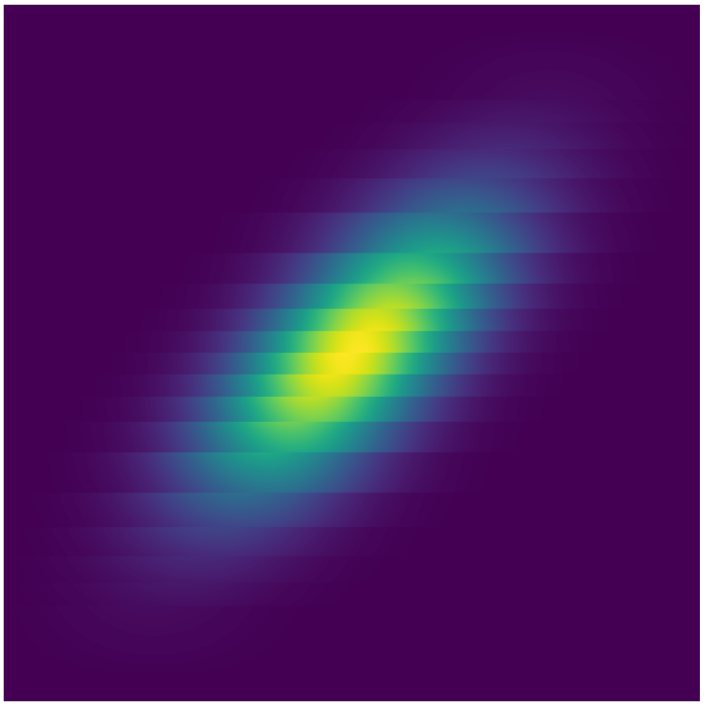
UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE



Bachelor thesis

Compiling Bayesian networks with continuous support into Sum-Product networks

Theo Rüter Würtzen

Advisor: Sebastian Weichwald

June 8, 2023

Abstract

Sum-Product networks (SPNs) are a deep architecture capable of modeling joint distributions supporting tractable exact integration. Bayesian networks (BNs) allow for direct modeling of independencies and relationships between variables, but integrating them (say, to normalize) is generally intractable. In this thesis, we propose an approach of compiling BNs into SPNs, that is, capturing the distribution of one model in another to leverage the advantages of both frameworks. By doing so we achieve tractable integration in a SPN that simultaneously captures a BN's relationships and independencies.

Contribution

Our work focuses on investigating and motivating a methodology for compiling Bayesian networks with the structure $X \to Y$ into SPNs. We then present the compilation of complex yet tractable linear Gaussian graphical probabilistic models into SPNs, providing a comparative analysis of the resulting representations. Following these results we present a generalized algorithm and framework for compiling *any* BN with continuous variables bounded to a finite interval into a SPN. We conduct a qualitative and quantitative evaluation, analyzing the procedure's effectiveness and limitations. Finally, we demonstrate promising future research directions. The Python implementation of our approach, built on the open-source SPN library `SPFlow`[Mol+19], is released for public use[1].

Prerequisites and Notation

Notation. $\mathcal{N}(x; \mu, \sigma^2)$ denotes the PDF $p_X(x)$ of the normally distributed RV X with mean μ and standard deviation σ^2. $\mathcal{U}(x; (a, b])$ denotes $p_X(x)$ of the uniformly distributed RV X with support $(a, b]$ and density $1/(b-a)$. A conditional distribution $p_{Y|X}(y|x)$ might also be written as $p_{Y|X=x}(y)$, to make explicit that the pdf of y is "selected" by x.

Bayesian Networks and the Linear Gaussian Probabilistic Graphical Model

Bayesian Networks (BNs) are a type of probabilistic graphical model used to represent and reason about relationships between random variables. In a BN, variables are represented as nodes in a directed acyclic graph (DAG), the I-map, from which all independence statements can be derived. If the I-map characterizes all [in]dependencies we call it faithful. Nodes are associated with the conditional probability distribution of their variable given their parents which parameterize the distribution. Combined, the nodes and directed edges represent a factorization of the joint probability of all included variables. Different inference algorithms exist (variable elimination, message-passing, recursive conditioning), but

[1] https://github.com/Whatisthisname/Bayesian-network-to-SPN

for an arbitrary BN integrals over varying domains become intractable. See chapter 3 of [KF] for an in depth coverage.

The **Linear Gaussian Probabilistic Graphical Model** (LGPGM) is an instantiation of a BN in which all conditional distributions are gaussian with their means a linear function of their parents and the variance remains fixed. Under these restrictions the multivariate normal distribution and the LGPGM are equivalent, which implies closed-form marginal distributions in the LGPGM, a useful property to aid in verification of compilation correctness making it a good candidate with which to evalute the compilation procedure (done later). Consult the very first part of example 0.2 for a simple LGPGM and for an in-depth coverage see chapter 7 of [KF].

Sum-Product Networks

SPNs [PD11] are probabilistic models that directly encode how to evalute the pdf of their represented distributions. They can be represented as computational graphs built from three types of operations (nodes/units): Sum-units, Product-units and Distribution-units. Every unit is associated with a scope, and the subtree rooted at it is itself a normalized PDF over the variables in its scope. A composition of sums and products lets them represent expressive deep factorized mixture models. Limiting the joint distribution to be composed of products and weighted sums of simple distributions is what enables tractable integration. If the composition respects certain restrictions, inference queries can be performed with computational cost proportional to the amount of nodes in the DAG.

Definition 0.1. ***Distribution-unit***
Distribution units (or Leaf units) are the fundamental building block of SPNs, they are what define the support and scope of the distribution. They tractably ($\mathcal{O}(1)$) support point evaluations, integrals, and moments and MAP. In pseudocode or when the exact distribution is left unspecified we denote it as `Leaf`*(scope). In this report the relevant leaf nodes are uniforms, slopyforms*[2] *and gaussians, all of which have scope of a single variable*[3]*, and all of which we can integrate and calculate moments for tractably. See the leftmost of fig 1 and 2 for an example.* ∘

Definition 0.2. ***Product-unit***
The product-unit computes a product of its children, which lets SPNs represent a joint distribution. In pseudocode, we denote it as `Product`*(factors). The scope of a Product-unit is the union of the scopes of its children. See the center of 1 and 2 for an example.* ∘

[2]See appendix for the full definition of this distribution.

[3]The general theory and algorithms built on the SPN framework is not subject to the single variable scope limitaiton. This report refrains from multi-variable leaves as you could in theory make one big leaf to capture the entire joint distribution (in particular a multivariate gaussian), after which we would have offloaded all the integration problems to our leaf and are no longer using the SPN framework. To generalize to any continuous joint distribution we refrain from doing this.

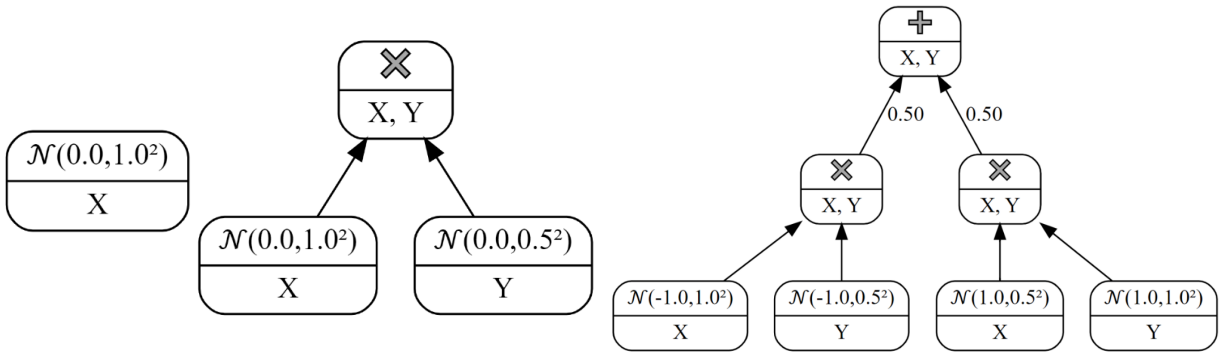


Figure 1: Three example SPN DAGs, showing the structure of the expressions (from left to right) of
1) $M(x) = \mathcal{N}(x; 0, 1^2)$
2) $M(x, y) = \mathcal{N}(x; 0, 1^2) \cdot \mathcal{N}(y; 0, 0.5^2)$
3) $M(x, y) = 0.5 \cdot \mathcal{N}(x; -1, 1^2) \cdot \mathcal{N}(y; -1, 0.5^2) + 0.5 \cdot \mathcal{N}(x; 1, 0.5^2) \cdot \mathcal{N}(y; 1, 1^2)$

Definition 0.3. ***Sum-unit***
The sum-unit, with some n children, assigns each child a weight w_i, for which $\sum_{i=1}^{n} w_i = 1$, which lets SPNs represent mixtures. We denote it as `Sum`*(summands, weights). The scope of a sum-unit is the union of the scopes of its children. See the rightmost of fig 1 and 2 for an example.* $\circ$

Structural Constraints and Tractable Queries
In probabilistic models, intractability often arises when computing integrals, even in low-dimensional cases. Computing $\int_S p(x_{1:n})\, \mathrm{d}x_{1:n}$ for a joint probability density $p(x_{1:n})$ and a contiguous subspace $S^n \subseteq \mathbb{R}^n$ is generally intractable. Marginalizing variable x_i (where $S_i = \mathbb{R}$) shows the relevance of this query. However, by imposing two structural constraints on SPNs, we can ensure tractable (linear in $|V|$) computations for such queries: *Completeness* holds when every mixture component of every `Sum` in an SPN has the same scope, and *Decomposability* ensures that all factors of every `Product` have disjoint scopes. These two properties allow for commuting integrals into `Sum`s and pulling independent factors out of integrals over `Product`s. Intuitively, we recursively "push" the integrals all the way down into the `Distribution`-units which can finally exactly and tractably compute them. Satisfying *completeness* and *decomposability* is both necessary and sufficient to enable tractable computation of marginals, conditional queries, and moments. Another query, $\arg\max_{x_{1:n}} p(x_{1:n})$, can be computed tractably under the additional *determinism* constraint. *Determinism* requires that the supports of components within each `Sum` in an SPN must be disjoint. This constraint effectively allows the max operator to commute with sums, similar to integrals. SPN inference algorithms are conveniently implemented recursively, with `Distribution`-units serving as the base case. For a comprehensive overview of SPNs and algorithms, refer to [CVB].

