Sum-Product Networks: An NN Alternative?

Erkin Türköz
Technical University Munich
erkin.turkoz@tum.de

Abstract

Sum-Product Networks (SPNs) are a kind of deep network in the form of a rooted directed acyclic graph. As the name suggests, SPNs contain sum and product nodes as internal nodes in addition to leaf nodes. At the leaves, SPNs hold univariate distributions. When the sum and product nodes satisfy some important conditions, each SPN node represents a probability distribution. This fact attaches a probabilistic meaning and generative modeling capability to SPNs. SPNs are distinguished with their ability to perform tractable and exact inference even with missing input values, which is not possible with Neural Networks (NNs). In this report, the definition and fundamental properties of SPNs, different SPN architectures, learning & inference algorithms will be presented and a comparison of SPNs against NNs will be made.

1. Introduction

Sum-Product Networks (SPNs) were first introduced by Poon and Domingos [1] in 2011. An SPN is essentially a rooted directed acyclic graph containing the sum, product and leaf nodes. Leaf nodes hold probability distributions. Under certain conditions, each internal node in an SPN also represents a probability distribution, which brings a probabilistic interpretation to SPNs. One of the most remarkable properties of SPNs is probably its ability to perform several probabilistic inference tasks in a tractable manner. More precisely, SPN's root node can exactly output the joint or marginal probabilities for specific configurations of its variables with linear complexity. Because of the ability of SPNs to handle missing features in the data via marginalization and the efficiency in exact probabilistic inference, the applicability of SPNs as an alternative solution to the problems for which NNs are mainly used is investigated by some researchers. [2, 3, 4]

There are some similarities of SPNs to the Probabilistic Graphical Models (PGMs), e.g. Bayesian or Markov Networks, and the Neural Networks (NNs). [3] However, compared to SPNs, PGMs are more convenient to highlight the conditional independence between variables [4], but it is proven that the class of functions that SPNs represent efficiently is more general than the set of tractable functions representable with PGMs. [1] On the other hand, there are distributions that NNs can efficiently represent while SPNs cannot. [5] Further differences between SPNs and NNs will be discussed later.

In this report, SPNs will be briefly reviewed. First, their definition and properties will be given in Section 2. Secondly, some variations of SPNs designed for different tasks will be introduced in Section 3. Then, some of the common inference tasks, parameter and structure learning techniques will be explained in Sections 4 & 5. Lastly, a detailed comparison between NNs and SPNs will be made in Section 6.

2. Definition and Properties of SPNs

In the original paper [1], SPNs are defined as a rooted directed acyclic graph containing the sum and product nodes. According to [1] and [6], SPNs have the following fundamental properties:

- Internal nodes are either sum or product nodes. If an SPN represents a continuous distribution, then sum nodes become integral nodes.
- Sum and product nodes are placed in an alternating fashion.
- Links emanating from sum nodes have non-negative weights.
- Each leaf is typically a univariate distribution. In the case of discrete variables, leaves are indicators, which activate a leaf only if a specific condition is met for a specific value, while in the continuous case, they represent univariate Gaussian or other univariate continuous distributions in general. However, leaves can be extended to contain multivariate distributions as well.

The definition of SPN is recursive. Therefore, any node v of an SPN is also an SPN rooted at v. [1] Based on this fact, note that there is no restriction regarding the type of

