol{x})[\![\boldsymbol{x} \in \mathsf{supp}(p)]\!]$ can be represented as a smooth, decomposable, and deterministic circuit over variables $\mathbf{X}$ in $\mathcal{O}(|p|)$ time. Moreover, if p is structured-decomposable, then a is structured-decomposable as well.*

Proof. The proof proceeds by construction and recursively builds $a(\boldsymbol{x})|_{\mathsf{supp}(p)}$. As the base case, we can assume to compute the restricted α-power of the input units of p and represent it as a single new unit. When we encounter a deterministic sum unit, the power will decompose into the sum of the powers of its inputs. Specifically, let p be a sum unit: $p(\mathbf{X}) = \sum_{i \in \mathsf{ch}(p)} \theta_i p_i(\mathbf{X})$. Then, its restricted real power circuit $a(\boldsymbol{x})|_{\mathsf{supp}(p)}$ can be expressed as

$$a(\boldsymbol{x})|_{\mathsf{supp}(p)} = \left(\sum_{i \in \mathsf{ch}(p)} \theta_i p_i(\boldsymbol{x}) \right)^\alpha [\![\boldsymbol{x} \in \mathsf{supp}(p)]\!] = \sum_{i \in \mathsf{ch}(p)} \theta_i^\alpha \big(p_i(\boldsymbol{x})\big)^\alpha [\![\boldsymbol{x} \in \mathsf{supp}(p_i)]\!].$$

Note that this construction is possible because only one input of p will be non-zero for any input (determinism). As such, the power circuit is retaining the same structure of the original

Algorithm 13 POWER$(p, \alpha, \mathsf{cache})$

Input: a smooth, deterministic and decomposable circuit $p(\mathbf{X})$, a scalar $\alpha \in \mathbb{R}$, and a cache for memoization
Output: a smooth, deterministic and decomposable circuit $a(\mathbf{X})$ encoding $p^\alpha(\mathbf{X})|_{\mathsf{supp}(p)}$
if $p \in \mathsf{cache}$ **then return** $\mathsf{cache}(p)$
if p is an input unit **then** $a \leftarrow \textsc{Input}(p^\alpha(\mathbf{X})|_{\mathsf{supp}(p)}, \phi(p))$
else if p is a sum unit **then** $a \leftarrow \textsc{Sum}(\{\textsc{power}(p_i, \alpha, \mathsf{cache})\}_{i=1}^{|\mathsf{ch}(p)|}), \{\theta_i^\alpha\}_{i=1}^{|\mathsf{ch}(p)|})$
else if p is a product unit **then** $a \leftarrow \textsc{Product}(\{\textsc{power}(p_i, \alpha, , \mathsf{cache})\}_{i=1}^{|\mathsf{ch}(p)|})$
$\mathsf{cache}(p) \leftarrow a$
return a

sum unit.

Next, for a decomposable product unit, its power will be the product of the powers of its inputs. Specifically, let p be a product unit: $p(\mathbf{X}) = p_1(\mathbf{X}_1) \cdot p_2(\mathbf{X}_2)$. Then, its restricted real power circuit $a(\boldsymbol{x})|_{\mathsf{supp}(p)}$ can be expressed as

$$\begin{aligned} a(\boldsymbol{x})|_{\mathsf{supp}(p)} &= \big(p_1(\boldsymbol{x}_1) \cdot p_2(\boldsymbol{x}_2)\big)^\alpha [\![\boldsymbol{x} \in \mathsf{supp}(p)]\!] \\ &= \big(p_1(\boldsymbol{x}_1)\big)^\alpha [\![\boldsymbol{x} \in \mathsf{supp}(p_1)]\!] \cdot \big(p_2(\boldsymbol{x}_2)\big)^\alpha [\![\boldsymbol{x} \in \mathsf{supp}(p_2)]\!]. \end{aligned}$$

Note that even this construction preserves the structure of p and hence its scope partitioning is retained throughout the whole algorithm. Hence, if p were also structured-decomposable, then a would be structured-decomposable. Algorithm 13 illustrates the whole algorithm in detail. □

A.2.4 Quotient of Circuits

Theorem 22 (Hardness of quotient of two circuits)**.** *Let p and q be two smooth and decomposable circuits over variables $\mathbf{X}$, and let $q(\boldsymbol{x}) \neq 0$ for every $\boldsymbol{x} \in \mathsf{val}(\mathbf{X})$. Then, computing their quotient $p(\mathbf{X})/q(\mathbf{X})$ as a decomposable circuit is #P-hard, even if they are compatible.*

Proof. This result follows from Theorem 4 by noting that computing the reciprocal of a circuit is a special case of computing the quotient of two circuits. In particular, let p be an

omni-compatible circuit representing the constant function 1 over variables $\mathbf{X}$, constructed as in Section A.1.2. Then computing the reciprocal of a structured-decomposable circuit q as a decomposable circuit reduces to computing the quotient p/q. □

Theorem 23 (Tractable restricted quotient of two circuits)**.** *Let p and q be two compatible circuits over variables $\mathbf{X}$, and let q be also deterministic. Then, their quotient restricted to $\mathsf{supp}(q)$ can be represented as a circuit compatible with p (and q) over variables $\mathbf{X}$ in $\mathcal{O}(|p||q|)$ time. Moreover, if p is also deterministic, then the quotient circuit is deterministic as well.*

Proof. We know from Theorem 5 that we can obtain the reciprocal circuit q^{-1} that is also compatible with q (and by extension p) in $\mathcal{O}(|q|)$ time. Then we can multiply p and q^{-1} in $\mathcal{O}(|p||q|)$ time using Theorem 2 to compute their quotient circuit that is still compatible with p and q. If p is also deterministic, then we are multiplying two deterministic circuits and therefore their product circuit is deterministic (Theorem 2). □

A.2.5 Logarithm of a PC

Theorem 24 (Logarithms)**.** ***(Tractability)*** *Let p be a smooth, deterministic and decomposable PC over variables $\mathbf{X}$. Then its logarithm circuit, restricted to the support of p and defined as*

$$l(\boldsymbol{x})|_{\mathsf{supp}(p)} = \begin{cases} \log p(\boldsymbol{x}) & \text{if } \boldsymbol{x} \in \mathsf{supp}(p) \\ 0 & \text{otherwise} \end{cases}$$

for every $\boldsymbol{x} \in \mathsf{val}(\mathbf{X})$ can be represented as a smooth and decomposable circuit that shares the scope partitioning of p in $\mathcal{O}(|p|)$ time. ***(Hardness)*** *Otherwise, if p is a smooth and decomposable PC, then computing its logarithm circuit $l(\mathbf{X}) := \log p(\mathbf{X})$ as a decomposable circuit is #P-hard, even if p is structured-decomposable.*

We will provide the proofs for tractability and hardness separately below.

Proof of tractability. The proof proceeds by recursively constructing $l(\boldsymbol{x})|_{\mathsf{supp}(p)}$. In the base case, we assume computing the logarithm of an input unit can be done in $\mathcal{O}(1)$ time. When

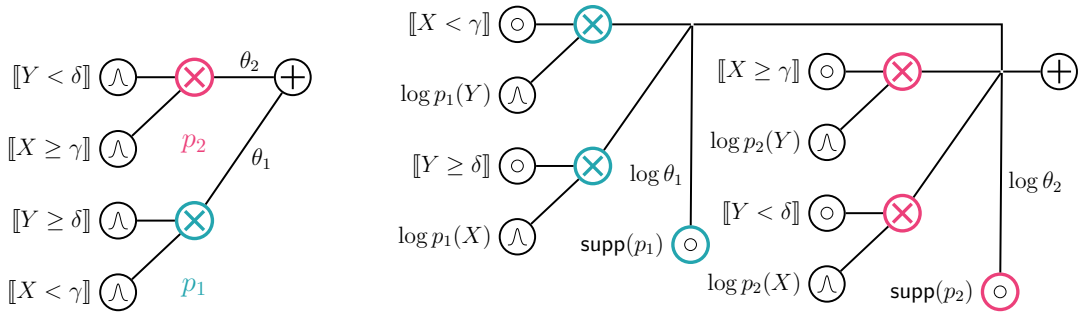


Figure A.1: Building the logarithmic circuit (right) for a deterministic PC (left) whose input units are labeled by their supports. A single sum unit is introduced over smoothed product units and additional dummy input units which share the same support across circuits if they have the same color.

we encounter a deterministic sum unit $p(\boldsymbol{x}) = \sum_{i \in |\mathsf{ch}(p)|} \theta_i p_i(\boldsymbol{x})$, its logarithm circuit consists of the sum of (i) the logarithm circuits of its child units and (ii) the support circuits of its children weighted by their respective weights $\{\theta_i\}_{i=1}^{|\mathsf{ch}(p)|}$:

$$
\begin{aligned}
l(\boldsymbol{x})|_{\mathsf{supp}(\boldsymbol{x})} &= \log\left(\sum_{i \in \mathsf{ch}(p)} \theta_i p_i(\boldsymbol{x})\right) \cdot [\![\boldsymbol{x} \in \mathsf{supp}(p)]\!] = \sum_{i \in |\mathsf{ch}(p)|} \log\Big(\theta_i p_i(\boldsymbol{x})\Big)[\![\boldsymbol{x} \in \mathsf{supp}(p_i)]\!] \\
&= \sum_{i \in |\mathsf{ch}(p)|} \log \theta_i [\![\boldsymbol{x} \in \mathsf{supp}(p_i)]\!] + \sum_{i \in |\mathsf{ch}(p)|} l_i(\boldsymbol{x})|_{\mathsf{supp}(p_i)} \,.
\end{aligned}
$$

For a smooth, decomposable, and deterministic product unit $p(\boldsymbol{x}) = p_1(\boldsymbol{x})p_2(\boldsymbol{x})$, its logarithm circuit can be decomposed as sum of the logarithm circuits of its child units:

$$
\begin{aligned}
&l(\boldsymbol{x})|_{\mathsf{supp}(\boldsymbol{x})} = \log\left(p_1(\boldsymbol{x}_1)p_2(\boldsymbol{x}_2)\right) \cdot [\![\boldsymbol{x} \in \mathsf{supp}(p)]\!] \\
&= \log p_1(\boldsymbol{x}_1)[\![\boldsymbol{x} \in \mathsf{supp}(p)]\!] + \log p_2(\boldsymbol{x}_2)[\![\boldsymbol{x} \in \mathsf{supp}(p)]\!] \\
&= \log p_1(\boldsymbol{x}_1)[\![\boldsymbol{x}_1 \in \mathsf{supp}(p_1)]\!][\![\boldsymbol{x}_2 \in \mathsf{supp}(p_2)]\!] + \log p_2(\boldsymbol{x}_2)[\![\boldsymbol{x}_2 \in \mathsf{supp}(p_2)]\!][\![\boldsymbol{x}_1 \in \mathsf{supp}(p_1)]\!] \\
&= l(\boldsymbol{x}_1)|_{\mathsf{supp}(p_1)}\, [\![\boldsymbol{x}_2 \in \mathsf{supp}(p_2)]\!] + l(\boldsymbol{x}_2)|_{\mathsf{supp}(p_2)}\, [\![\boldsymbol{x}_1 \in \mathsf{supp}(p_1)]\!].
\end{aligned}
$$

Note that in both case, the support circuits (e.g., $[\![\boldsymbol{x} \in \mathsf{supp}(p)]\!]$) are used to enforce smoothness in the output circuit. Algorithm 14 illustrates the whole algorithm in detail, showing that the construction of these support circuits can be done in linear time by caching intermediate sub-circuits while calling Algorithm 9. Furthermore, the newly introduced

product units, i.e., $l(\boldsymbol{x}_1)|_{\mathsf{supp}(p_1)}\,[\![\boldsymbol{x}_2 \in \mathsf{supp}(p_2)]\!]$, $l(\boldsymbol{x}_2)|_{\mathsf{supp}(p_2)}\,[\![\boldsymbol{x}_1 \in \mathsf{supp}(p_1)]\!]$, and the additional support input unit $\log\theta_i[\![\boldsymbol{x} \in \mathsf{supp}(p_i)]\!]$ share the same support of p by construction. Fig. A.1 illustrates this property with one example. This implies that when a deterministic circuit and its logarithmic circuit are going to be multiplied, e.g., when computing entropies (Section A.3.2), we can check for their support to overlap in linear time (Algorithm 11). □

Proof of hardness. We will prove hardness of computing the logarithm by showing hardness of computing the partition function of the logarithm of a circuit. Let $\mathbf{X} = \{X_1, \ldots, X_n\}$ be a collection of binary variables, and p a smooth and decomposable PC over $\mathbf{X}$ where $p(\boldsymbol{x}) > 0$ for all $\boldsymbol{x} \in \mathsf{val}(\mathbf{X})$. Then computing the quantity

$$\sum_{\boldsymbol{x}\in\mathsf{val}(\mathbf{X})} \log p(\boldsymbol{x}) \tag{LOGPC}$$

is #P-hard.

The proof is by reduction from #NUMPAR, the counting problem of the number partitioning problem (NUMPAR) defined as follows. Given n positive integers $k_1, \ldots, k_n$, we want to decide whether there exists a subset $S \subset [n]$ such that $\sum_{i\in S} k_i = \sum_{i\notin S} k_i$. NUMPAR is NP-complete, and #NUMPAR which asks for the number of solutions is known to be #P-hard.

We will show that we can solve #NUMPAR using an oracle for LOGPC, which will imply that LOGPC is also #P-hard. First, consider the following quantity SL for a given weight function $w(\cdot)$:

$$\begin{aligned}\mathsf{SL} &:= \sum_{\boldsymbol{x}\in\mathsf{val}(\mathbf{X})} \log(\sigma(w(\boldsymbol{x})) + 1) = \sum_{\boldsymbol{x}\in\mathsf{val}(\mathbf{X})} \log\left(\frac{1}{1+e^{-w(\boldsymbol{x})}} + 1\right) = \sum_{\boldsymbol{x}\in\mathsf{val}(\mathbf{X})} \log\left(\frac{2+e^{-w(\boldsymbol{x})}}{1+e^{-w(\boldsymbol{x})}}\right) \\ &= \sum_{\boldsymbol{x}\in\mathsf{val}(\mathbf{X})} \log(2+e^{-w(\boldsymbol{x})}) - \sum_{\boldsymbol{x}\in\mathsf{val}(\mathbf{X})} \log(1+e^{-w(\boldsymbol{x})}).\end{aligned}$$

Similar to the construction in the proof of Theorem 4, we can construct smooth and decomposable, unnormalized PCs for $2 + e^{-w(\boldsymbol{x})}$ and $1 + e^{-w(\boldsymbol{x})}$ of size linear in n. Then, we can compute SL via two calls to the oracle for LOGPC on these PCs.

Next, we choose the weight function $w(\cdot)$ such that SL can be used to answer #NUMPAR. For a given instance of NUMPAR described by $k_1, \ldots, k_n$ and a large integer m, which will be chosen later, we define the following weight function:

$$w(\boldsymbol{x}) := -\frac{m}{2} - m\sum_i k_i + 2m\sum_i k_i x_i.$$

In other words, $w(\boldsymbol{x}) = w_0 + \sum_i w_i x_i$ where $w_0 = -m/2 - m\sum_i k_i$ and $w_i = 2mk_i$ for $i = 1, \ldots, n$. Here, an assignment $\boldsymbol{x}$ corresponds to a subset $S_{\boldsymbol{x}} = \{i | x_i = 1, x_i \in \boldsymbol{x}\}$. Then the assignment $1 - \boldsymbol{x}$ corresponds to the complement $S_{1-\boldsymbol{x}} = \overline{S_{\boldsymbol{x}}}$. In the following, we will consider pairs of assignments $(\boldsymbol{x}, 1-\boldsymbol{x})$ and say that it is a solution to NUMPAR if $S_{\boldsymbol{x}}$ and by extension $S_{1-\boldsymbol{x}}$ are solutions to NUMPAR.

Observe that if $(\boldsymbol{x}, 1-\boldsymbol{x})$ is a solution to NUMPAR, then $w(\boldsymbol{x}) = w(1-\boldsymbol{x}) = -m/2$. Otherwise, one of their weights must be $\geq m/2$ and the other $\leq -3m/2$. We can then deduce the following facts about the *contribution* of each pair to SL, defined as $c(\boldsymbol{x}, 1-\boldsymbol{x}) = \log(\sigma(w(\boldsymbol{x})) + 1) + \log(\sigma(w(1-\boldsymbol{x})) + 1)$.

If the pair $(\boldsymbol{x}, 1-\boldsymbol{x})$ is a solution to NUMPAR, then its contribution to SL is going to be:

$$c(\boldsymbol{x}, 1-\boldsymbol{x}) = 2\log(\sigma(-m/2) + 1).$$

Otherwise, we can bound its contribution as follows:

$$\log(\sigma(m/2) + 1) \leq c(\boldsymbol{x}, 1-\boldsymbol{x}) \leq 1 + \log(\sigma(-3m/2) + 1)$$

If there are k pairs that are solutions to the NUMPAR problem, then using the above observations we have the following bounds on SL:

$$\mathsf{SL} \geq (2^{n-1} - k)\log(\sigma(m/2) + 1) + 2k\log(\sigma(-m/2) + 1) \geq (2^{n-1} - k)\log(\sigma(m/2) + 1), \tag{A.1}$$

$$\mathsf{SL} \leq (2^{n-1} - k)(1 + \log(\sigma(-3m/2) + 1)) + 2k \log(\sigma(-m/2) + 1). \tag{A.2}$$

Suppose for some given $\epsilon > 0$, we select m such that it satisfies both $1 - \epsilon \leq \log(\sigma(m/2)+1)$ and $\log(\sigma(-m/2) + 1) \leq \epsilon$. First, this implies that m also satisfies the following:

$$1 + \log(\sigma(-3m/2) + 1)) \leq 1 + \log(\sigma(-m/2) + 1) \leq 1 + \epsilon.$$

Plugging in above inequalities to Eqs. (A.1) and (A.2), we get the following bounds on SL w.r.t. ϵ and k:

$$(2^{n-1} - k)(1 - \epsilon) \leq \mathsf{SL} \leq (2^{n-1} - k)(1 + \epsilon) + 2k\epsilon.$$

We can alternatively express this as the following bounds on k:

$$\frac{2^{n-1}(1 - \epsilon) - \mathsf{SL}}{1 - \epsilon} \leq k \leq \frac{2^{n-1}(1 + \epsilon) - \mathsf{SL}}{1 - \epsilon}.$$

The difference between the upper and lower bounds on k is equal to $2^n\epsilon/(1 - \epsilon)$. If this difference is less than 1—e.g. by setting $\epsilon = 1/(2^n + 2)$—we can exactly solve for k. In particular, it must be equal to the ceiling of the lower bound as well as the floor of the upper bound. Moreover, the answer to $\#\mathsf{NUMPAR}$ is given by $2k$. This concludes the proof that computing LOGPC is #P-hard. □

A.2.6 Exponential Function of a Circuit

Theorem 25 (Hardness of the exponential of a circuit)**.** *Let p be a smooth and decomposable circuit over variables $\mathbf{X}$. Then, computing its exponential $\exp(p(\mathbf{X}))$ as a decomposable circuit is #P-hard, even if p is structured-decomposable.*

Proof. We will prove hardness of computing the exponential by showing hardness of computing

Algorithm 14 LOGARITHM$(p, \mathsf{cache}_l, \mathsf{cache}_s)$

Input: a smooth, deterministic and decomposable PC $p(\mathbf{X})$ and two caches for memoization (cache_l for the logarithmic circuit and cache_s for the support circuit).
Output: a smooth and decomposable circuit $l(\mathbf{X})$ encoding $\log(p(\mathbf{X}))$
if $p \in \mathsf{cache}_l$ **then return** $\mathsf{cache}_l(p)$
if p is an input unit **then**
  $l \leftarrow \textsc{Input}(\log\left(p_{|\mathsf{supp}(p)}\right), \phi(p))$
else if p is a sum unit **then**
  $n \leftarrow \{\}$
  for $i = 1$ **to** $|\mathsf{ch}(p)|$ **do**
    $n \leftarrow n \cup \{\textsc{Support}(p_i, \mathsf{cache}_s)\} \cup \{\textsc{Logarithm}(p_i, \mathsf{cache}_l)\}$
  $l \leftarrow \textsc{Sum}(n, \{\log\theta_1, 1, \log\theta_2, 1, \ldots, \log\theta_{|\mathsf{ch}(p)|}, 1\})$
else if p is a product unit **then**
  $n \leftarrow \{\}$
  for $i = 1$ **to** $|\mathsf{ch}(p)|$ **do**
    $n \leftarrow n \cup \{\textsc{Product}(\{\textsc{Logarithm}(p_i, \mathsf{cache}_l)\} \cup \{\textsc{Support}(p_j, \mathsf{cache}_s)\}_{j\neq i})\}$
  $l \leftarrow \textsc{Sum}(n, \{1\}_{i=1}^{|\mathsf{ch}(p)|})$
$\mathsf{cache}_l(p) \leftarrow l$
return l

the partition function of the exponential of a circuit. Let $\mathbf{X} = \{X_1, \ldots, X_n\}$ be a collection of binary variables with values in $\{-1, +1\}$ and let p be a smooth and decomposable PC over $\mathbf{X}$ then computing the quantity

$$\sum_{\boldsymbol{x}\in\mathsf{val}(\mathbf{X})} \exp\left(p(\boldsymbol{x})\right) \tag{EXPOPC}$$

is #P-hard.

The proof is a reduction from the problem of computing the partition function of an Ising model, ISING which is known to be #P-complete [70]. Given a graph $G = (V, E)$ with n vertexes, computing the partition function of an Ising model associated to G and equipped with potentials associated to its edges ($\{w_{u,v}\}_{(u,v)\in E}$) and vertexes ($\{w_v\}_{v\in V}$) equals to

$$\sum_{\boldsymbol{x}\in\mathsf{val}(\mathbf{X})} \exp\left(\sum_{(u,v)\in E} w_{u,v}x_u x_v + \sum_{v\in V} w_v x_v\right). \tag{ISING}$$

The reduction is made by constructing a smooth and decomposable circuit $p(\mathbf{X})$ that computes $\sum_{(u,v)\in E} w_{u,v}x_u x_v + \sum_{v\in V}$. This can be done by introducing a sum units with

$|E|+|V|$ inputs that are product units and with weights $\{w_{u,v}\}_{(u,v)\in E} \cup \{w_v\}_{v\in V}$. The first $|E|$ product units receive inputs from n input distributions where only 2 corresponds to the binary indicator inputs X_u and X_v for an edge $(u, v) \in E$ while the remaining $n-2$ are uniform distributions outputting 1 for all the possible states of variables $\mathbf{X} \setminus \{X_u, X_v\}$. Analogously, the remaining $|V|$ product units receive input from n of which only one, corresponding to the vertex $v \in V$ is an indicator unit over X_v, while the remaining are uniform distributions for variables in $\mathbf{X} \setminus \{X_v\}$. □

Theorem 26 (Tractable exponential of a linear circuit)**.** *Let p be a linear circuit over variables $\mathbf{X}$, i.e., $p(\mathbf{X}) = \sum_i \theta_i \cdot X_i$. Then $\exp(p(\mathbf{X}))$ can be represented as an omni-compatible circuit with a single product unit in $\mathcal{O}(|p|)$ time.*

Proof. The proof follows immediately by the properties of exponentials of sums. Algorithm 15 formalizes the construction. □

Algorithm 15 EXPONENTIAL(p)

Input: a smooth circuit p encoding $p(\mathbf{X}) = \theta_0 + \sum_{i=1}^{n} \theta_i X_i$
Output: its exponential circuit encoding $\exp(p(\mathbf{X}))$
$e \leftarrow \{\text{INPUT}(\exp(\theta_0 + \theta_1 X_1), X_1)\}$
for $i = 2$ **to** n **do**
    $e \leftarrow e \cup \{\text{INPUT}(\exp(\theta_i X_i), X_i)\}$
return PRODUCT(e)

A.2.7 Other tractable operators over circuits

This section proves Lemma 2, which states that any operator over circuits that should yield a decomposable and smooth circuit as output must take the form of a sum, power, logarithm or exponential.

Lemma 2 (Atlas Completeness)**.** *Let f be a continuous function. If (1) $f : \mathbb{R} \to \mathbb{R}$ satisfies $f(x + y) = f(x) + f(y)$ then it is a linear function $\beta \cdot x$; if (2) $f : \mathbb{R}_+ \to \mathbb{R}_+$ satisfies*

$f(x \cdot y) = f(x) \cdot f(y)$, then it takes the form x^β; if (3) instead $f : \mathbb{R}_+ \to \mathbb{R}$ satisfies $f(x \cdot y) = f(x) + f(y)$, then it takes the form $\beta \log(x)$; and if (4) $f : \mathbb{R} \to \mathbb{R}_+$ satisfies that $f(x + y) = f(x) \cdot f(y)$ then it is of the form $\exp(\beta \cdot x)$, for a certain $\beta \in \mathbb{R}$.

Proof. The proof of all properties follows from constructing f such that we obtain a *Cauchy functional equation* [71, 152].

The condition (1) exactly takes the form of a Cauchy functional equation, then it must hold that $f(x) = \beta \cdot x$.

For condition (2), let $g(x) = \log(f(\exp(x)))$ for all $x \in \mathbb{R}$, which is continuous because f is. Then, it follows that

$$\begin{aligned} g(x+y) &= \log(f(\exp(x+y))) = \log(f(\exp(x) \cdot \exp(y))) = \log(f(\exp(x))) + \log(f(\exp(y))) \\ &= g(x) + g(y). \end{aligned}$$

Therefore, $g(x)$ assumes the Cauchy functional form and, as in case (1), it is equal to $\beta \cdot x$. β can be retrieved by solving $\beta \cdot x = \log(f(\exp(x)))$ for $x = 1$. This gives $\beta = \log(f(e))$. Applying the definition of g, we can hence write

$$f(\exp(x)) = e^{g(x)} = e^{\beta \cdot x} = (e^x)^\beta$$

Let $y \in \mathbb{R}_+$. Using the identity $y = e^{\log(y)}$ it follows that:

$$f(y) = f(e^{\log(y)}) = \left(e^{\log(y)}\right)^\beta = y^\beta.$$

Condition (3) follows an analogous pattern. Let $g(x) = f(\exp(x))$ for all $x \in \mathbb{R}$, which is continuous as f is. Once again, g satisfies the Cauchy functional form:

$$g(x+y) = f(\exp(x+y)) = f(\exp(x) \cdot \exp(y)) = f(\exp(x)) + f(\exp(y)) = g(x) + g(y).$$

Therefore, $g(x)$ must be of the form $\beta \cdot x$ for $\beta = f(e)$. Hence, $f(y) = \beta \log(y)$.

Lastly, for condition (4), $g(x) = \log(f(x))$ for all $x \in \mathbb{R}$, which is continuous if f is. Then, we can retrieve the Cauchy functional by

$$g(x+y) = \log(f(x+y)) = \log(f(x) \cdot f(y)) = \log(f(x)) + \log(f(y)) = g(x) + g(y).$$

Therefore, $g(x)$ must be of the form $\beta \cdot x$. Hence, $f(y) = \exp(\beta \cdot y)$. □

In summary, Lemma 2 states that if we want to enlarge our atlas beyond sum and product circuit operators, we need to focus our attention over powers, logarithms and exponentials. At the same time, it states that no operator with a different functional form and yet yielding a circuit made of sum and product units can be found. Extending our atlas to deal with a new language of circuits is an interesting future research direction.

A.3 Complex Information-Theoretic Queries

This section collects the complete tractability and hardness results for the queries in Table 2.2. Note that the tractability proofs are succinct thanks to our atlas which allows to define a tractable model class effortlessly. Some hardness proofs also benefit from the hardness results we provided for the simple operators in the previous section.

A.3.1 Cross Entropy

Theorem 27. *Let p and q be two compatible PCs over variables $\mathbf{X}$, and also let q be deterministic. Then their cross-entropy, i.e.,*

$$-\int_{\mathsf{val}(\mathbf{X})} p(\boldsymbol{x}) \log(q(\boldsymbol{x})) d\mathbf{X},$$

restricted to the support of q can be exactly computed in $\mathcal{O}(|p||q|)$ time. If q is not deterministic, then computing their cross-entropy is #P-hard, even if p and q are compatible over $\mathbf{X}$.

Proof. ***(Tractability)*** From Theorem 24 we know that we can compute the logarithm of q in polytime, which is a PC of size $\mathcal{O}(|q|)$ that is compatible with q and hence with p. Therefore, multiplying p and $\log q$ according to Theorem 1 can be done exactly in polytime and yields a circuit of size $\mathcal{O}(|p||q|)$ that is still smooth and decomposable, hence we can tractably compute its partition function.

(Hardness) The proof consists of a simple reduction from LOGPC from Theorem 24. We know that computing LOGPC for a smooth and decomposable PC over binary variables $\mathbf{X}$ is #P-hard. We can reduce this to computing the cross entropy between $p = 1$, which can be constructed as an omni-compatible circuit (Section A.1.2), and the original PC of the LOGPC problem. Thus, the cross-entropy of two compatible circuits is a #P-hard problem. □

A.3.2 Entropy

Theorem 28. *Let p be a smooth, deterministic, and decomposable PC over variables $\mathbf{X}$. Then its entropy,*[2] *defined as*

$$-\int_{\mathsf{val}(\mathbf{X})} p(\boldsymbol{x}) \log p(\boldsymbol{x})\, d\mathbf{X}$$

can be exactly computed in $\mathcal{O}(|p|)$ time. If p is smooth and decomposable but not deterministic, then computing its Shannon entropy, defined as

$$\textit{ENT}(p) := -\sum_{\mathsf{val}(\mathbf{X})} p(\boldsymbol{x}) \log(p(\boldsymbol{x}))d\mathbf{X} \qquad \text{(ENTPC)}$$

is coNP-hard.

Proof. ***(Tractability)*** Using Theorem 24 we can compute the logarithm of p in polytime as a smooth and decomposable PC of size $\mathcal{O}(|p|)$ which furthermore shares the same support partitioning with p. Therefore, multiplying p and $\log p$ according to Algorithm 11 can be done in polytime and yields a smooth and decomposable circuit of size $\mathcal{O}(|p|)$ since $\log p$ shares the

[2]For the continuous case this quantity refers to the *differential entropy*, while for the discrete case it is the Shannon entropy.

same support structure of p (Theorem 24). Therefore, we can compute the partition function of the resulting circuit in time linear in its size.

(Hardness) The hardness proof contains a polytime reduction from the coNP-hard 3UNSAT problem, defined as follows: given a set of n Boolean variables $\mathbf{X} = \{X_1, \ldots, X_n\}$ and a CNF with m clauses $\{c_1, \ldots, c_m\}$ (each clause contains exactly 3 literals), decide whether the CNF is unsatisfiable.

The reduction borrows two gadget circuits p_β and p_γ defined in Section A.1.3. They each represent a logical formula over an auxiliary set of variables, which we denote here $\mathbf{X}'$, and thus outputs 0 or 1 for all values of $\mathbf{X}'$. Moreover, by construction, $p_\beta \cdot p_\gamma$ is the constant function 0 if and only if the original CNF is unsatisfiable.

We further construct a circuit p_α as the summation over p_β and p_γ. Recall that p_β and p_γ can efficiently be constructed as smooth and decomposable circuits, and thus their sum can be represented as a smooth and decomposable circuit in polynomial time. We will now show that 3UNSAT can be reduced to checking whether the entropy of p_α is zero.

First, observe that for any assignment $\boldsymbol{x}'$ to $\mathbf{X}'$, $p_\alpha(\boldsymbol{x}')$ evaluates to 0, 1, or 2, because p_β and p_γ always evaluates to either 0 or 1. Moreover, if p_α only outputs 0 or 1 for all values of $\mathbf{X}'$, then $p_\beta \cdot p_\gamma$ must always be 0, implying that the original CNF is unsatisfiable. Lastly, in such a case, the entropy of p_α must be 0, whereas the entropy will be nonzero if there is an assignment $\boldsymbol{x}'$ such that $p_\alpha(\boldsymbol{x}') = 2$. This concludes the proof that computing the entropy of a smooth and decomposable PC is coNP-hard. □

A.3.3 Mutual Information

Theorem 29. *Let p be a deterministic and structured-decomposable PC over variables $\mathbf{Z} = \mathbf{X} \cup \mathbf{Y}$ ($\mathbf{X} \cap \mathbf{Y} = \varnothing$). Then the mutual information between $\mathbf{X}$ and $\mathbf{Y}$, defined as*

$$MI(p; \mathbf{X}, \mathbf{Y}) := \int_{\mathsf{val}(\mathbf{Z})} p(\boldsymbol{x}, \boldsymbol{y}) \log \frac{p(\boldsymbol{x}, \boldsymbol{y})}{p(\boldsymbol{x}) \cdot p(\boldsymbol{y})} d\mathbf{X} d\mathbf{Y},$$

can be exactly computed in $\mathcal{O}(|p|)$ time if p is still deterministic after marginalizing out $\mathbf{Y}$ as well as after marginalizing out $\mathbf{X}$.[3] *If p is instead smooth, decomposable, and deterministic, then computing the mutual information between $\mathbf{X}$ and $\mathbf{Y}$ is coNP-hard.*

Proof. ***(Tractability)*** From Theorem 24 we know that the logarithm circuits of $p(\mathbf{X}, \mathbf{Y})$, $p(\mathbf{X})[\![\boldsymbol{y} \in \mathsf{supp}(p(\mathbf{Y}))]\!]$, and $p(\mathbf{Y})[\![\boldsymbol{x} \in \mathsf{supp}(p(\mathbf{X}))]\!]$ can be computed in polytime and are smooth and decomposable circuits of size $\mathcal{O}(|p|)$ that furthermore share the same support partitioning with $p(\mathbf{Y}, \mathbf{Z})$. Therefore, we can multiply $p(\mathbf{X}, \mathbf{Y})$ with each of these logarithm circuits efficiently according to Theorem 2 to yield circuits of size $\mathcal{O}(|p|)$. These are still smooth and decomposable circuits. Hence we can compute their partition functions and compute the mutual information between $\mathbf{X}$ and $\mathbf{Y}$ w.r.t. p.

(Hardness) We show hardness for the case of Boolean inputs, which implies hardness in the general case. This proof largely follows the hardness proof of Theorem 28 to show that there is a polytime reduction from 3UNSAT to the mutual information of PCs. For a given CNF, suppose we construct p_β, p_γ, and $p_\alpha = p_\beta + p_\gamma$ over a set of Boolean variables, say $\mathbf{X}$, as shown in Section A.1.2 and Theorem 28.

Let $\mathbf{Y} = \{Y\}$ be a single Boolean variable, and define p_δ as:

$$p_\delta := p_\beta \times [\![Y = 1]\!] + p_\gamma \times [\![Y = 0]\!].$$

That is, we first construct two product units q_1, q_2 with inputs $\{p_\beta, [\![Y = 1]\!]\}$ and $\{p_\gamma, [\![Y = 0]\!]\}$, respectively, and build a sum unit p_δ with inputs $\{q_1, q_2\}$ and weights $\{1, 1\}$. Then p_δ has the following properties: **(1)** p_δ is smooth, decomposable, and deterministic, following from the fact that p_β and p_γ are also smooth, decomposable, and deterministic, and that q_1 and q_2 have no overlapping support. **(2)** $\textsc{ent}(p_\delta)$ can be computed in linear-time w.r.t. the circuit size by Theorem 28. **(3)** $p_\delta(Y = 1)$ and $p_\delta(Y = 0)$ can be computed in linear time

[3] This structural property of circuits is also known as marginal determinism [14] and has been introduced in the context of marginal MAP inference and the computation of same-decision probabilities of Bayesian classifiers [19, 128].

(w.r.t. size of the circuit p_δ), as p_δ admits tractable marginalization. **(4)** For any $\boldsymbol{x} \in \mathsf{val}(\mathbf{X})$, $p_\delta(\boldsymbol{x}) = p_\beta(\boldsymbol{x}) + p_\gamma(\boldsymbol{x}) = p_\alpha(\boldsymbol{x})$.

We can express the mutual information $\text{MI}(p_\delta; \mathbf{X}, \mathbf{Y})$ as:

$$\text{MI}(p_\delta; \mathbf{X}, \mathbf{Y}) = \text{ENT}(p_\delta) - p_\delta(Y=1)\log p_\delta(Y=1) - p_\delta(Y=0)\log p_\delta(Y=0) - \text{ENT}(p_\alpha).$$

Therefore, given an oracle that computes $\text{MI}(p_\delta; \mathbf{X}, \mathbf{Y})$, we can check if it is equal to $\text{ENT}(p_\delta) - p_\delta(Y=1)\log p_\delta(Y=1) - p_\delta(Y=0)\log p_\delta(Y=0)$, which is equivalent to checking $\text{ENT}(p_\alpha) = 0$, and decide whether the original CNF is unsatisfiable. Hence, computing the mutual information of smooth, deterministic, and decomposable PCs is a coNP-hard problem. □

A.3.4 Kullback-Leibler Divergence

Theorem 30. *Let p and q be two deterministic and compatible PCs over variables $\mathbf{X}$. Then, their intersectional Kullback-Leibler divergence (KLD), defined as*

$$\mathbb{D}_{\mathsf{KL}}(p \parallel q) = \int_{\mathsf{supp}(p)\cap\mathsf{supp}(q)} p(\boldsymbol{x}) \log \frac{p(\boldsymbol{x})}{q(\boldsymbol{x})} d\mathbf{X},$$

can exactly be computed in $\mathcal{O}(|p||q|)$ time. If p and q are not deterministic, then computing their KLD is #P-hard, even if they are compatible.

Proof. ***(Tractability)*** Tractability of the intersectional KLD can be concluded directly from the tractability of cross entropy and entropy (Theorem 27 and 28). Specifically, KLD can be expressed as the difference between cross entropy and entropy:

$$\int p(\boldsymbol{x}) \log \frac{p(\boldsymbol{x})}{q(\boldsymbol{x})}\, d\mathbf{X} = \int p(\boldsymbol{x}) \log p(\boldsymbol{x})\, d\mathbf{X} - \int p(\boldsymbol{x}) \log q(\boldsymbol{x})\, d\mathbf{X}.$$

We can compute the entropy of a smooth, decomposable, and deterministic PC p in $\mathcal{O}(|p|)$; and the cross entropy between two deterministic and compatible PCs p and q in $\mathcal{O}(|p||q|)$ time.