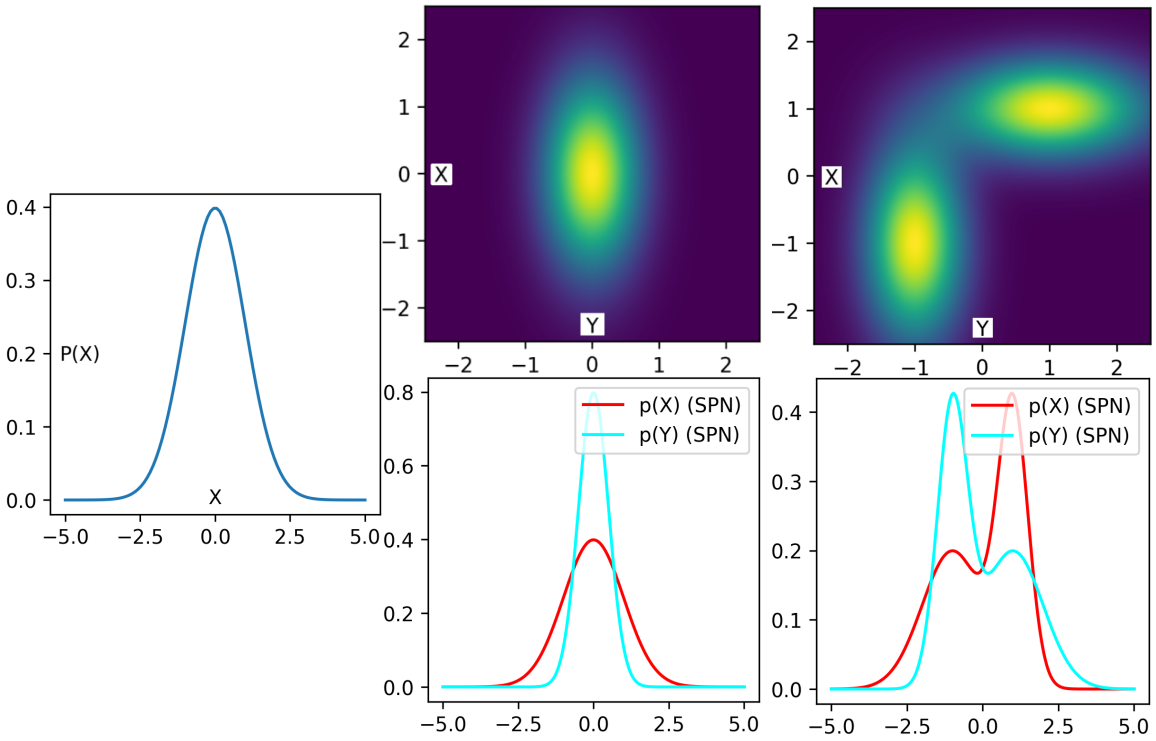


Figure 2: (Left to right) the corresponding densities (and marginals densities below) of **1)**, **2)**, and **3)** from fig. 1. Colors of the heatmaps have been scaled so 2) and 3) are not directly comparable.

For the purposes of tractable integration, we require that the our final SPN is *smooth* and *decomposable*, which can be enforced by adhering to the following substitution rules:

Definition 0.4. ***Smooth and decomposable SPN grammar*** $\mathcal{G}$.
Let $x_{\boldsymbol{A}}$ *denote the vector* x *indexed by set* $\boldsymbol{A}$. *Members of the grammar* $\mathcal{G} =$

$$\begin{aligned} S \rightarrow\ & M(\boldsymbol{x}_{1:n}) \\ M(\boldsymbol{x}_A) \rightarrow\ & \texttt{Leaf}(\boldsymbol{x}_A) \\ & \mid \sum_{i=1}^{k} \theta_i \cdot M(\boldsymbol{x}_A) \qquad \textit{and} \sum_{i=1}^{k} \theta = 1 \\ & \mid \prod_{P_i \in P} M(\boldsymbol{x}_{P_i}) \qquad \textit{for } P \in \textit{Partitions}(A) \end{aligned}$$

are smooth and decomposable SPNs over the RVs $x_1, ..., x_n$. *Member-functions in this grammar will be denoted* M. $\circ$

Members of G will support tractable integration. Exactly compiling a BN into a SPN for effective inference is essentially asking two questions: Does M exist such that $P(x_1) \cdot ... \cdot P(x_n|x_{1:n-1}) = M(x_{1:n}) \in \mathcal{G}$ and how do we find it?

The Approach

Compiling a simple LGPGM

The Bayesian network faithfully represented by $X \rightarrow Y$ will be our initial explanatory example in this section. This structure explicitly factors the joint pdf into $p_{X,Y}(x, y) = p_X(x) \cdot p_{Y|X}(y|x)$. X and Y are assumed to be continuous, and the distribution deliberately left unspecified.
As we restrict ourselves to leaves of single scope it will be quickly become clear that we cannot generally construct $M \in \mathcal{G}$ such that $M(x, y) = p_X(x) \cdot p_{Y|X}(y|x)$ as the right hand side of the product will always depend on both x and y whereas the left only on x. This is problematic because $\mathcal{G}$ requires disjoint scope in products (*decomposability).* This implies that we cannot *exactly* capture any *interesting*[4] joint distribution, because that would imply overlapping scope between at least two of the terms. We will therefore have to make do with an approximate compilation. To move forward, notice that *decomposability* implies that, in the distribution given by a subtree rooted at a `Product`, the children will be independent. We must therefore use `Sums` to capture the dependecies between variables. We will motivate the chosen approach from the following equality, where $\delta_a(x)$ denotes the

[4]Interesting means there is at least one dependence between variables. If they are all independent, the I-map of the BN would be fully disconneted and the SPN would just be a product of all the nodes, $M(x_{1:n}) = \prod_{i=1}^{n} \text{pdf}_{x_i}(x_i)$

dirac delta centered at a:

$$p_X(x) \cdot p_{Y|X=x}(y) = \int_{\text{Domain}(X)} \delta_t(x) \cdot p_X(t) \cdot p_{Y|X=t}(y) \, \mathrm{d}t$$

Equality holds by the definition of $\delta_a(x)$. This expression is not yet directly compatible with SPNs because of the integral. We can however approximate the integral with a sum-unit and replace the delta with a weighted indicator function $\mathbb{1}$. This gives the following model $M \in \mathcal{G}$, for some partition P of $\text{Domain}(X)$.

$$\approx M(x,y) = \sum_{(a,b] \in P} \frac{\mathbb{1}(a < x \leq b)}{b-a} \cdot \frac{p_X\left(\frac{a+b}{2}\right)}{Z} \cdot p_{Y|X=\frac{a+b}{2}}(y) \quad \text{for } Z = \sum_{(a,b] \in P} p_X\left(\frac{a+b}{2}\right)$$

Since $\frac{\mathbb{1}(a<x\leq b)}{b-a} = \mathcal{U}(x;a,b)$, we can rephrase this as

$$= \sum_{[a,b] \in P} \underbrace{\frac{p_X\left(\frac{a+b}{2}\right)}{Z}}_{\text{normalized weight}} \cdot \underbrace{\mathcal{U}(x;a,b)}_{\text{Bin } \texttt{Leaf} \text{ with scope } x} \cdot \underbrace{p_{Y|X=\frac{a+b}{2}}(y)}_{\texttt{Leaf} \text{ with scope } y} \tag{1}$$

The above expression is in $\mathcal{G}$ as it is a `Sum` of `Products` of independent `Leaf` units, fulfilling *completeness* and *decomposabiity*. Additionally, only one of the terms in the `Sum` will ever be nonzero, which fulfills *determinism*[5]. As we increase the granularity of our partition P, we get a more accurate[6] model, as we better approximate the integral (think Riemann integral), because $a \to b \implies \mathcal{U}(x;a,b) \to \delta_b(x)$[7]. Intuitively, what we have done is quantized and truncated the outcomes of X into nonoverlapping bins[8]. We can now, instead of conditioning on the exact taken value of X, know that it has fallen inside a specific bin and assume it fell in the middle of it.

Example 0.1. *In pseudocode eq.* 1 *is doing the following:*

$$M(x,y) = \begin{cases} \textit{for each interval (a,b]} \in P\textit{:} \\ \quad \textit{if } x \in \textit{(a,b]:} \\ \quad\quad c \leftarrow \textit{expected value of item in bin} = (a+b)/2 \\ \quad\quad w \leftarrow \frac{1}{Z \cdot (b-a)} \\ \quad\quad \textit{return } w \cdot p_X(c) \cdot p_{Y|X=c}(y) \end{cases}$$

Essentially eq. 1 is constructing a decision tree, using the graph of the BN. The same general idea applies when we scale the procedure to complex BNs. ∘

[5]Not only desirable for tractable MAP, but also for a later analysis of approximation error bounds.

[6]See **Partitioning Procedure**

[7]In the limit, the uniform distribution will go from an interval-indicator to a point-indicator as the width shrinks.