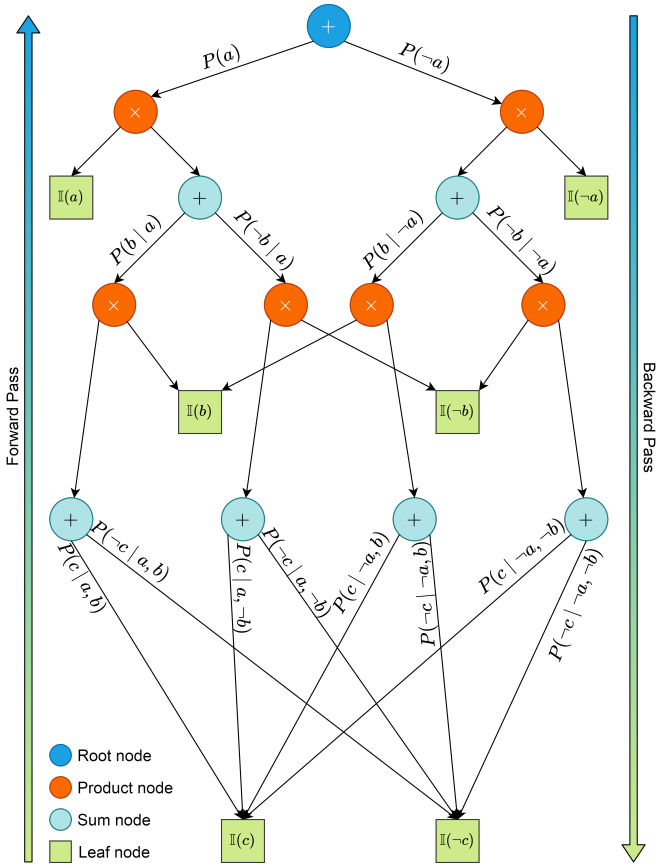

Figure 1. A Complete and Decomposable SPN Example: Network has 3 variables A, B & C. Variables are discrete and binary-valued. Therefore, there are 6 leaf nodes each of which holds indicator function ($\mathbb{I}$). Sum node weights can be thought as conditional probabilities and a bottom-up or forward pass can be considered as the application of the chain rule for probabilities. (Figure is taken from [6].)

the root node. In other words, it can be a sum or product node.

The value of a variable is a particular assignment to that variable. The scope of a node refers to the set of variables associated with at least one leaf node accessible from that node. [1]

In SPNs, leaf nodes act as inputs and the root node is the output thereof. Furthermore, each leaf node represents a simple distribution (or probability density function). Therefore, the overall SPN model can be interpreted as a hierarchy of probability distributions under certain conditions. In this probabilistic framework, sum nodes are factors of a joint distribution (defined by the parent product node) and, at the same time, they represent mixtures of different distributions (defined by child product nodes). They can be also viewed as latent variables of the SPN. Product nodes, on the other hand, are the constituents of the mixtures (defined by parent sum node) and, at the same time, they represent the joint probability of their factors (defined by child sum nodes). [1, 3] As SPNs directly model the joint distributions of the variables, they are generative models. [7]

As mentioned, for this probabilistic interpretation to hold, an SPN has to fulfill some conditions: [1, 6]

- Completeness: A sum node is complete if and only if all of its children have the same scope. An SPN is complete if and only if all of its sum nodes are complete.
- Decomposability: A product node is decomposable if and only if all of its children have pairwise disjoint scopes. An SPN is decomposable if and only if all of its product nodes are decomposable.

If an SPN satisfies completeness and decomposability, it is said to be valid. A valid SPN represents the unnormalized probability distribution over its variables. Completeness and decomposability are sufficient but not necessary for the validity of overall SPN. On the other hand, they are necessary to enforce every sub-SPN to be valid. When an SPN is valid and every sum node of it has weights normalized to 1, then SPN represents the normalized probability distribution. [1] Then, the root node of the SPN can exactly and efficiently evaluate joint, marginal, and conditional probabilities with respect to its input values. [6] A simple example for a complete and decomposable SPN can be seen in Figure 1.

These two properties are significant because while completeness enforces sum nodes to produce valid mixture models, decomposability dictates that children of product nodes are independent of one another and factorized distributions computed by the product nodes are valid. [8]

Sum nodes of an SPN can also have selectiveness property. If a sum node v is selective, then for all possible complete configurations to the variables in the scope of v, at most one child has a positive output value. However, an important detail in this statement is that if configurations are incomplete (i.e. partial, one or more variables are missing) then there can be more than one child of v with a positive output value and this does not break the condition for the selectivity of the sum node v. An SPN is selective when all of its sum nodes are selective. [6] Selective SPNs are useful in practise because parameter learning with MLE can be performed with a closed-form expression [9] and exact MPE inference becomes tractable. [10]