(Hardness) The proof proceeds similarly to the hardness proof of Theorem 27. Recall that the LOGPC problem from Theorem 24 is #P-hard for a smooth and decomposable PC over binary variables. We can reduce this to computing the negative of KL divergence between $p = 1$, which can be constructed as an omni-compatible circuit (Section A.1.2), and q the original PC of the LOGPC problem. Thus, the KLD of two compatible circuits is a #P-hard problem. □

A.3.5 Rényi Entropy

Definition 12 (Rényi entropy)**.** The Rényi entropy of order $\alpha \in \mathbb{R}$ of a PC p is defined as

$$\frac{1}{1-\alpha} \log \int_{\mathsf{supp}(p)} p^{\alpha}(\boldsymbol{x}) d\mathbf{X}.$$

Theorem 31 (Rényi entropy for natural α)**.** *Let p be a structured-decomposable PC over variables $\mathbf{X}$ and $\alpha \in \mathbb{N}$. Its Rényi entropy can be computed in $\mathcal{O}(|p|^{\alpha})$ time. If p is instead smooth and decomposable, then computing its Rényi entropy of order α is #P-hard.*

Proof. ***(Tractability)*** Tractability easily follows from computing the natural power circuit of p, which takes $\mathcal{O}(|p|^{\alpha})$ time according to Theorem 3.

(Hardness) We show hardness for the case of discrete inputs. The hardness of computing the Rényi entropy for natural number α is implied by the hardness of computing the natural power of smooth and decomposable PCs. Specifically, we conclude the proof by observing that there exists a polytime reduction from POW2PC, defined as $\sum_{\boldsymbol{x} \in \mathsf{val}(\mathbf{X})} p^2(\boldsymbol{x})$, a #P-hard problem as proved in Theorem 3, to Rényi entropy with $\alpha = 2$. □

Theorem 32 (Rényi entropy for real α)**.** *Let p be a smooth, decomposable, and deterministic PC over variables $\mathbf{X}$ and $\alpha \in \mathbb{R}_+$. Its Rényi entropy can be computed in $\mathcal{O}(|p|)$ time. If p is not deterministic, then computing its Rényi entropy of order α is #P-hard, even if p is structured-decomposable.*

Proof. ***(Tractability)*** Tractability easily follows from computing the power circuit of p, which takes $\mathcal{O}(|p|)$ time according to Theorem 5.

(Hardness) Similar to the hardness proof of Theorem 31, this hardness result follows from the fact that computing the reciprocal of a structured-decomposable circuit is #P-hard (Theorem 4). Again, this is demonstrated by a polytime reduction from INVPC (i.e., $\sum_{\boldsymbol{x}\in\mathsf{val}(\mathbf{X})} p^{-1}(\boldsymbol{x})$) to Rényi entropy with $\alpha = -1$. □

A.3.6 Rényi's alpha divergence

Definition 13 (Rényi's α-divergence)**.** The Rényi's α-divergence of two PCs p and q is defined as

$$\mathbb{D}_\alpha(p \parallel q) = \frac{1}{1-\alpha} \log \int_{\mathsf{supp}(p)\cap\mathsf{supp}(q)} p^\alpha(\boldsymbol{x}) q^{1-\alpha}(\boldsymbol{x}) d\mathbf{X}.$$

Theorem 33 (Hardness of alpha divergence of two PCs)**.** *Let p and q be two smooth and decomposable PCs over variables $\mathbf{X}$. Then computing their Rényi's α-divergence for $\alpha \in \mathbb{R} \setminus \{1\}$ is #P-hard, even if p and q are compatible.*

Proof. Suppose p is a smooth and decomposable PC $\mathbf{X}$ representing the constant function 1, which can be constructed as in Section A.1.2. Then p^α is also a constant 1. Hence, computing Rényi's 2-divergence between p and another smooth and decomposable PC q is as hard as computing the reciprocal of q, which is #P-hard (Theorem 4). □

Theorem 34 (Tractable alpha divergence of two PCs)**.** *Let p and q be compatible PCs over variables $\mathbf{X}$. Then their Rényi's α-divergence can be exactly computed in $\mathcal{O}(|p|^\alpha|q|)$ time for $\alpha \in \mathbb{N}, \alpha > 1$ if q is deterministic or in $\mathcal{O}(|p||q|)$ for $\alpha \in \mathbb{R}, \alpha \neq 1$ if p and q are both deterministic.*

Proof. The proof easily follows from first computing the power circuit of p and q according to Theorem 5 or Theorem 3 in polytime. Depending on the value of α, the resulting circuits will

have size $\mathcal{O}(|p|^\alpha)$ and $\mathcal{O}(|q|)$ for $\alpha \in \mathbb{N}$ or $\mathcal{O}(|p|)$ and $\mathcal{O}(|q|)$ for $\alpha \in \mathbb{R}$ and will be compatible with the input circuits. Then, since they are compatible between themselves, their product can be done in polytime (Theorem 2) and it is going to be a smooth and decomposable PC of size $\mathcal{O}(|p|^\alpha|q|)$ (for $\alpha \in \mathbb{N}$) or $\mathcal{O}(|p||q|)$ (for $\alpha \in \mathbb{R}$), for which the partition function can be computed in time linear in its size. □

A.3.7 Itakura-Saito Divergence

Theorem 35. *Let p and q be two deterministic and compatible PCs over variables $\mathbf{X}$, with bounded intersectional support $\mathsf{supp}(p) \cap \mathsf{supp}(q)$. Then their Itakura-Saito divergence, defined as*

$$\mathbb{D}_{\mathsf{IS}}(p \parallel q) = \int_{\mathsf{supp}(p)\cap\mathsf{supp}(q)} \left(\frac{p(\boldsymbol{x})}{q(\boldsymbol{x})} - \log \frac{p(\boldsymbol{x})}{q(\boldsymbol{x})} - 1 \right) d\mathbf{X}, \tag{A.3}$$

can be exactly computed in $\mathcal{O}(|p||q|)$ time. If p and q are instead compatible but not deterministic, then computing their Itakura-Saito divergence is #P-hard.

Proof. ***(Tractability)*** The proof easily follows from noting that the integral decomposes into three integrals over the inner sum: $\int_{\mathsf{supp}(p)\cap\mathsf{supp}(q)} \frac{p(\boldsymbol{x})}{q(\boldsymbol{x})}\, d\mathbf{X} - \int_{\mathsf{supp}(p)\cap\mathsf{supp}(q)} \log \frac{p(\boldsymbol{x})}{q(\boldsymbol{x})}\, d\mathbf{X}$ - $\int_{\mathsf{supp}(p)\cap\mathsf{supp}(q)} 1\, d\mathbf{X}$.. Then, the first integral over the quotient can be solved $\mathcal{O}(|p||q|)$ (Theorem 23); the second integral over the log of a quotient of two PCs can be computed in time $\mathcal{O}(|p||q|)$ (Theorem 23 and 24) and finally the last one integrates to the dimensionality of $|\mathsf{supp}(p) \cap \mathsf{supp}(q)|$, which we assume to exist.

(Hardness) We show hardness for the case of binary variables $\mathbf{X} = \{X_1, \ldots, X_n\}$. Suppose q is an omni-compatible circuit representing the constant function 1, which can be constructed as in Section A.1.2. As such, integration in Eq. (A.3) becomes the summation $\sum_{\mathsf{val}(\mathbf{X})} p(\boldsymbol{x}) - \sum_{\mathsf{val}(\mathbf{X})} \log p(\boldsymbol{x}) - 2^n$. Hence, computing $\mathbb{D}_{\mathsf{IS}}$ must be as hard as computing $\sum_{\mathsf{val}(\mathbf{X})} \log p(\boldsymbol{x})$, since the first sum can be efficiently computed as p must be smooth and decomposable by assumption and the last one is a constant. That is, we reduced the problem of

computing the logarithm of the non-deterministic circuit (LOGPC, Theorem 24) to computing $\mathbb{D}_{\mathsf{IS}}$. □

A.3.8 Cauchy-Schwarz Divergence

Theorem 36. *Let p and q be two structured-decomposable and compatible PCs over variables $\mathbf{X}$. Then their Cauchy-Schwarz divergence, defined as*

$$\mathbb{D}_{\mathsf{CS}}(p \parallel q) = -\log \frac{\int_{\boldsymbol{x}\in\mathsf{val}(\mathbf{X})} p(\boldsymbol{x})q(\boldsymbol{x})\, d\mathbf{X}}{\sqrt{\int_{\boldsymbol{x}\in\mathsf{val}(\mathbf{X})} p^2(x)\, d\mathbf{X} \int_{\boldsymbol{x}\in\mathsf{val}(\mathbf{X})} q^2(x)\, d\mathbf{X}}},$$

can be exactly computed in time $\mathcal{O}(|p||q|+|p|^2+|q|^2)$. If p and q are instead structured-decomposable but not compatible, then computing their Cauchy-Schwarz divergence is #P-hard.

Proof. ***(Tractability)*** The proof easily follows from noting that the numerator inside the log can be computed in $\mathcal{O}(|p||q|)$ time as a product of two compatible circuits (Theorem 2); and the integrals inside the square root at the denominator can both be solved in $\mathcal{O}(|p|^2)$ and $\mathcal{O}(|q|^2)$ respectively as natural powers of structured-decomposable circuits (Theorem 3).

(Hardness) The proof follows by noting that if p and q are structured-decomposable, then computing the denominator inside the log can be exactly done in $|p|^2 + |q|^2$ because they are natural powers of structured-decomposable circuits (Theorem 3). Then $\mathbb{D}_{\mathsf{CS}}$ must be as hard as a the product of two non-compatible circuits. Therefore we can reduce MULPC (Theorem 1) to computing $\mathbb{D}_{\mathsf{CS}}$. □

A.3.9 Squared Loss Divergence

Theorem 37. *Let p and q be two structured-decomposable and compatible PCs over variables $\mathbf{X}$. Then their squared loss, defined as*

$$\mathbb{D}_{\mathsf{SL}}(p \parallel q) = \int_{\mathsf{val}(\mathbf{X})} \left(p(\boldsymbol{x}) - q(\boldsymbol{x})\right)^2\, d\mathbf{X},$$

can be computed exactly in time $\mathcal{O}(|p||q|+|p|^2+|q|^2)$. *If* p *and* q *are structured-decomposable but not compatible, then computing their squared loss is #P-hard.*

Proof. ***(Tractability)*** Proof follows by noting that the integral decomposes over the expanded square as $\int_{\mathsf{val}(\mathbf{X})} p^2(\boldsymbol{x})\, d\mathbf{X} + \int_{\mathsf{val}(\mathbf{X})} q^2(\boldsymbol{x})\, d\mathbf{X} - 2\int_{\mathsf{val}(\mathbf{X})} p(\boldsymbol{x})q(\boldsymbol{x})\, d\mathbf{X}$ and as such each integral can be computed by leveraging the tractable power of structured-decomposable circuits (Theorem 3) and the tractable product of compatible circuits (Theorem 2) and therefore the overall complexity is given by the maximum of the three.

(Hardness) Proof follows by noting that the integral decomposes over the expanded square as $\int_{\mathsf{val}(\mathbf{X})} p^2(\boldsymbol{x})\, d\mathbf{X} + \int_{\mathsf{val}(\mathbf{X})} q^2(\boldsymbol{x})\, d\mathbf{X} - 2\int_{\mathsf{val}(\mathbf{X})} p(\boldsymbol{x})q(\boldsymbol{x})\, d\mathbf{X}$ and that the first two terms can be computed in polytime as natural powers of structured-decomposable circuits (Theorem 3), hence computing $\mathbb{D}_{\mathsf{SL}}$ must be as hard as computing the product of two non-compatible circuits. Therefore we can reduce MULPC (Theorem 1) to computing $\mathbb{D}_{\mathsf{SL}}$. □

A.4 Expectation-based queries

This section completes the discussion around the complex queries that can be dealt with our atlas and details the expectations briefly discussed at the end of Section 2.5.

A.4.1 Moments of a distribution

Proposition 10 (Tractable moments of a PC)**.** *Let* $p(\mathbf{X})$ *be a smooth and decomposable PC over variables* $\mathbf{X} = \{X_1, \ldots, X_d\}$*, then for a set of natural numbers* $\mathbf{k} = (k_1, \ldots, k_d)$*, its* $\mathbf{k}-$*moment, defined as*

$$\int_{\mathsf{val}(\mathbf{X})} x_1^{k_1} x_2^{k_2} \ldots x_d^{k_d} p(\boldsymbol{x})\, d\mathbf{X}$$

can be computed exactly in time $\mathcal{O}(|p|)$.

Proof. The proof directly follows from representing $x_1^{k_1} x_2^{k_2} \ldots x_d^{k_d}$ as an omni-compatible

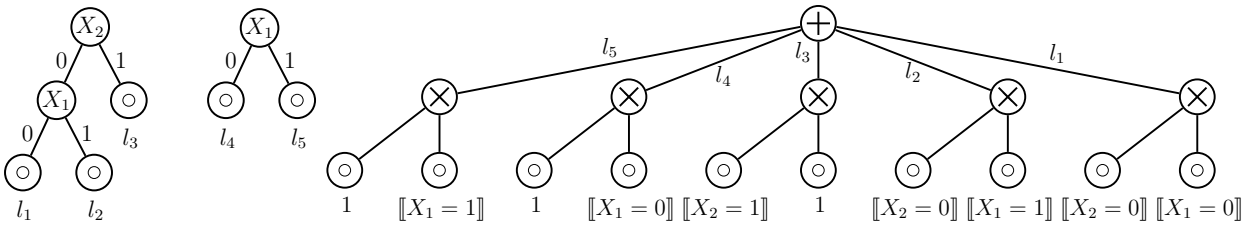


Figure A.2: Encoding an additive ensemble of two trees over $\mathbf{X} = \{X_1, X_2\}$ (left) in an omni-compatible circuit over $\mathbf{X}$ (right).

circuit comprising a single product unit over d input units, each encoding $x_i^{k_i}$, and then applying Corollary 1. □

A.4.2 Probability of logical formulas

Proposition 11 (Tractable probability of a logical formula)**.** *Let p be a smooth and decomposable PC over variables $\mathbf{X}$ and f an indicator function that represents a logical formula over $\mathbf{X}$ that can be compiled into a circuit compatible with p.*[4] *Then computing $\mathbb{P}_p[f]$ can be done in $\mathcal{O}(|p||f|)$ time.*

Proof. It follows directly from Theorem 1, by noting that $\mathbb{P}_p[f] = \mathbb{E}_{\boldsymbol{x}\sim p(\mathbf{X})}[f(\boldsymbol{x})]$ and hence a tractable product between p and f suffices. □

A.4.3 Expected predictions

Example 1 (Decision trees as circuits)**.** Let $\mathcal{F}$ be an additive ensemble of (decision or regression) trees over variables $\mathbf{X}$, also called a *forest*, and computing

$$\mathcal{F}(\boldsymbol{x}) = \sum_{\mathcal{T}_i \in \mathcal{F}} \theta_i \mathcal{T}_i(\boldsymbol{x})$$

[4] E.g. by compiling it into an SDD [12, 36] whose vtree encodes the hierarchical scope partitioning of p.

for some input configuration $\boldsymbol{x} \in \mathsf{val}(\mathbf{X})$ and each $\mathcal{T}_i$ realizing a tree, i.e., a function of the form

$$\mathcal{T}(\boldsymbol{x}) = \sum_{p_j \in \mathsf{paths}(\mathcal{T})} l_j \cdot \prod_{X_k \in \phi(p_j)} [\![x_k \leq \delta_k]\!]$$

where the outer sum ranges over all possible paths in tree $\mathcal{T}$, $l_j \in \mathbb{R}$ is the label (class or predicted real) associated to the leaf of that path, and the product is over indicator functions encoding the decision to take one branch of the tree in path p_j if x_k, the observed value for variable X_k appearing in the decision node, i.e., satisfies the condition $[\![x_k \leq \delta_k]\!]$ for a certain threshold $\delta_k \in \mathbb{R}$.

Then, it is easy to transform $\mathcal{F}$ into an omni-compatible circuit $p(\mathbf{X})$ of the form

$$p(\boldsymbol{x}) = \sum_{\mathcal{T}_i \in \mathcal{F}, p_j \in \mathsf{paths}(\mathcal{T}_i)} l_j \cdot \prod_{X_k \in \phi(p_j)} [\![x_k \leq \delta_k]\!] \cdot \prod_{X'_k \notin \phi(p_j)} 1$$

with a single sum unit realizing the outer sum and as many input product units as paths in the forest, each of which realizing a fully-factorized model over $\mathbf{X}$, and weighted by l_j. One example is shown in Fig. A.2.

Proposition 12 (Tractable expected predictions of additive ensembles of trees)**.** *Let p be a smooth and decomposable PC and f an additive ensemble of k decision trees over variables $\mathbf{X}$ and bounded depth. Then, its expected predictions can be exactly computed in $\mathcal{O}(k|p|)$.*

Proof. Recall that an additive ensemble of decision trees can be encoded as an omni-compatible circuit. Then, proof follows from Corollary 1. □

Proposition 13 (Tractable expected predictions of deep regressors (regression circuits))**.** *Let p be a structured-decomposable PC over variables $\mathbf{X}$ and f be a regression circuit [75]*

Algorithm 16 RGCtoCircuit$(r, \mathsf{cache}_r, \mathsf{cache}_s)$

Input: a regression circuit r over variables $\mathbf{X}$ and two caches for memoization (i.e., cache_r and cache_s).
Output: its representation as a circuit $p(\mathbf{X})$.
if $r \in \mathsf{cache}_r$ **then return** $\mathsf{cache}_r(r)$
if r is an input gate **then**
  $p \leftarrow \textsc{Input}(0, \phi(r))$
else if r is a sum gate **then**
  $n \leftarrow \{\}$
  for $i = 1$ **to** $|\mathsf{ch}(r)|$ **do**
    $n \leftarrow n \cup \{\textsc{Support}(r_i, \mathsf{cache}_s)\} \cup \{\text{RGCtoCircuit}(r_i, \mathsf{cache}_r)\}$
  $p \leftarrow \textsc{Sum}(n, \{\theta_i, 1_1, \ldots, 1_{\mathsf{ch}(p)}\}_{i=1}^{|\mathsf{ch}(r)|})$
else if r is a product gate **then**
  for $i = 1$ **to** $|\mathsf{ch}(r)|$ **do**
    $p \leftarrow \textsc{Product}(\{\text{RGCtoCircuit}(r_i, \mathsf{cache}_r)\} \cup \{\textsc{Support}(r_j, \mathsf{cache}_s)\}_{j \neq i})$
$\mathsf{cache}_r(r) \leftarrow p$
return p

compatible with p over $\mathbf{X}$, and defined as

$$f_n(\boldsymbol{x}) = \begin{cases} 0 & \text{if } n \text{ is an input} \\ f_{n_\mathsf{L}}(\boldsymbol{x}_\mathsf{L}) + f_{n_\mathsf{R}}(\boldsymbol{x}_\mathsf{R}) & \text{if } n \text{ is an AND} \\ \sum_{c \in \mathsf{ch}(n)} s_c(\boldsymbol{x}) \left(\phi_c + f_c(\boldsymbol{x})\right) & \text{if } n \text{ is an OR} \end{cases}$$

where $s_c(\boldsymbol{x}) = [\![\boldsymbol{x} \in \mathsf{supp}(c)]\!]$. Then, its expected predictions can be exactly computed in $\mathcal{O}(|p||h|)$ time, where h is its circuit representation as computed by Algorithm 16.

Proof. Proof follows from noting that Algorithm 16 outputs a polysize circuit representation h in polytime. Then, computing $\mathbb{E}_{\boldsymbol{x} \sim p(\mathbf{X})}\left[h(\boldsymbol{x})\right]$ can be done in $\mathcal{O}(|p||h|)$ time by Theorem 2. □

A.5 Experiments

Generated PCs All adopted PCs were generated by running Strudel [32] on the twenty density estimation benchmarks [176]. For every dataset, we ran Strudel twice with 200

Table A.1: Sizes of the intermediate and final circuits as processed by the operators in the pipelines of the Shannon and Rényi (for $\alpha = 1.5$) entropies and Kullback-Leibler and Alpha (for $\alpha = 1.5$) divergences when computed for two input circuits p and q learned from 20 different real-world datasets as in [32].

Dataset	p	q	p^α	$q^{1-\alpha}$	$r = \log(q)$	$s = p/q$	$t = \log(s)$	$p \times q$	$p \times r$	$p \times t$	$p^\alpha \times q^{1-\alpha}$
nltcs	2779	7174	2779	7174	26155	7202	26239	7202	26183	26239	7202
msnbc	2765	6614	2765	6614	24111	6634	24171	6634	24131	24171	6634
kdd	4963	50377	4963	50377	184575	50417	184695	50417	184615	184695	50417
plants	12909	64018	12909	64018	234661	64070	234817	64070	234713	234817	64070
audio	10278	45864	10278	45864	168062	45950	168320	45950	168148	168320	45950
jester	6475	35369	6475	35369	129579	35479	129909	35479	129689	129909	35479
netflix	5068	14636	5068	14636	53571	14706	53781	14706	53641	53781	14706
accidents	3193	8183	3193	8183	29891	8299	30239	8299	30007	30239	8299
retail	4790	14926	4790	14926	54554	14994	54758	14994	54622	54758	14994
pumsb	4277	12461	4277	12461	45500	12595	45902	12595	45634	45902	12595
dna	73828	856955	73828	856955	3141981	857029	3142203	857029	3142055	3142203	857029
kosarek	5115	12988	5115	12988	47354	13106	47708	13106	47472	47708	13106
msnweb	4859	9025	4859	9025	32675	9175	33125	9175	32825	33125	9175
book	7718	12731	7718	12731	45985	12943	46621	12943	46197	46621	12943
movie	8309	11732	8309	11732	42374	11926	42956	11926	42568	42956	11926
webkb	10598	13397	10598	13397	47859	13653	48627	13653	48115	48627	13653
cr52	10912	14348	10912	14348	51094	14546	51688	14546	51292	51688	14546
c20ng	11386	14630	11386	14630	52120	14886	52888	14886	52376	52888	14886
bbc	13884	17016	13884	17016	60857	17282	61655	17282	61123	61655	17282
ad	17744	21676	17744	21676	76870	21920	77602	21920	77114	77602	21920

and 500 iterations, respectively. All other hyperparameters were selected following Dang et al. [32].

Server specifications All our experiments were run on a server with 72 CPUs, 512G Memory, and 2 TITAN RTX GPUs.

Implementations Code snippets for the five adopted queries (i.e., Kullback-Leibler divergence, Cross Entropy, Entropy, Alpha divergence, and Cauchy-Schwarz divergence) are shown in Fig. A.3. Note that they are simple compositions of the modular operators introduced in Section 2.4.

Table A.2: Times in seconds to compute the Shannon entropy (ENT), the cross-entropy (XENT), Kullback-Leibler (KLD), Alpha (for $\alpha = 1.5$) divergence, Rényi entropy (RényiEnt), and Cauchy-Schwarz divergence (CSDiv) over the circuits learned from 20 different real-world datasets by either using the algorithm distilled by our pipelines (see Table A.1 and Fig. A.3) compared to the custom and highly-optimized implementations of the same ENT [158] and KLD [91] algorithms as available in Juice.jl [29].

Dataset	ENT		KLD		XENT		AlphaDiv		RényiEnt		CSDiv	
	ours	Juice	ours	Juice	ours	Juice	ours	Juice	ours	Juice	ours	Juice
nltcs	0.143	0.001	0.830	0.207	0.422	-	0.140	-	0.013	-	0.300	-
msnbc	0.109	0.001	0.369	0.182	0.297	-	0.105	-	0.018	-	0.227	-
kdd	0.157	0.001	3.154	0.790	2.180	-	0.885	-	0.016	-	1.136	-
plants	0.679	0.005	3.983	3.909	3.739	-	1.160	-	0.088	-	1.572	-
audio	0.406	0.003	2.736	1.681	1.873	-	0.537	-	0.029	-	0.771	-
jester	0.764	0.003	1.019	0.432	0.805	-	0.351	-	0.024	-	0.476	-
netflix	0.106	0.002	0.352	0.175	0.264	-	0.100	-	0.017	-	0.201	-
accidents	0.055	0.001	0.207	0.039	0.542	-	0.091	-	0.009	-	0.124	-
retail	0.108	0.001	0.508	0.153	0.415	-	0.184	-	0.013	-	0.197	-
pumsb	0.092	0.001	0.701	0.133	0.316	-	0.119	-	0.012	-	0.214	-
dna	4.365	0.027	64.664	220.377	52.997	-	15.609	-	0.255	-	22.901	-
kosarek	0.182	0.002	0.477	0.106	0.379	-	0.139	-	0.011	-	0.735	-
msnweb	0.128	0.002	0.261	0.047	0.211	-	0.342	-	0.015	-	0.135	-
book	0.086	0.003	0.215	0.036	0.202	-	0.075	-	0.020	-	0.115	-
movie	0.272	0.002	0.443	0.063	0.373	-	0.172	-	0.015	-	0.194	-
webkb	0.138	0.003	0.241	0.031	0.164	-	0.079	-	0.023	-	0.098	-
cr52	0.141	0.004	0.260	0.035	0.188	-	0.087	-	0.031	-	0.143	-
c20ng	0.118	0.003	0.264	0.034	0.194	-	0.088	-	0.032	-	0.101	-
bbc	0.205	0.005	0.308	0.037	0.225	-	0.110	-	0.038	-	0.189	-
ad	0.193	0.007	0.346	0.046	0.281	-	0.151	-	0.031	-	0.207	-

```
function kld(p, q)
    r = quotient(p, q)
    s = log(r)
    t = product(p, s)
    return integrate(t)
end
```

```
function xent(p, q)
    r = log(q)
    s = product(p, r)
    return -integrate(s)
end
```

```
function csdiv(p, q)
    r = product(p, q)
    s = real_pow(p, 2.0)
    t = real_pow(q, 2.0)
    a = integrate(r)
    b = integrate(s)
    c = integrate(t)
    return -log(a / sqrt(b * c))
end
```

```
function ent(p)
    q = log(p)
    r = product(p, q)
    return -integrate(s)
end
```

```
function alphadiv(p, q, alpha=1.5)
    r = real_pow(p, alpha)
    s = real_pow(q, 1.0-alpha)
    t = product(r, s)
    return log(integrate(t)) / (1.0-alpha)
end
```

Figure A.3: The modular operators defined in Section 2.4 can be easily composed to implement tractable algorithms for novel query classes. Here we show the code snippet for five queries: Kullback-Leibler divergence (`kld`), Cross Entropy (`xent`), Entropy (`ent`), Alpha divergence (`alphadiv`), and Cauchy-Schwarz divergence (`csdiv`).

Appendix B

Scalable Learning of Probabilistic Circuits – Algorithmic Side

B.1 Learning Sparse PCs with Pruning and Growing

B.1.1 Pseudocode

In this section, we list the detailed algorithms for pruning (Section 3.2.1), growing (Section 3.2.3), circuit flows computation (Definition 7), and mini-batch Expectation Maximization (Section 3.2.3).

Algorithm 17 shows how to prune k percentage edges from PC $\mathcal{C}$ following heuristic h.

Algorithm 18 shows show a feedforward implementation of growing operation.

Algorithm 19 computes the circuit flows of a sample $\boldsymbol{x}$ given PC $\mathcal{C}$ with parameters $\boldsymbol{\theta}$ though one forward pass (line 1) and one backward pass (line 2-8).

Algorithm 20 shows the pipeline of mini-batches Expectation Maximization algorithm given PC $\mathcal{C}$, dataset $\mathcal{D}$, batch size B and learning rate α.

Algorithm 17 PRUNE($\mathcal{C}, h, k$)

Input: A non-deterministic PC $\mathcal{C}$; a heuristic h to score edges (e.g., EFLOW, ERAND, or EPARAM); percentage k of edges to prune
Output: A pruned PC $\mathcal{C}'$
Initialize `old2new` $\leftarrow$ mapping from each node $n \in \mathcal{C}$ to its copy in the pruned PC
Define $s(n, c) \leftarrow$ score of edge (n, c) according to heuristic h
Initialize $f(n, c) \leftarrow$ **false** for all edges
Set $f(n, c) \leftarrow$ **true** if $s(n, c)$ ranks in the lowest $k\%$ among edges from n
for each node $n \in \mathcal{C}$ in post-order (children before parents) **do**
  if n is a leaf **then**
    `old2new`$[n] \leftarrow n$
  else if n is a sum unit **then**
    `old2new`$[n] \leftarrow \bigoplus \left([\texttt{old2new}(c) \text{ for } c \in \texttt{ch}(n) \text{ if } \neg f(n, c)]\right)$
  else
    `old2new`$[n] \leftarrow \bigotimes \left([\texttt{old2new}(c) \text{ for } c \in \texttt{ch}(n)]\right)$
return `old2new`$[n_r]$ where n_r is the root of $\mathcal{C}$

B.1.2 Proofs

In this section, we provide detailed proofs of Theorem 9 (Section B.1.2) and Theorem 10 (Section B.1.2).

Pruning One Edge over One Example

Lemma 1 (Pruning One Edge Log-Likelihood Lower Bound)**.** *For a PC $\mathcal{C}$ and a sample $\boldsymbol{x}$, the loss of log-likelihood by pruning away edge (n, c) is*

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

Proof. For notation simplicit, denote the probability of units m (resp. n) in the original (resp. pruned) PC given sample $\boldsymbol{x}$ as $p_m(\boldsymbol{x})$ (resp. $p'_n(\boldsymbol{x})$). As a slight extension of Definition 7, we define $F_n(\boldsymbol{x}; m)$ as the flow of unit n w.r.t. the PC rooted at m.

The proof proceeds by induction over the PC's root unit. That is, we first consider pruning (n, c) w.r.t. the PC rooted at n. Then, in the induction step, we prove that if the lemma holds for PC rooted at m, then it also holds for PC rooted at any parent unit of m. Instead of directly proving the statement in Lemma 1, we first prove that for any root node m, the

Algorithm 18 GROW($\mathcal{C}, \sigma^2$)

Input: A PC $\mathcal{C}$; Gaussian noise variance σ^2
Output: A new PC $\mathcal{C}'$ after growing operation
Initialize `old2new` $\leftarrow$ dictionary mapping input units $n \in \mathcal{C}$ to units of the grown PC
for each node $n \in \mathcal{C}$ in post-order (children before parents) **do**
  if n is an input unit **then**
    `old2new`$[n] \leftarrow (n, \mathsf{deepcopy}(n))$
  else
    `chs_1, chs_2` $\leftarrow$ first and second copies of children:

$$[\texttt{old2new}[c][0] \text{ for } c \in \mathsf{ch}(n)], \quad [\texttt{old2new}[c][1] \text{ for } c \in \mathsf{ch}(n)]$$

    if n is a product unit **then**
      `old2new`$[n] \leftarrow \big(\bigotimes(\texttt{chs\_1}), \bigotimes(\texttt{chs\_2})\big)$
    else if n is a sum unit **then**
      $n_1, n_2 \leftarrow \bigoplus([\texttt{chs\_1}, \texttt{chs\_2}]), \bigoplus([\texttt{chs\_1}, \texttt{chs\_2}])$
      $\boldsymbol{\theta}_{|n_i} \leftarrow \mathsf{normalize}([\boldsymbol{\theta}_{|n}, \boldsymbol{\theta}_{|n}]) \times \boldsymbol{\epsilon}, \quad \epsilon \sim \mathcal{N}(\mathbf{1}, \sigma^2)$ for $i \in \{1, 2\}$
      `old2new`$[n] \leftarrow (n_1, n_2)$
return `old2new`$[r][0]$ `// r is the root unit of` $\mathcal{C}$

following holds:

$$p_m(\boldsymbol{x}) - p'_m(\boldsymbol{x}) = F_n(\boldsymbol{x}; m) \cdot p_m(\boldsymbol{x}) \cdot \left(\frac{1}{1-\theta} \frac{\mathrm{F}_{n,c}(\boldsymbol{x}; m)}{\mathrm{F}_n(\boldsymbol{x}; m)} - \frac{\theta}{1-\theta} \right). \tag{B.1}$$

Base case: pruning an edge of the root unit. That is, the root unit of the PC is n. In this case, we have

$$\begin{aligned} p_n(\boldsymbol{x}) - p'_n(\boldsymbol{x}) &= \sum_{c' \in \mathsf{ch}(n)} \theta_{c'|n} \cdot p_c(\boldsymbol{x}) - \sum_{c' \in \mathsf{ch}(n) \setminus c} \theta'_{c'|n} \cdot p'_c(\boldsymbol{x}), \\ &= \theta_{c|n} \cdot p_c(\boldsymbol{x}) + \sum_{c' \in \mathsf{ch}(n) \setminus c} \theta_{c'|n} \cdot p_c(\boldsymbol{x}) - \sum_{c' \in \mathsf{ch}(n) \setminus c} \theta'_{c'|n} \cdot p_c(\boldsymbol{x}), \end{aligned} \tag{B.2}$$

where $\theta'_{c|n}$ denotes the normalized parameter corresponding to edge (n, c) in the pruned PC. Specifically, we have

$$\forall m \in \mathsf{ch}(n) \setminus c, \quad \theta'_{m|n} = \frac{\theta_{m|n}}{\sum_{c' \in \mathsf{ch}(n) \setminus c} \theta_{c'|n}} = \frac{\theta_{m|n}}{1 - \theta_{c|n}}.$$

Algorithm 19 $\mathsf{CircuitFlow}(\mathcal{C}, \boldsymbol{\theta}, \boldsymbol{x})$

Input: PC $\mathcal{C}$ with parameters $\boldsymbol{\theta}$; sample $\boldsymbol{x}$
Output: Circuit flow $\mathsf{flow}[n, c]$ for each edge (n, c) and $\mathsf{flow}[n]$ for each node n
For all $n \in \mathcal{C}$, compute marginal probability $\mathsf{p}[n] \leftarrow p_n(\boldsymbol{x})$
Set root flow: $\mathsf{flow}[n_r] \leftarrow 1$
for each node $n \in \mathcal{C}$ in post-order **do**
  $\mathsf{flow}[n] \leftarrow \sum_{g \in \mathsf{pa}(n)} \mathsf{flow}[g]$
  if n is a sum node **then**
    for each child $c \in \mathsf{ch}(n)$ **do**
      $\mathsf{flow}[n, c] \leftarrow \theta_{c|n} \cdot \frac{\mathsf{p}[c]}{\mathsf{p}[n]} \cdot \mathsf{flow}[n]$
  else
    for each child $c \in \mathsf{ch}(n)$ **do**
      $\mathsf{flow}[n, c] \leftarrow \mathsf{flow}[n]$

Algorithm 20 $\mathsf{StochasticEM}(\mathcal{C}, \mathcal{D}; B, \alpha)$

Input: PC $\mathcal{C}$; dataset $\mathcal{D}$; batch size B; learning rate α
Output: Estimated parameters $\boldsymbol{\theta}$ from $\mathcal{D}$
Initialize $\theta \leftarrow$ random values
Set root flow: $\mathsf{flow}[n_r] \leftarrow 1$
while not converged or early stopped **do**
  Sample mini-batch: $\mathcal{D}' \leftarrow B$ random samples from $\mathcal{D}$
  Compute batch flow: $\mathsf{flow} \leftarrow \sum_{\boldsymbol{x} \in \mathcal{D}'} \mathsf{CircuitFlow}(\mathcal{C}, \boldsymbol{\theta}, \boldsymbol{x})$
  for each sum unit n and child c **do**
    $\theta^{\text{new}}_{c|n} \leftarrow \frac{\mathsf{flow}[n,c]}{\mathsf{flow}[n]}$
    Update: $\theta_{c|n} \leftarrow \alpha \cdot \theta^{\text{new}}_{c|n} + (1 - \alpha) \cdot \theta_{c|n}$

For notation simplicity, denote $\theta := \theta_{c|n}$. Plug in the above definition into Equation B.2, we have

$$
\begin{aligned}
p_n(\boldsymbol{x}) - p'_n(\boldsymbol{x}) &= \theta_{c|n} \cdot p_c(\boldsymbol{x}) + \sum_{c' \in \mathsf{ch}(n) \setminus c} \theta_{c'|n} \cdot p_c(\boldsymbol{x}) - \frac{1}{1-\theta} \sum_{c' \in \mathsf{ch}(n) \setminus c} \theta_{c'|n} \cdot p_c(\boldsymbol{x}), \\
&= \theta_{c|n} \cdot p_c(\boldsymbol{x}) - \frac{\theta}{1-\theta} \sum_{c' \in \mathsf{ch}(n) \setminus c} \theta_{c'|n} \cdot p_c(\boldsymbol{x}), \\
&= \theta_{c|n} \cdot p_c(\boldsymbol{x}) - \frac{\theta}{1-\theta} (p_n(\boldsymbol{x}) - \theta_{c|n} p_c(\boldsymbol{x})), \\
&= \frac{1}{1-\theta} \cdot \theta_{c|n} \cdot p_c(\boldsymbol{x}) - \frac{\theta}{1-\theta} \cdot p_n(\boldsymbol{x}), \\
&\overset{(a)}{=} \frac{1}{1-\theta} \cdot p_n(\boldsymbol{x}) \cdot \frac{\mathrm{F}_{n,c}(\boldsymbol{x}; n)}{\mathrm{F}_n(\boldsymbol{x}; n)} - \frac{\theta}{1-\theta} \cdot p_n(\boldsymbol{x}),
\end{aligned}
$$

$$= F_n(\boldsymbol{x}; n) \cdot p_n(\boldsymbol{x}) \cdot \left(\frac{1}{1-\theta} \frac{\mathrm{F}_{n,c}(\boldsymbol{x}; n)}{\mathrm{F}_n(\boldsymbol{x}; n)} - \frac{\theta}{1-\theta} \right), \tag{B.3}$$

where (a) follows from the fact that $F_n(\boldsymbol{x}; n) = 1$ and $F_{n,c}(\boldsymbol{x}; n) = \theta_{c|n} p_c(\boldsymbol{x}) / p_n(\boldsymbol{x})$.

Inductive case #1: suppose Equation B.1 holds for m. If product unit d is a parent of m, we show that Equation B.1 also holds for d:

$$\begin{aligned}
p_d(\boldsymbol{x}) - p'_d(\boldsymbol{x}) &= \prod_{n' \in \mathsf{ch}(d)} p_{n'}(\boldsymbol{x}) - \prod_{n' \in \mathsf{ch}(d)} p'_{n'}(\boldsymbol{x}), \\
&= (p_m(\boldsymbol{x}) - p'_m(\boldsymbol{x})) \prod_{n' \in \mathsf{ch}(d) \setminus m} p_{n'}(\boldsymbol{x}), \\
&\overset{(a)}{=} F_n(\boldsymbol{x}; m) \cdot p_m(\boldsymbol{x}) \cdot \left(\frac{1}{1-\theta} \frac{\mathrm{F}_{n,c}(\boldsymbol{x}; m)}{\mathrm{F}_n(\boldsymbol{x}; m)} - \frac{\theta}{1-\theta} \right) \cdot \prod_{n' \in \mathsf{ch}(d) \setminus m} p_{n'}(\boldsymbol{x}), \\
&\overset{(b)}{=} F_n(\boldsymbol{x}; d) \cdot p_d(\boldsymbol{x}) \cdot \left(\frac{1}{1-\theta} \frac{\mathrm{F}_{n,c}(\boldsymbol{x}; d)}{\mathrm{F}_n(\boldsymbol{x}; d)} - \frac{\theta}{1-\theta} \right),
\end{aligned}$$

where (a) is the inductive step that applies Equation B.3; (b) follows from the fact that (note that d is a product unit) $F_n(\boldsymbol{x}; m) = F_n(\boldsymbol{x}; d)$ and $F_{n,c}(\boldsymbol{x}; m) = F_{n,c}(\boldsymbol{x}; d)$.