[8]Bins is to be thought of the same way as bins is used in a histogram. The procedure of dividing a function into bins is implemented in the `Mixify` algorithm 1.

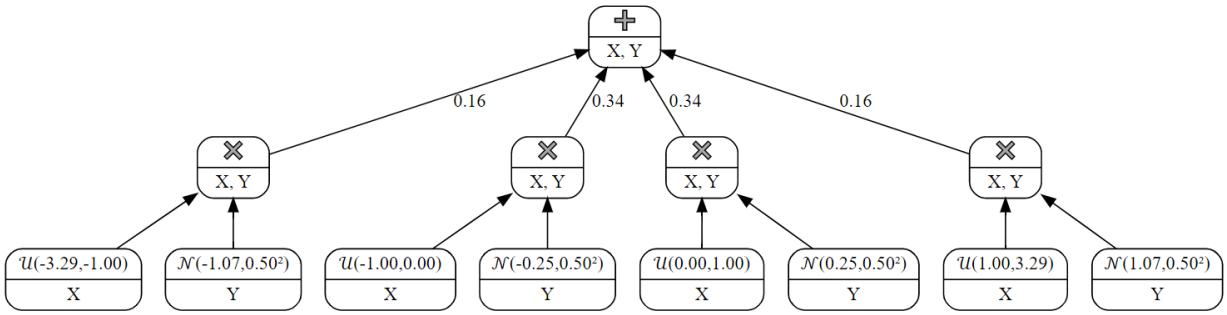


Figure 3: The SPN of example 0.2. The bottom of each node is marked with its scope, so you may verify that this SPN belongs to $\mathcal{G}$ (*decomposability, completeness*) and additionallyfulfills *determinism*.

In the above motivation we used uniform distibutions to represent our bins to keep things simple. We are however not limited to this, but can also internally weigh the bins by having varying density within. To further this idea, we develop the slopyform distribution $\mathcal{S}$ (see appendix for full definition), which generalizes the uniform distribution to include a constant slope, allowing us to fit the derivative of the distribution as well.
The only assumption used in eq. 1 is that each interval is bounded. `Sums` are limited to finite mixtures[9], and a finite amount of bounded intervals cannot partition an infinite domain, so SPNs compiled with eq. 1 cannot represent distributions with infinite support (e.g. gaussian). In case our BN is a LGPGM we can resort to symmetrically truncating the tails of each conditional distribution, removing $p/2$ probability mass on each side, with p chosen as some small positive number. We define this shrunk renormalized version as the truncated gaussian with PDF

$$\mathcal{N}^p(x;\mu,\sigma^2) = \begin{cases} \frac{\mathcal{N}(x;\mu,\sigma^2)}{1-p} & x \in \left(\mathrm{CDF}_{\mathcal{N}}^{-1}(p/2), \mathrm{CDF}_{\mathcal{N}}^{-1}(1-p/2)\right] \\ 0 & \text{otherwise} \end{cases}$$

Example 0.2. *We will demonstrate a simple LGPGM compiled using the ideas of eq. 1. Let our network be $X \to Y$ with factorized joint distribution given as:*

$$p(x,y) = \mathcal{N}(x;0,1^2) \cdot \mathcal{N}(y;x,0.5^2) = \mathcal{N}\left(\begin{bmatrix} x \\ y \end{bmatrix}; \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1, & \sqrt{2}^{-1} \\ \sqrt{2}^{-1}, & 1 \end{bmatrix}\right)$$

For our truncation, we choose $p = 10^{-3}$. Partitioning both conditional distributions into the intervals (offset from their μ) {(-3.29,-1], (-1, 0], (0, 1], (1, 3.29]} we can construct the SPN below, following eq. 1, here using uniforms for simplicity. See the resulting DAG and density in fig 3 and 4. ∘

[9]See a discussion of this in **Future work**.

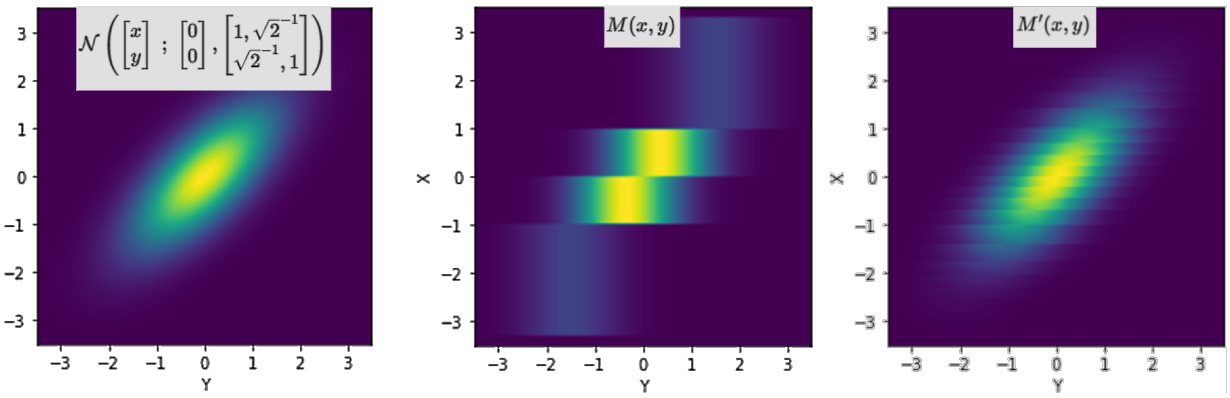


Figure 4: We show the densities to give a qualitative comparison. Left is the true non-truncated density, the center shows the density represented by the constructed SPN from the example, and right shows the finer approximation of a second SPN $M'(x, y)$, which uses 20 components and slopyforms[11].

Partitioning procedures

We give three distinct partitioning procedures and compare their approximation abilities qualitatively and quantitavely. Algorithm 1, `Mixify`[12], takes in a distribution and returns weights and parameters applicable to fit a slopyform (or uniform[13]). The default (and simplest) implementation, `even partition`, takes some number n and returns n intervals of equal width and their weights. The last two procedures, `relative error` and `absolute error`, parametrized by ε require knowledge of critical points and inflection points of the conditional distribution. Respectively, they then output parameters for mixture M for which these hold:

$$\varepsilon > 1, \forall x : \; g(x)\varepsilon^{-1} \leq M(x) \leq g(x)\varepsilon$$

$$or \quad \varepsilon > 0, \forall x : |g(x) - M(x)| < \varepsilon$$

where g is the pdf of the target distribution. This property is proved to hold by construction by careful selection of partitions and component weights, see the appendix for the case where g is a gaussian. For a qualitative comparison, see fig. 7 and 8 showing the fit to a gaussian. For a quantitative comparison, we fit SPNs to gaussians and compare KL divergences, see fig. 6. Experiments point towards slopyform bins being more expressive and requiring less components for a better fit. Uniforms bins can be recovered by setting the slope parameter to 0 in cases where the derivative is undefined.

[11] constructed with `absolute error` bound with tolerance $\varepsilon = 0.005$ as described in **Partitioning procedures** and the appendix.

[12] Named as such because it turns a pdf into a mixture and is short and catchy.

[13] One can discard the slope parameter and just use the interval and weight for a uniform distribution instead.

Algorithm 1 `Mixify`

Require: pdf of `node` in `BN`, amount of components n (or parameter ε)
Ensure: All parents of `node` must be observed or `node` must be a root.

$$\left.\begin{array}{lll} \text{bins} & \leftarrow n \text{ intervals of domain partition} & \\ \text{widths} & \leftarrow \text{width of each bin} & \\ \text{means} & \leftarrow \text{mean of each bin} & \\ \text{density}_i & \leftarrow \text{pdf}(\text{means}_i), & i \in \{1,...,n\} \\ \text{weights}_i & \leftarrow \frac{\text{width}_i \cdot \text{density}_i}{\langle \textbf{width}, \textbf{densities} \rangle}, & i \in \{1,...,n\} \end{array}\right\} \begin{array}{l} \text{These can also be determined by} \\ \texttt{relative error} \text{ or } \texttt{absolute} \\ \texttt{error} \text{ partitioning procedure if} \\ \text{given } \varepsilon \text{ and locations of critical} \\ \text{and inflection points of pdf} \end{array}$$

$$\text{slopes}_i \leftarrow \text{pdf}'(\text{means}_i)/\text{weights}_i,\ i \in \{1,...,n\}$$

return $(\text{intervals}_i, \text{weights}_i, \text{slopes}_i), i \in \{1,...,n\}$

Compiling a complicated LGPGM

Eq. 1 only covers the conversion of the simplest nontrivial BN structure. V-structures and multiple children in a the BN complicate the matter and require additional bookkeeping, for which we implement a `BN` struct in code.

Definition 0.5. *`BN`*
`BN` represents the graph of a BN, each node being associated with a variable, its conditional distribution given the parents, and the associated (possibly by truncation) finite domain[14]*. `BN` tracks given evidence as `evi`, involved variables of which are directly marginalized out of the network and its scope reduced*[15]*. The function `cc`(n) returns the set of lists of all the reachable nodes from n, ignoring directedness and not traversing nodes $\in$ `evi`. By construction of `evi`, for every A and B within every returned list we have $A \not\perp\!\!\!\perp B \mid$ `evi`. The property `roots` returns a list of roots for each connected components in `BN` in case the BN consists of disjoint graphs. `cc`(n) and `roots` can be implemented using standard graph algorithms.* $\circ$

Example 0.3. *`roots` and `cc`*
`BN`$(A, Y, Z|X = x)$ representing $A \leftarrow X \rightarrow Y \rightarrow Z$ will return $\{[A], [Y, Z]\}$ upon a call to `cc`(X), but only $\{[Y, Z]\}$ for `cc`(Y) or `cc`(Z), as it will not traverse X if not starting there. If `evi`$=\emptyset$, then all inputs to `cc` return $\{[X, A, Y, Z]\}$.
This same `BN` will return $\{[A], [Y]\}$ upon accessing `roots`, but just $\{[X]\}$ if `evi`$=\emptyset$. $\circ$

[14]The truncation can be dynamic and a function of the parents as well. See the examples of the generalized BN $\rightarrow$ SPN mapping in the appendix.