If an SPN is not selective, it can be augmented to a selective one. Resulting SPN is called augmented SPN and it represents the same probability function as the original one. However, during the augmentation, new variables and links are introduced. Therefore, augmented SPN has a different inner structure from the original SPN. The interpretation for newly-inserted nodes is that such nodes were latent in the

original SPN and they are revealed when the SPN is augmented. As mentioned previously, selective SPNs are very desirable for some learning and inference tasks. Because the augmentation operation generates a selective SPN from a non-selective one, this transformation makes it possible to use existing algorithms for selective SPNs on the augmented version. However, the results obtained from augmented SPN is only an approximation for the original SPN and not guaranteed to be a good one. [6]

3. Different SPN Architectures

In this section, some examples of different architectures derived from basic SPN definition to tackle different specialized tasks will be presented.

3.1. Deep Generalized Convolutional Sum-Product Networks (DGC-SPN)

Deep Generalized Convolutional Sum-Product Networks (DGC-SPNs) [3] are introduced to equip SPNs with the ability to capture spatial features in images. It is similar to Convolutional Neural Networks (CNNs) in this sense. However, DGC-SPN maintains the probabilistic interpretation of SPNs. It can be trained for both generative and discriminative image tasks. Contrary to Deep Convolutional SPNs (DCSPNs), which is again an SPN-based architecture for spatial data by [2], DGC-SPN can take overlapping patches from the image with a new stride and dilation parametrization without violating validity. With overlapping patches, the authors emphasize that DGC-SPN captures features with a finer scale and coverage than DCSPNs. Therefore, the class of distributions representable by DGC-SPN is broader than DCSPNs.

In their terms, all nodes along the channel dimension of a specific spatial location in the patch form a cell. The sum layer is the result after summing nodes over channels per cell. Therefore, each sum operation computes a mixture of nodes of a specific cell. The result of each sum yields a new node in the respective output cell. As it is possible to define multiple single-cell sums by varying the weights, the number of channels per output cell can be arbitrarily chosen. The product layer is the outcome of multiplying single channels from different cells. Therefore, the product layer combines inputs spatially. Sum and product layer computations are performed alternatingly. (See Figure 2.)

The resulting model is valid because completeness is achieved by ensuring that each node in the same cell has the same scopes and decomposability is maintained with exponentially growing dilation (dilation rate is doubled at every product layer, see Figure 2). This parametrization of products ensures that nodes with common scopes are skipped, yielding disjoint scopes for multiplications and maintaining decomposability.

3.2. Dynamic SPNs (DSPN)

Regular SPNs can represent distributions of only a fixed number of inputs. Therefore, they are not directly applicable to sequential data. For that reason, in [4], authors introduce Dynamic SPNs (DSPN) to generalize SPNs to sequential data with varying number of inputs. They define 3 main types of networks and interface nodes to connect these networks to achieve this goal:

- Interface Node: Contrary to the regular leaf or root nodes, interface nodes connect pairs of networks. The connection is established by sharing interface nodes between network pairs, e.g. output interfaces of the previous network are passed as input interfaces to the next network. Interface nodes can be sum or product nodes and they can be root or leaf.

- Template Network: For n binary variables, template network consists of $2n$ standard leaf nodes, k interface nodes as input from the previous network, and k root nodes as an output interface to the next network. Hence, there are $2n + k$ leaves in total and k roots.

- Bottom Network: The bottom network processes the start of the sequence. Therefore, it has no interface nodes as input. For n binary variables, it has $2n$ leaves and k root nodes as an output interface to the next network.

- Top Network: The top network combines outputs of the previous network that processes the end of the sequence. Therefore, it has k leaves and 1 root.

Modeling sequential data of length T requires 1 bottom network followed by a template network replicated $T-1$ times and 1 top network at the end. Figure 3 depicts an example for a DSPN.

The authors define the invariance property to check whether the template network satisfies completeness and decomposability properties. Furthermore, they provide a theorem incorporating conditions regarding invariance property for template networks as well as conditions for completeness and decomposability of bottom and top networks to ensure that overall DSPN is valid.