Inductive case #2: for sum unit d, suppose Equation B.1 holds for m, where $m \in \mathcal{A}$ iff $m \in \mathsf{ch}(d)$ and m is an ancester of n and c. Assume all other children of d are not ancestoer of n, we show that Equation B.1 also holds for d:

$$\begin{aligned}
p_d(\boldsymbol{x}) - p'_d(\boldsymbol{x}) &= \theta_{m|d} \cdot (p_m(\boldsymbol{x}) - p'_m(\boldsymbol{x})) \\
&= \theta_{m|d} \cdot F_n(\boldsymbol{x}; m) \cdot p_m(\boldsymbol{x}) \cdot \left(\frac{1}{1-\theta} \frac{\mathrm{F}_{n,c}(\boldsymbol{x}; m)}{\mathrm{F}_n(\boldsymbol{x}; m)} - \frac{\theta}{1-\theta} \right), \\
&= \theta_{m|d} \cdot F_n(\boldsymbol{x}; m) \cdot p_m(\boldsymbol{x}) \cdot \left(\frac{1}{1-\theta} \frac{\mathrm{F}_{n,c}(\boldsymbol{x}; d)}{\mathrm{F}_n(\boldsymbol{x}; d)} - \frac{\theta}{1-\theta} \right), \\
&= \theta_{m|d} \cdot F_n(\boldsymbol{x}; d) \cdot \frac{\sum_{m' \in \mathsf{ch}(d)} \theta_{m'|d} p_{m'}(\boldsymbol{x})}{\theta_{m|d} p_m(\boldsymbol{x})} \cdot p_m(\boldsymbol{x}) \cdot \left(\frac{1}{1-\theta} \frac{\mathrm{F}_{n,c}(\boldsymbol{x}; d)}{\mathrm{F}_n(\boldsymbol{x}; d)} - \frac{\theta}{1-\theta} \right),
\end{aligned}$$

$$= F_n(\boldsymbol{x}; d) \cdot \left(\sum_{m' \in \mathsf{ch}(d)} \theta_{m'|d} p_{m'}(\boldsymbol{x}) \right) \cdot \left(\frac{1}{1-\theta} \frac{\mathrm{F}_{n,c}(\boldsymbol{x}; d)}{\mathrm{F}_n(\boldsymbol{x}; d)} - \frac{\theta}{1-\theta} \right),$$
$$= F_n(\boldsymbol{x}; d) \cdot p_d(\boldsymbol{x}) \cdot \left(\frac{1}{1-\theta} \frac{\mathrm{F}_{n,c}(\boldsymbol{x}; d)}{\mathrm{F}_n(\boldsymbol{x}; d)} - \frac{\theta}{1-\theta} \right).$$

Therefore, following Equation B.1 for root r, we have

$$\frac{p_r(\boldsymbol{x}) - p'_r(\boldsymbol{x})}{p_r(\boldsymbol{x})} = \frac{1}{1-\theta} \mathrm{F}_{n,c}(\boldsymbol{x}; r) - \frac{\theta}{1-\theta} F_n(\boldsymbol{x}; r),$$
$$\Leftrightarrow \quad \frac{p'_r(\boldsymbol{x})}{p_r(\boldsymbol{x})} = 1 + \frac{\theta}{1-\theta} F_n(\boldsymbol{x}; r) - \frac{1}{1-\theta} \mathrm{F}_{n,c}(\boldsymbol{x}; r).$$

Therefore, we have

$$\begin{aligned} \Delta\mathrm{LL}(\{\boldsymbol{x}\}, \mathcal{C}, \{(n,c)\}) &= \log p_r(\boldsymbol{x}) - \log p'_r(\boldsymbol{x}), \\ &= \frac{1}{|\mathcal{D}|} \sum_{\boldsymbol{x} \in \mathcal{D}} \log \left(\frac{1-\theta_{c|n}}{1-\theta_{c|n}+\theta_{c|n}\mathrm{F}_n(\boldsymbol{x}; r)-\mathrm{F}_{n,c}(\boldsymbol{x}; r)} \right), \\ &\overset{(a)}{\leq} -\log(1 - F_{n,c}(\boldsymbol{x})), \end{aligned}$$

where (a) follows from the fact that $F_{n,c}(\boldsymbol{x}) \leq F_n(\boldsymbol{x})$. □

Theorem 9 follows directly from Lemma 1 by noting that for any dataset $\mathcal{D}$, $\Delta\mathrm{LL}(\mathcal{D}, \mathcal{C}, \{(n,c)\}) = \frac{1}{|\mathcal{D}|} \Delta\mathrm{LL}(\{\boldsymbol{x}\}, \mathcal{C}, \{(n,c)\})$.

Pruning Multiple Edges

Proof. Similar to the proof of Lemma 1, we prove Theorem 10 by induction. Different from Lemma 1, we induce a slightly different objective:

$$p_m(\boldsymbol{x}) - p'_m(\boldsymbol{x}) \leq \sum_{(n,c) \in \mathcal{E} \cap \mathsf{des}(m)} F_n(\boldsymbol{x}; m) \cdot p_m(\boldsymbol{x}) \cdot \left(\frac{1}{1-\theta_{c|n}} \frac{\mathrm{F}_{n,c}(\boldsymbol{x}; m)}{\mathrm{F}_n(\boldsymbol{x}; m)} - \frac{\theta_{c|n}}{1-\theta_{c|n}} \right), \quad \text{(B.4)}$$

where $\mathsf{des}(n)$ is the set of descendent units of n.

Base case: the base case follows directly from the proof of Lemma 1, and lead to the

conclusion in Equation B.3.

Inductive case #1: suppose for all children of a product unit d, Equation B.4 holds, we show that Equation B.4 also holds for d:

$$
\begin{aligned}
p_d(\boldsymbol{x}) - p'_d(\boldsymbol{x}) &= \prod_{m \in \mathsf{ch}(d)} p_m(\boldsymbol{x}) - \prod_{m \in \mathsf{ch}(d)} p'_m(\boldsymbol{x}), \\
&= \prod_{m \in \mathsf{ch}(d)} p_m(\boldsymbol{x}) - \prod_{m \in \mathsf{ch}(d)} \Big(p_m(\boldsymbol{x}) - (p_m(\boldsymbol{x}) - p'_m(\boldsymbol{x}))\Big), \\
&\leq \sum_{m \in \mathsf{ch}(d)} \Big(p_m(\boldsymbol{x}) - p'_m(\boldsymbol{x}))\Big) \cdot \prod_{m' \in \mathsf{ch}(d) \setminus m} p_{m'}(\boldsymbol{x}), \\
&\overset{(a)}{\leq} \sum_{m \in \mathsf{ch}(d)} \sum_{(n,c) \in \mathcal{E} \cap \mathsf{des}(m)} F_n(\boldsymbol{x}; d) \cdot p_d(\boldsymbol{x}) \cdot \left(\frac{1}{1-\theta_{c|n}} \frac{\mathrm{F}_{n,c}(\boldsymbol{x}; m)}{\mathrm{F}_n(\boldsymbol{x}; m)} - \frac{\theta_{c|n}}{1-\theta_{c|n}} \right), \\
&\leq \sum_{(n,c) \in \mathcal{E} \cap \mathsf{des}(d)} F_n(\boldsymbol{x}; d) \cdot p_d(\boldsymbol{x}) \cdot \left(\frac{1}{1-\theta_{c|n}} \frac{\mathrm{F}_{n,c}(\boldsymbol{x}; d)}{\mathrm{F}_n(\boldsymbol{x}; d)} - \frac{\theta_{c|n}}{1-\theta_{c|n}} \right),
\end{aligned}
$$

where (a) uses the definition that $p_d(\boldsymbol{x}) = \prod_{m \in \mathsf{ch}(d)} p_m(\boldsymbol{x})$.

Inductive case #2: suppose for all children of a sum unit d, Equation B.4 holds, we show that Equation B.4 also holds for d:

$$
\begin{aligned}
p_d(\boldsymbol{x}) - p'_d(\boldsymbol{x}) &= \sum_{m \in \mathsf{ch}(d) \cap (d,m) \notin \mathcal{E}} \theta_{m|d} \cdot \Big(p_m(\boldsymbol{x}) - p'_m(\boldsymbol{x})\Big) + \sum_{m \in \mathsf{ch}(d) \cap (d,m) \in \mathcal{E}} \theta_{m|d} \cdot \Big(p_m(\boldsymbol{x}) - p'_m(\boldsymbol{x})\Big), \\
&\overset{(a)}{=} \sum_{m \in \mathsf{ch}(d) \cap (d,m) \notin \mathcal{E}} \theta_{m|d} \cdot \Big(p_m(\boldsymbol{x}) - p'_m(\boldsymbol{x})\Big), \\
&\quad + \sum_{m \in \mathsf{ch}(d) \cap (d,m) \in \mathcal{E}} \theta_{m|d} \cdot F_n(\boldsymbol{x}; m) \cdot p_m(\boldsymbol{x}) \cdot \left(\frac{1}{1-\theta_{c|n}} \frac{\mathrm{F}_{n,c}(\boldsymbol{x}; m)}{\mathrm{F}_n(\boldsymbol{x}; m)} - \frac{\theta_{c|n}}{1-\theta_{c|n}} \right),
\end{aligned}
$$

where (a) follows from the base case of the induction. Next, we focus on the first term of the above equation:

$$
\begin{aligned}
&\sum_{m \in \mathsf{ch}(d) \cap (d,m) \notin \mathcal{E}} \theta_{m|d} \cdot \Big(p_m(\boldsymbol{x}) - p'_m(\boldsymbol{x})\Big), \\
\leq &\sum_{m \in \mathsf{ch}(d) \cap (d,m) \notin \mathcal{E}} \sum_{(n,c) \in \mathcal{E} \cap \mathsf{des}(m)} \theta_{m|d} \cdot \Big(p_m(\boldsymbol{x}) - p'_m(\boldsymbol{x})\Big),
\end{aligned}
$$

$$\leq \sum_{m \in \mathsf{ch}(d) \cap (d,m) \notin \mathcal{E}} \sum_{(n,c) \in \mathcal{E} \cap \mathsf{des}(m)} \theta_{m|d} \cdot F_n(\boldsymbol{x}; m) \cdot p_m(\boldsymbol{x}) \cdot \left(\frac{1}{1-\theta_{c|n}} \frac{\mathrm{F}_{n,c}(\boldsymbol{x}; m)}{\mathrm{F}_n(\boldsymbol{x}; m)} - \frac{\theta_{c|n}}{1-\theta_{c|n}} \right),$$
$$\leq \sum_{(n,c) \in \mathcal{E} \cap \mathsf{des}(d)} F_n(\boldsymbol{x}; d) \cdot p_d(\boldsymbol{x}) \cdot \left(\frac{1}{1-\theta_{c|n}} \frac{\mathrm{F}_{n,c}(\boldsymbol{x}; d)}{\mathrm{F}_n(\boldsymbol{x}; d)} - \frac{\theta_{c|n}}{1-\theta_{c|n}} \right),$$

where the derivation of the last inequality follows from the corresponding steps in the proof of Lemma 1.

Therefore, from Equation B.4, we can conclude that

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

Finally, we prove the approximation step in Equation 3.2. Let $\epsilon(\cdot) = \sum_{(n,c) \in \mathcal{E}} \mathrm{F}_{n,c}(\cdot) \in [0, 1)$. We have,

$$\begin{aligned} \mathrm{RHS} &= -\sum_{\boldsymbol{x} \in \mathcal{D}} \log(1 - \epsilon(\boldsymbol{x})) = -\sum_{\boldsymbol{x} \in \mathcal{D}} \sum_{k=1}^{\infty} -\frac{\epsilon(\boldsymbol{x})^k}{k} (\text{Taylor expansion}) \leq \sum_{\boldsymbol{x} \in \mathcal{D}} \sum_{k=1}^{\infty} \epsilon(\boldsymbol{x})^k, \\ &= \sum_{\boldsymbol{x} \in \mathcal{D}} \frac{\epsilon(\boldsymbol{x})}{1 - \epsilon(\boldsymbol{x})} = \frac{1}{1-\epsilon} \sum_{\boldsymbol{x} \in \mathcal{D}} \epsilon(\boldsymbol{x}) = \frac{1}{1-\epsilon} \sum_{(n,c) \in \mathcal{E}} \sum_{\boldsymbol{x} \in \mathcal{D}} \mathrm{F}_{n,c}(\boldsymbol{x}) = \frac{1}{1-\epsilon} \sum_{(n,c) \in \mathcal{E}} \mathrm{F}_{n,c}(\mathcal{D}). \end{aligned}$$

□

B.1.3 Experiments Details

Hardware specifications All experiments are performed on a server with 32 CPUs, 126G Memory, and NVIDIA RTX A5000 GPUs with 26G Memory. In all experiments, we only use a single GPU on the server.

Datasets

For MNIST-family datasets, we split 5% of training set as validation set for early stopping. For Penn Tree Bank dataset, we follow the setting in [117] to split a training, validation, and test set. Table B.1 lists the all the dataset statistics.

Table B.1: Dataset statistics including number of variables (#**vars**), number of categories for each variable (#**cat**), and number of samples for training, validation and test set (#**train**, #**valid**, #**test**).

Dataset	n (#**vars**)	k (#**cat**)	#**train**	#**valid**	#**test**
MNIST	28×28	256	57000	3000	10000
EMNIST(MNIST)	28×28	256	57000	3000	10000
EMNIST(Letters)	28×28	256	118560	6240	20800
EMNIST(Balanced)	28×28	256	107160	5640	18800
EMNIST(ByClass)	28×28	256	663035	34897	116323
FashionMNIST	28×28	256	57000	3000	10000
Penn Tree Bank	288	50	42068	3370	3761

Learning Hidden Chow-Liu Trees

HCLT structures. Adopting hidden chow liu tree (HCLT) PC architecture as in [98], we reimplement the learning process to speed it up and use a different training pipeline and hyper-parameters tuning.

EM parameter learning We adopt the EM parameter learning algorithm introduced in [18], which computes the EM update target parameters using circuit flows. We use a stochastic mini-batches EM algorithm. Denoting $\theta^{\texttt{new}}$ as the EM update target computed from a mini-batch of samples, and we update the targeting parameter with a learning rate α: $\theta^{t+1} \leftarrow \alpha\theta^{\texttt{new}} + (1-\alpha)\theta^{t}$. α is piecewise-linearly annealed from $[1.0, 0.1]$, $[0.1, 0.01]$, $[0.01, 0.001]$, and each piece is trained T epochs.

Hyper-parameters searching. For all the experiments, the hyper-parameters are searched from

- $h \in \{8, 16, 32, 64, 128, 256\}$, the hidden size of HCLT structures;
- $\gamma \in \{0.0001, 0.001, 0.01, 0.1, 1.0\}$, Laplace smoothing factor;
- $B \in \{128, 256, 512, 1024\}$, batch-size in mini-batches EM algorithm;

- α piecewise-linearly annealed from $[1.0, 0.1]$, $[0.1, 0.01]$, $[0.01, 0.001]$, where each piece is called one mini-batch EM phase. Usually the algorithm will start to overfit as validation set and stop at the third phase;
- $T = 100$, number of epochs for each mini-batch EM phase.

The PC size is quadratically growing with hidden size h, thus it is inefficient to do a grid search among the entire hyper-parameters space. What we do is to fist do a grid search when $h = 8$ or $h = 16$ to find the best Laplace smoothing factor γ and batch-size B for each dataset, and then fix γ and B to train a PC with larger hidden size $h \in \{32, 64, 128, 256\}$. The best tuned B is in $\{256, 512\}$, which is different for different hidden size h, and the best tuned γ is 0.01.

Details of Section 3.2.5

Sparse PC (ours). Given an HCLT learned in Section B.1.3 as initial PC, we use the structure learning process proposed in Section 3.2.3. Specifically, starts from initial HCLT, for each iteration, we (1) prune 75% of the PC parameters, and (2) grow PC size with Gaussian variance ϵ, (3) finetuing PC using mini-batches EM parameter learning with learning rate α. We prune and grow PC iteratively until the validation set likelihood is overfitted . The hyper-parameters are searched from

- $\epsilon \in \{0.1, 0.3, 0.5\}$, Gaussian variance in growing operation;
- α, piecewise-linearly annealed from $[0.1, 0.01]$, $[0.01, 0.001]$;
- $T = 50$, number of epochs for each mini-batch EM phase;
- for γ and B, we use the tuned best number from Section B.1.3.

HCLT. The HLCT experiments in Table 3.1 are performed following the original paper (Code https://github.com/UCLA-StarAI/Tractable-PC-Regularization), which is different from the leaning pipeline we use as our inital PC (Section B.1.3).

SPN. We reimplement the SPN architecture ourselves following [133] and train it with the same mini-batch pipeline as HCLT.

IDF. We run all experiments with the code in the GitHub repo provided by the authors. We adopt an IDF model with the following hyperparameters: 8 flow layers per level; 2 levels; densenets with depth 6 and 512 channels; base learning rate 0.001; learning rate decay 0.999. The algorithm adopts an CPU-based entropy coder rANS.

BitSwap. We train all models using the following author-provided script: https://github.com/fhkingma/bitswap/blob/master/model/mnist_train.

BB-ANS. All experiments are performed using the following official code https://github.com/bits-back/bits-back.

McBits. All experiments are performed using the following official code https://github.com/ryoungj/mcbits.

Details of Section 3.2.6

For all experiments in Section 3.2.6, we use the best tuned γ and B from Section B.1.3 and hidden size h ranging from $\{16, 32, 64, 128\}$. For experiments "What is the Smallest PC for the Same Likelihood?", the hyper-parameters are searched from

- $k \in \{0.05, 0.1, 0.3\}$, percentage of parameters to prune each iteration;
- α, piecewise-linearly annealed from $[0.3, 0.1]$, $[0.1, 0.01]$, $[0.01, 0.001]$;
- $T = 50$, number of epochs for each mini-batch EM phase;

For experiments "What is the Best PC Given the Same Size?", we use the same setting as in Section B.1.3.

B.2 Latent Variable Distillation

B.2.1 Proofs

In this section, we provide detailed proofs of Lemma 1.

Proof of Lemma 1

Proof. Define $\mathbf{Y} := \mathbf{X} \backslash \mathbf{W}$. To prove that variables $\mathbf{W}$ is conditional independent with $\mathbf{Y}$ given Z, it is sufficient to show that $\forall \mathbf{w} \in \mathsf{val}(\mathbf{W}), z \in \mathsf{val}(Z), \boldsymbol{y} \in \mathsf{val}(\mathbf{Y})$, we have $p(\mathbf{w}|z) = p(\mathbf{w}|z, \boldsymbol{y})$.

Define $S^{\text{sum}}_{p,\mathbf{W}}$ and $S^{\text{prod}}_{p,\mathbf{W}}$ as the set of sum and product units with scope $\mathbf{W}$, respectively. $\forall n \in S^{\text{sum}}_{p,\mathbf{W}}$, since the scope of n does not contain variables $\mathbf{Y}$, we can immediately conclude that $p_n(\mathbf{w}|z) = p(\mathbf{w}|z, \boldsymbol{y})$. In order to show that this equation also holds for the root PC unit, we only need to show that for each PC unit n in p, if all its children (whose scope contains $\mathbf{W}$) satisfy this equation, then n also does. The reason is that the root unit n_r of p must be an ancestor unit of every unit in $S^{\text{sum}}_{p,\mathbf{W}}$.

We start with the case that n is a product unit. Since the PC is assumed to be decomposable, only one child, denoted m, satisfies $\mathbf{W} \subseteq \phi(m)$. Therefore, the distribution of n can be written as

$$p_n(\boldsymbol{x}) = p_m(\boldsymbol{x}) \cdot \prod_{c \in \mathsf{ch}(n), c \neq m} p_c(\boldsymbol{x}) \overset{(a)}{=} p_m(\boldsymbol{x}) \cdot \prod_{c \in \mathsf{ch}(n), c \neq m} p_c(\boldsymbol{y}),$$

where (a) holds because $\forall c \in \mathsf{ch}(n), c \neq m$, we have $\phi(c) \cap \mathbf{W} = \varnothing$. Therefore, we have $p_n(\mathbf{w}|z) = p_m(\mathbf{w}|z)$ and $p_n(\mathbf{w}|z, \boldsymbol{y}) = p_m(\mathbf{w}|z, \boldsymbol{y})$. Taking the two equations together and use the assumption from the induction step: $p_m(\mathbf{w}|z) = p_m(\mathbf{w}|z, \boldsymbol{y})$, we conclude that $p_n(\mathbf{w}|z) = p_n(\mathbf{w}|z, \boldsymbol{y})$.

Define n_z as the product unit in $S^{\text{prod}}_{p,\mathbf{W}}$ that is augmented with input unit $Z = z$ by Algorithm 3. Before proving the main result, we highlight that $\forall n$ whose scope contains $\mathbf{W}$, $p_n(\boldsymbol{x})$ can be written as $p_{n_z}(\mathbf{w}) \cdot g_n(\boldsymbol{y})$, where $g_n(\boldsymbol{y})$ is independent with $\mathbf{w}$. This is because

$\forall m \in S^{\text{prod}}_{p,\mathbf{W}}$ and $m \neq n_z$, $p_m(\mathbf{w}, z) = 0 (\forall \mathbf{w} \in \mathsf{val}(\mathbf{W}))$.

Next, assume n is a sum unit whose scope contains $\mathbf{W}$. Using the above result, we know that every child c of n satisfies the following: $p_c(\boldsymbol{x}) = p_{n_z}(\mathbf{w}) \cdot g_c(\boldsymbol{y})$. Thus, we have

$$p_n(\boldsymbol{x}) = \sum_{c \in \mathsf{ch}(n)} \theta_{c|n} \cdot p_{n_z}(\mathbf{w}) \cdot g_c(\boldsymbol{y}) = p_{n_z}(\mathbf{w}) \cdot \Big(\sum_{c \in \mathsf{ch}(n)} \theta_{c|n} \cdot g_c(\boldsymbol{y}) \Big).$$

Since $p_{n_z}(\mathbf{w}|z) = p_{n_z}(\mathbf{w}|z, \boldsymbol{y})$, we have $p_n(\mathbf{w}|z) = p_n(\mathbf{w}|z, \boldsymbol{y})$.

Taking the above two inductive cases (i.e., for sum and product units, respectively), we can conclude that for the root unit n_r, $p_{n_r}(\mathbf{w}|z) = p_{n_r}(\mathbf{w}|z, \boldsymbol{y})$. □

B.2.2 Details for Latent Variable Distillation

This section provides additional details for latent variable distillation (LVD), including description of the adopted EM algorithm and details of the LV extraction step.

Parameter Estimation

We adopt a stochastic mini-batch version of the Expectation-Maximization algorithm. Specifically, a mini-batch of samples are drown from the dataset, and the EM algorithm for PCs [16, 29] is used to compute a set of new parameters $\boldsymbol{\theta}^{\text{new}}$, which is updated with a learning rate α: $\boldsymbol{\theta}_{t+1} \leftarrow \alpha \cdot \boldsymbol{\theta}^{\text{new}} + (1 - \alpha) \cdot \boldsymbol{\theta}_t$.

Details of the MAE-based LV Extraction Step

We use the official code (https://github.com/facebookresearch/mae) to train MAE models on the adopted datasets (i.e., CIFAR, ImageNet32, and ImageNet64). At each training step, the percentage of masked patches is chosen uniformly from 10% to 90%. After training, the LV extraction step follows the description in Section 3.3.3.

B.2.3 Experiment Details

In this section, we describe experiment details of all four TPMs adopted in Section 3.3.1 and Section 3.3.4.

Hardware specification All experiments are run on servers/workstations with the following configuration:

- 32 CPUs, 128G Mem, 4 × NVIDIA A5000 GPU;
- 32 CPUs, 64G Mem, 1 × NVIDIA GeForce RTX 3090 GPU;
- 64 CPUs, 128G Mem, 3 × NVIDIA A100 GPU.

HMM The HMM models are trained with varying hidden states $h = 128$, 256, 512, 750, 1024 and 1250, with and without LVD. All HMM models are trained with mini-batch EM (Section B.2.2) for two phases: in phase 1, the model is trained with learning rate 0.1 for 20 epochs; in phase 2, the model is trained with learning rate 0.01 for 5 epochs. Note that for HMM models with hidden states ≥ 750, we train for 30 epochs in phase 1. The number of epochs are selected such that all model converges before training stops.

LVD For every subset $\mathbf{X}_i$, the number of hidden categories, i.e., $\{M_i\}_{i=1}^k$ are set to values in $\{8, 16, 32, 64, 128, 256\}$. For the latent-conditioned distribution $\{p(\mathbf{X}_i|Z_i = j)\}_{i,j}$, we adopt HCLTs with hidden size 16, and for the latent distribution $p(\mathbf{Z})$, a HCLT with hidden size M_i is adopted. When optimizing the model with the MLE lower bound, we adopt mini-batch EM (Section B.2.2) with learning rate annealed linearly from 0.1 to 0.01. In the latent distribution training step (Section 3.3.2), we anneal learning rate from 0.1 to 0.001.

HCLT We use the publicly available implementation of HCLT at https://github.com/Juice-jl/ProbabilisticCircuits.jl/blob/master/src/structures/hclts.jl. The hidden size is chosen from $\{16, 32, 64, 128, 256, 512, 1024\}$. We anneal the EM learning rate from 0.1 to 0.01 and train for 100 epochs, and then anneal the learning rate from 0.01 to 0.001 and train for another 100 epochs.

RAT-SPN We adopt the publicly available implementation at https://github.com/Juice-jl/ProbabilisticCircuits.jl/blob/master/src/structures/rat.jl. num_nodes_region,

num_features, and num_nodes_leaf are set to the same value, which is chosen from

$$\{16, 32, 64, 128, 256, 512, 1024\}.$$

Learning rate schedule is same with HCLTs.

EiNet We use the official implementation on GitHub: https://github.com/cambridge-mlg/EinsumNetworks. We use the PD structure provided in the codebase. We select hyperparameter delta from $\{4, 6, 8\}$ and select num_sums from $\{16, 32, 64, 128, 256\}$. Learning rate is set to 0.001.

B.2.4 Efficiency Analysis

This section provides the breakdown of the runtime for each stage of the LVD algorithm for the PC that achieves 4.38 bpd on ImageNet32. The PC has 836M parameters. All experiments are done on a single NVIDIA A5000 GPU.

For this PC, training all latent conditioned distributions $\{p(\boldsymbol{x}_i|Z_i = j)\}_{i,j}$ take $\sim$8 hours, and training the latent distribution $p(\boldsymbol{z})$ takes $\sim$0.5 hours. Finally, the fine-tuning stage takes $\sim$1 hour.

As shown in the above computation time breakdown, the most time-consuming part is to train the latent conditioned distributions. However, we note that this is not a fundamental problem of LVD: we are training every latent conditioned distributions *independently*, while there could be massive structure/parameter sharing among such distributions. We left this to future work.

B.2.5 Additional Ablation Studies

To better understand the effectiveness of LVD as a general PC learning approach, this section conducts ablation studies on various components of the LVD pipeline.

Architecture of the Deep Generative Model

In addition to the MAE model, we apply LVD using LV assignments generated by a pretrained VQ-VAE model [175]. Specifically, the encoder of the adopted VQ-VAE transforms every 4×4 image patch directly into a discrete latent code. The discrete codes are used directly as LV assignments by LVD. On ImageNet32, with 256 categories for each latent variable (same with the best adopted MAE model), VQ-VAE + LVD achieved 4.44 bpd. This is significantly better than the performance of the best TPM/PC without LVD, and is only slightly worse than the MAE + LVD. This result shows that LVD works well across different deep generative model architectures.

Different Patch Sizes

In the original experiments, we select a patch size 4 for the MAE model. That is, every 4×4 patch is materialized as a LV Z. To study the influence caused by the number of materialized LVs, we perform LVD using two new MAE models with patch sizes 2 and 8, respectively. The results are shown in the table below.

Patch size	2	4	8
ImageNet32 bpd	4.57	4.39	4.44

We observe that with patch size 8, the performance becomes slightly worse and with patch size 2 the performance becomes much worse. However, we highlight that the inferior performance of the patch-size-2 model is not because of the use of more materialized latent variables. The reason is that the sub-PCs suffer from worse likelihood since the receptive field is limited to 2×2. In addition, the 2×2 patch MAE is harder to train and has a larger RMSE compared to its 4×4 variant. Therefore, the performance of LVD is also determined by the deep generative model and the strategy to choose LVs.

Appendix C

Scalable Learning of Probabilistic Circuits – System Side

C.1 Algorithm Details

In this section, we provide additional details of the design of PyJuice. Specifically, we introduce the layer partitioning algorithm that divides a layer into groups of node blocks with a similar number of children in Section C.1.1, and describe the details of the backpropagation algorithm in Section C.1.2.

C.1.1 The Layer Partitioning Algorithm

The layer partitioning algorithm receives as input a vector of integers `nchs` where each number denotes the number of child node blocks connected to a node block in the layer. It also receives as input the maximum number of groups to be considered (denoted `G`) and a sparsity tolerance threshold $\texttt{tol} \in (0, 1]$. Our goal is to search for a set of n (at most `G`) groups with capacities $g_1, \ldots, g_n$, respectively. Every number in `nchs` is then placed into the group with the smallest capacity it can fit in. Every number in `nchs` must fit in a group. Assume there are k_i numbers assigned to group i, the overhead/cost w.r.t. a partitioning

Algorithm 21 Partition a layer into groups

Inputs: a list of child node (block) counts of the current layer $\texttt{nchs} \in \mathbb{Z}^N$ (N is the number of node blocks in the layer)
Inputs: the maximum number of groups `G`, the sparsity tolerance threshold $\texttt{tol} \in (0, 1]$
$\texttt{uni\_nchs}, \texttt{counts} \leftarrow \texttt{unique}(\texttt{nchs}, \texttt{sorted} = \texttt{True})$ (get the unique values and their appearance counts; we require the numbers in `uni_nchs` to be sorted in ascending order)
$\texttt{L} \leftarrow \texttt{length}(\texttt{uni\_nchs})$
$\texttt{target\_overhead} \leftarrow \lceil \texttt{sum}(\texttt{uni\_nchs} * \texttt{counts}) * (1.0 + \texttt{tol}) \rceil$ (get the target overhead)
$\texttt{cum\_counts} \leftarrow \texttt{cumsum}(\texttt{counts})$
$\texttt{dp}, \texttt{backtrace} \leftarrow (0)_{\texttt{L}\times\texttt{G}+1} \in \mathbb{R}^{\texttt{L}\times\texttt{G}+1}, (0)_{\texttt{L}\times\texttt{G}+1} \in \mathbb{Z}^{\texttt{L}\times\texttt{G}+1}$
for $i = 0$ **to** $\texttt{L} - 1$ **do**
  $\texttt{dp}[i, 1] \leftarrow \texttt{uni\_nchs}[i] * \texttt{cum\_counts}[i]$
Main DP algorithm
$\texttt{target\_n\_group} \leftarrow \texttt{G}$
for $\texttt{n\_group} = 2$ **to** `G` **do**
  $\texttt{dp}[0, \texttt{n\_group}] \leftarrow \texttt{uni\_nchs}[0] * \texttt{cum\_counts}[0]$
  $\texttt{backtrace}[0, \texttt{n\_group}] \leftarrow 0$
  for $\texttt{i} = 1$ **to** $\texttt{L} - 1$ **do**
    $\texttt{min\_overhead}, \texttt{best\_idx} \leftarrow \texttt{inf}, -1$
    for $\texttt{j} = 0$ **to** $\texttt{i} - 1$ **do**
      $\texttt{curr\_overhead} \leftarrow \texttt{dp}[\texttt{j}, \texttt{n\_group} - 1] + \texttt{uni\_nchs}[\texttt{i}] * (\texttt{cum\_counts}[\texttt{i}] - \texttt{cum\_counts}[\texttt{j}])$
      if $\texttt{curr\_overhead} < \texttt{min\_overhead}$ **then**
        $\texttt{min\_overhead}, \texttt{best\_idx} \leftarrow \texttt{curr\_overhead}, \texttt{j}$
    $\texttt{dp}[\texttt{i}, \texttt{n\_group}], \texttt{backtrace}[\texttt{i}, \texttt{n\_group}] \leftarrow \texttt{min\_overhead}, \texttt{best\_idx}$
  if $\texttt{dp}[-1, \texttt{n\_group}] <= \texttt{target\_overhead}$ **then**
    $\texttt{target\_n\_group} \leftarrow \texttt{n\_group}$
Backtrace
$\texttt{group\_sizes} \leftarrow (0)_{\texttt{target\_n\_group}} \in \mathbb{Z}^{\texttt{target\_n\_group}}$
$\texttt{i} \leftarrow \texttt{L} - 1$
for $n = \texttt{target\_n\_group}$ **to** 1 **do**
  $\texttt{group\_sizes}[n - 1] \leftarrow \texttt{i}$
  $\texttt{i} \leftarrow \texttt{backtrace}[\texttt{i}, \texttt{target\_n\_group}]$
return `group_sizes`

$\{g_1, \ldots, g_n\}$ is defined as $\sum_{i\in[n]} k_i \cdot g_i$. Our goal is to find a partitioning with overhead smaller than $\texttt{sum}(\texttt{nchs}) \cdot (1 + \texttt{tol})$.

We use a dynamic programming algorithm that is based on the following main idea. We first sort the numbers in `nchs` in ascending order. Denote `L` as the size of `nchs`, we maintain a scratch table of size $\texttt{L} \times \texttt{G}$ whose ith row and jth column indicates the best possible overhead achieved by the first i numbers in `nchs` when having in total at most j partitions. The update formula of the DP table is

$$\texttt{dp}[i, j] \leftarrow \min_{k\in[i-1]} \texttt{dp}[k, j - 1] + \texttt{nchs}[i] \cdot (i - k), \tag{C.1}$$

where we try to find the best place (k) to put a new group/partition. By simultaneously

maintaining a matrix for backtracking, we can retrieve the best partition found by the algorithm.

The algorithm is shown in Algorithm 21. A practical trick to speed it up is to coalesce the identical values in `nchs` as done in line 3. Lines 7-9 initialize the buffers, and lines 11-23 are the main loop of the DP algorithm. Finally, the result partitioning is retrieved using lines 25-29.

Theoretical guarantee. Algorithm 21 is guaranteed to find an optimal grouping given a pre-specified number of groups, and is fairly efficient in practice. We formally state the problem in the following and provide the proof and analysis as follows.

As described in Appendix A.1, the grouping algorithm essentially takes as input a list of "# child node blocks" for each parent node block in a layer, and the goal is to partition all parent node blocks into K groups such that we minimize the following cost: the sum of the cost of each group, where the cost of a group is the maximum "# child node blocks" in the group times the number of parent node blocks in the group. In the following, we first demonstrate that the proposed dynamic programming (DP) algorithm (Algorithm 2) can retain the optimal cost for every K. We then proceed to analyze the time and space complexity of the algorithm.

To simplify notations, we assume the input is a vector of integers $[n_1, \ldots, n_N]$. We assume without loss of generality that the numbers are sorted because if not, we can apply any sorting algorithm. The main idea of the DP algorithm is to maintain a table termed dp of size N times K, where $\texttt{dp}[i, j]$ indicates the optimal cost when partitioning the first i integers into j groups. For the base cases, we can set $\texttt{dp}[i, 1] = n_i(\forall i)$ and $\texttt{dp}[1, j] = n_1(\forall j)$. For the inductive case, we have Eq. (C.1). It is straightforward to verify that when $\texttt{dp}[k, j-1](\forall k \in [1, i-1])$ are optimal, $\texttt{dp}[i, j]$ is also optimal. Therefore, for any K, Algorithm 21 computes the optimal grouping strategy for K groups.

Efficiency. We then focus on the runtime. Given N and K, Algorithm 21 requires $\mathcal{O}(KN^2)$ runtime and $\mathcal{O}(KN)$ memory, which is undesired for large N (in practice, we set

Algorithm 22 Backward pass of a sum layer group w.r.t. parameters

Inputs: log-probs of product nodes $\boldsymbol{l}_{\text{prod}}$, log-probs of sum nodes $\boldsymbol{l}_{\text{sum}}$, flows of sum nodes $\texttt{f}_{\text{sum}}$, flattened parameter vector $\boldsymbol{\theta}_{\text{flat}}$, `sum_ids`, `prod_ids`, `param_ids`
Inputs: # sum nodes: M, # product nodes: N, batch size: B
Inputs: block sizes K_M, K_N, K_B for the sum node, product node, and batch dimensions, respectively
Inputs: number of sum node blocks C_M; number of product node blocks C_N; number of batch blocks C_B
Outputs: flows of params $\texttt{f}_{\text{params}}$
Kernel launch: schedule to launch $C_M \times C_N$ thread-blocks with $\texttt{m}=0,\dots,C_M-1$ and $\texttt{n}=0,\dots,C_N-1$
$\texttt{cum} \leftarrow (0)_{K_M \times K_N} \in \mathbb{R}^{K_M \times K_N}$ ▷ Scratch space on SRAM
$\texttt{ms}, \texttt{me} \leftarrow \texttt{sum\_ids}[\texttt{m}], \texttt{sum\_ids}[\texttt{m}] + K_M$
$\texttt{ns}, \texttt{ne} \leftarrow \texttt{prod\_ids}[\texttt{m}, \texttt{n}], \texttt{prod\_ids}[\texttt{m}, \texttt{n}] + K_N$
for $\texttt{b} = 0$ **to** $C_B - 1$ **do**
  $\texttt{bs}, \texttt{be} \leftarrow \texttt{b} \cdot K_B, (\texttt{b}+1) \cdot K_B$ ▷ Start and end batch index
  Load $\boldsymbol{f}_{\text{s}} \leftarrow \texttt{f}_{\text{sum}}[\texttt{ms:me}, \texttt{bs:be}] \in \mathbb{R}^{K_M \times K_B}$ and $\boldsymbol{l}_{\text{s}} \leftarrow \boldsymbol{l}_{\text{sum}}[\texttt{ms:me}, \texttt{bs:be}] \in \mathbb{R}^{K_M \times K_B}$ to SRAM
  Load $\boldsymbol{l}_{\text{p}} \leftarrow \boldsymbol{l}_{\text{prod}}[\texttt{ns:ne}, \texttt{bs:be}] \in \mathbb{R}^{K_N \times K_B}$ to SRAM
  $\texttt{log\_nf} \leftarrow \log(\boldsymbol{f}_{\text{s}}) - \boldsymbol{l}_{\text{s}}$
  $\texttt{log\_nf\_max} \leftarrow \texttt{max}(\texttt{log\_nf}, \texttt{dim}=0) \in \mathbb{R}^{1 \times K_B}$ ▷ Compute on chip
  $\texttt{log\_nf\_sub} \leftarrow \texttt{exp}(\texttt{log\_nf} - \texttt{log\_nf\_max}) \in \mathbb{R}^{K_M \times K_B}$
  $\texttt{scaled\_emars} \leftarrow \texttt{transpose}(\texttt{exp}(\boldsymbol{p}_{\text{p}} + \texttt{log\_nf\_max})) \in \mathbb{R}^{K_B \times K_N}$
  $\texttt{partial\_flows} \leftarrow \texttt{matmul}(\texttt{log\_nf\_sub}, \texttt{scaled\_emars}) \in \mathbb{R}^{K_M \times K_N}$ ▷ With Tensor Cores
  $\texttt{cum} \leftarrow \texttt{cum} + \texttt{partial\_flows}$
$\texttt{ps}, \texttt{pe} \leftarrow \texttt{param\_ids}[\texttt{m}, \texttt{n}], \texttt{param\_ids}[\texttt{m}, \texttt{n}] + K_M \cdot K_N$
$\texttt{f}_{\text{params}}[\texttt{ps:pe}] \leftarrow \texttt{f}_{\text{params}}[\texttt{ps:pe}] + \boldsymbol{\theta}_{\text{flat}}[\texttt{ps:pe}] * \texttt{cum.view}(K_M * K_N)$

K to be smaller than 10). However, as demonstrated in Algorithm 21 (line 3), we only need to enumerate through the unique values in $[n_1, \dots, n_N]$, which could potentially lower the computation cost significantly. Even when we are dealing with highly non-structured PCs, we can always round the numbers up to a minimum integer that is divisible by a small integer such as 10. This allows us to achieve a decent approximated solution with much less computation time.