[15]This is tractable and exact because we only provide evidence to nodes whose marginal distribution is already known, e.g. they have no parents or their parents are also already observed.

Because we are for now concerned with the simpler LGPGM, we develop a specialization of `BN`, `LGPGM`, which uses the closed-form marginalisation of multivariate gaussians to keep track of the exact marginal distributions given the conditioned data. This proves useful when comparing the SPN density visually.

Definition 0.6. *`LGPGM`*
This object inherits the traits of `BN` and additionally keeps track of the covariance matrix and mean vector of the joint distribution, taking into account evidence given. It essentially doing exact inference in the background to compute marginal distributions, which are useful to validate that the compilation procedure works as expected. ∘

The complete compilation function $C : \texttt{BN} \to \mathcal{G}$ is defined recursively, and is given below in mathematical notation with comments. For the equivalent pseudocode see the appendix.

$$
\begin{aligned}
C\Big(\texttt{BN}(x_{\mathbf{A}}|\texttt{evi})\Big) \;=\; & \prod_{R\in\texttt{roots}} \texttt{Product over the lists of roots of connected components.} \\
& \sum_{((a,b],w,s)^{|R|}\in\bigtimes_{r\in R}\texttt{Mixify}(r)} \texttt{Mixture (Sum) with one component per element in cartesian product of bins of roots. [16]} \\
& \prod_{i=1}^{|R|}\Big(w_i\cdot\mathcal{S}(R_i;s_i,(a_i,b_i])\Big) \texttt{Quantizing the outcome of the roots into a bin} \in\mathbb{R}^{|R|} \texttt{ of Leaves.} \\
& \cdot\; C\Big(\texttt{BN}\left(x_{\texttt{cc}(R)\setminus R}\Big|\texttt{evi}\cup\left\{r_i=\mathop{\mathbb{E}}_{x\sim\text{bin}_i}(x)\right\}_{i=1}^{|R|}\right)\Big) \texttt{Adding } R \texttt{ to evi and reducing scope.}[17]
\end{aligned}
$$

This is a generalization of eq. 1 which only worked for BNs with structure $X \to Y$[18]. See the appendix for an example of a complex LGPGM compiled to SPN using this procedure.

Error-bounds on the compiled LGPGMs

If we compile LGPGM $P(x_{1:n})$ to a SPN $M(x_{1:n})$ with the `relative error` criterion and tolerance $\varepsilon_i > 1$ for each node, the resulting SPN joint distribution also has bounded

[18]Why is it a generalization? In eq. 1 we were working with the BN $X \to Y$. Follow the definition of C step by step: A call to `roots` in this network will return $\{[X]\}$, so just one element, $[X]$, so the product disappears. In the sum, as the cartesian product over just one element (`Mixify`(X)) is the element itself, we have reconstructed the sum of eq. 1, over all the disjoint intervals of the domain of X, the associated normalized weight of which is computed in the `Mixify` procedure. The product over $|R|$ disappears again, leaving just the weight times the chosen leaf node as in eq. 1 (recover uniform by forcing $s_i = 0$). At last, for the recursive call to build the pdf of Y, see the footnote linked to the sum.

[18]If `BN` is `LGPGM` and $|R| = 1$ with no children, replace this `Sum` with an appropriate gaussian `Leaf`. There is no reason to unnecessarily truncate and mixture-ize, as it blows up the amount of nodes in SPN.

[18]We can freely marginalize the `BN` to be only over the connected components by simply "chopping off" what's not connected in the DAG.

relative error:

$$\prod_{i=1}^{n} \varepsilon_i^{-1} \leq \frac{M(x_{1:n})}{P(x_{1:n})} \leq \prod_{i=1}^{n} \varepsilon_i$$

A tighter variant on the marginal joint distribution holds if leaves of the BN are marginalized out. Leaves in the BN can be removed in the SPN removed because nothing depends on them, resulting in fewer quantization steps and less possible error. The corresponding ε_i terms in the bound will disappear as the factorized joint distribution will contain less factors.
One can efficiently and exactly evaluate the full joint density for any input in any BN so these current pointwise `relative` or `absolute error` bounds on the SPN by themselves are nothing but informative and reassuring.

Going from LGPGMs to continuous BNs

We can generalize the procedure to support conversion of any BN over continuous variables to a SPN. `Mixify` supports arbitrary conditional distributions if provided a bound on the domain, which was easily to define in the gaussian case. Evenly partitioning that same domain gives promising results in practice (fig. 6), but at a loss of guarantees[19]. See fig. 9 for some examples of some evenly `Mixified` functions. Because `Mixify` includes a normalization step, the passed functions do not have to be so themselves. A nonnegative function on a bounded domain is a valid input. See the appendix for multiple examples of BNs compiled to SPNs using this procedure.

Future work

Developing bounds on the error of `even partition`

One might be able to derive a bound given a Lipschitz constant of the function when using uniforms or, in the slopyform case, if given a Lipschitz constant on its derivative. Symbolic solvers might automatically be able to determine this for some functions but not in general, so this direction was not further explored.

Developing bounds on the integration error

Care has been taken to ensure that integrals over the entire domain of a variable always evaluates to 1 by renormalizing weights, but the integration error over subsets of the domain

[19]With the LGPGM, we knew that all conditional distributions were gaussian and we used the fact that we knew about the placement of the critical and inflection points to propose a subdividable partition of which we could determine the maximum error of. For arbitrary conditional distributions, we know of no analytic way of determining where these sections are, so these partitioning methods with guarantees do not carry over. We therefore resort to the even partition, for which a bound may also be derivable. See **Future work** for a further discussion on this.

has not been analyzed. The bounds on pointwise (conditional) densities derived in this report may aid in further analysis on the integration error.

Further implications of using slopyforms over uniforms

Using slopyforms instead of uniforms results in the SPN density $M(x_{1:n}) = C(\texttt{BN}(x_{1:n}))$ having nonzero gradient w.r.t. the distributional parameters θ[20]. To briefly explore ramifications of this, we implement algorithm 1 in PyTorch[Pas+19] and `Mixify` a univariate gaussian, computing the slope using automatic differentation while retaining the computation graph of the slope. When computing densities of given samples, we then backpropagate and successfully nudge the parameters (μ, σ) to maximize the likelihood. See fig. 10. With a differentiable compilation we can optimize the parameters of a general deep BN with respect to some differentiable objective functions:

- Joint, marginal, and conditional likelihood: We can now integrate out any variables completely or partially to attain the joint, marginal, or conditional distributions which will still be differentiable w.r.t. the network parameters. We can optimize the density of ε-hypercube around inputs to make the SPN robust to perturbations in the input. With a differentiable CDF we can optimize for specific locations of the percentiles of the SPN.

- Reparameterization trick[KW14]: We can optimize over $\mathbb{E}_{\mathbf{X}\sim\text{SPN}(\cdot)}[f(\mathbf{X})]$ for some differentiable f if can invert the CDF and thereby transform samples from $\mathbf{U} \sim \mathcal{U}(x_i; 0, 1)^n$ into $\text{CDF}^{-1}(\mathbf{U}) \sim \text{SPN}(\cdot)$ using the reparameterization trick. In the case of a mixture with components with disjoint support, CDF^{-1} can be derived in closed form, while, for an arbitrary mixture, one could potentially devise an on-the-fly inversion.

Lazy construction and evaluation of SPNs

Optimizing through differentiable compilation with the current procedure using the `SPFlow` framework is prohibitively slow for two main reasons: Firstly, SPNs have to be fully constructed and represented in memory before being queried. After the parameters of the underlying BN are repeatedly nudged the SPN has to be fully rebuilt again from scratch. `Mixify` becomes a computationally heavy process as the desired accuracy is increased. Secondly, `PyTorch` constructs a large computational graph in the background to enable subsequent backpropagation, which requires a lot of bookkeeping, as mentioned in fig. 10. To alleviate these issues, we propose an alternative take on SPNs: constructing them on the fly (lazy evaluation) and short-circuiting of `Sum`-units when we have *determinism*. With *determinism* the amount of branching per sum node is limited to the amount of inputs[21].

[20] This could be $\mu_{1:n}, \sigma_{1:n}$, and parent weights in the case of a LGPGM.

[21] One query (for simplicity assuming we are querying the joint distribution) can only fall in a single bin per `Sum`.

The remaining mixture components can effectively be fully ignored, never to be explicitly constructed as they would evaluate to 0. Informally, the SPN only has to exist in the places of the DAG that are important for the query. If constructing subtrees of the SPN ad hoc, the precision of the SPN is no longer memory bound by the width of the DAG but only by the depth. One immediate issue with lazy construction is that all components need to be constructed before the normalization factor can be determined. But if one can evaluate the antiderivative of the conditional distribution p, the bin $(a, b]$ can be exactly weighted by $\int_a^b p(x)\,\mathrm{d}x$. This is different from the current approach, which tries to fit the density at the center of the bin and distributes the resulting normalization-error across all bins. Taking it one step further, lazy evaluation of a *deterministic* `Sum` allows it to implicitly have an countably infinite amount of mixture components, which allows us to represent distributions with support on $\mathbb{R}$ using finite support slopyforms. This would however introduce another problem as exact marginalization in SPNs is tractable only because we can turn a complicated integral into a finite sum of simple integrals. With infinite mixture components, this becomes a prohibitive infinite sum of simple integrals.