C.1.2 Details of the Backpropagation Algorithm for Sum Layers

We compute the backward pass with respect to the inputs and the parameters of the sum layer in two different kernels as we need two different layer partitioning strategies to improve efficiency. In the following, we first introduce the backpropagation algorithm for the parameters since it reuses the index tensors compiled for the forward pass (i.e., `sum_ids`, `prod_ids`, and `param_ids`).

The algorithm is shown in Algorithm 22. In addition to the log-probabilities of the product nodes (i.e., $\boldsymbol{l}_{\text{prod}}$), the log-probabilities of the sum nodes (i.e., $\boldsymbol{l}_{\text{sum}}$), and the flattened parameters (i.e., $\boldsymbol{\theta}_{\text{flat}}$), the algorithm takes as input the flows $\texttt{f}_{\text{sum}}$ computed for the sum nodes. Following Definition 8, we can compute the flow w.r.t. the sum parameters as

$$\mathrm{F}_{n,c}(\boldsymbol{x}) := \theta_{n,c} \cdot p_c(\boldsymbol{x})/p_n(\boldsymbol{x}) \cdot \mathrm{F}_n(\boldsymbol{x}).$$

Similar to Algorithm 4, we partition the sum nodes, product nodes, and samples into blocks of size K_M, K_N, and K_B, respectively. We schedule to launch $C_M \times C_N$ thread-blocks, each responsible for computing the parameter flows for a block of $K_M \times K_N$ parameter flows. The main loop (line 10) iterates through blocks of K_B samples. In every iteration, we first load the log-probabilities (i.e., $\boldsymbol{l}_{\text{s}}$ and $\boldsymbol{l}_{\text{p}}$) and the sum node flows (i.e., $\boldsymbol{f}_{\text{s}}$) to compute the partial flow $p_c(\boldsymbol{x})/p_n(\boldsymbol{x}) \cdot \mathrm{F}_n(\boldsymbol{x})$ for the block of samples (note that this equals $\mathrm{F}_{n,c}(\boldsymbol{x})/\theta_{n,c}$. The partial flows are accumulated in the matrix `cum` initialized in line 7. After processing all blocks of samples, we add back the parameter flows by accumulating `cum` $*$ [the corresponding parameters] in line 21.

As elaborated in Section 4.4, if we use the same set of index tensors used in the forward pass, we have the problem of different thread-blocks needing to write (partial) flows to the same input product node blocks. Therefore, we do a separate compilation step for the backward pass. Consider a sum layer with sum node blocks of size K_M and child product node blocks of size K_N. We first partition the C_N children into groups such that every child node block in a group has a similar number of parents. This is done by the dynamic programming algorithm described in Section C.1.1.

Similar to the compilation procedure of the forward pass, for a group with C_N child node blocks (assume every block has C_M blocks of parents), we generate three index tensors: $\texttt{ch\_ids} \in \mathbb{Z}^{C_N}$ and $\texttt{par\_ids}, \texttt{par\_param\_ids} \in \mathbb{Z}^{C_N \times C_M}$. `ch_ids` contains the initial index of all C_N child node blocks belonging to the group. For the ith node block in the group

(i.e., the product node block with the initial index $\texttt{ch\_ids}[i]$), $\texttt{par\_ids}[i,:]$ encode the start indices of its parent sum node blocks, and $\texttt{par\_param\_ids}[i,:]$ represent the corresponding initial parameter indices.

The main algorithmic procedure is very similar to *Algorithm* 4. Specifically, the kernel schedules to launch $C_N \times C_B$ thread-blocks each computing a block of $K_N \times K_B$ product node flows. In the main loop (line 9), we iterate through all C_M parent node blocks. In lines 13-16, we are essentially computing $\theta_{n,c}/p_n(\boldsymbol{x}) \cdot \mathrm{F}_n(\boldsymbol{x})$ (notations inherited from Definition 8) for the block of $K_N \times K_B$ values using the logsumexp trick. Finally, we store the results back to $\texttt{f}_{\text{prod}}$.

Algorithm 23 Backward pass of a sum layer group w.r.t. inputs

1: **Inputs:** log-probs of product nodes $\boldsymbol{l}_{\text{prod}}$, log-probs of sum nodes $\boldsymbol{l}_{\text{sum}}$, flows of sum nodes $\texttt{f}_{\text{sum}}$, flattened parameter vector $\boldsymbol{\theta}_{\text{flat}}$, $\texttt{ch\_ids}$, $\texttt{par\_ids}$, $\texttt{par\_param\_ids}$
2: **Inputs:** # sum nodes: M, # product nodes: N, batch size: B
3: **Inputs:** block sizes K_M, K_N, K_B for the sum node, product node, and batch dimensions, respectively
4: **Inputs:** number of sum node blocks C_M; number of product node blocks C_N; number of batch blocks C_B
5: **Outputs:** flows of inputs $\texttt{f}_{\text{prod}}$
6: **Kernel launch:** schedule to launch $C_N \times C_B$ thread-blocks with $\texttt{n}=0,\ldots,C_N-1$ and $\texttt{b}=0,\ldots,C_B-1$
7: $\texttt{cum} \leftarrow (-\infty)_{K_N \times K_B} \in \mathbb{R}^{K_N \times K_B}$ ▷ Scratch space on SRAM
8: $\texttt{bs}, \texttt{be} \leftarrow \texttt{b} \cdot K_B, (\texttt{b}+1) \cdot K_B$
9: **for** $\texttt{m} = 0$ **to** $C_M - 1$ **do**
10: $\quad \texttt{ps}, \texttt{pe} \leftarrow \texttt{par\_param\_ids}[\texttt{n}, \texttt{m}]$
11: $\quad$ Load $\boldsymbol{f}_{\text{s}} \leftarrow \texttt{f}_{\text{sum}}[\texttt{ms:me}, \texttt{bs:be}] \in \mathbb{R}^{K_M \times K_B}$ and $\boldsymbol{l}_{\text{s}} \leftarrow \boldsymbol{l}_{\text{sum}}[\texttt{ms:me}, \texttt{bs:be}] \in \mathbb{R}^{K_M \times K_B}$ to SRAM
12: $\quad$ Load $\boldsymbol{\theta} \leftarrow \texttt{transpose}(\boldsymbol{\theta}_{\text{flat}}[\texttt{ps:pe}].\texttt{view}(K_M, K_N)) \in \mathbb{R}^{K_N \times K_M}$ to SRAM
13: $\quad \texttt{log\_nf} \leftarrow \log(\boldsymbol{f}_{\text{s}}) - \boldsymbol{l}_{\text{s}}$
14: $\quad \texttt{log\_nf\_max} \leftarrow \texttt{max}(\texttt{log\_nf}, \texttt{dim}=0) \in \mathbb{R}^{1 \times K_B}$ ▷ Compute on chip
15: $\quad \texttt{log\_nf\_sub} \leftarrow \texttt{exp}(\texttt{log\_nf} - \texttt{log\_nf\_max}) \in \mathbb{R}^{K_M \times K_B}$
16: $\quad \texttt{partial\_flows} \leftarrow \texttt{matmul}(\boldsymbol{\theta}, \texttt{log\_nf\_sub}) \in \mathbb{R}^{K_M \times K_N}$ ▷ With Tensor Cores
$\quad\quad\quad\quad \texttt{cum} \leftarrow \texttt{where}(\texttt{log\_nf\_max} > \texttt{cum},$
$\quad\quad\quad\quad\quad\quad \texttt{log}(\texttt{partial\_flows} + \texttt{exp}(\texttt{cum} - \texttt{log\_nf\_max}) + \texttt{log\_nf\_max},$
$\quad\quad\quad\quad\quad\quad \texttt{log}(\texttt{exp}(\texttt{log\_nf\_max} - \texttt{cum}) \cdot \texttt{partial\_flows} + 1) + \texttt{cum})$
17:
18: $\texttt{ns}, \texttt{ne} \leftarrow \texttt{ch\_ids}[\texttt{n}], \texttt{ch\_ids}[\texttt{n}] + K_N$
19: $\texttt{f}_{\text{prod}}[\texttt{ns:ne}, \texttt{bs:be}] \leftarrow \exp(\texttt{cum} + \boldsymbol{l}_{\text{prod}}[\texttt{ns:ne}, \texttt{bs:be}])$

C.1.3 PCs with Tied Parameters

Formally, PCs with tied parameters are PCs containing same sub-structures in different parts of its DAG. Although the nodes in these sub-structures could have different semantics, they can have shared/tied parameters. For example, in homogeneous HMMs, although the transition probabilities between different pairs of consecutive latent variables are represented

by different sets of nodes and edges in the PC, they all have the *same* set of probability parameters.

PyJuice can be readily adapted to PCs with tied parameters. For the forward pass, we just need the compiler to assign the same parameter indices in `param_ids`. Similarly, we only need to slightly change the compilation procedure of `par_param_ids`. One notable difference is that in the backward pass w.r.t. the parameters, multiple thread-blocks would need to write partial flows to the same memory addresses, which leads to inter-thread-block barriers. We implemented a memory-IO tradeoff by letting the compiler create new sets of memory addresses to store the parameter flows when the number of thread-blocks writing to the same address is greater than a predefined threshold (by default set to 4).

C.2 Additional Technical Details

C.2.1 Block-Sparsity of Common PC Structures

Most commonly-adopted PC structures such as PD [136], RAT-SPN [133], and HCLT [97] have block-sparse sum layers because one of the key building blocks of the structure is a set of sum nodes fully connected to their inputs. Therefore, every sum layer must contain multiple fully-connected blocks of sum and product nodes, and hence they are block sparse.

C.2.2 Relation Between PC Flows and Gradients

We first show the equality for the node flows:

$$\mathrm{F}_n(\boldsymbol{x}) = \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial \log p_n(\boldsymbol{x})}. \tag{C.2}$$

We do the proof by induction. As a base case, we have by definition that $\mathrm{F}_{n_r}(\boldsymbol{x}) = \partial \log p_{n_r}(\boldsymbol{x}) / \partial \log p_{n_r}(\boldsymbol{x}) = 1$.

Next, suppose n is a sum or an input node, and for all its parents m, we have Eq. (C.2)

is satisfied by induction. Since all parents of n are product nodes, we have

$$\mathrm{F}_n(\boldsymbol{x}) = \sum_{m \in \mathsf{pa}(n)} \mathrm{F}_m(\boldsymbol{x}) = \sum_{m \in \mathsf{pa}(n)} \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial \log p_m(\boldsymbol{x})} = \sum_{m \in \mathsf{pa}(n)} \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial \log p_{n \to m}(\boldsymbol{x})} = \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial \log p_n(\boldsymbol{x})},$$

where $p_{n \to m}(\boldsymbol{x})$ denotes the probability carried by the edge from n to m.

Finally, suppose n is a product node and thus all its parents are sum nodes. We have

$$\mathrm{F}_n(\boldsymbol{x}) = \sum_{m \in \mathsf{pa}(n)} \frac{\theta_{m,n} \cdot p_n(\boldsymbol{x})}{p_m(\boldsymbol{x})} \cdot \mathrm{F}_m(\boldsymbol{x}) = \sum_{m \in \mathsf{pa}(n)} \frac{\theta_{m,n} \cdot p_n(\boldsymbol{x})}{p_m(\boldsymbol{x})} \cdot \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial \log p_m(\boldsymbol{x})}, \tag{C.3}$$

$$= \sum_{m \in \mathsf{pa}(n)} \theta_{m,n} \cdot p_n(\boldsymbol{x}) \cdot \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial p_m(\boldsymbol{x})}. \tag{C.4}$$

Denote $p_{n \to m}(\boldsymbol{x}) = \theta_{m,n} \cdot p_n(\boldsymbol{x})$ as the probability carried on the edge (m, n). Since $p_m(\boldsymbol{x}) = \sum_{n' \in \mathsf{ch}(m)} p_{n' \to m}(\boldsymbol{x})$, we have

$$\forall n \in \mathsf{ch}(m), \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial p_m(\boldsymbol{x})} = \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial p_{n \to m}(\boldsymbol{x})}.$$

Plug in the above equation on $\mathrm{F}_n(\boldsymbol{x})$, this results in

$$\mathrm{F}_n(\boldsymbol{x}) = \sum_{m \in \mathsf{pa}(n)} p_{n \to m}(\boldsymbol{x}) \cdot \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial p_{n \to m}(\boldsymbol{x})} = \sum_{m \in \mathsf{pa}(n)} \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial \log p_{n \to m}(\boldsymbol{x})} = \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial \log p_n(\boldsymbol{x})}. \tag{C.5}$$

We move on to demonstrate the following relation:

$$\mathrm{F}_{n,c}(\boldsymbol{x}) = \theta_{n,c} \cdot \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial \theta_{n,c}} = \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial \log \theta_{n,c}},$$

where n is a sum node and c is one of its children. We reuse the results derived in Eqs. (C.4) and (C.5), where we replace n with c and m with n:

$$\mathrm{F}_{n,c}(\boldsymbol{x}) = \frac{\theta_{n,c} \cdot p_c(\boldsymbol{x})}{p_n(\boldsymbol{x})} \cdot \mathrm{F}_n(\boldsymbol{x}) = \theta_{n,c} \cdot p_c(\boldsymbol{x}) \cdot \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial p_n(\boldsymbol{x})} = \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial \log p_{c \to n}(\boldsymbol{x})} = \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial \log \theta_{n,c}}.$$

C.3 Experimental Details

C.3.1 The Adopted Block-Sparse PC Layer

The PC layer contains 200 independent fully-connected sets of nodes. Every connected subset consists of 1024 sum nodes and 1024 product nodes. When compiling the layer, we divide the layer into blocks of size 32. When dropping 32×32 edge blocks from the layer, we ensure that every sum node has at least one child.

C.3.2 Details of Training the HMM Language Model

Following [192], we first fine-tune a GPT-2 model with the CommonGen dataset. We then sample 8M sequences of length 32 from the fine-tuned GPT-2. After initializing the HMM parameters with latent variable distillation, we fine-tune the HMM with the sampled data. Specifically, following [192], we divide the 8M samples into 40 equally-sized subsets, and run full-batch EM on the 40 subsets repeatedly. Another set of 800K samples is drawn from the fine-tuned GPT as the validation set.

C.3.3 Details of Training the Sparse Image Model

Following [100], we fine-tune the model with an equivalent batch size of 6400 and a step size of 0.01 in the mini-batch EM algorithm. Specifically, suppose $\boldsymbol{\theta}$ are the current parameters, $\boldsymbol{\theta}^{\text{new}}$ are the new set of parameters computed by the EM update. Given step size α, the update formula is $\boldsymbol{\theta} \leftarrow (1-\alpha)\boldsymbol{\theta} + \alpha\boldsymbol{\theta}^{\text{new}}$.

C.3.4 Additional Benchmark Results

Hyperparameters of the adopted HCLTs. We adopt two HCLTs [97] with hidden sizes 256 and 512, respectively. The backbone CLT structure is constructed using 20,000 randomly selected training samples.

Table C.2: **Average (± standard deviation of 5 runs) runtime (in seconds) per training epoch** of 60K samples for PyJuice and the baselines on five RAT-SPNs [133] with different sizes. All other settings are the same as described in Section 4.5.1.

# nodes	58K	116K	232K	465K	930K
# edges	616K	2.2M	8.6M	33.4M	132M
EiNet	$60.29_{\pm 0.30}$	$136.85_{\pm 0.13}$	$282.58_{\pm 0.27}$	$690.73_{\pm 0.08}$	$1936.28_{\pm 0.26}$
Juice.jl	$4.41_{\pm 0.21}$	$11.57_{\pm 0.07}$	$32.74_{\pm 1.86}$	$121.25_{\pm 0.43}$	$331.98_{\pm 2.87}$
PyJuice	$\mathbf{1.53}_{\pm 0.07}$	$\mathbf{3.11}_{\pm 0.07}$	$\mathbf{6.47}_{\pm 0.08}$	$\mathbf{13.62}_{\pm 0.37}$	$\mathbf{30.69}_{\pm 0.19}$

Hyperparameters of the adopted PDs. Starting from the set of all random variables, the PD structure recursively splits the subset with product nodes. Specifically, consider an image represented as a $H \times W \times C$ (H is the hight; W is the width; C is the number of channels), the PD structure recursively splits over both the height and the width coordinates, where every coordinate has a set of pre-defined split points. For both the height and the width coordinates, we add split points with interval 2. PD-mid has a hidden dimension of 128 and PD-large has 256.

Benchmark results on WikiText-103. Table C.1 illustrates results on WikiText-103. We train the model on sequences with 64 tokens. We adopt two (homogeneous) HMM models, HMM-mid and HMM-large with hidden sizes 2048 and 4096, respectively.

Table C.1: Density estimation performance of PCs on the WikiText-103 dataset. Reported numbers are test set perplexity.

Dataset	HMM-mid	HMM-large
WikiText-103	146.59	167.65

C.4 Additional Experiments

C.4.1 Runtime on Different GPUs

In addition to the RTX 4090 GPU adopted in the experiments in Table 4.1, we compare the runtime of PyJuice with the baselines on an NVIDIA A40 GPU. As shown in the following table, PyJuice is still significantly faster than all baselines for PCs of different sizes.

C.4.2 Runtime on Different Batch Sizes

As a supplement to Table 4.1, we report the runtime for a RAT-SPN [133] with $465K$ nodes and $33.4M$ edges using batch sizes $\{8, 16, 32, 64, 128, 256, 512\}$. To minimize distractions, we only record the time to compute the forward and backward process, but not the time used for EM updates. Results are shown in the table below.

Table C.3: **Average ($\pm$ standard deviation of 5 runs) runtime (in seconds) per training epoch (excluding EM updates)** of 60K samples for PyJuice and the baselines on a RAT-SPNs [133] with $465K$ nodes and $33.4M$ edges. All other settings are the same as described in Section 4.5.1. OOM denotes out-of-memory.

Batch size	8	16	32	64	128	256	512
EiNet	$332.87_{\pm 0.21}$	OOM	OOM	OOM	OOM	OOM	OOM
Juice.jl	$1045.04_{\pm 0.06}$	$853.15_{\pm 0.03}$	$775.87_{\pm 0.02}$	$642.54_{\pm 0.04}$	$324.23_{\pm 0.02}$	$163.68_{\pm 0.02}$	$80.57_{\pm 0.01}$
PyJuice	$\mathbf{43.09}_{\pm 0.04}$	$\mathbf{18.63}_{\pm 0.02}$	$\mathbf{7.38}_{\pm 0.01}$	$\mathbf{4.58}_{\pm 0.01}$	$\mathbf{3.50}_{\pm 0.01}$	$\mathbf{3.04}_{\pm 0.01}$	$\mathbf{2.76}_{\pm 0.03}$

Appendix D

Applications

D.1 Image Inpainting via Tractable Steering of Diffusion Models

D.1.1 Proof of Theorem 11

The proof contains two main pairs: (i) shows that the forward pass computes $\sum_{\boldsymbol{x}_0} \prod_i w_i(\boldsymbol{x}_0^i) \cdot p(\boldsymbol{x}_0)$, and (ii) demonstrates the backward pass computes the conditional probabilities $p_{\text{TPM}}(\tilde{x}_0^i | \boldsymbol{x}_t, \boldsymbol{x}_0^{\text{k}})$.

Correctness of the forward pass We show by induction that the forward value $\texttt{fw}_n$ of every node n computes $\sum_{\boldsymbol{x}_0} \prod_{i \in \phi(n)} w_i(x_0^i) \cdot p(\boldsymbol{x}_0)$.

• Base case: input nodes. By definition, for every input node defined on variable $X_0^i := \phi(n)$, its forward value is $\sum_{\boldsymbol{x}_0} w_i(x_0^i) \cdot p_n(\boldsymbol{x}_0)$.

• Inductive case: product nodes. For every product node n, assume the forward value of its every child node c satisfies $\texttt{fw}_c = \sum_{\boldsymbol{x}_0} \prod_{i \in \phi(c)} w_i(x_0^i) \cdot p_m(\boldsymbol{x}_0)$. Note that the forward value of product nodes is computed according to Eq. (2.1), we have

$$
\begin{aligned}
\texttt{fw}_n &= \prod_{c \in \mathsf{ch}(n)} \texttt{fw}_c, \\
&= \prod_{c \in \mathsf{ch}(n)} \sum_{\boldsymbol{x}_0} \prod_{i \in \phi(c)} w_i(x_0^i) \cdot p_c(\boldsymbol{x}_0), \\
&\overset{(a)}{=} \sum_{\boldsymbol{x}_0} \prod_{c \in \mathsf{ch}(n)} \prod_{i \in \phi(c)} w_i(x_0^i) \cdot p_c(\boldsymbol{x}_0), \\
&\overset{(b)}{=} \sum_{\boldsymbol{x}_0} \prod_{i \in \phi(n)} w_i(x_0^i) \prod_{c \in \mathsf{ch}(n)} p_c(\boldsymbol{x}_0), \\
&\overset{(c)}{=} \sum_{\boldsymbol{x}_0} \prod_{i \in \phi(n)} w_i(x_0^i) \cdot p_n(\boldsymbol{x}_0),
\end{aligned}
$$

where (a) follows from the fact that scopes of the children $\phi(c)$ are disjoint, and the child PCs $\{p_c(\boldsymbol{x}_0)\}_c$ are defined on disjoint sets; (b) follows from the fact that $\phi(n) = \bigcup_{c \in \mathsf{ch}(n)} \phi(c)$; (c) follows the definition in Eq. (2.1).

• Inductive case: sum nodes. Similar to the case of product nodes, for every sum node n, we assume the forward value of its children satisfies the induction condition. We have

$$\begin{aligned}
\mathtt{fw}_n &= \sum_{c\in\mathsf{ch}(n)} \theta_{n,c}\cdot \mathtt{fw}_c,\\
&= \sum_{c\in\mathsf{ch}(n)} \theta_{n,c} \sum_{\boldsymbol{x}_0} \prod_{i\in\phi(c)} w_i(x_0^i)\cdot p_c(\boldsymbol{x}_0),\\
&\overset{(a)}{=} \sum_{\boldsymbol{x}_0} \prod_{i\in\phi(n)} w_i(x_0^i)\cdot \sum_{c\in\mathsf{ch}(n)} \phi_{n,c}\cdot p_c(\boldsymbol{x}_0),\\
&\overset{(b)}{=} \sum_{\boldsymbol{x}_0} \prod_{i\in\phi(n)} w_i(x_0^i)\cdot p_n(\boldsymbol{x}_0),
\end{aligned}$$

where (a) holds since $\forall c\in\mathsf{ch}(n), \phi(c)=\phi(n)$, and (b) follows from Eq. (2.1).

Therefore, the forward value of every node n is defined by $\mathtt{fw}_n = \sum_{\boldsymbol{x}_0}\prod_{i\in\phi(n)} w_i(x_0^i)\cdot p_m(\boldsymbol{x}_0)$.

Correctness of the backward pass We first provide an intuitive semantics for the backward value $\mathtt{bk}_n$ of every node: for every node n, if its forward value $\mathtt{fw}_n$ were to set to $\mathtt{fw}'_n$ (the other inputs of the PC remains unchanged), the value at the root node n_r would change to $(1-\mathtt{bk}_n)\cdot\mathtt{fw}_{n_r} + \mathtt{bk}_n\cdot\mathtt{fw}_{n_r}\cdot\mathtt{fw}'_n/\mathtt{fw}_n$. In the following, we prove this result by induction over the root node.

• Base case: the PC rooted at n. Denote $\mathtt{bk}_n^{n_r}$ as the backward value of node n w.r.t. the PC rooted at n_r. Since by definition $\mathtt{bk}_n^n = 1$, we have that when the value of n is changed to $\mathtt{fw}'_n$, the root node's value becomes

$$(1-\mathtt{bk}_n^n)\cdot\mathtt{fw}_{n_r} + \mathtt{bk}_n^n\cdot\mathtt{fw}_n\cdot\mathtt{fw}'_n/\mathtt{fw}_n = \mathtt{fw}'_n.$$

• Inductive case: sum node. Suppose the statement holds for all children of a sum node m. Define $\mathtt{bk}_{n,c}^m$ as the backward value of edge (n,c) for the PC rooted at m. Following the definition of the backward values, we have $\sum_{c\in\mathsf{ch}(m)}\mathtt{bk}_{n,c}^m = \mathtt{bk}_n^m$. When the value of n is

changed to $\mathtt{fw}'_n$, the value of m becomes:

$$\begin{aligned}
\sum_{c\in\mathsf{ch}(m)} \theta_{m,c}\cdot \mathtt{fw}'_c &= \sum_{c\in\mathsf{ch}(m)} \theta_{m,c}\cdot\big[(1-\mathtt{bk}_n^c)\cdot\mathtt{fw}_c + \mathtt{bk}_n^c\cdot\mathtt{fw}_c\cdot\mathtt{fw}'_n/\mathtt{fw}_n\big],\\
&= \underbrace{\sum_{c\in\mathsf{ch}(m)} \theta_{m,c}\cdot\mathtt{fw}_c}_{\mathtt{fw}_m} + \sum_{c\in\mathsf{ch}(m)} \theta_{m,c}\cdot\mathtt{bk}_n^c\cdot\mathtt{fw}_c\cdot\big(\mathtt{fw}'_n/\mathtt{fw}_n-1\big),\\
&= \mathtt{fw}_m + \sum_{c\in\mathsf{ch}(m)} \theta_{m,c}\cdot\underbrace{\Big(\mathtt{bk}_{n,c}^m\cdot\frac{\mathtt{fw}_m}{\theta_{m,c}\cdot\mathtt{fw}_c}\Big)}_{\mathtt{bk}_n^c}\cdot\mathtt{fw}_c\cdot\big(\mathtt{fw}'_n/\mathtt{fw}_n-1\big),\\
&= \mathtt{fw}_m + \underbrace{\sum_{c\in\mathsf{ch}(m)} \mathtt{bk}_{n,c}^m}_{\mathtt{bk}_n^m}\cdot\mathtt{fw}_m\cdot\big(\mathtt{fw}'_n/\mathtt{fw}_n-1\big),\\
&= (1-\mathtt{bk}_n^m)\cdot\mathtt{fw}_m + \mathtt{bk}_n^m\cdot\mathtt{fw}_m\cdot\mathtt{fw}'_n/\mathtt{fw}_n,
\end{aligned}$$

• Inductive case: product node. Suppose the statement holds for all children of a product node m. Thanks to decomposability, at most one of m's children can be an ancestor of n. Denote this child as c'. When the value of n is changed to $\mathtt{fw}'_n$, the value of m becomes:

$$\begin{aligned}
\prod_{c\in\mathsf{ch}(m),c\neq c'} \mathtt{fw}'_c &= \prod_{c\in\mathsf{ch}(m),c\neq c'} \mathtt{fw}_c\cdot\big[(1-\mathtt{bk}_n^{c'})\cdot\mathtt{fw}_{c'} + \mathtt{bk}_n^{c'}\cdot\mathtt{fw}_{c'}\cdot\mathtt{fw}'_n/\mathtt{fw}_n\big],\\
&\overset{(a)}{=} \prod_{c\in\mathsf{ch}(m),c\neq c'} \mathtt{fw}_c\cdot\big[(1-\mathtt{bk}_n^{m})\cdot\mathtt{fw}_{c'} + \mathtt{bk}_n^{m}\cdot\mathtt{fw}_{c'}\cdot\mathtt{fw}'_n/\mathtt{fw}_n\big],\\
&\overset{(b)}{=} (1-\mathtt{bk}_n^m)\cdot\mathtt{fw}_{c'} + \mathtt{bk}_n^m\cdot\mathtt{fw}_{c'}\cdot\mathtt{fw}'_n/\mathtt{fw}_n,
\end{aligned}$$

where (a) holds since $\mathtt{bk}_n^m = \mathtt{bk}_n^{c'}$ and (b) follows from the definition of product nodes in Eq. (2.1).

Next, assume that the input nodes are all indicators in the form of $\mathbb{1}[X_i = x_i]$. In fact, any discrete univariate distribution can be represented as a mixture (sum node) of indicators. By induction, we can show that the sum of backward values of all input nodes corresponding to a variable X_i is 1, since sum nodes only "divide" the backward value, and product nodes

preserve the backward value sent to them. By setting the value of the input node $\mathbb{1}[X_i = x_i]$, we are essentially computing $\prod_{j \neq i} w_j(x_j) \cdot \mathbb{1}[X_i = x_i] \cdot p(\boldsymbol{x})$. Therefore, the backward values of these input nodes are proportional to the target conditional probability $p_{\text{TPM}}(\tilde{x}_0^i | \boldsymbol{x}_t, \boldsymbol{x}_0^{\text{k}})$.

We are left with proving that the sum of backward values of all input nodes corresponding to variable X_i equals 1. To see this, consider the subset of nodes whose scope contains X_i. This subset of nodes represents a DAG with the root node as the only source node and input nodes of variables X_i as the sink. Consider the backward algorithm as computing flows in the DAG. For every node non-sink node, the amount of flow it accepts equals the amount it sends. Specifically, product nodes send all their flow to their only child node (according to decomposability, at most one child of a product node contains X_i in its scope); for every sum node n, the sum of flows sent to its children equal to the flow it receives. Since the flow sent by all source nodes equals the flow received by all sink nodes, we conclude that the backward values of the input nodes for variable X_i sum up to 1. □

D.1.2 Design Choices for High-Resolution Guided Image Inpainting

In all experiments, we compute $w_i^z(\tilde{z}_0^i)$ by first drawing 4 samples from $\frac{1}{Z}\prod_j w_j(\tilde{x}_0^j)$, and then feed them to the VQ-GAN's encoder. For every sample, we get a distribution over variable $\tilde{Z}_0^i$. $w_i^z(\tilde{z}_0^i)$ is then computed as the mean of the four distributions. In the following decoding phase that computes $p_{\text{TPM}}(\tilde{x}_0 | \boldsymbol{x}_t, \boldsymbol{x}_0^{\text{k}}) := \mathbb{E}_{\tilde{\boldsymbol{z}}_0 \sim p_{\text{TPM}}(\cdot | \boldsymbol{x}_t, \boldsymbol{x}_0^{\text{k}})}[p(\tilde{\boldsymbol{x}}_0 | \tilde{\boldsymbol{z}}_0)]$, we draw 8 random samples from $p_{\text{TPM}}(\cdot | \boldsymbol{x}_t, \boldsymbol{x}_0^{\text{k}})$ to estimate $p_{\text{TPM}}(\tilde{x}_0 | \boldsymbol{x}_t, \boldsymbol{x}_0^{\text{k}})$.

In the following, we describe the hyperparameters of the adopted VQ-GAN for all three datasets:

Table D.1: Hyperparameters of the adopted VQ-GAN models for Tiramisu.

	CelebA-HQ	ImageNet	LSUN-Bedroom
# latent variables	16×16	16×16	16×16
Vocab size	1024	16384	16384

Additional hyperparameters of Tiramisu For the distribution mixing hyperparameter α (cf. Section 5.1.2), we use an exponential decay schedule from a to b with a temperature parameter λ. Specifically, the mixing hyperparameter at time step T is

$$(b-a)\cdot\exp(-\lambda\cdot t/T)+a. \tag{D.1}$$

We further define a cutoff parameter t_{cut} such that when $t \leq t_{\text{cut}}$, the TPM guidance is not used. In all experiments, we used the first three samples in the validation set to tune the mixing hyperparameters. Hyperparameters are given in the following table. In all settings, we have $T = 250$.

Table D.2: Mixing hyperparameters of Tiramisu.

	CelebA-HQ	ImageNet	LSUN-Bedroom
a	0.8	0.8	0.8
b	1.0	1.0	1.0
λ	2.0	2.0	2.0
t_{cut}	200	235	235

D.1.3 PC Learning Details

The EM Parameter Learning Algorithm

When performing a feedforward evaluation (i.e., Eq. (2.1)), a PC takes as input a sample $\boldsymbol{x}$ and outputs its probability $p_n(\boldsymbol{x})$. Given a dataset $\mathcal{D}$, our goal is to learn a set of PC parameters (including sum edge parameters and input node/distribution parameters) to maximize the MLE objective: $\sum_{\boldsymbol{x}\in\mathcal{D}} \log p_n(\boldsymbol{x})$. The Expectation-Maximization algorithm is a natural way to learn PC parameters since PCs can be viewed as latent variable models with a hierarchically nested latent space [131]. There are two interpretations of the EM learning algorithm for PCs: one based on gradients [132] and the other based on a new concept called circuit flows [16]. We use the gradient-based interpretation since it is easier to understand.

Note that the feedforward computation of PCs is differentiable and can be modeled by a

computation graph. Therefore, after computing the log-likelihood $\log p_n(\boldsymbol{x})$ of a sample $\boldsymbol{x}$, we can efficiently compute its gradient with respect to all PC parameters via the backpropagation algorithm. Given a mini-batch of samples, we use the backpropagation algorithm to accumulate gradients for every parameter. Take sum parameters as an example. Define the cumulative gradient of $\theta_{n,c}$ as $g_{n,c}$, the updated parameters $\{\theta_{n,c}\}_{c\in\mathsf{ch}(n)}$ for every sum node n is given by:

$$\theta_{n,c} \leftarrow (1-\alpha)\cdot\theta_{n,c} + \alpha\cdot\frac{g_{n,c}\cdot\theta_{n,c}}{\sum_{m\in\mathsf{ch}(n)} g_{n,m}\cdot\theta_{n,m}},$$

where $\alpha \in (0, 1]$ is the step size. For input nodes, since we only use categorical distributions that can be equivalently represented as a mixture over indicator leaves (i.e., a sum node connecting these indicators), optimizing leaf parameters is equivalent to the sum parameter learning process.

Details of the PC Learning Pipeline

PC structure We adopt a variant of the original PD structure proposed in [136]. Specifically, starting from the whole image, the PD structure gradually uses product nodes to horizontally or vertically split the variable scope into half. As a result, the scope of every node represents a patch (of variable size) in the original image. We use categorical input nodes in accordance with the VQ-GAN's latent space. We treat the set of parameters belonging to nodes with the same scope as the parameters of the scope. Based on the original structure, we further tie the parameters representing every pixel and every 2×2 patch. Since the marginal distribution of every latent variable should be similar thanks to the spatial invariance of images.

Parameter learning This paper uses the latent variable distillation (LVD) technique [99,100] to initiate the PC parameters before further fine-tuning them with the EM algorithm described in Section D.1.3. Intuitively, LVD provides extra supervision to PC optimizers through semantic-aware latent variable assignments extracted from deep generative models.

We refer readers to the original papers for more details.

After initializing PC parameters with LVD, we further fine-tune the parameters with EM with the following hyperparameters:

Table D.3: Hyperparameters of EM fine-tuning process.

Name	Value
Step size	1.0
Batch size	20000
Pseudocount	0.1
# iterations	200

D.1.4 Details of the Main Experiments and the Baselines

Pretrained Models For all inpainting algorithms, we adopt the same diffusion model checkpoint pretrained by [108] (for CelebA-HQ) and OpenAI (for ImageNet and LSUN-Bedroom; https://github.com/openai/guided-diffusion). The links to the checkpoints for all three datasets are listed below.

• CelebA-HQ: https://drive.google.com/uc?id=1norNWWGYP3EZ_o05DmoW1ryKuKMmhlCX

• ImageNet: https://openaipublic.blob.core.windows.net/diffusion/jul-2021/256x256_diffusion.pt and https://openaipublic.blob.core.windows.net/diffusion/jul-2021/256x256_classifier.pt.

• LSUN-Bedroom: https://openaipublic.blob.core.windows.net/diffusion/jul-2021/lsun_bedroom.pt.

Data For CelebA-HQ and LSUN-Bedroom, we use the first 100 images in the validation set. We adopt the validation split of CelebA-HQ following [166]. For ImageNet, we use a random validation image for the first 100 classes.

Table D.4: User study results. We report the vote difference (%), i.e., [percentage of votes to Tiramisu] - [percentage of votes to the baseline]. The higher the vote difference value, the more the annotators prefer images generated by Tiramisu compared to the baseline.

Tasks		Algorithms			
Dataset	Mask	Tiramisu (ours)	CoPaint	RePaint	DPS
CelebA-HQ	Expand1	Reference	22	34	14
	Wide	Reference	26	30	22
ImageNet	Expand1	Reference	32	20	24
	Wide	Reference	14	6	12
LSUN-Bedroom	Expand1	Reference	18	2	8
	Wide	Reference	4	6	-6

D.1.5 Additional Experiments

User Study

Since image inpainting is an ill-posed problem and LPIPS alone may not be sufficient to indicate the performance of each algorithm, we recruited human evaluators to evaluate the quality of inpainted images. Specifically, for every baseline method, we sample inpainted image pairs from the baseline and Tiramisu using the same inputs (source image and mask). For every image pair, the evaluator is provided with both inpainted images and is tasked to select the better one based on the following criterion: images that visually look more natural and without artifacts (e.g., blurry, distorted). A screenshot of the interface is shown in Fig. D.1.

The user study is conducted on the three strongest baselines based on their overall LPIPS scores: CoPaint [191], RePaint [108], and DPS [23]. We use two mask types for comparison: "expand1" and "wide". For every comparison, we report the vote difference (%), which is the percentage of votes to Tiramisu subtracted by that of the baseline. A positive vote difference value means images generated by Tiramisu are preferred compared to the baselines, while a negative value suggests that the baseline is better than Tiramisu.

We adopt the three most competitive baselines, i.e., CoPaint, RePaint, and DPS, based on their average LPIPS scores (Table 5.1). For all three datasets, we conduct user studies

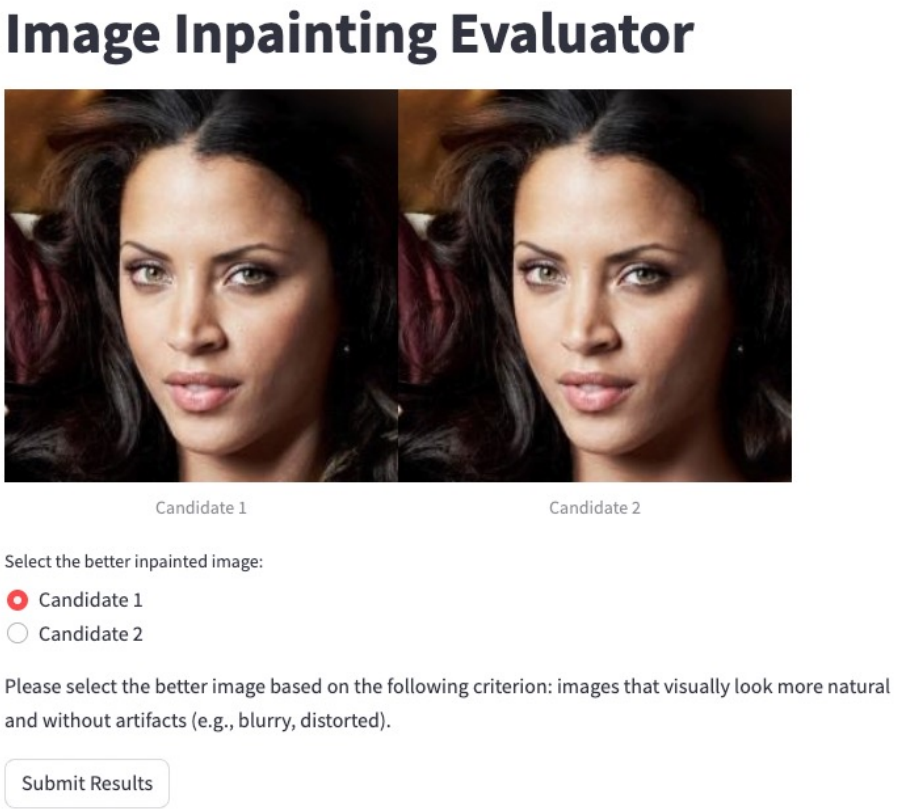


Figure D.1: User study interface.

on two types of masks: "expand1" and "wide". Results are shown in Table D.4. The vote difference scores are mostly high, indicating the superior inpainting performance of Tiramisu. Additionally, we observe that Tiramisu generally performs better with the "expand1" mask (with larger to-be-inpainted regions), which suggests that Tiramisu may be more helpful in the case of large-hole image inpainting.

Additional Qualitative Results

Please refer to Figs. D.2 to D.4 for additional qualitative results on all three adopted datasets.

D.1.6 Details of the Semantic Fusion Experiment

The mixing hyperparameters are the same as described in Section D.1.2.

The VQ-GAN encoder first generates an embedding $\mathbf{e}$ for every latent variable $\tilde{Z}_0^i$, and it is then discretized with the VQ codebook by selecting the ID in the codebook that has the minimum L2 distance with $\mathbf{e}$. We soften this process by setting $w_i^z(j) = \exp(-\|\mathbf{e} - \mathbf{e}_j\|_2/\lambda_{\text{sf}})$, where $\mathbf{e}_j$ is the jth embedding in the codebook, and λ_{sf} is the temperature that controls the semantic coherence level of the generated images. The closer λ_{sf} is to 0, the higher the coherence level.

Figure D.2: Additional qualitative results on CelebA-HQ with six mask types.

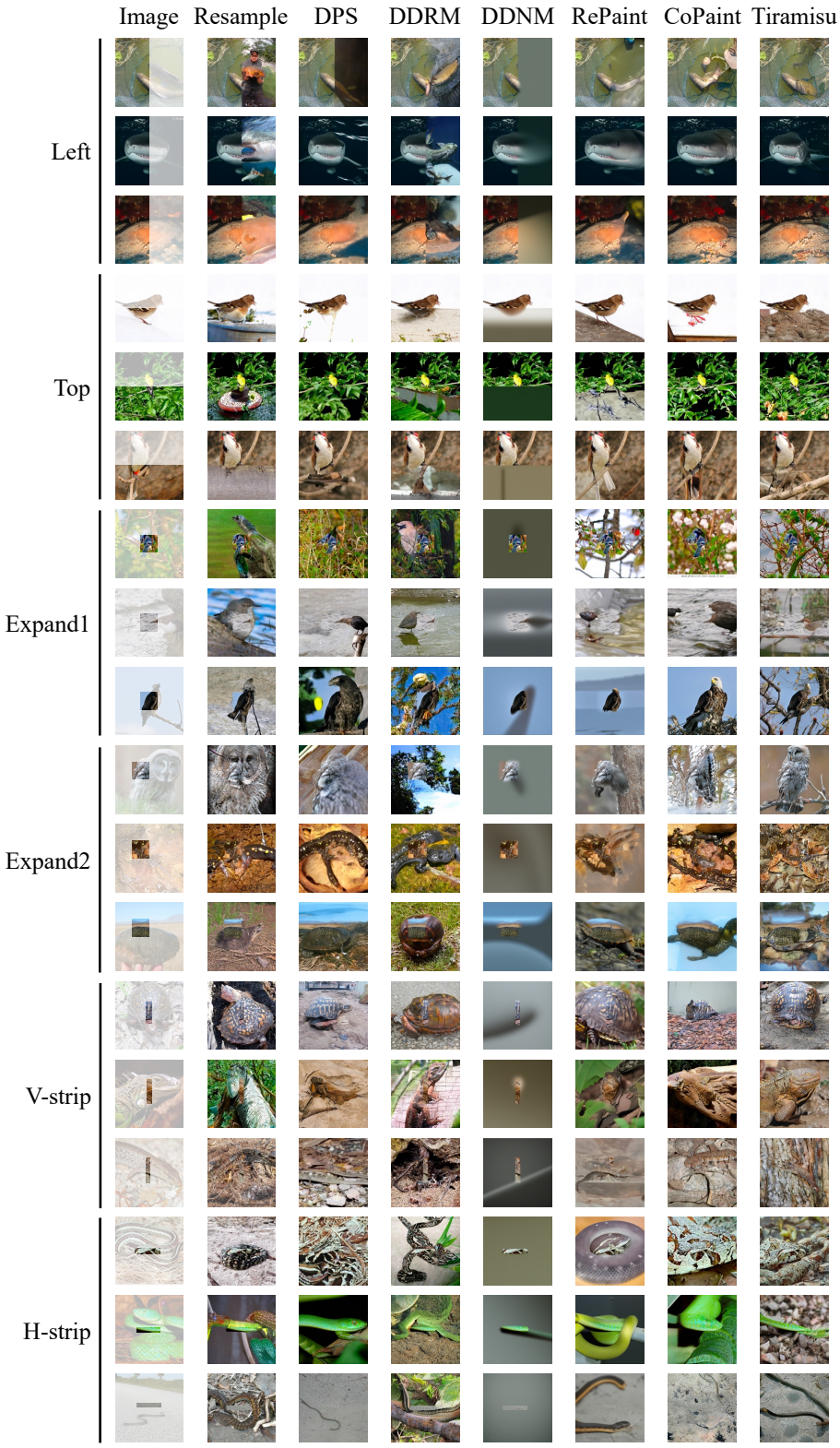


Figure D.3: Additional qualitative results on ImageNet with six mask types.

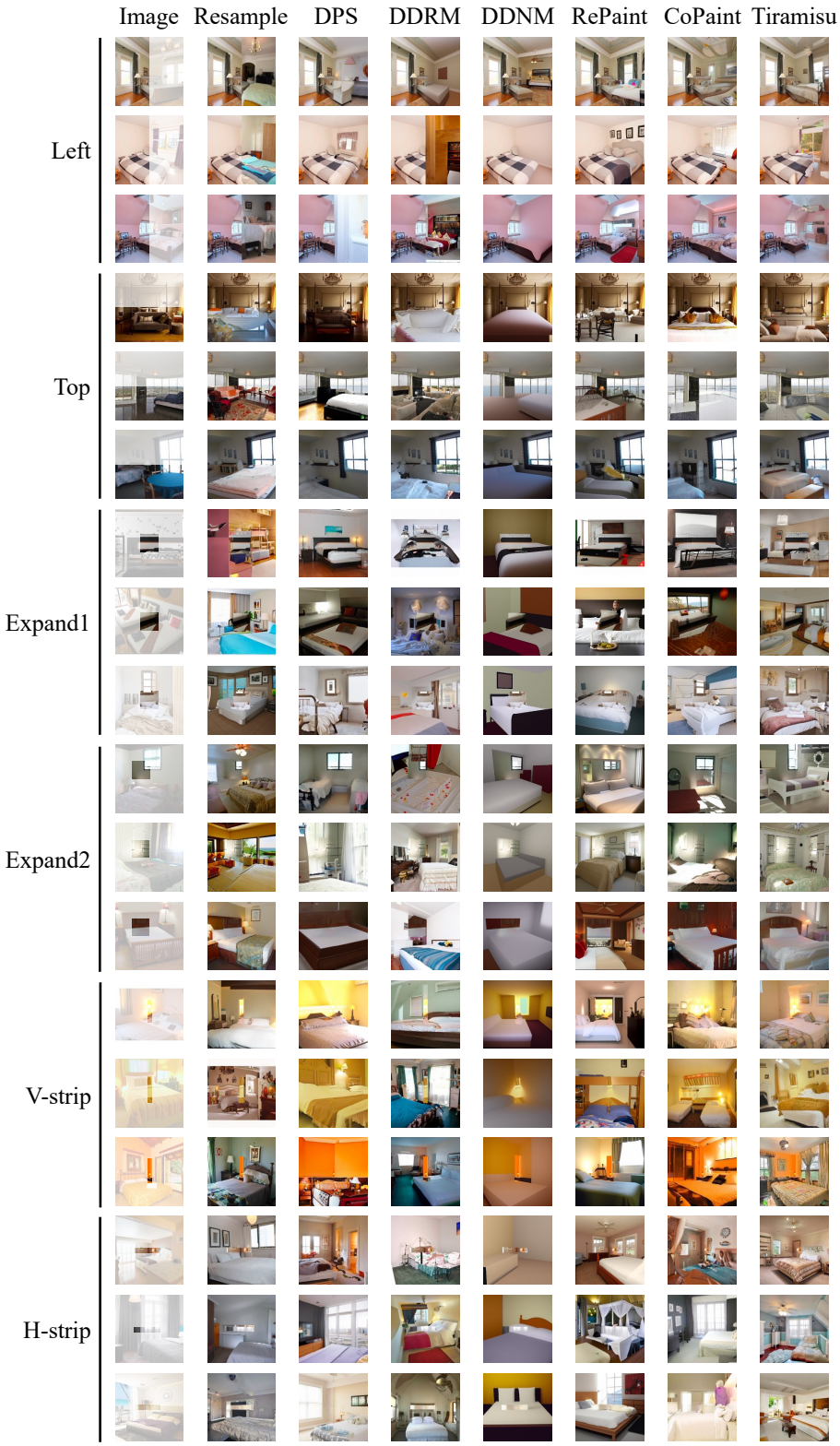


Figure D.4: Additional qualitative results on LSUN-Bedroom with six mask types.

D.2 Lossless Data Compression

D.2.1 Proof of Theorem 13

As hinted by the proof sketch given in the main text, this proof consists of three main parts — (i) construction of the optimal variable order π^* given a smooth and structured-decomposable PC, (ii) justify the correctness of Algorithm 6, and (iii) prove that $F_{\pi^*}(\boldsymbol{x})$ can be computed by evaluating no more than $\mathcal{O}(\log(K)\cdot|p|)$ PC units (i.e., analyze the time complexity of Algorithm 6).

Construction of an optimal variable order For ease of illustration, we first transform the original smooth and structured-decomposable PC into an equivalent PC where every product node has two children. Fig. D.5 illustrates this transformation on any product node with more than two children. Note that this operation will not change the number of parameters in a PC, and will only incur at most $2\cdot|p|$ edges.

We are now ready to define the variable tree (*vtree*) [82] of a smooth and structured-decomposable PC. Specifically, a *vtree* is a binary tree structure whose leaf nodes are labeled with a PC's input features/variables $\mathbf{X}$ (every leaf node is labeled with one variable). A PC conforms to a vtree if for every product unit n, there is a corresponding vtree node v such that children of n split the variable scope $\phi(n)$ in the same way as the children of the vtree node v. According to its definition, every smooth and structured-decomposable PC whose product units all have two children must conform to a vtree [82]. For example, the PC shown in Fig. D.6(a) conforms to the vtree illustrated in Fig. D.6(b). Similar to PCs, we define the scope $\phi(v)$ of a vtree node v as the set of all descendent leaf variables of v.

We say that a unit n in a smooth and structured-decomposable PC conforms to a node v in the PC's corresponding vtree if their scopes are identical. For ease of presentation, define $\varphi(p, v)$ as the set of PC units that conform to vtree node v. Additionally, we define $\varphi_{\text{sum}}(p, v)$ and $\varphi_{\text{prod}}(p, v)$ as the set of sum and product units in $\varphi(p, v)$, respectively.

Next, we define an operation that changes a vtree into an *ordered vtree*, where for each inner node v, its left child has more descendent leaf nodes than its right child. See Fig. D.6(c-d) as an example. The vtree in Fig. D.6(b) is transformed into an ordered vtree illustrated in Fig. D.6(c); the corresponding PC (Fig. D.6(a)) is converted into an *ordered PC* (Fig. D.6(d)). This transformation can be performed by all smooth and structured-decomposable PCs.

We are ready to define the optimal variable order. For a pair of ordered PC and ordered vtree, the optimal variable order π^* is defined as the order the leaf vtree nodes (each corresponds to a variable) are accessed following an inorder traverse of the vtree (left child accessed before right child).

Correctness of Algorithm 6 Assume we have access to a smooth, structured-decomposable, and ordered PC p and its corresponding vtree. Recall from the above construction, the optimal variable order π^* is the order following an inorder traverse of the vtree.

We show that it is sufficient to only evaluate the set of PC units stated in line 6 of Algorithm 6. Using our new definition of vtrees, we state line 6 in the following equivalent way. At iteration i (i.e., we want to compute the ith term in $F_\pi(\boldsymbol{x})$: $p(x_{\pi_1}, \ldots, x_{\pi_i})$), we need to evaluate all PC units that conform to any vtree node in the set $T_{p,i}$. Here $T_{p,i}$ is defined as the set of vtree nodes v that satisfy the following condition: $X_{\pi_i} \in \phi(v)$ and there does not exist a child c of v such that $\{X_{\pi_j}\}_{j=1}^{i} \in \phi(c)$. For ease of presentation, we refer to evaluate PC units $\varphi(p, v)$ when we say "evaluate a vtree node v".

First, we don't need to evaluate vtree units v where $X_{\pi_i} \notin \phi(v)$ because the probability of these PC units will be identical to that at iteration $i-1$ (i.e., when computing $p(x_{\pi_1}, \ldots, x_{\pi_{i-1}})$). Therefore, we only need to cache these probabilities computed in previous iterations.

Second, we don't need to evaluate vtree units v where at least one of its children c satisfy $\{X_{\pi_j}\}_{j=1}^{i-1} \in \phi(c)$ because we can obtain the target marginal probability $p(x_{\pi_1}, \ldots, x_{\pi_i})$ following lines 7-9 of Algorithm 6. We proceed to show how this is done in the following.

Denote the "highest" in $T_{p,i}$ as $v_{r,i}$ (i.e., the parent of $v_{r,i}$ is not in $T_{p,i}$). According to the

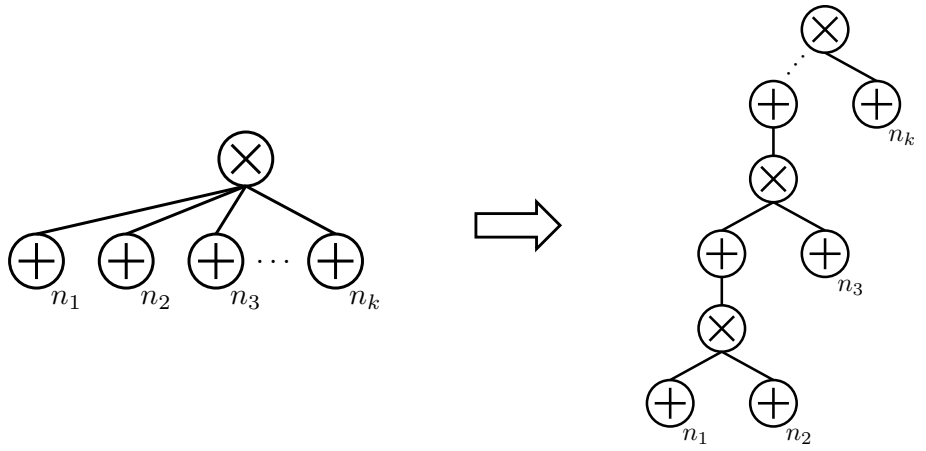


Figure D.5: Convert a product unit with k children into an equivalent PC where every product node has two children.

variable order π^*, $v_{r,i}$ uniquely exist for any $i \in [D]$. According to Algorithm 7, the top-down probabilities of PC units is defined as follows

• $p_{\text{down}}(n_r) = 1$, where n_r is the PC's root unit.

• For any product unit n, $p_{\text{down}}(n) = \sum_{m \in \text{par}(n)} p_{\text{down}}(m) \cdot \theta_{m,n}$, where $\text{par}(n)$ is the set of parent (sum) units of n.

• For any sum unit n, $p_{\text{down}}(n) = \sum_{m \in \text{par}(n)} p_{\text{down}}(m)$, where $\text{par}(n)$ is the set of parent (product) units of n.

We now prove that

$$p(x_{\pi_1}, \ldots, x_{\pi_i}) = \sum_{n \in \varphi_{\text{sum}}(p,v)} p_{\text{down}}(n) \cdot p_n(\boldsymbol{x}) \tag{D.2}$$

holds when $v = v_{r,i}$.

• **Base case:** If v is the vtree node correspond to n_r, then $\varphi_{\text{sum}}(p, v) = \{n_r\}$ and it is easy to verify that

$$p(x_{\pi_1}, \ldots, x_{\pi_i}) = p_{\text{down}}(n_r) \cdot p_{n_r}(\boldsymbol{x}) = \sum_{n \in \varphi_{\text{sum}}(p,v)} p_{\text{down}}(n) \cdot p_n(\boldsymbol{x})$$

• **Inductive case:** Suppose v is an ancestor of $v_{r,i}$ and the parent vtree node v_p of v satisfy

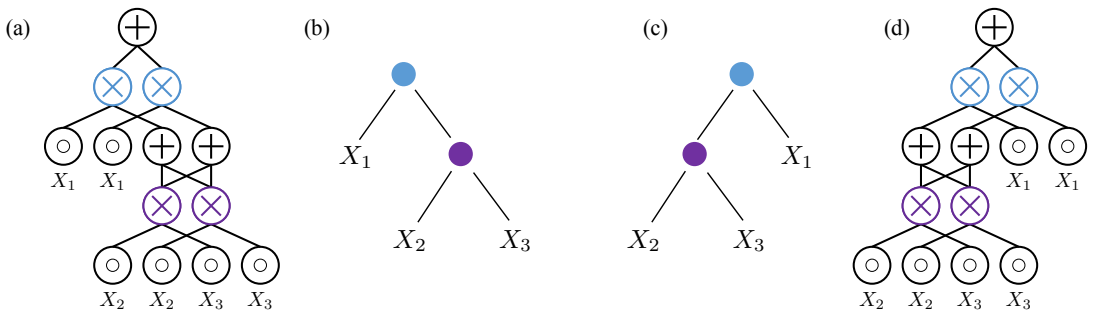


Figure D.6: (a-b): An example structured-decomposable PC and a corresponding *vtree*. (c-d): Converting (b) into an ordered vtree. (d) The converted ordered PC that is equivalent to (a).

Eq. (D.2). We have

$$
\begin{aligned}
p(x_{\pi_1}, \ldots, x_{\pi_i}) &= \sum_{m \in \varphi_{\text{sum}}(p, v_p)} p_{\text{down}}(m) \cdot p_m(\boldsymbol{x}), \\
&= \sum_{m \in \varphi_{\text{sum}}(p, v_p)} \sum_{n \in \mathsf{ch}(m)} p_{\text{down}}(m) \cdot \theta_{m,n} \cdot p_n(\boldsymbol{x}), \\
&\overset{(a)}{=} \sum_{n \in \varphi_{\text{prod}}(p, v_p)} \underbrace{\sum_{m \in \text{par}(n)} p_{\text{down}}(m) \cdot \theta_{m,n}}_{p_{\text{down}}(n)} \cdot p_n(\boldsymbol{x}), \\
&= \sum_{n \in \varphi_{\text{prod}}(p, v_p)} p_{\text{down}}(n) \cdot p_n(\boldsymbol{x}), \\
&\overset{(b)}{=} \sum_{n \in \varphi_{\text{prod}}(p, v_p)} \sum_{o \in \{o : o \in \mathsf{ch}(n), \{X_j\}_{j=1}^{i} \in \phi(o)\}} p_{\text{down}}(n) \cdot p_o(\boldsymbol{x}), \\
&\overset{(c)}{=} \sum_{o \in \varphi_{\text{sum}}(p, v)} \underbrace{\sum_{n \in \text{par}(o)} p_{\text{down}}(n) \cdot p_o(\boldsymbol{x})}_{p_{\text{down}}(o)}, \\
&= \sum_{o \in \varphi_{\text{sum}}(p, v)} p_{\text{down}}(o) \cdot p_o(\boldsymbol{x}),
\end{aligned}
$$

where (a) reorders the terms for summation; (b) holds since $\forall n \in \varphi_{\text{prod}}(p, v_p)$, $p_n(\boldsymbol{x}) =$

$\prod_{o\in\mathsf{ch}(n)} p_o(\boldsymbol{x})$ and $\forall o \in \mathsf{ch}(n)$ such that $\{X_j\}_{j=1}^{i} \cap \phi(o) = \varnothing$, $p_o(\boldsymbol{x}) = 1$;[1] (c) holds because

$$\bigcup_{n\in\varphi_{\text{prod}}(p,v_p)} \{o : o \in \mathsf{ch}(n), \{X_j\}_{j=1}^{i} \in \phi(o)\} = \varphi_{\text{sum}}(p, v).$$

Thus, we have prove that Eq. (D.2) holds for $v = v_{r,i}$, and hence the probability $p(x_{\pi_1}, \ldots, x_{\pi_i})$ can be computed by weighting the probability of PC units $\varphi_{\text{sum}}(p, v_{r,i})$ (line 8 in Algorithm 6) with the corresponding top-down probabilities (line 9 in Algorithm 6).

Efficiency of following the optimal variable order We proceed to show that when using the optimal variable order π^*, Algorithm 6 evaluates no more than $\mathcal{O}(\log(D)\cdot|p|)$ PC units.

According to the previous paragraphs, whenever Algorithm 6 evaluates a PC unit n w.r.t. vtree node v, it will evaluate all PC units in $\varphi(p, v)$. Therefore, we instead count the total number of vtree nodes need to be evaluated by Algorithm 6. Since the PC is assumed to be balanced Definition 9, for every v, we have $\varphi(p, v) = \mathcal{O}(|p|/D)$. Therefore, we only need to show that Algorithm 6 evaluates $\mathcal{O}(D \cdot \log(D))$ vtree nodes in total.

We start with the base case, which is PCs correspond to a single vtree leaf node v. In this case, $F_{\pi^*}(\boldsymbol{x})$ boils down to computing a single marginal probability $p(x_{\pi_1^*})$, which needs to evaluate PC units $\varphi(p, v)$ once.

Define $f(x)$ as the number of vtree nodes that need to be evaluated given a PC that corresponds to a vtree node with x descendent leaf nodes. From the base case, we know that $f(1)=1$.

Next, consider the inductive case where v is an inner node that has x descendent leaf nodes. Define the left and right child node of v as c_1 and c_2, respectively. Let c_1 and c_2 have y and z descendent leaf nodes, respectively. We want to compute $F_{\pi^*}(\boldsymbol{x})$, which can be

[1] This is because the scope of these PC units does not contain any of the variables in $\{X_{\pi_j}\}_{j=1}^{i}$.

broken down into computing following two sets of marginals:

$$\text{Set 1: } \left\{p(x_{\pi_1^*}, \cdots, x_{\pi_i^*})\right\}_{i=1}^{y}, \qquad \text{Set 2: } \left\{p(x_{\pi_1^*}, \cdots, x_{\pi_i^*})\right\}_{i=y+1}^{y+z}.$$

Since π^* follows the in-order traverse of v, to compute the first term, we only need to evaluate c_1 and its descendents, that is, we need to evaluate $f(y)$ vtree nodes. This is because the marginal probabilities in set 1 are only defined on variables in $\phi(c_1)$. To compute the second term, in addition to evaluating PC units corresponding to c_2 (that is $f(z)$ vtree nodes in total),[2] we also need to re-evaluate the PC units $\varphi(p, v)$ every time, which means we need to evaluate z more vtree nodes. In summary, we need to evaluate

$$f(x) = f(y) + f(z) + z \quad (y \geq z, y + z = x)$$

vtree nodes.

To complete the proof, we upper bound the number of vtree nodes that need to be evaluated. Define $g(\cdot)$ as follows:

$$g(x) = \max_{y \in \{1, \ldots, \lfloor \frac{x}{2} \rfloor\}} y + g(y) + g(x - y).$$

It is not hard to verify that $\forall x \in \mathbb{Z}, g(x) \geq f(x)$. Next, we prove that

$$\forall x \in \mathbb{Z}\,(x \geq 2),\ g(x) \leq 3x \log x.$$

First, we can directly verify that $g(2) \leq 3 \cdot 2 \log_2 2 \approx 4.1$. Next, for $x \geq 3$,

$$g(x) = \max_{y \in \{1, \ldots, \lfloor \frac{x}{2} \rfloor\}} y + g(y) + g(x - y),$$

[2] As justified in the second part of this proof, all probabilities of PC units that conform to descendents of c_1 will be unchanged when computing the marginals in set 2. Hence, we only need to cache these probabilities.

$$
\begin{aligned}
&\leq \max_{y\in\{1,\dots,\lfloor\frac{x}{2}\rfloor\}} \underbrace{y + 3y\log y + 3(x-y)\log(x-y)}_{h(y)},\\
&\overset{(a)}{\leq} \max\left(1 + 3(x-1)\log(x-1), \left\lfloor\frac{x}{2}\right\rfloor + 3\left\lfloor\frac{x}{2}\right\rfloor\log\left\lfloor\frac{x}{2}\right\rfloor + 3\left(x - \left\lfloor\frac{x}{2}\right\rfloor\right)\log\left(x - \left\lfloor\frac{x}{2}\right\rfloor\right)\right),\\
&\leq \max\left(1 + 3(x-1)\log(x-1), \left\lfloor\frac{x}{2}\right\rfloor + 3(x+1)\log\frac{x+1}{2}\right),\\
&\leq 3x\log x,
\end{aligned}
$$

where (a) holds since according to its derivative, $h(y)$ obtains its maximum value at either $y = 1$ or $y = \lfloor\frac{x}{2}\rfloor$.

For a structured-decomposable PC with D variables, $g(D) \leq 3D\log D$ vtree nodes need to be evaluated. Since each vtree node corresponds to $\mathcal{O}(\frac{|p|}{D})$ PC units, we need to evaluate $\mathcal{O}(\log(D)\cdot|p|)$ PC units to compute $F_{\pi^*}(\boldsymbol{x})$.

HCLTs, EiNets, and RAT-SPNs are Balanced

Consider the compilation from a PGM to an HCLT (Section 3.1). We first note that each PGM node g uniquely corresponds to a variable scope ϕ of the PC. That is, all PC units correspond to g have the same variable scope. Please first refer to Section D.2.2 for details on how to generate a HCLT given its PGM representation.

In the main loop of Algorithm 24 (lines 5-10), for each PGM node g such that $\text{var}(g) \in \mathbf{Z}$, the number of computed PC units are the same (M product units compiled in line 9 and M sum units compiled in line 10). Therefore, for any variable scopes ϕ_1 and ϕ_2 possessed by some PC units, we have $|\text{nodes}(p, \phi(m))| \approx |\text{nodes}(p, \phi(n))|$. Since there are in total $\Theta(D)$ different variable scopes in p, we have: for any scope ϕ' exists in an HCLT p, $\text{nodes}(p, \phi') = \mathcal{O}(|p|/D)$.

EiNets and RAT-SPNs are also balanced since they also have an equivalent PGM representation of their PCs. The main difference between these models and HCLTs is the different variable splitting strategy in the product units.

D.2.2 Methods and Experiment Details

Learning HCLTs

Computing Mutual Information As mentioned in the main text, computing the pairwise mutual information between variables $\mathbf{X}$ is the first step to compute the Chow-Liu Tree. Since we are dealing with categorical data (e.g., 0-255 for pixels), we compute mutual information by following its definition:

$$I(X;Y) = \sum_{i=1}^{C_X}\sum_{j=1}^{C_Y} P(X=i, Y=j)\log_2 \frac{P(X=i,Y=j)}{P(X=i)P(Y=j)},$$

where C_X and C_Y are the number of categories for variables X and Y, respectively. To lower the computation cost, for image data, we truncate the data by only using 3 most-significant bits. That is, we treat the variables as categorical variables with $2^3 = 8$ categories during the construction of the CLT. Note that we use the full data when constructing/learning the PC.

Training pipeline We adopt two types of EM updates — mini-batch and full-batch. In mini-batch EM, parameters are updated according to a step size η: $\boldsymbol{\theta}^{(k+1)} \leftarrow (1-\eta)\boldsymbol{\theta}^{(k)} + \eta\boldsymbol{\theta}^{(\text{new})}$, where $\boldsymbol{\theta}^{(\text{new})}$ is the EM target computed with a batch of samples; full-batch EM updates the parameters by the EM target computed using the whole dataset. In this paper, HCLTs are trained by first running mini-batch EM with batch size 1024 and η changing linearly from 0.1 to 0.05; full-batch EM is then used to finetune the parameters.

Generating PCs Following the HCLT Structure

After generating the PGM representation of a HCLT model, we are now left with the final step of compiling the PGM representation of the model into an equivalent PC. Recall that we define the latent variables $\{Z_i\}_{i=1}^4$ as categorical variables with M categories, where M is a hyperparameter. As demonstrated in Algorithm 24, we incrementally compile every PGM node into an equivalent PC unit though a bottom-up traverse (line 5) of the PGM. Specifically,

Algorithm 24 Compile the PGM representation of a HCLT into an equivalent PC

1: **Input:** A PGM representation of a HCLT $\mathcal{G}$ (e.g., Fig. 3.1(c)); hyperparameter M
2: **Output:** A smooth and structured-decomposable PC p equivalent to $\mathcal{G}$
3: **Initialize:** cache $\leftarrow$ dict() a dictionary storing intermediate PC units
4: **Sub-routines:** $\mathbf{PC\_leaf}(X_i)$ returns a PC input unit of variable X_i; $\mathbf{PC\_prod}(\{n_i\}_{i=1}^m)$ (resp. $\mathbf{PC\_sum}(\{n_i\}_{i=1}^m)$) returns a product (resp. sum) unit over child nodes $\{n_i\}_{i=1}^m$.
5: **foreach** node g traversed in postorder (bottom-up) of $\mathcal{G}$ **do**
6: **if** $\text{var}(g) \in \mathbf{X}$ **then** $\text{cache}[g] \leftarrow \big[\mathbf{PC\_leaf}\big(\text{var}(g)\big)$ for $i = 1 : M\big]$
7: **else** # That is, $\text{var}(g) \in \mathbf{Z}$
8: $\text{chs\_cache} \leftarrow \big[\text{cache}[c]$ for c in $\text{children}(g)\big]$ # $\text{children}(g)$ is the set of children of g
9: $\text{prod\_nodes} \leftarrow \big[\mathbf{PC\_prod}\big(\big[\text{nodes}[i]$ for nodes in $\text{chs\_cache}\big]\big)$ for $i = 1 : M\big]$
10: $\text{cache}[g] \leftarrow \big[\mathbf{PC\_sum}\big(\text{prod\_nodes}\big)$ for $i = 1 : M\big]$
11: **return** $\text{cache}[\text{root}(\mathcal{G})][0]$

leaf PGM nodes corresponding to observed variables X_i are compiled into PC input units of X_i (line 6), and inner PGM nodes corresponding to latent variables are compiled by taking products and sums (implemented by product and sum units) of its child nodes' PC units (lines 8-10). Leaf units generated by $\mathbf{PC\_leaf}(X)$ can be any simple univariate distribution of X. We used categorical leaf units in our HCLT experiments. Fig. 3.1(d) demonstrates the result PC after running Algorithm 24 with the PGM in Fig. 3.1(c) and $M = 2$.

Implementation Details of the PC Learning Algorithm

We adopted the EM parameter learning algorithm introduced in [18], which computes the EM update targets using *expected flows*. Following [97], we use a hybrid EM algorithm, which uses mini-batch EM updates to initiate the training process, and switch to full-batch EM updates afterwards.

• Mini-batch EM: denote $\boldsymbol{\theta}^{(\text{EM})}$ as the EM update target computed with a mini-batch of samples. An update with step-size η is: $\boldsymbol{\theta}^{(k+1)} \leftarrow (1-\eta)\boldsymbol{\theta}^{(k)} + \eta\boldsymbol{\theta}^{(\text{EM})}$.

• Full-batch EM: denote $\boldsymbol{\theta}^{(\text{EM})}$ as the EM update target computed with the whole dataset. Full-batch EM updates the parameters with $\boldsymbol{\theta}^{(\text{EM})}$ at each iteration.

In our experiments, we trained the HCLTs with 100 mini-batch EM epochs and 20 full-batch EM epochs. During mini-batch EM updates, η was annealed linearly from 0.15 to 0.05.

Details of the Compression/Decompression Experiment

Hardware specifications All experiments are performed on a server with 72 CPUs, 512G Memory, and 2 TITAN RTX GPUs. In all experiments, we only use a single GPU on the server.

IDF We ran all experiments with the code in the GitHub repo provided by the authors. We adopted an IDF model with the following hyperparameters: 8 flow layers per level; 2 levels; densenets with depth 6 and 512 channels; base learning rate 0.001; learning rate decay 0.999. The algorithm adopts an CPU-based entropy coder rANS. For (de)compression, we used the following script: https://github.com/jornpeters/integer_discrete_flows/blob/master/experiment_coding.py.

BitSwap We trained all models using the following author-provided script: https://github.com/fhkingma/bitswap/blob/master/model/mnist_train.py. The algorithm adopts an CPU-based entropy coder rANS. And we used the following code for (de)compression: https://github.com/fhkingma/bitswap/blob/master/mnist_compress.py.

BB-ANS All experiments were performed using the following official code: https://github.com/bits-back/bits-back.

Details of the PC+IDF Model

The adopted IDF architecture follows the original paper [64]. For the PCs, we adopted EiNets [132] with hyperparameters $K = 12$ and $R = 4$. Instead of using random binary trees to define the model architecture, we used binary trees where "closer" latent variables in $\boldsymbol{z}$ will be put closer in the binary tree.

Parameter learning was performed by the following steps. First, compute the average log-likelihood over a mini-batch of samples. The negative average log-likelihood is the loss we use. Second, compute the gradients w.r.t. all model parameters by backpropagating the loss.

Finally, update the IDF and PCs using the gradients individually: for IDF, following [64], the Adamax optimizer was used; for PCs, following [132], we use the gradients to compute the EM target of the parameters and performed mini-batch EM updates.

D.3 Offline Reinforcement Learning

D.3.1 Proof of Theorem 14

To improve the clarity of the proof, we first simplify the notations in Theorem 14: define $\mathbf{X}$ as the boolean action variables $a_t := \{a_t^i\}_{i=1}^k$, and Y as the variable V_t, which is a categorical variable with two categories 0 and 1. We can equivalently interpret Y as a boolean variable where the category 0 corresponds to `F` and 1 corresponds to `T`. Dropping the condition on s_t everywhere for notation simplicity, we have converted the problem into the following one:

Assume boolean variables $\mathbf{X} := \{X_i\}_{i=1}^k$ and Y follow a Naive Bayes distribution: $p(\boldsymbol{x}, y) := p(y) \cdot \prod_i p(x_i|y)$. We want to prove that computing $p(\boldsymbol{x}|\mathbb{E}[y] \geq v)$, which is defined as follows, is NP-hard.

$$p(\boldsymbol{x}|\mathbb{E}[y] \geq v) := \frac{1}{Z} \begin{cases} p(\boldsymbol{x}) & \text{if } \mathbb{E}_{y \sim p(\cdot|\boldsymbol{x})}[y] \geq v, \\ 0 & \text{otherwise.} \end{cases} \tag{D.3}$$

By the definition of Y as a categorical variable with two categories 0 and 1, we have

$$\mathbb{E}_{y \sim p(\cdot|\boldsymbol{x})}[y] = p(y = \texttt{T}|\boldsymbol{x}) \cdot 1 + p(y = \texttt{F}|\boldsymbol{x}) \cdot 0 = p(y = \texttt{T}|\boldsymbol{x}).$$

Therefore, we can rewrite $p(\boldsymbol{x}|\mathbb{E}[y] \geq v)$ as

$$p(\boldsymbol{x}|\mathbb{E}[y] \geq v) := \frac{1}{Z} \cdot p(\boldsymbol{x}) \cdot \mathbb{1}[p(y = \texttt{T}|\boldsymbol{x}) \geq v],$$

where $\mathbb{1}[\cdot]$ is the indicator function. In the following, we show that computing the normalizing constant $Z := \sum_{\boldsymbol{x}} p(\boldsymbol{x}) \cdot \mathbb{1}[p(y = \texttt{T}|\boldsymbol{x}) \geq v]$ is NP-hard by reduction from the number partition problem, which is a known NP-hard problem. Specifically, for a set of k numbers $n_1, \ldots, n_k$ ($\forall i, n_i \in \mathbb{Z}^+$), the number partition problem aims to decide whether there exists a subset

$S \subseteq [k]$ (define $[k] := \{1, \ldots, k\}$) that partition the numbers into two sets with equal sums: $\sum_{i \in S} n_i = \sum_{j \notin S} n_j$.