Conclusion

We have explored the compilation of BNs into SPNs to combine the advantages of both frameworks. Our conceptually simple compilation procedure shows results that respect the independencies and relationships encoded in the BN. The resulting SPN satisfies structural constraints *decomposability, completeness*, and *determinism* which guarantuee tractable integration and MAP. We introduce the slopyform to bestow the SPN with a differentiable density, which gives better representations with fewer nodes and enables gradient-based optimiziation of interesting objective functions.

[PD11] Hoifung Poon and Pedro Domingos. "Sum-Product Networks: A New Deep Architecture". In: *Computer Science & Engineering* (2011).

[KW14] Diederik P Kingma and Max Welling. "Auto-Encoding Variational Bayes". In: *International Conference on Learning Representations*. 2014. URL: `https://arxiv.org/abs/1312.6114`.

[Mol+19] Alejandro Molina et al. *SPFlow: An Easy and Extensible Library for Deep Probabilistic Learning using Sum-Product Networks*. 2019. eprint: `arXiv:1901.03704`.

[Pas+19] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[CVB] YooJung Choi, Antonio Vergari, and Guy Van den Broeck. *Probabilistic Circuits: A Unifying Framework for Tractable Probabilistic Models*.

[KF] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques*. Adaptive computation and machine learning. ISBN: 978-0-262-01319-2.

Appendix

The slopyform distribution

The slopyform family of distributions is a generalization of the family of uniform distributions. Within $(a, b]$ it has constant derivative s with $|s| \leq \frac{1}{2(b-a)^2}$ to ensure nonnegativity. Setting $s = 0$ we recover the uniform distribution. Let $X \sim \mathcal{S}((a, b], s)$. Then:

$$p_X(x) = \begin{cases} (x-a) \cdot s + \frac{1-\frac{1}{2}s\cdot(b-a)^2}{b-a} & a < x \leq b \\ 0 & \text{otherwise} \end{cases}$$

$$\mathbb{E}[X] = \frac{a+b}{2} - \frac{(a-b)^3}{12}s \qquad \text{Var}[X] = \frac{(b-a)^3}{6}s$$

$$P(X < x) = \begin{cases} 0 & x \leq a \\ \frac{(s\cdot(b-x)\cdot a+2-b\cdot(b-x)\cdot s)\cdot(a-x)}{-2\cdot b+2\cdot a} & \text{otherwise} \\ 1 & b < x \end{cases}$$

See figure 5 for some pdfs and cdfs.

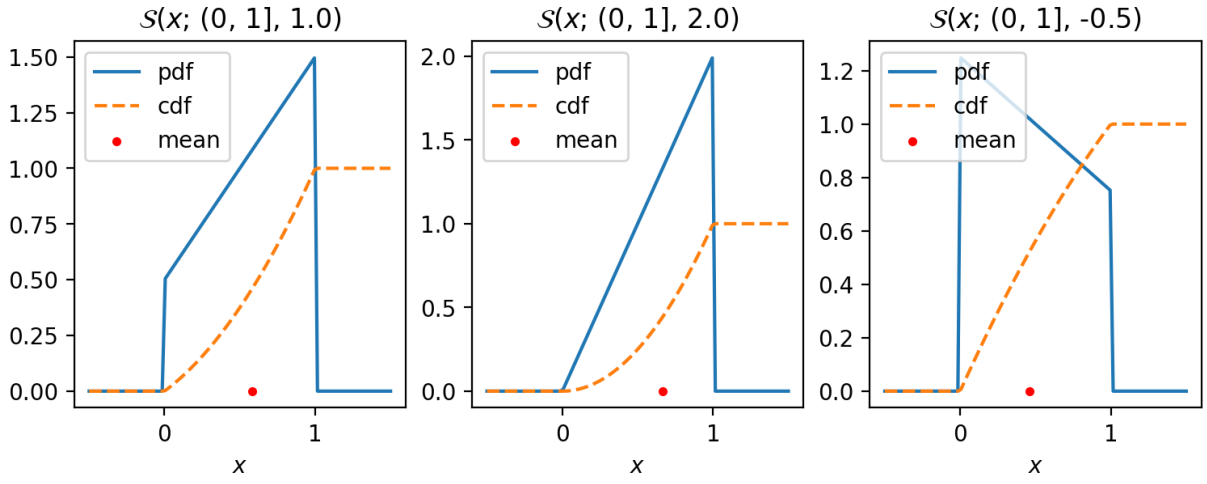


Figure 5: The slopyform pdf and cdf for various parameter choices. The second picture illustrates the maximum setting of the slope parameter for the given width.

Adaptive partition

In our $(a, b]$ truncated gaussian g case, we know that $g'(x) \geq 0, x \in L = (a, \mu]$ and $g'(x) < 0, x \in R = (\mu, b]$. If we start with proposal partition $(a, \mu], (\mu, b]$ with associated weights (area of graph of truncated gaussian) $w_1 = 0.5, w_2 = 0.5$, we can iteratively

subdivide each interval again until it satisfies some property, such as a sufficiently low error. When splitting an interval $(i, j]$ with weight w, the splitting point k is chosen so $(i, k], (k, j)$ both account for $w/2$ of the area the pdf under (i, j). The location of k can be found using the CDF and inverse CDF of the truncated gaussian. We will derive the procedure for the uniform and the slopyform approximation separately:

Uniform approximation

A mixture component $u(x) = w \cdot \mathcal{U}(x; i, j)$ in our deterministic sum-unit approximates g in the domain $(i, j]$, where either $(i, j] \in L$ or $(i, j] \in R$. As the partition gets finer and the renormalization constant $Z \to 1$, $u(x) \to g(0.5(i + j))$, meaning that g is approximated in the domain by its taken value in the center of the domain. If we define the error

$$e^{i,j}_{\texttt{diff}}(x) = |g(x) - u(x)|$$

it will be maximised at input i or j because of the monotonicity of g within the domain. This implies that

$$\forall x \in (i, j) : \quad \max(e^{i,j}_{\texttt{diff}}(i), e^{i,j}_{\texttt{diff}}(j)) \geq e^{i,j}_{\texttt{diff}}(x)$$

If we also define

$$e^{i,j}_{\texttt{ratio}}(x) = \max\left(\frac{g(0.5(i + j)}{g(x)}, \frac{g(x)}{g(0.5(i + j)}\right) = \exp\left|\log\left(\frac{g(0.5(i + j)}{g(x)}\right)\right|$$

we similarly have

$$\forall x \in (i, j) : \quad \max(e^{i,j}_{\texttt{ratio}}(i), e^{i,j}_{\texttt{ratio}}(j)) \geq e^{i,j}_{\texttt{ratio}}(x)$$

In both cases $e^{i,j}_{\texttt{ratio}}$ and $e^{i,j}_{\texttt{diff}}$, the error is reduced by shrinking our interval $(i, j]$ because of appropriate reweighting, the intermediate value theorem g and monotonicity. We can therefore select some maximum allowable error ε compare it only to the error in the endpoints of each interval, splitting the interval it into two new ones if the error is too big.

Slopyform approximation

We define the same error measures e as for the slopyform approximation. To facilitate checking only endpoints again we have to assume that any proposal interval (i, k) satisfies not only monotonicity of g but also of g'. This ensures that our approximation error consistently grows as $\mathcal{O}(n^2)$ as we deviate from the center of the interval. The additional requirement on the signedness of g'' can be met with the initial proposal partition

$$\Big((a, \mu - \sigma],\ (\mu - \sigma, \mu],\ (\mu, \mu + \sigma],\ (\mu + \sigma, b]\Big) = (L_1, L_2, R_1, R_2)$$

for which

$$g''(x) \geq 0 \text{ for } x \in (i, j) \text{ if } (i, j] \in L_1 \text{ or } (i, j] \in R_2$$

and

$$g''(x) \leq 0 \text{ for } x \in (i, j) \text{ if } (i, j] \in L_2 \cup R_1$$

Another consideration is that the slopyforms cannot represent all possible slopes, as the s parameter is bounded as described earlier, $|s| \leq \frac{1}{2(b-a)^2}$. If the slope of the considered function lies outside of those bounds, the slopyform will be split again, decreasing $b-a$ which increases the valid range of s. Repeat until within range. In practice, the inital proposal partition (L_1, L_2, R_1, R_2) with appropriate weights is used both for the uniform and slopyform partition which results in at least 4 components, which is reflected in example 0.2.

Proof of the full SPN relative error bound

If the conditional distribution of each node i is quantized by the `relative error` procedure with tolerance ε_i, then by construction the following holds:

$$\varepsilon > 1, \forall x: \quad ||\log \mathrm{M}(x) - \log \mathrm{P}(x)|| \leq \log(\varepsilon) \quad \Longleftrightarrow \quad \mathrm{P}(x)\varepsilon^{-1} \leq \mathrm{M}(x) \leq \mathrm{P}(x)\varepsilon$$

We derive an upper and lower bound on the ratio separately and combine them for the final bound. A key assumption of this proof is that the mixture components do not interact, that is, their domain is disjoint (*determinism*), which holds for our SPNs.