For every number partition problem $\{n_i\}_{i=1}^k$, we define a corresponding Naive Bayes distribution $p(\boldsymbol{x}, y)$ with the following parameterization: $p(y = \mathtt{T}) = 0.5$ and[3]

$$\forall i \in [k],\ p(x_i = \mathtt{T} | y = \mathtt{T}) = \frac{1 - e^{-n_i}}{e^{n_i} - e^{-n_i}} \ \text{ and } \ p(x_i = \mathtt{T} | y = \mathtt{F}) = e^{n_i} \cdot \frac{1 - e^{-n_i}}{e^{n_i} - e^{-n_i}}.$$

It is easy to verify that the above definitions lead to a valid Naive Bayes distribution. Further, we have

$$\forall i \in [k],\ \log \frac{p(x_i = \mathtt{T} | y = \mathtt{T})}{p(x_i = \mathtt{T} | y = \mathtt{F})} = n_i \ \text{ and } \ \log \frac{p(x_i = \mathtt{F} | y = \mathtt{T})}{p(x_i = \mathtt{F} | y = \mathtt{F})} = -n_i. \tag{D.4}$$

We pair every partition S in the number partition problem with an instance $\boldsymbol{x}$ such that $\forall i$, $x_i = \mathtt{T}$ if $i \in S$ and $x_i = \mathtt{F}$ otherwise. Choose $v = 2/3$, the normalizing constant Z can be written as

$$Z = \sum_{\boldsymbol{x} \in \mathsf{val}(\mathbf{X})} p(\boldsymbol{x}) \cdot \mathbb{1}\big[p(y = \mathtt{T} | \boldsymbol{x}) \geq 2/3\big]. \tag{D.5}$$

Recall the one-to-one correspondence between S and $\boldsymbol{x}$, we rewrite $p(y = \mathtt{T} | \boldsymbol{x})$ with the Bayes formula:

$$\begin{aligned} p(y = \mathtt{T} | \boldsymbol{x}) &= \frac{p(y = \mathtt{T}) \prod_i p(x_i | y = \mathtt{T})}{p(y = \mathtt{T}) \prod_i p(x_i | y = \mathtt{T}) + p(y = \mathtt{F}) \prod_i p(x_i | y = \mathtt{F})}, \\ &= \frac{1}{1 + e^{-\sum_i \log \frac{p(x_i | y = \mathtt{T})}{p(x_i | y = \mathtt{F})}}}, \\ &= \frac{1}{1 + e^{-(\sum_{i \in S} n_i - \sum_{j \notin S} n_j)}}, \end{aligned}$$

[3]Note that we assume the naive Bayes model is parameterized using log probabilities.

where the last equation follows from Eq. (D.4). After some simplifications, we have

$$\mathbb{1}\big[p(y=\mathtt{T}|\boldsymbol{x})\geq 2/3\big]=\mathbb{1}\big[\sum_{i\in S}n_i-\sum_{j\notin S}n_j\geq 1\big].$$

Plug back to Eq. (D.5), we have

$$\begin{aligned}Z&=\sum_{S\subseteq[k]}p(\boldsymbol{x})\cdot\mathbb{1}\big[\sum_{i\in S}n_i-\sum_{j\notin S}n_j\geq 1\big],\\&=\frac{1}{2}\sum_{S\subseteq[k]}p(\boldsymbol{x})\cdot\mathbb{1}\big[\sum_{i\in S}n_i-\sum_{j\notin S}n_j\neq 0\big],\end{aligned}$$

where the last equation follows from the fact that (i) if $\boldsymbol{x}$ satisfy $\sum_{i\in S}n_i-\sum_{j\notin S}n_j\geq 1$ then $\bar{\boldsymbol{x}}$ has $\sum_{i\in S}n_i-\sum_{j\notin S}n_j\leq -1$ and vise versa, and (ii) $\sum_{i\in S}n_i-\sum_{j\notin S}n_j$ must be an integer.

Note that for every solution S to the number partition problem, $\sum_{i\in S}n_i-\sum_{j\notin S}n_j=0$ holds. Therefore, there exists a solution to the defined number partition problem if $Z<\frac{1}{2}$. □

D.3.2 Algorithm Details of Trifle

This section provides a detailed description of the algorithmic procedure of Trifle with single-/multi-step value estimates.

Adopted PC Structure And Parameter Learning Algorithm

For all tasks/offline datasets, we adopt the Hidden Chow-Liu Tree (HCLT) PC structure proposed by [97] as it has been shown to perform well across different data types.

Following the definition in Eq. (2.1), a PC takes as input a sample $\boldsymbol{x}$ and outputs the corresponding probability $p_n(\boldsymbol{x})$. Given a dataset $\mathcal{D}$, the PC optimizer takes the PC parameters (consisting of sum edge parameters and input node/distribution parameters) as input and aims to maximize the MLE objective $\sum_{\boldsymbol{x}\in\mathcal{D}}\log p_n(x)$. Since PCs can be deemed as latent variable models with hierarchically nested latent space [131], the Expectation-

Maximization (EM) algorithm is usually the default choice for PC parameter learning. We adopt the full-batch EM algorithm proposed in [136].

Before tuning the parameters with EM, we adopt the latent variable distillation (LVD) technique proposed in [99] to initialize the PC parameters. Specifically, the neural embeddings used for LVD are acquired by a BERT-like Transformer [39] trained with the Masked Language Model task. To acquire the embeddings of a subset of variables ϕ, we feed the Transformer with all other variables and concatenate the last Transformer layer's output for the variables ϕ. Please refer to the original paper for more details.

We use the same quantile dataset discretized from the original Gym-MuJoCo dataset as done by TT [68], where each raw continuous variable is divided into 100 categoricals, and each categorical represents an equal amount of probability mass under the empirical data distribution.

Trifle with Single-/Multi-Step Value Estimates

Similar to other RvS algorithms, Trifle first trains sequence models given truncated trajectories $\{(s_t, a_t, r_t, \mathrm{RTG}_t)\}_t$. Specifically, we fit two sequence models: an autoregressive Transformer following prior work [68] as well as a PC, where the training details are introduced in Section D.3.2.

During the evaluation phase, at time step t, Trifle is tasked to generate a_t given $s_{\leq t}$ and other relevant information (such as rewards collected in past steps). As introduced in Section 5.1.3, Trifle generally works in two phases: rejection sampling for action generation and beam search for action selection. The main algorithm is illustrated in Algorithm 25, where we take the current state s_t as well as the past trajectory $\tau_{<t}$ as input, utilize the specified value estimate f_v as a heuristic to guide beam search, and output the best trajectory. Note that f_v is a subroutine of our algorithm that uses the trained sequence models to compute certain quantities, which will be detailed in subsequent parts. After that, we extract the current action a_t from the output trajectory to execute in the environment.

At the first step of the beam search, we perform rejection sampling to obtain a candidate action set $\mathbf{a_t}$ (line 4 of Algorithm 25). The concrete rejection sampling procedure for s-Trifle is detailed in Algorithm 26. The major modification of m-Trifle compared to s-Trifle is the adoption of a multi-step value estimate instead of the single-step value estimate, which is also shown in Algorithm 27. Specifically, Algorithm 27 is used to replace the value function f_v shown in Algorithm 25.

Algorithm 25 Trifle with Beam Search

1: **Input:** past trajectory $\tau_{<t}$, current state s_t, beam width N, beam horizon H, scaling ratio λ, sequence model $\mathcal{M}$, value function f_v ▷ $f_v = \mathbb{E}[V_t]$ for s-Trifle and $\mathbb{E}[V_t^m]$ for m-Trifle
2: **Output:** The best action a_t
3: Let $\mathbf{x_t} \leftarrow \texttt{concat}(\tau_{<t}, s_t).\texttt{reshape}(1, -1).\texttt{repeat}(N, \texttt{dim} = 0)$ ▷ Batchify the input trajectory
4: Perform rejection sampling to obtain $\mathbf{a_t}$ using Algorithm 26 ▷ cf. Algorithm 26
5: Initialize $X_0 = \texttt{concat}(\mathbf{x_t}, \mathbf{a_t})$
6: **foreach** $t = 1, ..., H$
7: $\quad X_{t-1} \leftarrow X_{t-1}.\texttt{repeat}(\lambda, \texttt{dim} = 0)$ ▷ Scale the number of trajectories from N to λN
8: $\quad \mathcal{C}_t \leftarrow \{\texttt{concat}(\mathrm{x_{t-1}}, x) \mid \forall \mathrm{x_{t-1}} \in X_{t-1}, \text{sample } x \sim p_{\mathcal{M}}(\cdot \mid \mathrm{x_{t-1}})\}$ ▷ Candidate next-token prediction
9: $\quad X_t \leftarrow \texttt{topk}_{X \in \mathcal{C}_t}\ (f_v(X), \mathtt{k} = N)$ ▷ keep N most rewarding trajectories
10: $X_m \leftarrow \texttt{argmax}_{X \in X_H}\ \ f_v(X)$
11: **return** a_t in X_m

Algorithm 26 Rejection Sampling with Single-step Value Estimate

1: **Input:** past trajectory $\tau_{<t}$, current state s_t, dimension of action k, rejection rate $\delta > 0$
2: **Output:** The sampled action $a_t^{1:k}$
3: Let $x_t \leftarrow \texttt{concat}(\tau_{<t}, s_t)$
4: **for** $i = 1, ..., k$ **do**
5: $\quad$ Compute $p_{\mathrm{GPT}}(a_t^i \mid x_t, a_t^{<i})$ Note that $a_t^{<1} = \varnothing$.
6: $\quad$ Compute $p_{\mathrm{TPM}}(V_t \mid x_t, a_t^{<i}) = \sum_{a_t^{i:k}} p_{\mathrm{TPM}}(V_t, a_t^{i:k} \mid x_t, a_t^{1:k})$ ▷ The marginal can be efficiently computed by PC in linear time.
7: $\quad$ Compute $v_\delta = \texttt{max}_v\{v \in \texttt{val}(V_t) \mid p_{\mathrm{TPM}}(V_t \geq v \mid x_t, a_t^{<i}) \geq 1 - \delta\}$, for each $a_t^i \in \texttt{val}(A_t^i)$
8: $\quad$ Compute $\tilde{p}(a_t^i \mid x_t, a_t^{<i}; v_\delta) = \frac{1}{Z} \cdot p_{\mathrm{GPT}}(a_t^i \mid x_t, a_t^{<i}) \cdot p_{\mathrm{TPM}}(V_t \geq v_\delta \mid x_t, a_t^{\leq i})$ ▷ Apply Eq. (5.10)
9: $\quad$ Sample $a_t^i \sim \tilde{p}(a_t^i \mid x_t, a_t^{<i}; v_\delta)$
10: **return** $a_t^{1:k}$

Algorithm 27 Multi-step Value Estimate

Input: $\tau_{\leq t} = (s_0, a_0, ..., s_t, a_t)$, sequence model $\mathcal{M}$, terminal timestep $t' > t$, discount γ

Output: The multi-step value estimate $\mathbb{E}\big[V_t^{\mathrm{m}}\big]$

Sample future actions $a_{t+1}, ..., a_{t'}$ from $\mathcal{M}$

Compute $p_{\mathrm{TPM}}(r_h \mid \tau_{\leq t}, a_{t+1:h}) = \sum_{s_{t+1:h}} p_{\mathrm{TPM}}(r_h, s_{t+1:h} \mid \tau_{\leq t'})$ for $h \in [t+1, t']$ ▷ Marginalize over intermediate states $s_{t+1:h}$

Compute $p_{\mathrm{TPM}}(\mathrm{RTG}_{t'} \mid \tau_{\leq t}, a_{t+1:t'}) = \sum_{s_{t+1:t'}} p_{\mathrm{TPM}}(\mathrm{RTG}_{t'} \mid \tau_{\leq t'})$

Compute

$$\mathbb{E}\big[V_t^{\mathrm{m}}\big] = \sum_{h=t}^{t'} \gamma^{h-t} \mathbb{E}_{r_h \sim p_{\mathrm{TPM}}(\cdot \mid \tau_{\leq t}, a_{t+1:h})}\big[r_h\big] + \gamma^{t'+1-t} \mathbb{E}_{\mathrm{RTG}_{t'} \sim p_{\mathrm{TPM}}(\cdot \mid \tau_{\leq t}, a_{t+1:t'})}\big[\mathrm{RTG}_{t'}\big]$$

return $\mathbb{E}\big[V_t^{\mathrm{m}}\big]$

Computing Multi-Step Value Estimates

In this section, we present an efficient algorithm that computes Eq. (5.11). From the decomposition of Eq. (5.11), we can calculate $\mathbb{E}\left[V_t^{\mathrm{m}}\right]$ if we have the probabilities $p(r_\tau | s_t, a_{t:t'})(t \leq \tau < t')$ and $p(\mathrm{RTG}_{t'} | s_t, a_{t:t'})$. A simple approach would be to compute each of the $t'-t+1$ probabilities separately by computing marginal probabilities (recall that conditional probabilities are a quotient of the corresponding marginal probabilities). However, this approach has an undesired time complexity that scales linearly with respect to $t'-t+1$.

Following [96], we describe an algorithm that can compute all desired quantities using a single feedforward and a backward pass to the PC.

The forward pass. The forward pass is similar to the one described in Section 2.1. Specifically, we set the evidence as $s_t, a_{t:t'}$ and execute the forward pass.

The backward pass. The backward pass consists of two steps: (i) traverse all nodes parents before children to compute a statistic termed flow for every node n: $\texttt{flow}_n$; (ii) compute the target probabilities using the flow of all input nodes. Recall that we assume without loss of generality that PCs alternate between sum and product layers. We further assume that all parents of input nodes are product nodes. We define the flow of the root node as 1. The flow of other nodes is defined recursively as (define p_n as the forward probability of

node n):

$$\texttt{flow}_n := \begin{cases} \sum_{m \in \texttt{pa}(n)} (\theta_{m,n} \cdot p_n / p_m) \cdot \texttt{flow}_m & n \text{ is a product node,} \\ \sum_{m \in \texttt{pa}(n)} \texttt{flow}_m & n \text{ is a input or sum node,} \end{cases}$$

where $\texttt{pa}(n)$ is the set of parents of node n.

Next for every variable $X \in \{R_t, \ldots, R_{t'-1}, \mathrm{RTG}_{t'}\}$, we first collect all input nodes defined on X. Define the set of input nodes as S. We have that

$$p(x|s_t, a_{t:t'}) := \frac{1}{Z} \sum_{n \in S} \texttt{flow}_n \cdot f_n(x),$$

where f_n is defined in Eq. (2.1) and Z is a normalizing constant.

D.3.3 Inference-time Optimality Score

We define the inference-time optimality score as a proxy for inference-time optimality. This score is primarily defined over a state-action pair (s_t, a_t) at each inference step. In Fig. 5.8 (middle) and Fig. 5.8 (right), each sample point represents a trajectory, and the corresponding inference-time optimality score is defined over the entire trajectory by averaging the scores of all inference steps.

The specific steps for calculating the score for a given inference step t, given s_t and a policy $p(a_t \mid s_t)$, are as follows:

1. Given s_t, sample a_t from $p_{\mathrm{TT}}(a_t \mid s_t)$, $p_{\mathrm{DT}}(a_t \mid s_t)$, or $p_{\mathrm{Trifle}}(a_t \mid s_t)$.
2. Compute the state-conditioned value distribution $p^s(V_t \mid s_t)$.
3. Compute $R_t := \mathbb{E}_{V_t \sim p^a(\mathrm{RTG}_t | s_t, a_t)}[V_t]$, which is the corresponding estimated expected value.
4. Output the quantile value S_t of R_t in $p^s(V_t \mid s_t)$.

To approximate the distributions $p^s(V_t \mid s_t)$ and $p^a(V_t \mid s_t, a_t)$ (where $V_t = \text{RTG} * t$) in steps 2 and 3, we train two auxiliary GPT models using the offline dataset. For instance, to approximate $p^s(V_t \mid s_t)$, we train the model on sequences $(s * t - k, V_{t-k}, \ldots, s_t, V_t)$.

Intuitively, $p^s(V_t \mid s_t)$ approximates $p(V_t \mid s_t) := \sum_{a_t} p(V_t \mid s_t, a_t) \cdot p(a_t \mid s_t)$. Therefore, S_t indicates the percentile of the sampled action in terms of achieving a high expected return, relative to the entire action space.

D.3.4 Additional Experimental Details

Gym-MuJoCo

Sampling Details. We take the single-step value estimate by setting $V_t = \text{RTG}_\text{t}$ and sample a_t from Eq. (5.10). When training the GPT used for querying $p_{\text{GPT}}(a_t^i | s_t, a_t^{<i})$, we adopt the same model specification and training pipeline as TT or DT. When computing $p_{\text{TPM}}(V_t \geq v | s_t, a_t^{<i})$, we first use the learned PC to estimate $p(V_t|s_t)$ by marginalizing out intermediate actions $a_{t:t'}$ and select the ϵ-quantile value of $p(V_t|s_t)$ as our prediction threshold v for each inference step. Empirically we fixed ϵ for each environment and ϵ ranges from 0.1 to 0.3.

Beam Search Hyperparameters. The maximum beam width N and planning horizon H that Trifle uses across 9 MuJoCo tasks are 15 and 64, respectively.

Comparison with Value-Based Algorithms. To shed light on how Trifle compares to methods that directly optimize the Q values while filtering actions by conditioning on high returns (as done in RvS algorithms), we compare Trifle with Q-learning Decision Transformer [185], which incorporates a contrastive Q-learning regime into the RvS framework. As shown in the table below, Trifle outperforms QDT in all six adopted MuJoCo benchmarks:

Table D.5: Normalized Scores of QDT and Trifle on Gym-MuJoCo benchmarks

Dataset	Environment	Trifle	QDT
Medium	Halfcheetah	**49.5**$\pm$0.2	42.3$\pm$0.4
Med-Replay	Halfcheetah	**45.0**$\pm$0.3	35.6$\pm$0.5
Medium	Hopper	**67.1**$\pm$4.3	66.5$\pm$6.3
Med-Replay	Hopper	**97.8**$\pm$0.3	52.1$\pm$20.3
Medium	Walker2d	**83.1**$\pm$0.8	67.1$\pm$3.2
Med-Replay	Walker2d	**88.3**$\pm$3.8	58.2$\pm$5.1

Stochastic Taxi Environment

Hyperparameters. Except for s-Trifle, the sequence length K modeled by TT, DT, and m-Trifle is all equal to 7. The inference algorithm of TT follows that of the MuJoCo experiment and DT follows its implementation in the Atati benchmark. Notably, during evaluation, we condition the pretrained DT on 6 different RTGs ranging from -100 to -350 and choose the best policy resulting from RTG=-300 to report in Fig. 5.9c. Beam width $N = 8$ and planning horizon $H = 3$ hold for TT and m-Trifle.

Additional Results on the Taxi benchmark. Besides the episode return, we adopt two metrics to better evaluate the adopted methods: (i) #penalty: the average number of executing illegal actions within an episode; (ii) $P(\texttt{failure})$: the probability of failing to transport the passenger to the destination within 300 steps.

Table D.6: Results on the stochastic Taxi environment. All the reported numbers are averaged over 1000 trials.

Methods	Episode return	# penalty	P(failure)
s-Trifle	-99	0.14	0.11
m-Trifle	**-57**	0.38	**0.02**
TT	-182	2.57	0.34
DT	-388	14.2	0.66
DoC	-146	**0**	0.28
dataset	-128	2.41	0

Ablation Study Regarding Action Filtering. In an attempt to justify the effectiveness/necessity of exact inference, we compare Trifle with value-based action filtering/value estimation in the following:

To begin with, we implemented the traditional Policy Evaluation algorithm on the Taxi offline dataset described in Section 5.3.5 of the paper. The policy evaluation algorithm is based on the Bellman update:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\big[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))\big]$$

Then we use the obtained Q function, denoted Q_{taxi}, to perform the following ablation studies. We still choose TT as our base RvS model. Recall that given s_t, TT first samples a_t from its learned prior policy $p_{TT}(a_t|s_t)$, which are subsequently fed to a beam search procedure that uses the learned value function $p_{TT}(V_t|s_t, a_t)$ to select the best action. Therefore, we consider ablations on two key components of TT: (i) the prior policy $p_{TT}(V_t|s_t, a_t)$ used to sample actions, and (ii) the value function $p_{TT}(V_t|s_t, a_t)$ used to evaluate and select actions.

1. **TT + Q_{taxi} action filtering**: weigh the prior policy $p_{TT}(a_t|s_t)$ with each action's exponentiated Q_{taxi} value (i.e., $exp(Q_{\text{taxi}}(s_t, a_t))$), but still adopt TT's value estimation.

In other words, in this experiment, we only use Q_{taxi} to improve the sampling quality as s-Trifle does.

2. **TT + Q_{taxi} value estimation**: replace $p_{TT}(V_t|s_t, a_t)$ with $Q_{\text{taxi}}(s_t, a_t)$ for action evaluation and selection, but still use TT's prior policy $p_{TT}(a_t|s_t)$.

3. **TT + full Q_{taxi}**: simultaneously use $exp(Q_{\text{taxi}}(s_t, a_t))$ for action filtering and $Q_{\text{taxi}}(s_t, a_t)$ for action evaluation.

We present the results of these ablation studies as follows:

Method	**Score**
TT	-182
TT + Q_{taxi} action filtering	-157
TT + Q_{taxi} value estimation	-147
TT + full Q_{taxi}	-138
m-Trifle	-58
s-Trifle	-99

From these results, we draw the following conclusions:

- m-Trifle and s-Trifle achieve the best performance.

- The rank of scores: TT + full Q_{taxi} > TT + Q_{taxi} value estimation > TT + Q_{taxi} action filtering > TT suggests that using Q_{taxi} for both action filtering and value estimation is beneficial; combining the two leads to the best performance.

- Specifically, the fact that s-Trifle outperforms the Q_{taxi} based action filtering demonstrates that our filtration with exact inference is much more effective. The superior performance of m-Trifle also provides strong evidence that explicit marginalization over future states leads to better value estimation.

D.3.5 Additional Experiments

Ablation Studies on Rejection Sampling and Beam Search

The key insight of Trifle to solve challenges elaborated in Scenario #1 is to utilize tractable probabilistic models to better approximate action samples from the desired distribution $p(a_t|s_{0:t}, \mathbb{E}[V_t] \geq v)$. We highlight that the most crucial design choice of our method for this goal is that: Trifle can effectively bias the per-action-dimension generation process of any base policy towards high expected returns, which is achieved by adding per-dimension correction terms $p_{TPM}(V_t \geq v|s_t, a_t^{\leq i})$ (Eq. (2) in the paper) to the base policy.

While the rejection sampling method can help us obtain more unbiased action samples through a post value(expected return)-estimation session, we only implement this component for TT-based Trifle (not for DT-based Trifle) for fair comparison, as the DT baseline doesn't perform explicit value estimation or adopt any rejection sampling methods. Therefore, the success of DT-based Trifle strongly justifies the effectiveness of the TPM components. Moreover, the beam search algorithm also comes from TT. Although it is a more effective way to do rejection sampling, it is not the necessary component of Trifle, either.

Table D.7: Ablations over Beam Search Hyperparameters on Halfcheetah Med-Replay. (a) With $H = 1$, the beam search degrades to naive rejection sampling (b) With $W = 1$, the algorithm doesn't perform rejection sampling. It samples a single action and applies it to the environment directly.

Table D.8: Varying Planning Horizon

Horizon H	Width W	TT	TT-based Trifle
5	32	41.9±2.5	**45.0±0.3**
4	32	40.1±2.0	**43.1±1.0**
3	32	41.6±1.3	**42.6±1.6**
2	32	39.7±2.5	**42.8±0.5**
1 (w/ naive rej sampling)	32	33.6±3.0	**39.6±0.7**

Table D.9: Varying Beam Width

Horizon H	Width W	TT	TT-based Trifle
5	32	41.9±2.5	**45.0±0.3**
5	16	42.5±1.9	**42.6±1.6**
5	8	42.9±0.4	**43.5±0.3**
5	4	38.7±0.3	**43.4±0.3**
5	1 (w/o rej sampling)	31.2±3.4	**36.7±1.8**

For TT-based Trifle, we adopted the same beam search hyperparameters as reported in the TT paper. We conduct ablation studies on beam search hyperparameters in Table D.7 to

investigate the effectiveness of Trifle's each component. From Table D.7, we can observe that:

- Trifle consistently outperforms TT across all beam search hyperparameters and is more robust to variations of both planning horizon and beam width.
- (a) Trifle w/ naive rejection sampling » TT w/ naive rejection sampling (b) Trifle w/o rejection sampling » TT w/o rejection sampling. In both cases, Trifle can positively guide action generation.
- Trifle w/ beam search > Trifle w/ naive rejection sampling > Trifle w/o rejection sampling » TT w/ naive rejection sampling. Although other design choices like rejection sampling/beam search help to better approximate samples from the high-expected-return-conditioned action distribution, the per-dimension correction terms computed by Trifle play a very significant role.

Computational Efficiency Analysis

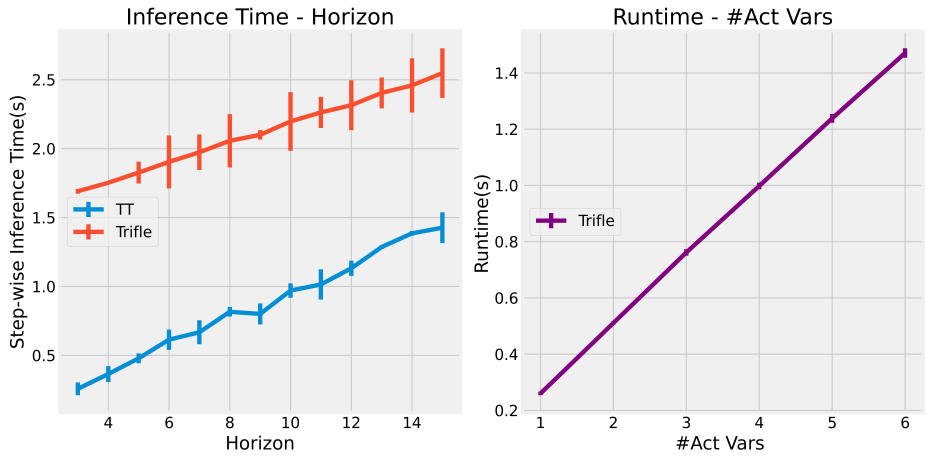
Since TPM-related computation consistently requires 1.45s computation time across different horizons, the relative slowdown of Trifle is diminishing as we increase the beam horizon. Specifically, in the Gym-Mujuco benchmark, the time consumption for one step (i.e., one interaction with the environment) of TT and Trifle with different beam horizons are listed here (Fig. D.7 (left) also plots an inference-time scaling curve of Trifle vs TT with varying horizons):

Table D.10: The one-step inference runtime of the Gym-MuJuCo benchmark

Horizon	TT	Trifle
5	0.5s	1.5s
15	1.2s	1.8s

Moreover, Fig. D.7 (right) shows that Trifle's runtime (TPM-related) scales *linearly* w.r.t.

Figure D.7: Scaling Curves of Inference Time. (Fix beam width = 32)



the number of action variables, which indicates its efficiency for handling high-dimensional action spaces.

Trifle is also efficient in training. It only takes 30-60 minutes (20s per epoch, 100-200 epochs) to train a PC on one GPU for each Gym-Mujuco task (*Note that a single PC model can be used to answer all conditional queries required by Trifle*). In comparison, training the GPT model for TT takes approximately 6-12 hours (80 epochs).

Ablation Studies on the Adaptive Thresholding Mechanism

The adaptive thresholding mechanism is adopted when computing the term $p_{TPM}(V_t \geq v|s_t, a_t^{<i})$ of Equation (2), where $i \in \{1, \ldots, k\}$, k is the number of action variables and a_t^i is the ith variable of a_t. Instead of using a fixed threshold v, we choose v to be the ϵ-quantile value of the distribution $p_{TPM}(V_t|s_t, a_t^{<i})$ computed by the TPM, which leverage the TPM's ability to exactly compute marginals given **incomplete** actions (marginalizing out $a_t^{i:k}$). Specifically, we compute v using $v = max_r\{r \in \mathbb{R}|p_{TPM}(V_t \geq r|s_t, a_t^{<i}) \geq 1 - \epsilon\}$. Empirically we fixed ϵ for each Gym-MuJoCo environment and $\epsilon = 0.2$ or 0.25, which is selected by running grid search on $\epsilon \in [0.1, 0.25]$.

Table D.11: Comparison of Adaptive and Fixed Thresholding Mechanisms

Table D.12: Ablations over Adaptive Thresholding (Varying ϵ) on Halfcheetah Med-Replay

Method	Score
TT	41.9±2.5
TT-based Trifle ($\epsilon = 0.25$)	**45.0±0.3**
TT-based Trifle ($\epsilon = 0.2$)	44.2±0.4
TT-based Trifle ($\epsilon = 0.15$)	44.4±0.3
TT-based Trifle ($\epsilon = 0.1$)	42.6±1.6

Table D.13: Performance of Fixed Thresholding (Varying v)

v	Halfcheetah Med-Replay	Walker2d Med-Expert
adaptive	**45.0±0.3**	**109.3±0.1**
90	44.8±0.3	109.1±0.2
80	39.5±2.8	108.9±0.2
70	44.9±0.3	108.4±0.4
60	42.6±1.6	105.8±0.3
50	41.4±2.0	107.5±1.5
40	42.6±1.6	107.5±1.4
30	44.0±0.4	98.3±5.4

We report the performance of TT-based Trifle with variant ϵ vs TT on Halfcheetah Med-Replay benchmark in Table D.12. We can see that Trifle is robust to the hyperparameter ϵ and consistently outperforms the base policy TT.

We also conduct ablation studies comparing the performance of the adaptive thresholding mechanism with the fixed thresholding mechanism on two environments in Table D.13. Specifically, given that V_t is discretized to be a categorical variable with 100 categoricals (0-99), we fix v to be 90,80,70,60,50,40,30 respectively.

The table shows that the adaptive approach consistently outperforms the fixed value threshold in both environments. Additionally, the performance variation of fixing v is larger compared to fixing ϵ as different v can be optimal for different states.

Appendix E

Tractability Matters in Diffusion Models

E.1 Proof of the Theoretical Results

Proof of Proposition 3. Following [63, 163], the negative ELBO $\mathcal{L}$ can be decomposed as follows:

$$\begin{aligned}
\mathcal{L} &= \mathbb{E}_q \left[-\log p(\boldsymbol{x}_T) - \sum_{t=1}^{T} \log \frac{p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)}{q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})} \right], \\
&= \mathbb{E}_q \left[-\log p(\boldsymbol{x}_T) - \sum_{t=1}^{T} \log \frac{p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t) \cdot q(\boldsymbol{x}_{t-1})}{q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t) \cdot q(\boldsymbol{x}_t)} \right], \\
&= \mathbb{E}_q \left[-\log \frac{p(\boldsymbol{x}_T)}{q(\boldsymbol{x}_T)} - \sum_{t=1}^{T} \log \frac{p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)}{q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)} - \log p(\boldsymbol{x}_0) \right], \\
&= \mathrm{D}_{\mathrm{KL}}(q(\boldsymbol{x}_T) \parallel p(\boldsymbol{x}_T)) + \mathbb{E}_q \left[\sum_{t=1}^{T} \mathrm{D}_{\mathrm{KL}}(q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t) \parallel p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)) \right] + \mathrm{H}(\boldsymbol{x}_0). \qquad \text{(E.1)}
\end{aligned}$$

The first term equals 0 as we assume the noise distribution $p(\mathbf{X}_T)$ is consistent in the noising and the denoising processes. Given the independent denoising assumption, when the denoising distribution are optimal, we have

$$\forall t \in \{1, \ldots, T\},\ p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t) = \prod_i q(x_{t-1}^i|\boldsymbol{x}_t).$$

Plug in Eq. (E.1) and using the definition of total correlation, we have:

$$\begin{aligned}\mathcal{L} &= \mathrm{D}_{\mathrm{KL}}(q(\boldsymbol{x}_T) \parallel p(\boldsymbol{x}_T)) + \mathbb{E}_q\left[\sum_{t=1}^{T} \mathrm{D}_{\mathrm{KL}}(q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t) \parallel \prod_i q(x^i_{t-1}|\boldsymbol{x}_t))\right] + \mathrm{H}(\boldsymbol{x}_0) \\ &= \mathrm{D}_{\mathrm{KL}}(q(\boldsymbol{x}_T) \parallel p(\boldsymbol{x}_T)) + \sum_{t=1}^{T} \mathrm{D}_{\mathrm{TC}}(q(\mathbf{X}_{t-1}|\mathbf{X}_t)) + \mathrm{H}(p(\mathbf{X}_0)) \\ &\geq \mathrm{H}(p(\mathbf{X}_0)) + \sum_{t=1}^{T} \mathrm{D}_{\mathrm{TC}}(q(\mathbf{X}_{t-1}|\mathbf{X}_t)).\end{aligned}$$

□

Proof of Proposition 4. According to Pythagoras' triangle-inequality theorem, if $\hat{p}$ is the I-projection of p_{est} onto $\mathcal{P}^{p_{\mathrm{tar}}}_{\mathrm{mar}}$, and $\mathcal{P}^{p_{\mathrm{tar}}}_{\mathrm{mar}}$ is convex (this can be shown by applying the definition of a convex set), the following holds for any $p' \in \mathcal{P}^{p_{\mathrm{tar}}}_{\mathrm{mar}}$:

$$\mathrm{D}_{\mathrm{KL}}(p' \parallel p_{\mathrm{est}}) \geq \mathrm{D}_{\mathrm{KL}}(p' \parallel \hat{p}) + \mathrm{D}_{\mathrm{KL}}(\hat{p} \parallel p_{\mathrm{est}}). \tag{E.2}$$

Choosing $p' = p_{\mathrm{tar}}$, we have

$$\mathrm{D}_{\mathrm{KL}}(p_{\mathrm{tar}} \parallel \hat{p}) \leq \mathrm{D}_{\mathrm{KL}}(p_{\mathrm{tar}} \parallel p_{\mathrm{est}}) - \mathrm{D}_{\mathrm{KL}}(\hat{p} \parallel p_{\mathrm{est}}) < \mathrm{D}_{\mathrm{KL}}(p_{\mathrm{tar}} \parallel p_{\mathrm{est}}),$$

where the last inequality holds since $\mathrm{D}_{\mathrm{KL}}(\hat{p} \parallel p_{\mathrm{est}}) > 0$ if the set of univariate marginals of p_{est} and p_{tar} are different (as assumed in the proposition).

□

Proof of Proposition 5. Following the definition of $\hat{p}$, we write down the constrained optimization problem as follows

$$\begin{aligned}&\underset{p'}{\text{minimize}}\ \mathrm{D}_{\mathrm{KL}}(p' \parallel p_{\mathrm{est}}) \\ &\text{s.t. } \forall i \in \{1, \ldots, N\}, x_i \in \{1, \ldots, C\},\ \sum_{\boldsymbol{x}_{\setminus i}} p'(\boldsymbol{x}_{\setminus i}, x_i) = p_{\mathrm{tar}}(x_i).\end{aligned}$$

To incorporate the constraints, we use the method of Lagrange multipliers. The Lagrangian for this problem is

$$\mathcal{L}(p', \{\lambda_i\}_{i=1}^N) = \sum_{\boldsymbol{x}} p'(\boldsymbol{x}) \log \frac{p'(\boldsymbol{x})}{p_{\text{est}}(\boldsymbol{x})} + \sum_{i=1}^{N} \sum_{x_i=1}^{C} \lambda_i(x_i) \cdot \left(\sum_{\boldsymbol{x}_{\setminus i}} p'(\boldsymbol{x}_{\setminus i}, x_i) - p_{\text{tar}}(x_i) \right),$$

where the Lagrange multipliers $\{\lambda_i\}_{i=1}^N$ enforce the univariate marginal constraints.

To minimize the Lagrangian with respect to $p'(\boldsymbol{x})$, we take the partial derivative of $\mathcal{L}(p', \{\lambda_i\}_{i=1}^N)$ with respect to $p'(\boldsymbol{x})$ and set it to 0:

$$\frac{\partial \mathcal{L}(p', \{\lambda_i\}_{i=1}^N)}{\partial p'(\boldsymbol{x})} = \log \frac{p'(\boldsymbol{x})}{p_{\text{est}}(\boldsymbol{x})} + 1 + \sum_i \lambda_i(x_i) = 0.$$

Simplifying this equation gives

$$p'(\boldsymbol{x}) = p_{\text{est}}(\boldsymbol{x}) \cdot \exp\left(-1 - \sum_i \lambda_i(x_i) \right).$$

Defining $\sigma_i(x_i) := \exp(-\lambda_i(x_i) - 1/N)$ gives $p'(\boldsymbol{x}) = p_{\text{est}}(\boldsymbol{x}) \prod_i \sigma_i(x_i)$.

Existence of the solution follows from the fact that (i) the objective function is convex and bounded (since probability values are in $[0, 1]$), and (ii) the set of constraints is feasible (e.g., $p'(\boldsymbol{x}) = \prod_i p_{\text{tar}}(x_i)$ or $p'(\boldsymbol{x}) = p_{\text{tar}}(\boldsymbol{x})$).

□

Proof of Theorem 15. We show that for any $\mathbf{V}^*$ that minimizes the objective function $\mathcal{L}(\mathbf{V}; p_{\text{tar}}, p_{\text{est}})$, the corresponding p' defined by $p'(\boldsymbol{x}) = p_{\text{est}}(\boldsymbol{x}) \cdot \prod_i \exp(\mathbf{V}[i, x_i])$ belongs to the set $\mathcal{P}_{\text{mar}}^{p_{\text{tar}}}$. Specifically, for any $\mathbf{V}$ that minimizes the objective, the partial derivative of $\mathcal{L}(\mathbf{V}; p_{\text{tar}}, p_{\text{est}})$ with respect to any $\mathbf{V}[i, x_i]$ should be 0:

$$\frac{\partial \mathcal{L}(\mathbf{V}; p_{\text{tar}}, p_{\text{est}})}{\partial \mathbf{V}[i, x_i]} = \exp(\mathbf{V}[i, x_i]) \sum_{\boldsymbol{x}_{\setminus i}} p_{\text{est}}(\boldsymbol{x}_{\setminus i}, x_i) \prod_{j \neq i} \exp(\mathbf{V}[j, x_j]) - p_{\text{tar}}(x_i) = 0.$$

Plug in the definition of p', we have

$$0 = \sum_{\boldsymbol{x}_{\setminus i}} p'(\boldsymbol{x}_{\setminus i}, x_i) - p_{\text{tar}}(x_i) = p'(x_i) - p_{\text{tar}}(x_i). \tag{E.3}$$

Since Eq. (E.3) holds for all (i, x_i) pairs, we have that every minimizer of $\mathcal{L}(\mathbf{V}; p_{\text{tar}}, p_{\text{est}})$ corresponds to a distribution p' in $\mathcal{P}_{\text{mar}}^{p_{\text{tar}}}$. Since $\mathcal{L}(\mathbf{V}; p_{\text{tar}}, p_{\text{est}})$ is convex, we can also argue the converse: if a distribution p' with the above-defined form belongs to $\mathcal{P}_{\text{mar}}^{p_{\text{tar}}}$, then the corresponding $\mathbf{V}$ is a minimizer of $\mathcal{L}(\mathbf{V}; p_{\text{tar}}, p_{\text{est}})$.

According to Proposition 5, the solution to the following I-projection exists and its solution $\hat{p}$ has the same form as p'.

$$\hat{p} = \operatorname*{arg\,min}_{p' \in \mathcal{P}_{\text{mar}}^{p}} \mathrm{D}_{\text{KL}}(p' \parallel p_{\text{est}}).$$

Since $\hat{p}$ has the same form as p' (by Prop. 5) and belongs to $\mathcal{P}_{\text{mar}}^{p_{\text{tar}}}$, it is the a minimizer of $\mathcal{L}(\mathbf{V}; p_{\text{tar}}, p_{\text{est}})$. □

Proof of Proposition 6. The copula of p is shown to be invariant under rescalings of the form $q(\boldsymbol{x}) \propto p(\boldsymbol{x}) \cdot \prod_i \exp(\mathbf{V}[i, x_i])$ for any $\mathbf{V} \in \mathbb{R}^{N \times C}$ by using the parameterization of a discrete copula by conditional odds ratios (Definition 14). The scaling factors cancel in the ratios as shown, e.g. by [149, Theorem 12.3]. □

Proof of Proposition 7. We start by writing the probability $q(\boldsymbol{x}_t | \boldsymbol{x}_{t+1})$ using the Bayes' rule:

$$\begin{aligned} q(\boldsymbol{x}_t | \boldsymbol{x}_{t+1}) &= q(\boldsymbol{x}_{t+1} | \boldsymbol{x}_t) \cdot \frac{q(\boldsymbol{x}_t)}{q(\boldsymbol{x}_{t+1})}, \\ &= \sum_{\boldsymbol{x}_0} \frac{1}{q(\boldsymbol{x}_{t+1})} \cdot q(\boldsymbol{x}_{t+1} | \boldsymbol{x}_t) \cdot q(\boldsymbol{x}_t | \boldsymbol{x}_0) \cdot p(\boldsymbol{x}_0), \end{aligned} \tag{E.4}$$

where the last equality follows from $q(\boldsymbol{x}_t) = \sum_{\boldsymbol{x}_0} q(\boldsymbol{x}_t | \boldsymbol{x}_0) \cdot p(\boldsymbol{x}_0)$. Recall from the proposition that I is defined as the set of variables i such that $x_{t+1}^i = \texttt{<MASK>}$ and J is the complement of I.

First, we must have $x_t^j = x_{t+1}^j$ for $j \in J$ since for any other value of X_t^j, we have $q(\boldsymbol{x}_{t+1}|\boldsymbol{x}_t)=0$ in Eq. (E.4). As a result, $q(\boldsymbol{x}_t|\boldsymbol{x}_{t+1})$ is also zero.

We then move our attention to the variables in I. We first consider the probability $q(X_t^i=\texttt{<MASK>}|\boldsymbol{x}_{t+1})$ for any $i \in I$. Following Eq. (E.4), we have

$$\begin{aligned}
q(X_t^i = \texttt{<MASK>}|\boldsymbol{x}_{t+1}) &= \sum_{\boldsymbol{x}_0} \sum_{\boldsymbol{x}_t^{\setminus i}} \frac{1}{q(\boldsymbol{x}_{t+1})} \cdot q(\boldsymbol{x}_{t+1}|\boldsymbol{x}_t) \cdot q(\boldsymbol{x}_t|\boldsymbol{x}_0) \cdot p(\boldsymbol{x}_0), \\
&= \sum_{\boldsymbol{x}_t^{\setminus i}} \frac{1}{q(\boldsymbol{x}_{t+1})} \cdot q(\boldsymbol{x}_{t+1}|\boldsymbol{x}_t) \cdot q(\boldsymbol{x}_t), \\
&= \frac{q(X_{t+1}^i = \texttt{<MASK>}|X_t^i = \texttt{<MASK>}) \cdot q(X_t^i = \texttt{<MASK>})}{q(X_{t+1}^i = \texttt{<MASK>})}, \\
&= \frac{q(X_t^i = \texttt{<MASK>})}{q(X_{t+1}^i = \texttt{<MASK>})} = \frac{\alpha_t}{\alpha_{t+1}}.
\end{aligned} \tag{E.5}$$

We then focus on $\mathbf{X}_t^I = \boldsymbol{x}_t^I$, where none of the value in $\boldsymbol{x}_t^I$ is <MASK>. Note that we also need to have $\mathbf{X}_t^J = \boldsymbol{x}_{t+1}^J$.

$$\begin{aligned}
q(\boldsymbol{x}_t|\boldsymbol{x}_{t+1}) &\propto \sum_{\boldsymbol{x}_0} q(\boldsymbol{x}_{t+1}|\boldsymbol{x}_t) \cdot q(\boldsymbol{x}_t|\boldsymbol{x}_0) \cdot q(\boldsymbol{x}_0), \\
&\overset{(a)}{=} q(\boldsymbol{x}_{t+1}|\boldsymbol{x}_t) \cdot q(\mathbf{X}_0 = \boldsymbol{x}_t), \\
&= \left(\frac{\alpha_{t+1} - \alpha_t}{1 - \alpha_t}\right)^{|I|} \cdot q(\mathbf{X}_0 = \boldsymbol{x}_t), \\
&\propto q(\mathbf{X}_0 = \boldsymbol{x}_t),
\end{aligned} \tag{E.6}$$

where $p(\mathbf{X}_0)$ is the data distribution; (a) follows from the fact that no value in $\boldsymbol{x}_t$ is <MASK>, hence $\boldsymbol{x}_0 = \boldsymbol{x}_t$; $\frac{\alpha_{t+1}-\alpha_t}{1-\alpha_t}$ is the probability of transitioning into the mask state from time t to time $t+1$.

Denote $\tilde{\mathbf{X}}_t$ as a set of variables with the same configuration and semantics as $\mathbf{X}_t$, with the only difference that the category <MASK> is excluded. By following Eq. (E.6) and apply

normalization, we conclude that

$$q(\tilde{\boldsymbol{x}}_t|\boldsymbol{x}_{t+1}) = p(\mathbf{X}_0^I = \tilde{\boldsymbol{x}}_t^I|\mathbf{X}_0^J = \boldsymbol{x}_{t+1}^J) \cdot \mathbb{1}[\tilde{\boldsymbol{x}}_t^J = \boldsymbol{x}_{t+1}^J]. \tag{E.7}$$

This matches the definition in Eq. (6.5).

Finally, we verify the correctness of the distribution $q(\mathbf{X}_t|\tilde{\boldsymbol{x}}_t, \boldsymbol{x}_{t+1})$ defined in the proposition by verifying the following for any $\boldsymbol{x}_t$

$$q(\boldsymbol{x}_t|\boldsymbol{x}_{t+1}) = \sum_{\tilde{\boldsymbol{x}}_t} q(\tilde{\boldsymbol{x}}_t|\boldsymbol{x}_{t+1}) \cdot q(\boldsymbol{x}_t|\tilde{\boldsymbol{x}}_t, \boldsymbol{x}_{t+1}). \tag{E.8}$$

Denote K as the set of variables i such that $\boldsymbol{x}_t = \texttt{<MASK>}$ and L as its complement. First, if $L \subseteq J$ (i.e., $I \subseteq K$), then both the left-hand side (LHS) and the right-hand sides (RHS) are zero. Specifically, the RHS is zero since according to the definition, $\forall i \in J \,\&\, i \in K$, we have $q(x_t^i|\tilde{x}_t^i, x_{t+1}^i) = 0$.

Next, if $K \subseteq I$, we can decompose $q(\boldsymbol{x}_t|\boldsymbol{x}_{t+1})$ as follows

$$q(\boldsymbol{x}_t|\boldsymbol{x}_{t+1}) = q(\boldsymbol{x}_t^{I\setminus K}|\boldsymbol{x}_{t+1}) \cdot \prod_{i\in K} q(x_t^i|\boldsymbol{x}_{t+1}) \cdot \prod_{j\in J} q(x_t^j|\boldsymbol{x}_{t+1}). \tag{E.9}$$

For any $j \in J$, if $x_t^j \neq x_{t+1}^j$ then both the LHS and the RHS of Eq. (E.8) are zero. Otherwise we always have $q(x_t^j|\boldsymbol{x}_{t+1}) = 1$. Therefore, Eq. (E.9) can be further simplified as

$$q(\boldsymbol{x}_t|\boldsymbol{x}_{t+1}) = q(\boldsymbol{x}_t^{I\setminus K}|\boldsymbol{x}_{t+1}) \cdot \prod_{i\in K} q(x_t^i|\boldsymbol{x}_{t+1}). \tag{E.10}$$

We then proceed to simplify the RHS of Eq. (E.8):

$$\begin{aligned}
&\sum_{\tilde{\boldsymbol{x}}_t} q(\tilde{\boldsymbol{x}}_t|\boldsymbol{x}_{t+1}) \cdot q(\boldsymbol{x}_t|\tilde{\boldsymbol{x}}_t, \boldsymbol{x}_{t+1}), \\
=&\sum_{\tilde{\boldsymbol{x}}_t^K} q(\tilde{\boldsymbol{x}}_t^K, \tilde{\boldsymbol{x}}_t^{I\setminus K}|\boldsymbol{x}_{t+1}) \cdot \left(\frac{\alpha_t}{\alpha_{t+1}}\right)^{|K|} \cdot \left(\frac{\alpha_{t+1}-\alpha_t}{\alpha_{t+1}}\right)^{|I|-|K|},
\end{aligned}$$

$$
\begin{aligned}
&\overset{(a)}{=} \sum_{\tilde{\boldsymbol{x}}_t^K} q(\tilde{\boldsymbol{x}}_t^K, \tilde{\boldsymbol{x}}_t^{I\setminus K}|\boldsymbol{x}_{t+1}) \cdot \left(\frac{\alpha_{t+1}-\alpha_t}{\alpha_{t+1}}\right)^{|I|-|K|} \cdot \prod_{i\in K} q(x_t^i|\boldsymbol{x}_{t+1}),\\
&= q(\tilde{\boldsymbol{x}}_t^{I\setminus K}|\boldsymbol{x}_{t+1}) \cdot \left(\frac{\alpha_{t+1}-\alpha_t}{\alpha_{t+1}}\right)^{|I|-|K|} \cdot \prod_{i\in K} q(x_t^i|\boldsymbol{x}_{t+1}),\\
&\overset{(b)}{\propto} p(\mathbf{X}_0^{I\setminus K} = \tilde{\boldsymbol{x}}_t^{I\setminus K}, \mathbf{X}_0^J = \tilde{\boldsymbol{x}}_t^J) \cdot \prod_{i\in K} q(x_t^i|\boldsymbol{x}_{t+1}),\\
&\overset{(c)}{\propto} q(\mathbf{X}_t^{I\setminus K} = \tilde{\boldsymbol{x}}_t^{I\setminus K}|\boldsymbol{x}_{t+1}) \cdot \prod_{i\in K} q(x_t^i|\boldsymbol{x}_{t+1}),
\end{aligned} \tag{E.11}
$$

where (a) follows from Eq. (E.5), (b) applies the definition in Eq. (E.7), and (c) is a result of applying Eq. (E.6) to the case where $\tilde{\boldsymbol{x}}_t^L = \{\tilde{\boldsymbol{x}}_t^{I\setminus K}, \tilde{\boldsymbol{x}}_t^J\}$ are not <MASK>.

By combining Eqs. (E.10) and (E.11), we conclude that the LHS and the RHS of Eq. (E.8) are proportional to each other. Since they are both properly-normalized distributions, they must also match exactly.

□

Proof of Proposition 8. We first state a more detailed version of the proposition: for each variable i and data category c ($c \neq$ <MASK>), we have

$$
q(\tilde{X}_t^i = c|\boldsymbol{x}_{t+1}) = \frac{1}{Z} \cdot q(X_t^i = c|\boldsymbol{x}_{t+1}), \text{ where } Z = \sum_{c\neq\texttt{<MASK>}} q(X_t^i = c|\boldsymbol{x}_{t+1}).
$$

According to the proof of Proposition 7, Eq. (E.8) holds for all $\boldsymbol{x}_t$. Therefore, we have that for each i and each data category $x_t^i \neq$ <MASK>,

$$
q(x_t^i|\boldsymbol{x}_{t+1}) = \sum_{\tilde{\boldsymbol{x}}_t} q(\tilde{\boldsymbol{x}}_t|\boldsymbol{x}_{t+1}) \cdot q(x_t^i|\tilde{\boldsymbol{x}}_t, \boldsymbol{x}_{t+1}). \tag{E.12}
$$

If $i \in J$, then both the LHS of the above equation and $q(x_t^i|\tilde{\boldsymbol{x}}_t, \boldsymbol{x}_{t+1})$ equals one if and only if $x_t^i = x_{t+1}^i$. Therefore, the result holds trivially.

Next, if $i \in I$, denote $I_{\setminus i} := I \setminus \{i\}$, Eq. (E.12) is simplified to

$$\begin{aligned} q(x_t^i|\boldsymbol{x}_{t+1}) &= \sum_{\tilde{\boldsymbol{x}}_t} q(\tilde{\boldsymbol{x}}_t|\boldsymbol{x}_{t+1}) \cdot q(x_t^i|\tilde{\boldsymbol{x}}_t, \boldsymbol{x}_{t+1}), \\ &= \sum_{\tilde{x}_t^i} \sum_{\tilde{\boldsymbol{x}}_t^{I_{\setminus i}}} q(\tilde{x}_t^i, \tilde{\boldsymbol{x}}_t^{I_{\setminus i}}|\boldsymbol{x}_{t+1}) \cdot q(x_t^i|\tilde{x}_t^i, x_{t+1}^i), \\ &= q(\tilde{X}_t^i = x_t^i|\boldsymbol{x}_{t+1}) \cdot q(x_t^i|\tilde{X}_t^i = x_t^i, x_{t+1}^i), \\ &= q(\tilde{X}_t^i = x_t^i|\boldsymbol{x}_{t+1}) \cdot \frac{\alpha_{t+1} - \alpha_t}{\alpha_{t+1}}. \end{aligned}$$

Therefore, we have

$$q(\tilde{X}_t^i = x_t^i|\boldsymbol{x}_{t+1}) = \frac{1}{Z} \cdot q(X_t^i = x_t^i|\boldsymbol{x}_{t+1}), \text{ where } Z = \sum_{x_t^i \neq \texttt{<MASK>}} q(X_t^i = x_t^i|\boldsymbol{x}_{t+1}).$$

□

E.2 Relation Between the Copula Objective and Matrix Scaling

The matrix scaling problem gives a matrix A as input and asks for diagonal 'scaling' matrices X and Y such that XAY is doubly stochastic (its row and column sums are all one). More generally, target row and column sum vectors r and c are provided and need not contain only ones. The solvability of this problem for positive matrices was established by [161], and its algorithms (sometimes called iterative proportional fitting), generalizations, and numerous applications have been studied thoroughly [2, 72, 151]; see [67] for a review. Taking the multidimensional generalization of the problem and interpreting the tensor as a (unnormalized) probability distribution yields the connection to our problem, with the target sums being the univariate marginal distributions.

E.3 Parameterizing Discrete Copulas by Odds Ratios

We start by formally defining odds ratios.

Definition 14 ([149])**.** Let p be a distribution over variables $\boldsymbol{X}$ each taking values in $\{0, 1\}$. For a partition of $\boldsymbol{X}$ into sets $\boldsymbol{A}$ and $\boldsymbol{B}$, the *conditional odds ratio* of variables $\boldsymbol{A}$ conditioned on the assignment $\boldsymbol{B} = \boldsymbol{b}$ is

$$\mathrm{COR}_p(\boldsymbol{A}|\boldsymbol{B} = \boldsymbol{b}) = \frac{\prod_{\boldsymbol{a}\in s} p(\boldsymbol{a}, \boldsymbol{b})}{\prod_{\boldsymbol{a}\in d} p(\boldsymbol{a}, \boldsymbol{b})},$$

where s is the set of assignments to $\boldsymbol{A}$ whose parity is the same as the number of variables in $\boldsymbol{A}$, and d is the set of assignments whose parity is different.

In the case of more than two categories per variable, $\mathrm{COR}_p(\boldsymbol{A}|\boldsymbol{B} = \boldsymbol{b})$ can generalized further to be a set of similarly defined ratios (see, e.g., [149]). Together the set of all conditional odds ratios $\mathrm{COR}_p(\boldsymbol{A}|\boldsymbol{B} = \boldsymbol{b})$ for partitions of $\boldsymbol{X}$ into sets $\boldsymbol{A}$ and $\boldsymbol{B}$ with $|\boldsymbol{A}| \geq 2$, completely specifies the association among the variables in the joint distribution p, as established by the following theorem.

Theorem 1 ([149])**.** *Let q and r be positive probability distributions on a the set of variables $\boldsymbol{X}$ each taking values in $\{0, 1, \ldots, k\}$. Then there exists a unique probability distribution p such that p has the same univariate marginal distributions as q, that is, for all i*

$$p(x_i) = q(x_i),$$

and p has the same copula as q, that is for all partitions of $\boldsymbol{X}$ into sets $\boldsymbol{A}$ and $\boldsymbol{B}$ with $|\boldsymbol{A}| \geq 2$,

$$COR_p(\boldsymbol{A}|\boldsymbol{B} = \boldsymbol{b}) = COR_r(\boldsymbol{A}|\boldsymbol{B} = \boldsymbol{b}).$$

Proof. This follows from [149, Theorem 10.2] by taking the descending set to contain the empty set and all singletons (and the ascending set, its complement). □

Theorem 1 shows how any distribution p can be viewed as combining independent marginal distributions (i.e., from r) and odds ratios (i.e., from q). Such a combination has desirable properties. For example, in the case of two variables with possibly many categories, it has been shown that among all distributions with the same margins as r, the distribution p minimizes the KL-divergence to q [53, Theorem 6.2], i.e. that p is the information projection of q onto the set of distributions with the margins of r.

E.4 Unbiased Univariate Marginals from Discrete Diffusion Models

In this section, we show that when their respective training losses are minimized, discrete-time and continuous-time discrete diffusion models recover the true univariate marginals.

Discrete-Time Diffusion Models. Discrete-time diffusion models [3] are trained to maximize the ELBO between the forward joint distribution $p(\boldsymbol{x}_0)q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)$, where $p(\boldsymbol{x}_0)$ is the data distribution, and the reverse joint distribution $p_\theta(\boldsymbol{x}_{0:T})$. The ELBO can be simplified to

$$\mathbb{E}_q \left[\log \frac{p(\boldsymbol{x}_T)}{q(\boldsymbol{x}_T)} + \sum_{t=1}^{T} \log \frac{p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)}{q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)} + \log p(\boldsymbol{x}_0) \right].$$

Assume that $p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ encodes fully-factorized distribution, the above objective can be simplified as

$$\sum_{t=1}^{T} \sum_{i} q(x_{t-1}^i|\boldsymbol{x}_t) \log \frac{p_\theta(x_{t-1}^i|\boldsymbol{x}_t)}{q(x_{t-1}^i|\boldsymbol{x}_t)} + \mathbb{E}_q \left[\log \frac{p(\boldsymbol{x}_T)}{q(\boldsymbol{x}_T)} + \log p(\boldsymbol{x}_0) \right],$$

where the second term is independent to p_θ. From the first term of the above formula, we can conclude that the ELBO objective is maximized when $p_\theta(x_{t-1}^i|\boldsymbol{x}_t) = q(x_{t-1}^i|\boldsymbol{x}_t)$ for every t and every i.

Continuous-Time Diffusion Models. As described in Section 6.2, many continuous-time diffusion models learn to approximate the likelihood ratio (defined as $s_\theta(\boldsymbol{x}_t, \boldsymbol{x}'_t; t)$) at all noise levels $t \in [0, T]$:

$$s_\theta(\boldsymbol{x}_t, \boldsymbol{x}'_t; t) := \frac{q(\mathbf{X}_t = \boldsymbol{x}'_t)}{q(\mathbf{X}_t = \boldsymbol{x}_t)}.$$

Specifically, [105, 113] directly parameterize a neural network to approximate the likelihood ratios, and [165] approximates the likelihood ratios with the conditional distributions $p_\theta(X_t^i | \boldsymbol{x}_t^{\setminus i})$ $(\forall i, t)$.

For each $\boldsymbol{x}_t$, since there are exponentially many possible $\boldsymbol{x}'_t$, it is infeasible to have a neural network to directly model the likelihood ratio for all pairs of $(\boldsymbol{x}_t, \boldsymbol{x}'_t)$. Instead, they focus on $(\boldsymbol{x}_t, \boldsymbol{x}'_t)$ pairs where $\boldsymbol{x}_t$ and $\boldsymbol{x}'_t$ are only different in one single variable, i.e., their Hamming distance is one. For example, in [105], they represent s_θ as $s_\theta(\boldsymbol{x}_t, y_t^i; t, i)$, which computes the likelihood ratio between $\boldsymbol{x}_t$ and $\boldsymbol{x}'_t = \{\boldsymbol{x}_t^{\setminus i}, y_t^i\}$. s_θ is trained by minimizing the following objective:

$$\mathbb{E}_{t, \boldsymbol{x}_t \sim q(\mathbf{X}_t)} \left[\sum_i \sum_{y_t^i \neq x_t^i} w_t \left(s_\theta(\boldsymbol{x}_t, y_t^i; t, i) - \frac{q(\mathbf{X}_t = \{\boldsymbol{x}_t^{\setminus i}, y_t^i\})}{q(\mathbf{X}_t = \boldsymbol{x}_t)} \log s_\theta(\boldsymbol{x}_t, y_t^i; t, i) \right) \right],$$

where $\{w_t\}_t$ are positive weights. When the above objective is minimized, s_θ recovers the correct likelihood ratios:

$$\forall i, t,\ s_\theta(\boldsymbol{x}_t, y_t^i; t, i) = \frac{q(\mathbf{X}_t = \{\boldsymbol{x}_t^{\setminus i}, y_t^i\})}{q(\mathbf{X}_t = \boldsymbol{x}_t)}. \tag{E.13}$$

At inference time, continuous-time discrete diffusion models select a list of time steps $0 < t_0 < \cdots < t_k = T$ to sample from: first sample from the prior $p(\mathbf{X}_{t_k})$ and then sample recursively from $\{p_\theta(\boldsymbol{x}_{t_{i-1}} | \boldsymbol{x}_{t_i})\}_{i=1}^k$, where $p_\theta(\boldsymbol{x}_{t_{i-1}} | \boldsymbol{x}_{t_i})$ is obtained from $s_\theta(\boldsymbol{x}_t, y_t^i; t, i)$ in an

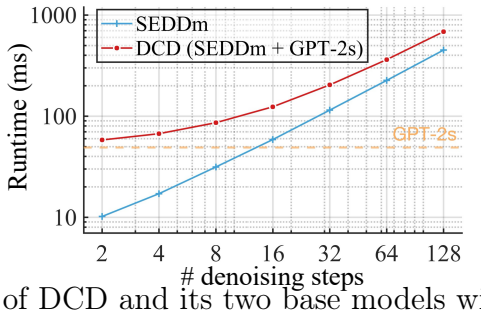


Figure E.1: Sampling time of DCD and its two base models with 2 to 128 denoising steps.

indirect manner. Specifically, assume $\frac{dp(\boldsymbol{x}_t)}{dt} = Q \cdot p(\boldsymbol{x}_t)$, we have[1]

$$\begin{aligned} q(\boldsymbol{x}_{t_{i-1}}|\boldsymbol{x}_{t_i}) &= q(\boldsymbol{x}_{t_i}|\boldsymbol{x}_{t_{i-1}}) \cdot \frac{q(\boldsymbol{x}_{t_{i-1}})}{q(\boldsymbol{x}_{t_i})}, \\ &= q(\boldsymbol{x}_{t_i}|\boldsymbol{x}_{t_{i-1}}) \cdot \left(\sum_{\boldsymbol{x}} \exp(-\Delta t \cdot Q)(\boldsymbol{x}_{t_{i-1}}, \boldsymbol{x}) \cdot \frac{q(\mathbf{X}_{t_i} = \boldsymbol{x})}{q(\mathbf{X}_{t_i} = \boldsymbol{x}_{t_i})} \right), \end{aligned}$$

where $\Delta t := t_i - t_{i-1}$ and $\exp(-\Delta t \cdot Q)(\boldsymbol{x}_{t_{i-1}}, \boldsymbol{x})$ denotes the product of $\exp(-\Delta t \cdot Q)(x^j_{t_{i-1}}, x^j)$, the $x^j_{t_{i-1}}$-th row and x^j-th column of $\exp(-\Delta t \cdot Q)$.

Plug in Eq. (E.13), we can compute the marginal of $x^j_{t_{i-1}}$ (i.e., $p_\theta(x^j_{t_{i-1}}|\boldsymbol{x}_{t_i})$) following

$$\begin{aligned} q(X^j_{t_{i-1}} = y|\boldsymbol{x}_{t_i}) &\propto q(\boldsymbol{x}_{t_i}|\boldsymbol{x}_{t_{i-1}}) \cdot \left(\sum_{y'} \exp(-\Delta t \cdot Q)(y, y') \cdot s_\theta(\boldsymbol{x}_{t_i}, y'; t_i, j) \right), \\ &= \exp(\Delta t \cdot Q)(y, x^j_{t_i}) \cdot \left(\sum_{y'} \exp(-\Delta t \cdot Q)(y, y') \cdot s_\theta(\boldsymbol{x}_{t_i}, y'; t_i, j) \right). \end{aligned}$$

Therefore, if s_θ perfectly learns the likelihood ratios between inputs with Hamming distance at most one, then the correct marginals $q(x^j_{t_{i-1}}|\boldsymbol{x}_{t_i})$ can be computed using s_θ.

E.5 Implementation Details of DCD

We describe details about the "autoregressive" version of DCD introduced in Section 6.5.3. According to Section 6.5.3, the first $(T-t-1)/T$ portion of the tokens in $\boldsymbol{x}_{t+1}$ are unmasked. At step t, we only need to additionally unmask the tokens spanning the $(T-t-1)/T$ to

[1]This argument largely follows Theorem 4.1 in [105]. We include it for the sake of completeness.

Algorithm 28 DCD with Autoregressive Copula Models and Using Autoregressive Sampling

Inputs: a diffusion model p_{dm}, an autoregressive model p_{ar}, number of time steps T, sequence length L
Outputs: a sample $\boldsymbol{x}_0$ from the discrete diffusion model augmented by the autoregressive model
Initialize: Sample $\boldsymbol{x}_T$ from the prior noise distribution $p(\mathbf{X}_T)$
for $t = T-1$ **to** 0
  $i_{\min}, i_{\max} = \frac{L}{T}\cdot(T-t-1), \frac{L}{T}\cdot(T-t)$ (w.l.o.g. assume L is divisible by T)
  Compute $\{p_{\text{dm}}(\tilde{X}_t^i|\boldsymbol{x}_{t+1})\}_i$ and $\{p_{\text{dm}}(\tilde{X}_t^i|\boldsymbol{x}_{t+1}^{<i})\}_i$ for each $i \in [i_{\min}, i_{\max})$ using the diffusion model
  Compute $\mathbf{V}[i, \tilde{x}_t^i] = \log p_{\text{dm}}(\tilde{x}_t^i|\boldsymbol{x}_{t+1}) - \log p_{\text{dm}}(\tilde{x}_t^i|\boldsymbol{x}_{t+1}^{<i})$ $(\forall i \in [i_{\min}, i_{\max}), \tilde{x}_t^i)$
  $\boldsymbol{x}_t \leftarrow \boldsymbol{x}_{t+1}$
  for $i = i_{\min}$ **to** $i_{\max} - 1$
    Sample x_t^i from $\hat{p}(x_t^i) \propto p_{\text{ar}}(x_t^i|\boldsymbol{x}_t^{<i}) \cdot \prod_i \exp(\mathbf{V}[i, x_t^i])$ and store it to $\boldsymbol{x}_t$

$(T-t)/T$ fraction of the sequence $\boldsymbol{x}_t$. We do this by caching the keys and values generated by the attention layers of tokens generated in previous denoising steps. So at step t, we will have the KV-caches of the first $(T-t-1)/T$ fraction of tokens. As a result, the computational cost for running the autoregressive Transformer is independent of the number of denoising steps.

Additional Runtime Analysis. Fig. E.1 displays the generation time per sample for $\text{SEDD}_{\texttt{M}}$, $\text{GPT-2}_{\texttt{S}}$, and DCD. When the number of denoising steps is small, the computation cost of running $\text{GPT-2}_{\texttt{S}}$ dominates the total runtime of DCD. However, as the number of denoising steps increases, this cost is amortized because, with KV-caching, the total computation cost for running $\text{GPT-2}_{\texttt{S}}$ stays constant.

E.6 Additional Unconditional Generation Experiments

To better understand the relation between quality (measured by generative perplexity), diversity (measured by sentence entropy[2]), and speed for DCD and its baselines. Specifically, we run the more efficient version of DCD described in the last paragraph of Section 6.5.3 and Section E.5 to generate text sequences of lengths 128 and 1024. In addition to SEDD and GPT2, the two base models used by DCD, we compare them with MDLM [153], a more recent discrete diffusion model that is more efficient than SEDD. Note that DCD can use any

[2]The sentence entropy of a sequence is the entropy of its token frequency distribution. The reported number is averaged across all samples.

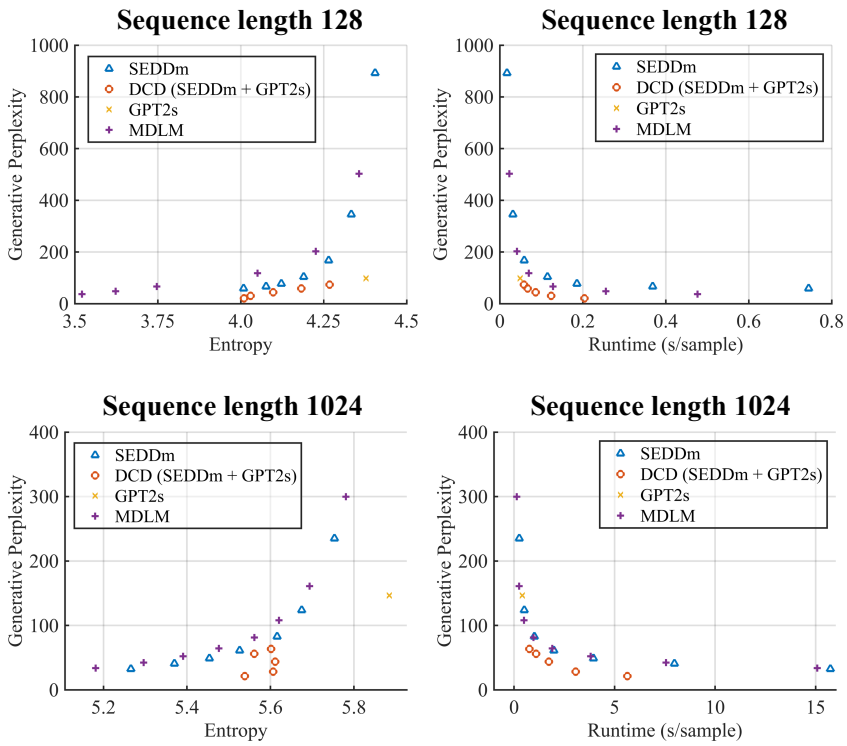


Figure E.2: Comparison between generative perplexity (↓), diversity (measured by sentence entropy; ↑), and runtime (↓) of DCD with baselines.

discrete diffusion model as its base model.

First, we compare the sample time and the generative perplexity (the second and the fourth sub-plot in Fig. E.2). Compared to SEDD, GPT, and MDLM, DCD consistently achieves better generative perplexity given a fixed runtime constraint. It also requires less time to achieve a desired perplexity value.

Additionally, we compare the perplexity and diversity of the generated text sequences. Following community standards, we adopt the sentence entropy to measure the diversity of generated text. Specifically, the entropy of each text sequence is the entropy of its token frequency distribution, and the final sentence entropy is the average entropy over all generated sequences. The desired behavior is to have low generative perplexity and high sentence entropy (which means high diversity). Results are shown in the table below and Fig. E.2's first and third sub-plot. Compared to the two discrete diffusion models (SEDD and MDLM), DCD achieves better generative perplexity under the same entropy, which offers a

better perplexity-diversity tradeoff. Compared to the autoregressive GPT model, although the entropy of DCD is lower, it achieves better generative perplexity with slightly worse entropy.

E.7 Additional Experimental Details

This section provides additional details of the experiments.

E.7.1 Unconditional Text Generation

SEDD. We adopt the SEDD-medium model with 320M non-embedding parameters trained on OpenWebText. The model is accessed through HuggingFace: https://huggingface.co/louaaron/sedd-medium. We follow the original paper [105] and use the log-linear noise schedule $\sigma(t) = -\log(1-(1-\epsilon t))$, which leads to the forward transition probabilities ($0 \le s < t \le T$):

$$q(\boldsymbol{x}_t|\boldsymbol{x}_s) := \mathrm{Cat}(\boldsymbol{x}_t; \exp(\sigma(t-s)\cdot Q)\cdot \boldsymbol{x}_s).$$

The absorbing mask forward noising process is used. The corresponding transition rate matrix is

$$Q := \begin{bmatrix} -1 & 0 & \cdots & 0 & 0 \\ 0 & -1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & -1 & 0 \\ 1 & 1 & \cdots & 1 & 0 \end{bmatrix},$$

where the last category is `<MASK>`.

GPT. The GPT-2-small model is obtained from HuggingFace: https://huggingface.

co/openai-community/gpt2.

DCD. We implement DCD by combining SEDD_{M} and GPT-2_{S} following the steps in Algorithm 8. In line 8, instead of masking tokens independently, we group chunks of 8 tokens together and mask/unmask them with the same probability given the noise schedule (i.e., α_t/α_{t+1} as shown in Prop. 7).

E.7.2 Conditional Text Generation

MAUVE Score. We adopt the MAUVE implementation available in the Python package `evaluate`. We use the default hyperparameters established by the original paper [134], which is also the default used by the package. We found that the number of samples and the number of samples given a fixed prompt influenced the score. Therefore, we randomly selected the 2,000 prompts and generated 5 samples for each prompt for all methods.

Detailed Runtime Analysis. As shown in Algorithm 8, in each denoising step of DCD, we need to run the discrete diffusion model twice: first to compute $\{p(\tilde{X}_t^i|\boldsymbol{x}_{t+1})\}_i$ and next to compute $\{p(\tilde{X}_t^i|\boldsymbol{x}_{t+1}^{<i})\}_i$ by applying causal attention masks to the same denoising neural network given that it is based on the Transformer architecture. Next, as discussed in Section E.5, the total runtime consumed by the autoregressive model remains constant across different numbers of denoising steps thanks to the KV-caching mechanism. Therefore, the runtime of DCD will be dominated by the computation cost of the autoregressive model with only a few denoising steps. As the number of denoising steps increases, the runtime of the autoregressive model will be amortized and the total computation cost will be dominated by the cost to evaluate the diffusion model.

SSD-LM. SSD-LM [60] is a semi-autoregressive model that uses techniques from discrete diffusion models to predict/denoise chunks of sequences in an autoregressive manner. Specifically, given a predefined chunk size, SSD-LM diffuses tokens in each chunk one by one conditioned on all previous chunks. As a result, the model is semi-autoregressive and cannot see suffix prompts.

While the official implementation on GitHub (https://github.com/xhan77/ssd-lm) only allows conditioning on tokens in previous prompts, we improved their code to also allow conditioning on tokens in the current chunk that is being diffused. Specifically, we replace the diffusion model's input corresponding to the prompt tokens with the ground truth token embeddings.

We followed the original paper to choose a chunk size of 32 and use top-p sampling with $p=0.95$. The remaining hyperparameters are kept as default.

E.7.3 Antibody Sequence Infilling

Detailed Task Description. The adopted antibodies with an immunoglobulin G (IgG) format, which comprises a heavy (H) chain and a light (L) chain. Each chain has three complementarity determining regions (CDRs) that are crucial toward the binding affinity to the target antigen.

Training NOS-D. We use the training script as well as the dataset provided in the official GitHub repo of NOS-D (https://github.com/ngruver/NOS). The model is trained with 50 epochs using the default settings (e.g., learning rate and its schedule).

Training GPT. We use the same dataset provided in the repository of NOS-D and use the GPT implementation from https://github.com/karpathy/nanoGPT/tree/master. The GPT model has 6 layers, an embedding size of 512, and 16 attention heads. The model is trained for 10 epochs with the default settings in the nanoGPT repository.

DCD. When implementing DCD for the antibody sequence infilling task, we add an additional scaling factor to the coefficients in $\mathbf{V}$. That is, $\mathbf{V}$ is updated in line 6 of Algorithm 8 following

$$\forall i, \tilde{x}_t^i,\ \mathbf{V}[i, \tilde{x}_t^i] = \beta \cdot \left(\log p_{\text{dm}}(\tilde{x}_t^i | \boldsymbol{x}_{t+1}) - \log p_{\text{dm}}(\tilde{x}_t^i | \boldsymbol{x}_{t+1}^{<i})\right),$$

where we set $\beta=0.1$ for this task. We note that $\beta=1$ works well for the language modeling

tasks. The need to choose a smaller β in this task may be caused by the fact that the dataset and the models are much smaller and are more prone to overfitting.

E.8 Additional Text Samples

We provide randomly selected unconditional samples in Figs. E.3 and E.4 and conditional samples in Figs. E.5 and E.6.

…, DHHS, Dion Todd of Detroit, Detroit Tigers team players Marcus Johnson of Detroit Lions, Minnesota Vikings team players Troy Polamalu of Detroit Tigers, mellow lines for opposing teams, but they do not share a common mentality.

“At the end of the day, if you‘re just grouping me but you also come for the tram stop, everybody plays by the rules,” Kalim said of Johnson. “If you start to play offense, you’re going to make mistakes. We don‘t just give them time because we don’t think they‘re going to win. We don’t just teach them how to take care …

Both 1-2 range panels are available for 3 hours of ongoing use (25 burning power consumption plus periodic gas combustion cycle). Also included moisture protection from agricultural insects and biological spills of greenhouse gases due to different costs of livestock sit idle in the energy cycle of so many nations.

The high yield yields from large, agricultural biofuels rely entirely on larger projections placed together by country by government into 2020. Such predictions assume an 1 billion tonne increase in capacity to 5 billion tonnes per year six billion years from now then consumption for nearly all full-time, undertaking- Stage 3 + 2.0 exponential growth.