Upper bound: We decompose the modeled conditional distributions of the SPN M into all the nodes given their parents. These distributions are exactly the ones constructed and fitted with error bounds, we we can upper bound them all by their bound, from which we collect terms to regain the original BN and an error factor.

$$\begin{aligned}
\forall \mathbf{x}: \quad \mathrm{M}(\mathbf{x}) &= \mathrm{M}(x_1) \cdot \mathrm{M}(x_2|x_1) \cdot \cdots \cdot \mathrm{M}(x_n|x_{1:n-1}) \\
&\leq (\varepsilon_1 \mathrm{P}(x_1)) \cdot (\varepsilon_2 \mathrm{P}(x_2|x_1)) \cdot \cdots \cdot (\varepsilon_n \mathrm{P}(x_n|x_{1:n-1})) \\
&= \prod_{i=1}^{n} \Big(\mathrm{P}(x_i|x_{1:i-1}) \Big) \cdot \prod_{i=1}^{n} \varepsilon_i \\
&= \mathrm{P}(\mathbf{x}) \prod_{i=1}^{n} \varepsilon_i
\end{aligned}$$

The **Lower bound** is derived analogously:

$$\begin{aligned}
\forall \mathbf{x}: \quad \mathrm{M}(\mathbf{x}) &= \mathrm{M}(x_1) \cdot \mathrm{M}(x_2|x_1) \cdot \cdots \cdot \mathrm{M}(x_n|x_{1:n-1}) \\
&\geq (\varepsilon_1^{-1} \mathrm{P}(x_1)) \cdot (\varepsilon_2^{-1} \mathrm{P}(x_2|x_1)) \cdot \cdots \cdot (\varepsilon_n^{-1} \mathrm{P}(x_n|x_{1:n-1})) \\
&= \prod_{i=1}^{n} \Big(\mathrm{P}(x_i|x_{1:i-1}) \Big) \cdot \prod_{i=1}^{n} \varepsilon_i^{-1} \\
&= \mathrm{P}(\mathbf{x}) \left(\prod_{i=1}^{n} \varepsilon_i \right)^{-1}
\end{aligned}$$

Isolating the error on both dervied inequalities and combining, we get

$$\prod_{i=1}^{n} \varepsilon_i^{-1} \leq \frac{M(\mathbf{x})}{P(\mathbf{x})} \leq \prod_{i=1}^{n} \varepsilon_i$$

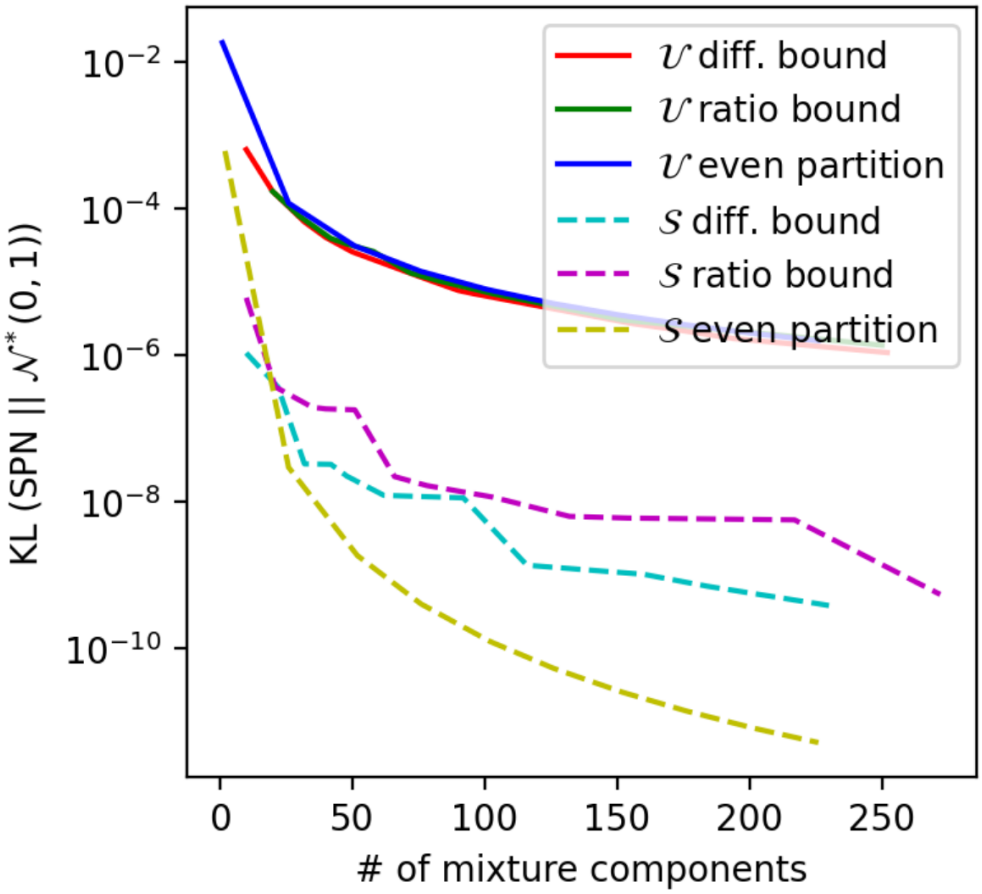


Figure 6: KL divergence between SPN with uniforms $\mathcal{U}$ and slopyforms $\mathcal{U}$ to truncated gaussian $\mathcal{N}^{10^{-3}}(x; 0, 1)$ for different partitioning schemes and amounts of mixture components. Evenly spacing gives the lowest distance, but provides no direct control over the error bound. We note that in all three cases, KL divergence of the slopyform approximation drops much faster than the uniform one, requiring fewer components, confirming that there is no reason to use uniforms.

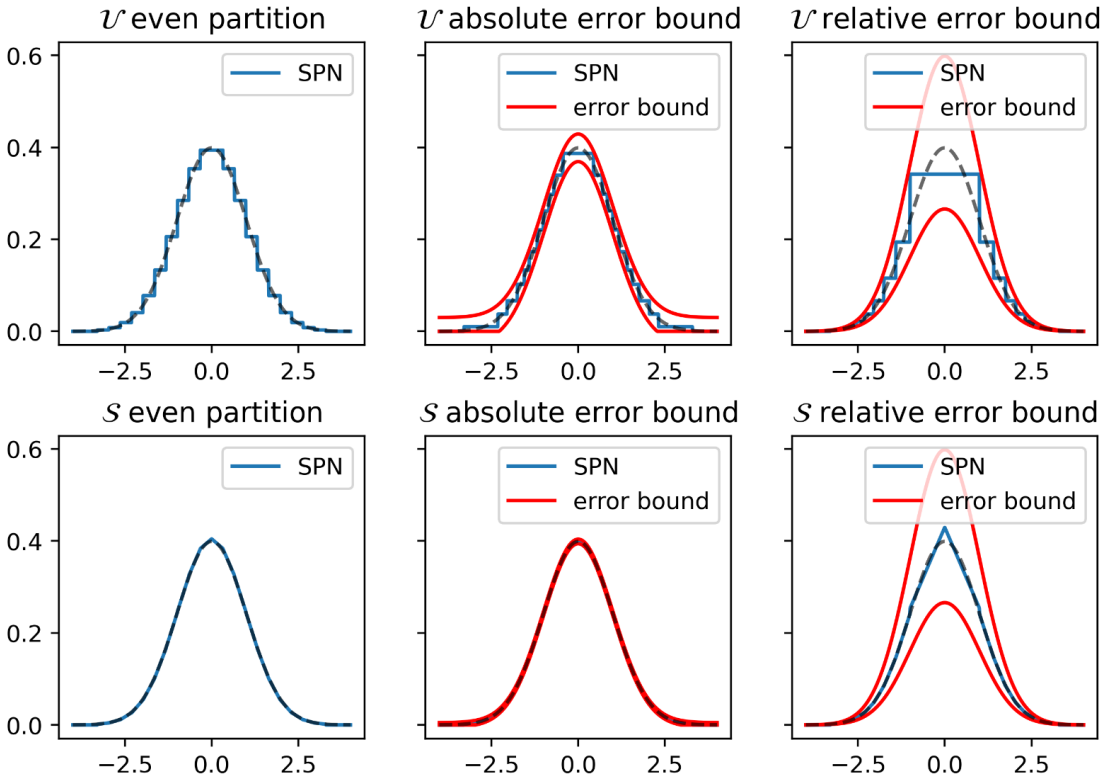


Figure 7: For these six `Mixifications` of a truncated standard gaussian the parameters n and ε were chosen in each instance to yield 20 ± 2 components. For the procedures yielding error bounds, these bounds have been visualized as well. Notice how the `even partition` and `absolute error` procedures allocate the intervals quite evenly in the domain, giving a (visually) more pleasing approximation. The `relative error` procedure requires a lot of care and detail on the areas with low probability, which in turn leaves only few for the likely areas. For `absolute error`, the slopyforms can achieve tighter bounds with the same amount of components, but this behaviour is not reflected with `relative error`.

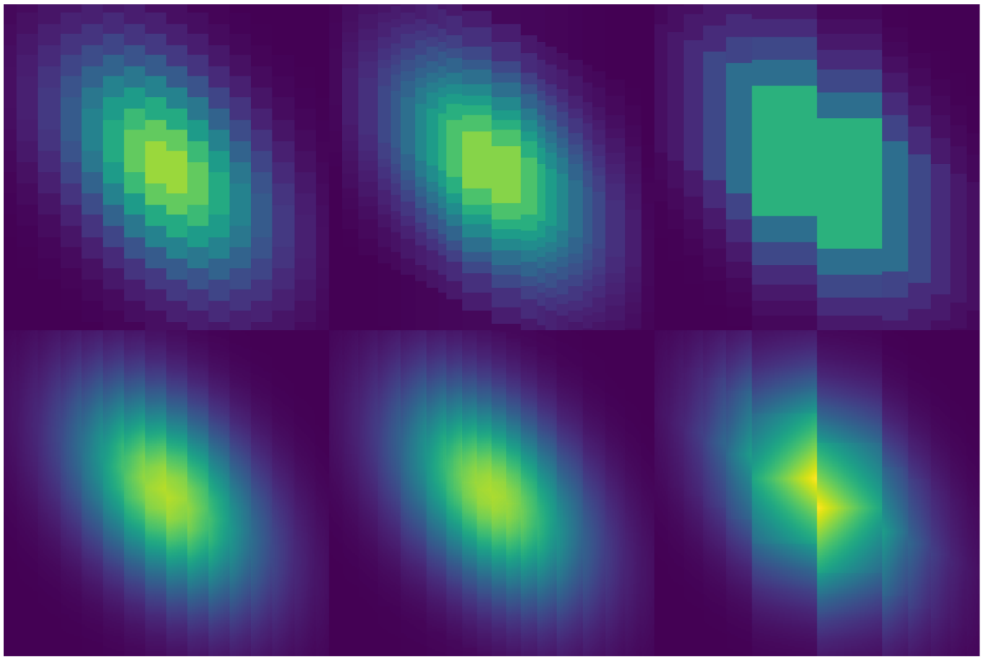


Figure 8: Here, we convert $\mathcal{N}(y; 0, 1^2) \cdot \mathcal{N}(x; 0.5y, 1^2)$ using eq. 1, using the `Mixifacation` parameters from fig. 7, respectively. Again we note the jaggedness of `relative error`.

Pseudocode for algorithm for full conversion of BNs to SPNs

Algorithm 2 `BN_to_SPN`

Require: Bayesian network `BN` with evidence `evi` ($= \emptyset$ on initial call)
  Factors ← empty list
  for R in `roots` **do**
    all_params ← List of `Mixify`(node) for node ∈ R
    outer_sum ← empty list
    outer_sum_weights ← empty list
    for binned_cluster ∈ Cartesian product of all_params **do**
      sub_factors ← empty list
      sum_weight ← 1
      `root_evi` ← empty list
      for ((a,b], weight, slope) ∈ binned_cluster **do**
        bin ← `Uniform` or `Slopyform` with parameters and corresponding scope
        append bin to sub_factors
        sum_weight ← sum_weight · weight
        append $\mathop{\mathbb{E}}_{x \sim \text{bin}(\cdot)}[x]$ to `root_evi`
      end for
      append `BN_to_SPN`(`BN` | `evi` ∪ `root_evi`) to sub_factors
      append `Product`(sub_factors) to summands
      append sum_weight to outer_sum_weights
    end for
    sum ← `Sum`(summands, outer_sum_weights)
    append sum to Factors
  end for
  return `Product`(Factors)

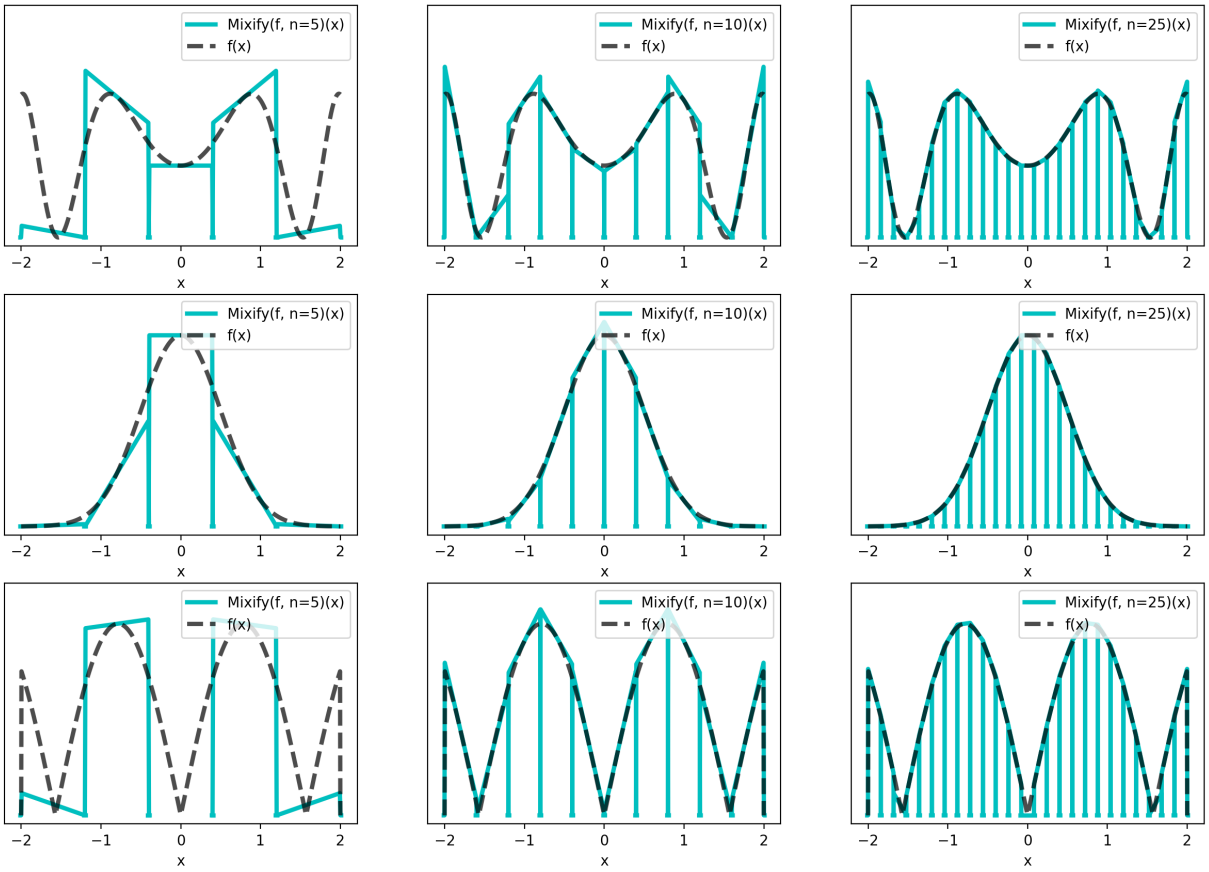


Figure 9: `Mixification` of three different differentiable functions with $n = \{5, 10, 25\}$, using the default `even partition` approach. The approximated function is overlaid to facilitate a qualitative comparison. Notice how at times we might still have a slope of 0 if the function is flat. Additionally see how, in the bottom left, the central bin gets weight 0, which illustrates how the weight is determined by the center of the bin.

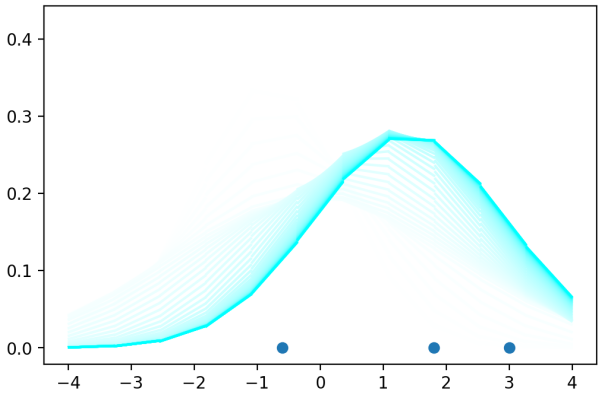


Figure 10: Optimizing the parameters (μ, σ)) for the likelihood of a univariate gaussian directly from its SPN representation through differentiable `Mixification`. The mean shifts to the center of the three samples (blue points), while the variance shrinks. Running this simple example for 50 epochs took around 2.7 seconds on a Lenovo P50, which highlights the large overhead of repeated calls to `Mixify` and `PyTorch`'s burden of having to keep track of the computation graph.

A library for constructing and compiling LGPGMs to SPNs

During the project, I implemented a syntax for constructing the `LGPGM`-object which enabled quick testing and is intuitive to use. Here's an example:
We construct a multivariate normal BN over RVs A, B, C, D factorized as

$$P(a,b,c,d) = \mathcal{N}(a;0.27,1^2)\cdot\mathcal{N}(b;a+2,1.25)\cdot\mathcal{N}(c;-a-1.5,1.25)\cdot\mathcal{N}(d;1+0.5(b+c),1.25)$$

using the `LGPGM` library developed alongside the project.

```
# we bind the variable name "A" to the result of the constant 0.27
    + N(0,1)
A = "A" @ (noise + 0.27)
# then we define the remaining RVs as compositions of N(0,1) and
   previously defined RVs.
B = "B" @ (A + 0.5*noise + 2)
C = "C" @ (-A + 0.5*noise - 1.5)
D = "D" @ (0.5*B + 0.5*C + 0.5*noise + 1)
# extracting the network structure
A.get_graph()
```

The resulting graph of this BN is reflected in fig. 11. The mean vector and covariance matrix (A, B, C, D) is

$$\begin{bmatrix} 0.27 \\ 2.27 \\ -1.77 \\ 1.25 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 1 & 1 & -1 & 0 \\ 1 & 1.25 & -1 & 0.125 \\ -1 & -1 & 1.25 & 0.125 \\ 0 & 0.125 & 0.125 & 0.375 \end{bmatrix}$$

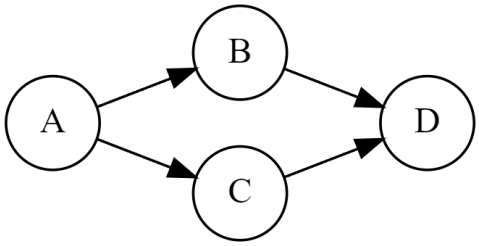


Figure 11: The structure of the BN that we define in the example. Before we know the conditional distribution of D, we must know what bins B and C have fallen into. We look at all their pairings with the cartesian product.