… acquisition jack. Add this to the negotiations, and the place starts going down. Veteran grocers like State Farm have halted stocking their own toll booths and warnings because they fear getting squeezed out by major retailers. Meanwhile, stores like Wal-Mart have reduced shelf space by as much as 8.6 percent. Wal-Mart Stores Canada has been the most profitable Wal-Mart store chain in Canada (green grocer WalMart Stores Canadian now accounts for nearly a third of sales, up from just 1 percent in 1996). Meanwhile Wal-Mart continues to aggressively sell Canadian goods. Since 1995, Wal-Mart Stores Canada has more than tripled the volume and …

I find myself divided. Like I was growing up in this world. I saw my dad constantly being obsessed about faith, constantly being remembered in my mind his name. My dad thought that I wasn't going to grow up to say myself, if I tried to make my family happy they wouldn't believe me anymore. Every time I looked in my eyes I thought I was crazy, but I thought I was my brother. I was worried I would always feel jealous of my father and slowly, I started to think about family. I found a family where everyone ruled me alone towards the end, in hell. Family was always there, it was

flap: Now you can lay your hands on a wavy pattern without touching the nerves, or if you fancy you can lay your arms on an occluded specialty? Any hypothesis relied from those vainly generalized action.[303] Reconciling the patient‘s single hairs with multiple llings stipulated that a single story could be shortened in half, but in fact lengthened in less flexible forms. If, however, little succeeded at the scholarly step, that compromise was subtlety in his notes, provided he lacked enough hairs to lie straight down. He could even weave rings for his harp—especially wheat—but there was …

Figure E.3: Randomly selected unconditional samples from DCD ($\text{SEDD}_{\texttt{M}} + \text{GPT-2}_{\texttt{S}}$) with 4 denoising steps.

March if they can overcome federal complaints.

Farmers say they have acquiesced to pressure from U.S. agribusiness giants to cut back on pesticides, while environmentalists say regulators allege a lack of oversight by officials.

"We won't tolerate anything bad," said Rutko Guerra, spokesman for Cornell University Extension. "It's a clear conflict of interest."

Cornell University Extension estimates about 400 pesticides are sold through the city each year in violation of the Public Health Act and other state laws, prompting over $49 million in fines — the largest ever levied by a public university.

Each time components are created a buffer is created. This buffer holds the components and should be updated whenever they touch on the screen. The buffer foundation passes every buffer value from the component to the component's buffer.

After the component has been created a markers is placed inside the marker stating which colors are used in the new paint direction. During the paint direction there are three modes: launch to draw pixels, around draw so baccarat will appear misaligned, and lowdraw so baccarat will appear perfectly aligned.

… in on their autobiographical stories of realizing happiness.The Stimulant Prophecy was an important spiritual awakening that occurred during the 1950s, 1960s and 1970s. It awakened believers to cultivate spiritual fortitude as well as resolve conflicts and lead to more successful relationships.According to Tages Jephzei (The Stimulant Prophecy), this event ultimately caused Muslims to develop compassion for one another by their communal experiences.It also fueled support for incongruity in mainline Muslim societies and gave hope for physical cleansings.In 1976, Tages Jephzei published his book The Stimulant Prophecy: Understanding Muslims …

…would proper address regulations affecting this country." He recalled how he supported Deputy Minority Leader Nancy Pelosi's (D-Calif.) efforts to explore Russian meddling in the 2016 election: "When Nancy Pelosi [D-Calif.] spoke to me, he wanted to consider Russian interference in our election." Kennedy also added that he believes a new law requiring commercial polluters to disclose their emissions during emissions tests will help clean up the air: "[I] hope that the EPA will follow through with two years of programs that are going to reduce [during gas] emissions in some form or another," Kennedy told reporters.

Figure E.4: Randomly selected unconditional samples from DCD ($\text{SEDD}_{\text{M}} + \text{GPT-2}_{\text{S}}$) with 32 denoising steps.

… are hear the exhortations “We ask of everyone to speak the voice of God” and “we ask to be loved **for the game and what they wanted to do next for the series**.” In fact, PS3 announcement executives confirmed this month that Sony Pictures Entertainment plans to release PS Vita versions of Sony PlayStation Classics PlayStationGS, The Last of Us, Ratchet, Square **did not come up with the “ revolutionary ” idea that would warrant a new entry for the PlayStation 3. Speaking in an interview**, it was revealed that Square Enix felt Square Enix could not offer a few new plugins without seeing Square Enix make Square Enix‘s “ visionary Battle …

… in the American version, and Warner Bros. added Nobuo Ukiura of Miyazaki Animation to directing on character design. **A large team of writers handled the script. The game‘s** story was developed by Kouki Watanabe in Chouki no Namco Europe, and Fujikyo Pictures Entertainment released it theatrically on May 25, 1999 in North America, followed **with an expanded edition in November of that year. It was also adapted into manga and an original video animation series. Due to low sales**, Warner Bros. suffered widespread cancellation due to lack of revenue. Warner Bros. also shut it down due to its failure to …

They save as many enemies as you can through Chrono Trigger Online missions, unlock quests in missions missions, unlock special quests, **having a higher difficulty than those found in the rest of the game.** These include boss & combat objectives. Chrono Trigger Online contains one Chrono Trigger EP with unlockable Chrono Trigger ARC girls.

This is the first patch which implements the PTZ **system, is carried over directly from Valkyira Chronicles. During missions, players select each unit using a top @**-left position. They determine their unit type, which determines their ability and the size of their field of vision. They can only activate …

The Final Fantasia in Japanese / **Media.Vision for the PlayStation Portable. Released in January 2011 in Japan, it** debuted as a novel while on hold in the North East Stand. It garnered reviews for its breathtaking narrative, disappointing plot, and creepy characters. A fourth graphic novel is in the first game **and follows the " Nameless ", a penal military unit serving the nation of Gallia during the Second Europan War.**

… scenery was was composed in short animation. When the game ended early on Mikami Sakura was drawn by Makoto Masui.. A large team of writers handled the script. The game‘s soundtrack lasted around 12 hours and Sugiyama Shogarashi, Makoto Masuyama and Kyoko Takamura included Takme Ibara. The music theme was originally released in 2009, **with an expanded edition in November of that year. It was also adapted into manga and an original video animation series. Due to low sales, the** game release was delayed to three weeks in spring 2011. Following its final release on May 23rd 2013. …

Figure E.5: Randomly selected conditional samples from DCD ($\text{SEDD}_{\text{M}} + \text{GPT-2}_{\text{S}}$) with 4 denoising steps. Prompt texts are bolded and in blue.

… every character has. You can choose combat situation best suit the **character. To learn Battle Potentials, each character has a unique** skills makes them invaluable. One of Potentials best suit the character is Point squirrel on the map, the character Leda can use skills like " Star Wars Matchmaker", the character Jaden **can activate " Direct Command " and move around the battlefield without depleting his Action Point gauge, the character Reila can shift** to melee objects to send morgues (so more reliable), Mira can change her story situation to battle nweire battle to ward strategise, a " Command Pointcher " is …

… this State building institution. We could be build a defensive system **to the United States Arsenal in this city (Little Rock).** This system seem really feasible and good. The name of the City that would to this scenario go by the Rocky Mountain Sound as the Academy as well is MADISON FIREWRIGHT NASHA ROCK. **-John M Harrel Telegram, January 31, 1861**
The item was intended simply as a piece of news, but it also served as an "opportunity" for the U.S. Fortresses on Little Rock. Setting aside the exception of the basisicks' menansi slogan, it was all …

… on the air at the end of those years run in early **1923. An original design for the society called The Darling of** the American for Being Unnamed was put forth on the air at the end of those years held early 1925. The unnamed pursuits of the American were previously documented by History Magazine called St. Luke's **Society for the Propagation of the Gospel in its November 1919 exhibition.**
Religious @-@ themed books include The Red Book, The Hidden Voyage, an opera which was written on behalf of John Ford and produced under the contract of the Protestant revival organization, The Evangelical Fund (without Contemplation) …

letter to city Evans noted in more details reminded Christine Barker to **supervise the household, and to give both her mother and sister** complete authority to their development. (See Evansdone & Sullivan) 79. Christine Barker continued her adult life but when she reached ends of age, during which time her big sister Ruth had died unexpectedly **of a heart attack. Barker was unable to pursue her art to any significant extent following her sister's death, as all** of her parents perished and she lacked the discipline, learning needed to be as a professional age. (See Evanssic) 80. Although moving art was a lifetime profession for Christine Barker, bear …

Pool : At Mumbai airport Shivaji Park. Women Technical girls **under @-@ 17 women's team competed in Confederation of** Asian Football's premier youth competition. 2011 year-13 results : :U 13 medal : NW15 qualification : A pool order : Of the 155 young women, five girls had to be narrowed **from an initial pool of 49 young women. Two girls from the SOS Children ' s Village Bakoteh** were chosen for the USC and two girls from the Meijer 's Village Bakoteh. The remaining Meijer girl was selected for the opening ceremony. After the AU's teams

Figure E.6: Randomly selected conditional samples from DCD ($\text{SEDD}_{\text{M}} + \text{GPT-2}_{\text{S}}$) with 32 denoising steps. Prompt texts are bolded and in blue.

Bibliography

[1] Anurag Ajay, Yilun Du, Abhi Gupta, Joshua B Tenenbaum, Tommi S Jaakkola, and Pulkit Agrawal. Is conditional generative modeling all you need for decision making? In *The Eleventh International Conference on Learning Representations*, 2022.

[2] Zeyuan Allen-Zhu, Yuanzhi Li, Rafael Oliveira, and Avi Wigderson. Much faster algorithms for matrix scaling. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 890–901. IEEE, 2017.

[3] Jacob Austin, Daniel D Johnson, Jonathan Ho, Daniel Tarlow, and Rianne Van Den Berg. Structured denoising diffusion models in discrete state-spaces. *Advances in Neural Information Processing Systems*, 34:17981–17993, 2021.

[4] Francis R Bach and Michael I Jordan. Thin junction trees. In *NIPS*, volume 14, pages 569–576, 2001.

[5] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018.

[6] David Brandfonbrener, Alberto Bietti, Jacob Buckman, Romain Laroche, and Joan Bruna. When does return-conditioned supervised learning work for offline reinforcement learning? *Advances in Neural Information Processing Systems*, 35:1542–1553, 2022.

[7] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[8] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[9] Cory J Butz, Jhonatan S Oliveira, André E Santos, André L Teixeira, Pascal Poupart, and Agastya Kalra. An empirical study of methods for spn learning and inference. In *International Conference on Probabilistic Graphical Models*, pages 49–60. PMLR, 2018.

[10] Andrew Campbell, Joe Benton, Valentin De Bortoli, Tom Rainforth, George Deligiannidis, and Arnaud Doucet. A continuous time framework for discrete denoising models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, pages 28266–28279, 2022.

[11] Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Misha Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. Decision transformer: Reinforcement learning via sequence modeling. *Advances in neural information processing systems*, 34:15084–15097, 2021.

[12] Arthur Choi, Doga Kisa, and Adnan Darwiche. Compiling probabilistic graphical models using sentential decision diagrams. In *European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty*, pages 121–132. Springer, 2013.

[13] Arthur Choi, Guy Van den Broeck, and Adnan Darwiche. Tractable learning for structured probability spaces: A case study in learning preference distributions. In *Proceedings of 24th International Joint Conference on Artificial Intelligence (IJCAI)*, volume 2015, pages 2861–2868, 2015.

[14] Y Choi, Antonio Vergari, and Guy Van den Broeck. Probabilistic circuits: A unifying framework for tractable probabilistic models. *UCLA. URL: http://starai. cs. ucla. edu/papers/ProbCirc20. pdf*, page 6, 2020.

[15] YooJung Choi, Meihua Dang, and Guy Van den Broeck. Group fairness by probabilistic modeling with latent fair decisions. *arXiv preprint arXiv:2009.09031*, 2020.

[16] YooJung Choi, Meihua Dang, and Guy Van den Broeck. Group fairness by probabilistic modeling with latent fair decisions. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 12051–12059, 2021.

[17] 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*, Feb 2021.

[18] 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*, Feb 2021.

[19] YooJung Choi, Adnan Darwiche, and Guy Van den Broeck. Optimal feature selection for decision robustness in bayesian networks. In *IJCAI*, pages 1554–1560, 2017.

[20] 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 AAAI Conference on Artificial Intelligence*, volume 34, pages 10077–10084, 2020.

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

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

[23] Hyungjin Chung, Jeongsol Kim, Michael T Mccann, Marc L Klasky, and Jong Chul Ye. Diffusion posterior sampling for general noisy inverse problems. *arXiv preprint arXiv:2209.14687*, 2022.

[24] 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.

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

[26] Alvaro HC Correia, Gennaro Gala, Erik Quaeghebeur, Cassio de Campos, and Robert Peharz. Continuous mixtures of tractable probabilistic models. *arXiv preprint arXiv:2209.10584*, 2022.

[27] Alvaro HC Correia, Gennaro Gala, Erik Quaeghebeur, Cassio de Campos, and Robert Peharz. Continuous mixtures of tractable probabilistic models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 7244–7252, 2023.

[28] Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. Towards general purpose acceleration by exploiting common data-dependence forms. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 924–939, 2019.

[29] 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.

[30] 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.

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

[32] Meihua Dang, Antonio Vergari, and Guy Broeck. Strudel: Learning structured-decomposable probabilistic circuits. In *International Conference on Probabilistic Graphical Models*, pages 137–148. PMLR, 2020.

[33] Tri Dao and Albert Gu. Transformers are SSMs: Generalized models and efficient algorithms through structured state space duality. In *International Conference on Machine Learning (ICML)*, 2024.

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

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

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

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

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

[39] Jacob Devlin. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[40] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT (1)*. Association for Computational Linguistics, 2019.

[41] Prafulla Dhariwal and Alex Nichol. Diffusion models beat gans on image synthesis. In *Proceedings of the 35th International Conference on Neural Information Processing Systems*, pages 8780–8794, 2021.

[42] Sander Dieleman, Laurent Sartran, Arman Roshannai, Nikolay Savinov, Yaroslav Ganin, Pierre H Richemond, Arnaud Doucet, Robin Strudel, Chris Dyer, Conor Durkan, et al. Continuous diffusion for categorical data. *arXiv preprint arXiv:2211.15089*, 2022.

[43] Thomas G Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of artificial intelligence research*, 13:227–303, 2000.

[44] Wenhao Ding, Tong Che, Ding Zhao, and Marco Pavone. Bayesian reparameterization of reward-conditioned reinforcement learning with energy-based models. *arXiv preprint arXiv:2305.11340*, 2023.

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

[46] Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. Density estimation using real nvp. In *International Conference on Learning Representations*, 2016.

[47] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[48] Jarek Duda. Asymmetric numeral systems: entropy coding combining speed of huffman coding with compression rate of arithmetic coding. *arXiv preprint arXiv:1311.2540*, 2013.

[49] Scott Emmons, Benjamin Eysenbach, Ilya Kostrikov, and Sergey Levine. Rvs: What is essential for offline rl via supervised learning? In *International Conference on Learning Representations*, 2021.

[50] Patrick Esser, Robin Rombach, and Bjorn Ommer. Taming transformers for high-resolution image synthesis. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 12873–12883, 2021.

[51] Justin Fu, Aviral Kumar, Ofir Nachum, George Tucker, and Sergey Levine. D4rl: Datasets for deep data-driven reinforcement learning, 2020.

[52] Scott Fujimoto, David Meger, and Doina Precup. Off-policy deep reinforcement learning without exploration. In *International conference on machine learning*, pages 2052–2062. PMLR, 2019.

[53] Gery Geenens. Copula modeling for discrete random vectors. *Dependence Modeling*, 8(1):417–440, 2020.

[54] Alison L Gibbs and Francis Edward Su. On choosing and bounding probability metrics. *International statistical review*, 70(3):419–435, 2002.

[55] Aaron Gokaslan and Vanya Cohen. Openwebtext corpus. http://Skylion007.github.io/OpenWebTextCorpus, 2019.

[56] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.

[57] Aditya Grover and Stefano Ermon. Boosted generative models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2018.

[58] Nate Gruver, Samuel Stanton, Nathan Frey, Tim GJ Rudner, Isidro Hotzel, Julien Lafrance-Vanasse, Arvind Rajpal, Kyunghyun Cho, and Andrew Gordon Wilson.

Protein design with guided discrete diffusion. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, pages 12489–12517, 2023.

[59] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, et al. Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905*, 2018.

[60] Xiaochuang Han, Sachin Kumar, and Yulia Tsvetkov. Ssd-lm: Semi-autoregressive simplex-based diffusion language model for text generation and modular control. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 11575–11596, 2023.

[61] Kaiming He, Xinlei Chen, Saining Xie, Yanghao Li, Piotr Dollár, and Ross Girshick. Masked autoencoders are scalable vision learners. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16000–16009, 2022.

[62] Geoffrey E Hinton and Drew Van Camp. Keeping the neural networks simple by minimizing the description length of the weights. In *Proceedings of the sixth annual conference on Computational learning theory*, pages 5–13, 1993.

[63] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *Advances in neural information processing systems*, 33:6840–6851, 2020.

[64] Emiel Hoogeboom, Jorn Peters, Rianne van den Berg, and Max Welling. Integer discrete flows and lossless compression. *Advances in Neural Information Processing Systems*, 32:12134–12144, 2019.

[65] Han Huang, Leilei Sun, Bowen Du, and Weifeng Lv. Conditional diffusion based on discrete graph structures for molecular graph generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 4302–4311, 2023.

[66] David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[67] Martin Idel. A review of matrix scaling and sinkhorn's normal form for matrices and positive maps. *arXiv preprint arXiv:1609.06349*, 2016.

[68] Michael Janner, Qiyang Li, and Sergey Levine. Offline reinforcement learning as one big sequence modeling problem. *Advances in neural information processing systems*, 34:1273–1286, 2021.

[69] Robert Jenssen, Jose C Principe, Deniz Erdogmus, and Torbjørn Eltoft. The cauchy–schwarz divergence and parzen windowing: Connections to graph theory and mercer kernels. *Journalof the Franklin Institute*, 343(6):614–629, 2006.

[70] Mark Jerrum and Alistair Sinclair. Polynomial-time approximation algorithms for the ising model. *SIAM Journal on computing*, 22(5):1087–1116, 1993.

[71] Wolfgang B Jurkat. On cauchy's functional equation. *Proceedings of the American Mathematical Society*, 16(4):683–686, 1965.

[72] Bahman Kalantari and Leonid Khachiyan. On the rate of convergence of deterministic and randomized ras matrix scaling algorithms. *Operations research letters*, 14(5):237–244, 1993.

[73] Kittipat Kampa, Erion Hasanbelliu, and Jose C Principe. Closed-form cauchy-schwarz pdf divergence for mixture of gaussians. In *The 2011 International Joint Conference on Neural Networks*, pages 2578–2585. IEEE, 2011.

[74] Bahjat Kawar, Michael Elad, Stefano Ermon, and Jiaming Song. Denoising diffusion restoration models. *Advances in Neural Information Processing Systems*, 35:23593–23606, 2022.

[75] 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*, pages 11169–11180, 2019.

[76] Pasha Khosravi, Yitao Liang, YooJung Choi, and Guy Van den Broeck. What to expect of classifiers? reasoning about logistic regression with missing features. In *IJCAI*, pages 2716–2724, 2019.

[77] Pasha Khosravi, Antonio Vergari, YooJung Choi, Yitao Liang, and Guy Van den Broeck. Handling missing data in decision trees: A probabilistic approach. In *Proceedings of The Art of Learning with Missing Values, Workshop at ICML*, 2020.

[78] Diederik Kingma, Tim Salimans, Ben Poole, and Jonathan Ho. Variational diffusion models. *Advances in neural information processing systems*, 34:21696–21707, 2021.

[79] Diederik P Kingma and Prafulla Dhariwal. Glow: generative flow with invertible $1\times$ 1 convolutions. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pages 10236–10245, 2018.

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

[81] 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.

[82] 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)*, pages 1–10, 2014.

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

[84] Ilya Kostrikov, Ashvin Nair, and Sergey Levine. Offline reinforcement learning with implicit q-learning. *arXiv preprint arXiv:2110.06169*, 2021.

[85] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images, 2009.

[86] Aviral Kumar, Aurick Zhou, George Tucker, and Sergey Levine. Conservative q-learning for offline reinforcement learning. *Advances in Neural Information Processing Systems*, 33:1179–1191, 2020.

[87] Yann Lecun, Sumit Chopra, Raia Hadsell, Marc Aurelio Ranzato, and Fu Jie Huang. A tutorial on energy-based learning. *Predicting structured data*, 2006.

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

[89] Sergey Levine, Aviral Kumar, George Tucker, and Justin Fu. Offline reinforcement learning: Tutorial, review, and perspectives on open problems. *arXiv preprint arXiv:2005.01643*, 2020.

[90] Xiang Lisa Li, John Thickstun, Ishaan Gulrajani, Percy Liang, and Tatsunori Hashimoto. Diffusion-lm improves controllable text generation. In *Advances in Neural Information Processing Systems*, 2022.

[91] Yitao Liang and Guy Van den Broeck. Towards compact interpretable models: Shrinking of learned probabilistic sentential decision diagrams. In *IJCAI 2017 Workshop on Explainable Artificial Intelligence (XAI)*, August 2017.

[92] Bill Yuchen Lin, Wangchunshu Zhou, Ming Shen, Pei Zhou, Chandra Bhagavatula, Yejin Choi, and Xiang Ren. Commongen: A constrained text generation challenge for generative commonsense reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1823–1840, 2020.

[93] Anji Liu, Kareem Ahmed, and Guy Van Den Broeck. Scaling tractable probabilistic circuits: A systems perspective. In *International Conference on Machine Learning*, pages 30630–30646. PMLR, 2024.

[94] Anji Liu, Oliver Broadrick, Mathias Niepert, and Guy Van den Broeck. Discrete copula diffusion. In *The Thirteenth International Conference on Learning Representations*, 2025.

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

[96] Anji Liu, Mathias Niepert, and Guy Van den Broeck. Image inpainting via tractable steering of diffusion models. *arXiv preprint arXiv:2401.03349*, 2023.

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

[98] Anji Liu and Guy Van den Broeck. Tractable regularization of probabilistic circuits. In *Advances in Neural Information Processing Systems 34 (NeurIPS)*, 2021.

[99] Anji Liu, Honghua Zhang, and Guy Van den Broeck. Scaling up probabilistic circuits by latent variable distillation. In *The Eleventh International Conference on Learning Representations*, 2022.

[100] Xuejie Liu, Anji Liu, Guy Van den Broeck, and Yitao Liang. Understanding the distillation process from deep generative models to tractable probabilistic circuits. In *International Conference on Machine Learning*, pages 21825–21838. PMLR, 2023.

[101] Xuejie Liu, Anji Liu, Guy Van den Broeck, and Yitao Liang. A tractable inference perspective of offline rl. *Advances in Neural Information Processing Systems*, 37:70953–70980, 2024.

[102] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 10012–10022, 2021.

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

[104] Stuart Lloyd. Least squares quantization in pcm. *IEEE transactions on information theory*, 28(2):129–137, 1982.

[105] Aaron Lou, Chenlin Meng, and Stefano Ermon. Discrete diffusion modeling by estimating the ratios of the data distribution. *arXiv preprint arXiv:2310.16834*, 2024.

[106] Daniel Lowd and Jesse Davis. Learning markov network structure with decision trees. In *2010 IEEE International Conference on Data Mining*, pages 334–343. IEEE, 2010.

[107] Daniel Lowd and Amirmohammad Rooshenas. The libra toolkit for probabilistic models. *Journal of Machine Learning Research*, 16:2459–2463, 2015.

[108] Andreas Lugmayr, Martin Danelljan, Andres Romero, Fisher Yu, Radu Timofte, and Luc Van Gool. Repaint: Inpainting using denoising diffusion probabilistic models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11461–11471, 2022.

[109] Lars Maaløe, Marco Fraccaro, Valentin Liévin, and Ole Winther. Biva: a very deep hierarchy of latent variables for generative modeling. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, pages 6551–6562, 2019.

[110] Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of english: The penn treebank. *Computational Linguistics*, 1993.

[111] Antonio Mari, Gennaro Vessio, and Antonio Vergari. Unifying and understanding overparameterized circuit representations via low-rank tensor decompositions. In *The 6th Workshop on Tractable Probabilistic Modeling*, 2023.

[112] Geoffrey J McLachlan, Sharon X Lee, and Suren I Rathnayake. Finite mixture models. *Annual review of statistics and its application*, 6:355–378, 2019.

[113] Chenlin Meng, Kristy Choi, Jiaming Song, and Stefano Ermon. Concrete score matching: Generalized score matching for discrete data. *Advances in Neural Information Processing Systems*, 35:34532–34545, 2022.

[114] Fabian Mentzer, Eirikur Agustsson, Michael Tschannen, Radu Timofte, and Luc Van Gool. Practical full resolution learned lossless image compression. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10629–10638, 2019.

[115] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

[116] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *International Conference on Learning Representations*, 2022.

[117] Tomáš Mikolov, Ilya Sutskever, Anoop Deoras, Hai-Son Le, and Stefan Kombrink. Subword language modeling with neural networks. *preprint (http://www.fit.vutbr.cz/imikolov/rnnlm/char.pdf)*, 2012.

[118] Thomas P. Minka. Expectation propagation for approximate bayesian inference. In *UAI*, pages 362–369. Morgan Kaufmann, 2001.

[119] Alejandro Molina, Antonio Vergari, Karl Stelzner, Robert Peharz, Pranav Subramani, Nicola Di Mauro, Pascal Poupart, and Kristian Kersting. Spflow: An easy and extensible library for deep probabilistic learning using sum-product networks. *arXiv preprint arXiv:1901.03704*, 2019.

[120] Serafín Moral, Rafael Rumí, and Antonio Salmerón. Mixtures of truncated exponentials in hybrid bayesian networks. In *ECSQARU*, volume 2143 of *Lecture Notes in Computer Science*, pages 156–167. Springer, 2001.

[121] Alex Morehead, Jeffrey Ruffolo, Aadyot Bhatnagar, and Ali Madani. Towards joint sequence-structure generation of nucleic acid and protein complexes with se (3)-discrete diffusion. *arXiv preprint arXiv:2401.06151*, 2023.

[122] Kevin Murphy, Scott Linderman, Peter G Chang, Xinglong Li, Aleyna Kara, Giles Harper-Donnelly, and Gerardo Duran-Martin. Dynamax, 2023.

[123] Roger B Nelsen. An introduction to copulas, 2006.

[124] Alexander Quinn Nichol and Prafulla Dhariwal. Improved denoising diffusion probabilistic models. In *International Conference on Machine Learning*, pages 8162–8171. PMLR, 2021.

[125] Frank Nielsen. Closed-form information-theoretic divergences for statistical mixtures. In *Proceedings of the 21st International Conference on Pattern Recognition (ICPR2012)*, pages 1723–1726. IEEE, 2012.

[126] Tobias H Olsen, Fergus Boyles, and Charlotte M Deane. Observed antibody space: A diverse database of cleaned, annotated, and translated unpaired and paired antibody sequences. *Protein Science*, 31(1):141–146, 2022.

[127] Manfred Opper, Ole Winther, and Michael J Jordan. Expectation consistent approximate inference. *Journal of Machine Learning Research*, 6(12), 2005.

[128] Umut Oztok, Arthur Choi, and Adnan Darwiche. Solving PP^{PP}-complete problems using knowledge compilation. In *Proceedings of the 15th International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 94–103, 2016.

[129] Keiran Paster, Sheila McIlraith, and Jimmy Ba. You can't count on luck: Why decision transformers and rvs fail in stochastic environments. *Advances in Neural Information Processing Systems*, 35:38966–38979, 2022.

[130] Robert Peharz, Robert Gens, and Pedro Domingos. Learning selective sum-product networks. In *LTPM workshop*, volume 32, 2014.

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

[132] 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.

[133] 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.

[134] Krishna Pillutla, Swabha Swayamdipta, Rowan Zellers, John Thickstun, Sean Welleck, Yejin Choi, and Zaid Harchaoui. Mauve: measuring the gap between neural text and human text using divergence frontiers. In *Proceedings of the 35th International Conference on Neural Information Processing Systems*, pages 4816–4828, 2021.

[135] Dean A Pomerleau. Alvinn: An autonomous land vehicle in a neural network. *Advances in neural information processing systems*, 1, 1988.

[136] 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.

[137] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems*, 5:606–624, 2023.

[138] Andrzej Pronobis, Avinash Ranganath, and Rajesh PN Rao. Libspn: A library for learning and inference with sum-product networks and tensorflow. In *Principled Approaches to Deep Learning Workshop*, 2017.

[139] Lawrence Rabiner and Biinghwang Juang. An introduction to hidden markov models. *ieee assp magazine*, 3(1):4–16, 1986.

[140] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners, 2019.

[141] Tahrima Rahman, Prasanna Kothalkar, and Vibhav Gogate. Cutset networks: A simple, tractable, and scalable approach for improving the accuracy of chow-liu trees. In *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2014, Nancy, France, September 15-19, 2014. Proceedings, Part II 14*, pages 630–645. Springer, 2014.

[142] Aditya Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. Hierarchical text-conditional image generation with clip latents. *arXiv preprint arXiv:2204.06125*, 2022.

[143] Marc'Aurelio Ranzato, Y-Lan Boureau, Sumit Chopra, and Yann LeCun. A unified energy-based framework for unsupervised learning. In *Artificial Intelligence and Statistics*, pages 371–379. PMLR, 2007.

[144] Alfréd Rényi et al. On measures of entropy and information. In *Proceedings of the Fourth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Contributions to the Theory of Statistics*. The Regents of the University of California, 1961.

[145] Jorma J Rissanen. Generalized kraft inequality and arithmetic coding. *IBM Journal of research and development*, 20(3):198–203, 1976.

[146] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10684–10695, 2022.

[147] Amirmohammad Rooshenas and Daniel Lowd. Learning sum-product networks with direct and indirect variable interactions. In *International Conference on Machine Learning*, pages 710–718. PMLR, 2014.

[148] Yangjun Ruan, Karen Ullrich, Daniel Severo, James Townsend, Ashish Khisti, Arnaud Doucet, Alireza Makhzani, and Chris J Maddison. Improving lossless compression rates via monte carlo bits-back coding. In *International Conference on Machine Learning*, 2021.

[149] Tamás Rudas. *Lectures on categorical data analysis*. Springer, 2018.

[150] Jeffrey A Ruffolo, Lee-Shin Chu, Sai Pooja Mahajan, and Jeffrey J Gray. Fast, accurate antibody structure prediction from deep learning on massive set of natural antibodies. *Nature communications*, 14(1):2389, 2023.

[151] Ludger Ruschendorf. Convergence of the iterative proportional fitting procedure. *The Annals of Statistics*, pages 1160–1174, 1995.

[152] Prasanna K Sahoo and Palaniappan Kannappan. *Introduction to functional equations*. CRC Press, 2011.

[153] Subham Sekhar Sahoo, Marianne Arriola, Yair Schiff, Aaron Gokaslan, Edgar Marroquin, Justin T Chiu, Alexander Rush, and Volodymyr Kuleshov. Simple and effective masked diffusion language models. *arXiv preprint arXiv:2406.07524*, 2024.

[154] Nimish Shah, Laura Isabel Galindez Olascoaga, Shirui Zhao, Wannes Meert, and Marian Verhelst. Dpu: Dag processing unit for irregular graphs with precision-scalable posit arithmetic in 28 nm. *IEEE Journal of Solid-State Circuits*, 57(8):2586–2596, 2021.

[155] Claude Elwood Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423, 1948.

[156] Yujia Shen, Arthur Choi, and Adnan Darwiche. Tractable operations for arithmetic circuits of probabilistic models. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, pages 3943–3951. Citeseer, 2016.

[157] Jiaxin Shi, Kehang Han, Zhe Wang, Arnaud Doucet, and Michalis K. Titsias. Simplified and generalized masked diffusion for discrete data. In *Advances in Neural Information Processing Systems*, 2024.

[158] Andy Shih and Stefano Ermon. Probabilistic circuits for variational inference in discrete graphical models. In *NeurIPS*, 2020.

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

[160] Andy Shih, Guy Van den Broeck, Paul Beame, and Antoine Amarilli. Smoothing structured decomposable circuits. *Advances in Neural Information Processing Systems*, 32:11416–11426, 2019.

[161] Richard Sinkhorn. A relationship between arbitrary positive matrices and doubly stochastic matrices. *The annals of mathematical statistics*, 35(2):876–879, 1964.

[162] M Sklar. Fonctions de répartition à n dimensions et leurs marges. In *Annales de l'ISUP*, volume 8, pages 229–231, 1959.

[163] Jascha Sohl-Dickstein, Eric Weiss, Niru Maheswaranathan, and Surya Ganguli. Deep unsupervised learning using nonequilibrium thermodynamics. In *International conference on machine learning*, pages 2256–2265. PMLR, 2015.

[164] Yang Song, Prafulla Dhariwal, Mark Chen, and Ilya Sutskever. Consistency models. In *International Conference on Machine Learning*, pages 32211–32252. PMLR, 2023.

[165] Haoran Sun, Lijun Yu, Bo Dai, Dale Schuurmans, and Hanjun Dai. Score-based continuous-time discrete diffusion models. In *The Eleventh International Conference on Learning Representations*, 2022.

[166] Roman Suvorov, Elizaveta Logacheva, Anton Mashikhin, Anastasia Remizova, Arsenii Ashukha, Aleksei Silvestrov, Naejin Kong, Harshith Goka, Kiwoong Park, and Victor Lempitsky. Resolution-robust large mask inpainting with fourier convolutions. In *Proceedings of the IEEE/CVF winter conference on applications of computer vision*, pages 2149–2159, 2022.

[167] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ international conference on intelligent robots and systems*, pages 5026–5033. IEEE, 2012.

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

[169] James Townsend, Thomas Bird, Julius Kunze, and David Barber. HiLLoC: lossless image compression with hierarchical latent variable models. In *International Conference on Learning Representations*, 2019.

[170] 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 32 (NeurIPS)*, 2019.

[171] Linh Tran, Maja Pantic, and Marc Peter Deisenroth. Cauchy-schwarz regularized autoencoder. *arXiv preprint arXiv:2101.02149*, 2021.

[172] Brian L Trippe, Jason Yim, Doug Tischer, David Baker, Tamara Broderick, Regina Barzilay, and Tommi Jaakkola. Diffusion probabilistic modeling of protein backbones in 3d for the motif-scaffolding problem. *arXiv preprint arXiv:2206.04119*, 2022.

[173] Arash Vahdat and Jan Kautz. Nvae: A deep hierarchical variational autoencoder. *arXiv preprint arXiv:2007.03898*, 2020.

[174] Guy Van den Broeck, Anton Lykov, Maximilian Schleich, and Dan Suciu. On the tractability of shap explanations. In *Proceedings of the 35th Conference on Artificial Intelligence (AAAI)*, 2021.

[175] Aaron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu. Neural discrete representation learning. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 6309–6318, 2017.

[176] Jan Van Haaren and Jesse Davis. Markov network structure learning: A randomized feature generation approach. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 26, 2012.

[177] Antonio Vergari, YooJung Choi, Anji Liu, Stefano Teso, and Guy Van den Broeck. A compositional atlas of tractable circuit operations for probabilistic inference. *Advances in Neural Information Processing Systems*, 34:13189–13201, 2021.

[178] Clement Vignac, Igor Krawczuk, Antoine Siraudin, Bohan Wang, Volkan Cevher, and Pascal Frossard. Digress: Discrete denoising diffusion for graph generation. In *The Eleventh International Conference on Learning Representations*, 2022.

[179] Eric Wang, Pasha Khosravi, and Guy Van den Broeck. Probabilistic sufficient explanations. *arXiv preprint arXiv:2105.10118*, 2021.

[180] Yinhuai Wang, Jiwen Yu, and Jian Zhang. Zero-shot image restoration using denoising diffusion null-space model. In *The Eleventh International Conference on Learning Representations*, 2022.

[181] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

[182] Bo Wei and Jerry D Gibson. Comparison of distance measures in discrete spectral modeling. Master's thesis, Citeseer, 2001.

[183] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.

[184] Jingyi Xu, Zilu Zhang, Tal Friedman, Yitao Liang, and Guy Van den Broeck. A semantic loss function for deep learning with symbolic knowledge. In *ICML*, volume 80 of *Proceedings of Machine Learning Research*, pages 5498–5507. PMLR, 2018.

[185] Taku Yamagata, Ahmed Khalil, and Raul Santos-Rodriguez. Q-learning decision transformer: Leveraging dynamic programming for conditional sequence modelling in offline rl. In *International Conference on Machine Learning*, pages 38989–39007. PMLR, 2023.

[186] Mengjiao Yang, Dale Schuurmans, Pieter Abbeel, and Ofir Nachum. Dichotomy of control: Separating what you can control from what you cannot. *arXiv preprint arXiv:2210.13435*, 2022.

[187] Yibo Yang, Stephan Mandt, and Lucas Theis. An introduction to neural data compression. *arXiv preprint arXiv:2202.06533*, 2022.

[188] Lingyun Yao, Martin Trapp, Karthekeyan Periasamy, Jelin Leslin, Gaurav Singh, and Martin Andraud. Logarithm-approximate floating-point multiplier for hardware-efficient inference in probabilistic circuits. In *The 6th Workshop on Tractable Probabilistic Modeling*, 2023.

[189] Fisher Yu, Yinda Zhang, Shuran Song, Ari Seff, and Jianxiong Xiao. Lsun: Construction of a large-scale image dataset using deep learning with humans in the loop. *arXiv preprint arXiv:1506.03365*, 2015.

[190] Zhe Zeng, Paolo Morettin, Fanqi Yan, Antonio Vergari, and Guy Van den Broeck. Scaling up hybrid probabilistic inference with logical and arithmetic constraints via message passing. In *ICML*, volume 119 of *Proceedings of Machine Learning Research*, pages 10990–11000. PMLR, 2020.

[191] Guanhua Zhang, Jiabao Ji, Yang Zhang, Mo Yu, Tommi Jaakkola, and Shiyu Chang. Towards coherent image inpainting using denoising diffusion implicit models. *arXiv preprint arXiv:2304.03322*, 2023.

[192] Honghua Zhang, Meihua Dang, Nanyun Peng, and Guy Van den Broeck. Tractable control for autoregressive language generation. In *International Conference on Machine Learning*, pages 40932–40945. PMLR, 2023.

[193] Richard Zhang, Phillip Isola, Alexei A Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 586–595, 2018.

[194] Lingxiao Zhao, Xueying Ding, Lijun Yu, and Leman Akoglu. Improving and unifying discrete&continuous-time discrete denoising diffusion. *arXiv preprint arXiv:2402.03701*, 2024.

[195] Zhilin Zheng and Li Sun. Disentangling latent space for vae by label relevant/irrelevant dimensions. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12192–12201, 2019.

[196] Mingyuan Zhou, Huangjie Zheng, Zhendong Wang, Mingzhang Yin, and Hai Huang. Score identity distillation: Exponentially fast distillation of pretrained diffusion models for one-step generation. In *International Conference on Machine Learning*, 2024.

[197] Zachary Ziegler and Alexander Rush. Latent normalizing flows for discrete sequences. In *Proceedings of the 36th International Conference on Machine Learning (ICML)*, 2019.