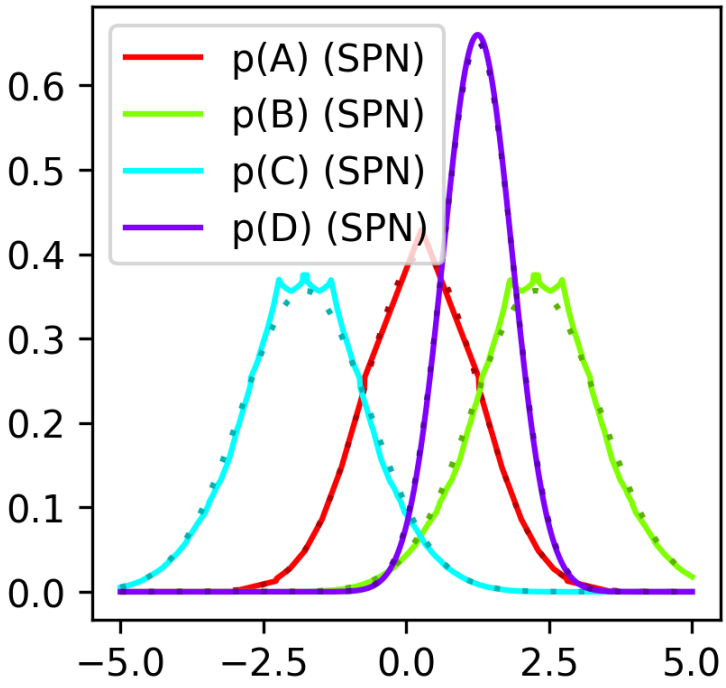


Figure 12: Here are the marginal distributions of each variable, easily extracted from the SPN with the `SPFlow` `marginalize` function. Overlaid in dotted lines are the true marginal distributions.

We construct the SPN, visualize marginals (fig. 12, dotted lines show exact gaussian marginal distributions) and sample:

```
# compiling the LGPGM to a SPN using the 'C' function.
spn = lgpgm_to_spn(A, sloped=True, # indicates slopyforms
                   crit=spnhelp.CRIT_slopyform_absolute_error,
                   crit_param=0.1) # = epsilon
# plotting true and spn marginal distributions
samples = sample_from_spn(spn, 10000) # sampling
plot_marginals(spn, A)
```

From the 10000 samples from the SPN we get the following sample mean and covariance (rounded):

$$\begin{bmatrix} 0.27 \\ 2.27 \\ -1.76 \\ 1.25 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 1.175 & 1.107 & -1.093 & 0.01 \\ 1.107 & 1.38 & -1.095 & 0.142 \\ -1.093 & -1.095 & 1.344 & 0.117 \\ 0.01 & 0.142 & 0.117 & 0.377 \end{bmatrix}$$

which shows that the structure has indeed been captured well.

A library for constructing and compiling BNs over continuous variables to SPNs

During the project, I also implemented a syntax for constructing BNs over continuous variables with the `BN` library. It relies on `PyTorch`[Pas+19] to compute slope information. Here, we re-create a LGPGM in with this more general library. We have to define the pdf, a bounded domain and distribution name to a `Distribution` object, which represents a node in a BN as a compound distribution of its parents. Let's define the conditional linear gaussian distribution, a `mean`, the `sd` and the parents weight on the mean `pws`:

```
from bn import Distribution as dist
def cond_linear_gauss(mean, sd, pws):
    pws = torch.tensor(pws)
    func = lambda Ps, x: torch.exp(-0.5*((x-mean-torch.sum(Ps *
   pws))/sd)**2)
    bounds = lambda Ps: (mean + torch.sum(Ps * pws) - 4*sd, mean +
    torch.sum(Ps * pws) + 4*sd)
    return dist(func, bounds, "linear_gauss")
```

This conditional distribution is a (unnormalized) gaussian, with the tails chopped at $\mu \pm 4\sigma$. Now we can construct the `BN`. The syntax is supposed to resemble the mathematical notation, but not all operators could be elegantly implemented. The $\sim$ (distributed as) symbol is replaced by a $>$, because $\sim$ is a unary operator in Python.

```
from bn import ParentList as pa
A = "A" > cond_linear_gauss(mean=0.0, sd=1.0, pws=[])
# The node "A" is distributed as N(0,1).
# pws is an empty list, as A (root) has no parents.
B = "B" |  pa(A) > cond_linear_gauss(1.5, 0.5, pws=[0.5])
# B | a is distributed as N(1.5+0.5a, 0.5^2)
# To be read as B | A ~ N(xxx)
C = "C" |  pa(A, B) > cond_linear_gauss(-4, 0.2, pws=[0.5, 0.5])
# C | a, b is distributed as N(-4 + 0.5a + 0.5b, 0.2^2)
A.get_graph() # get BN structure
```

See fig. 13 for the BN structure. We then plot the marginals and compare with a SPN compiled from the same BN structure, just defined in the LGPGM class which also support exact marginals. See fig 14 for a comparison between the LGPGM library which has been fitted specifically to work with gaussians.

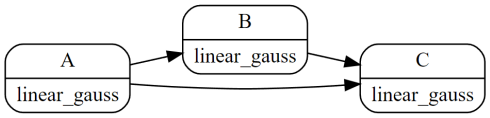


Figure 13: The resulting BN structure.

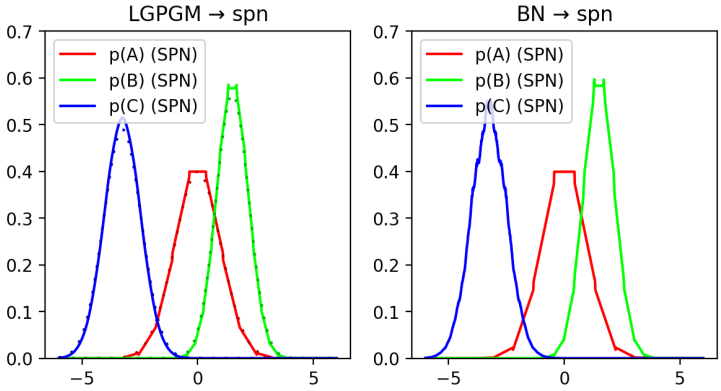


Figure 14: Comparing the two compiled SPNs. They are indeed similar. When using the general `BN` library, one loses approximation error bounds and exact marginals to compare against.

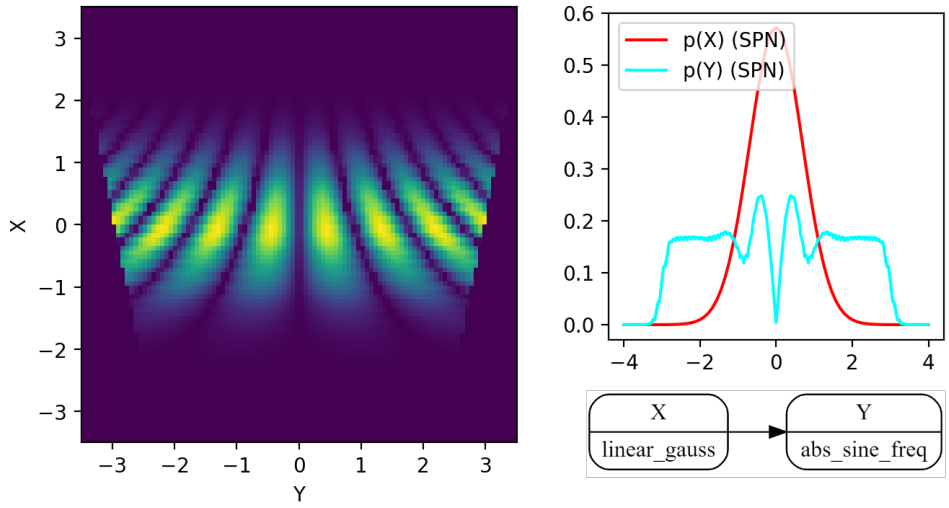


Figure 15: This is the joint distribution of the BN $X \to Y$ with $X \sim \mathcal{N}(0,1)$ and $P(Y = y|X = x) \propto \begin{cases} |\sin(y) \cdot 0.1(x+6)^2| & -3 + 0.2x < y < 3 - 0.2x \\ 0 & \text{otherwise} \end{cases}$ In this case, an unnormalized conditional distribution over Y is specified, and it is subsequently renormalized by `Mixify`.

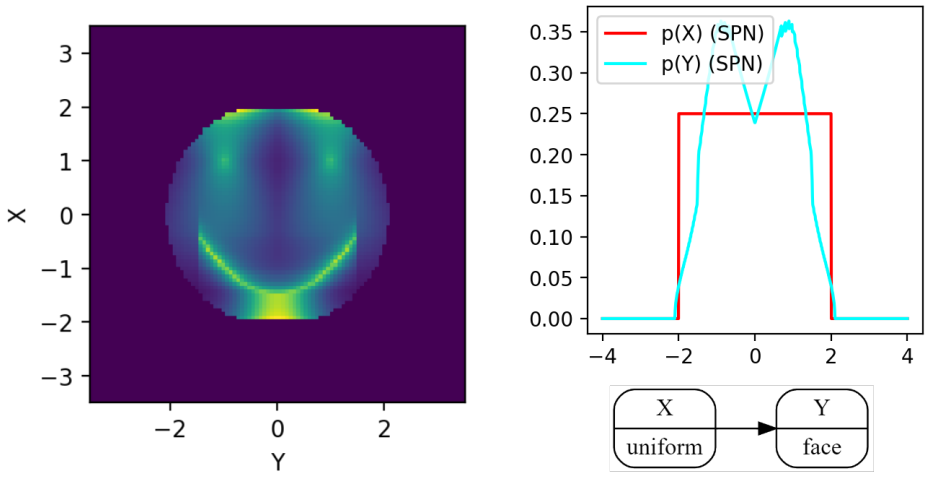


Figure 16: Another BN. Very complex conditional distributions can be specified. In this case, the conditional distribution and support of Y is defined to create something that resembles a face.