3.3. Sum-Product-Quotient Networks (SPQN)

In [7], authors present Sum-Product-Quotient Networks (SPQN), which extends the basic SPN architecture by introducing a quotient node which accepts two SPNs as numerator (e.g. $P(A, B)$) and denominator (e.g. $P(B)$) and directly computes conditional distributions with Bayes' Rule ($P(A|B) = \frac{P(A,B)}{P(B)}$). Therefore, in SPQNs, every internal node represents a conditional distribution. (See Figure 4 for an illustration.)

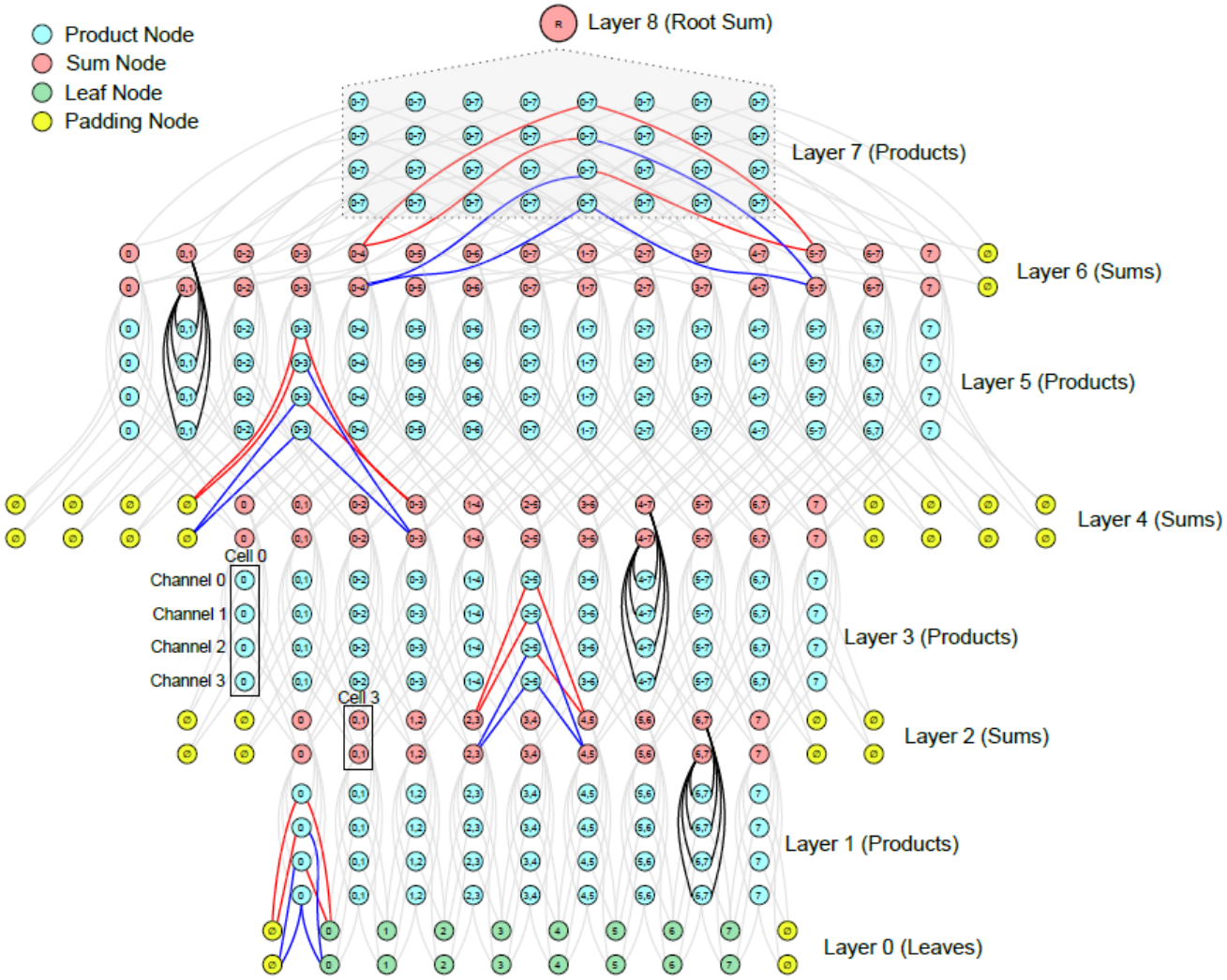

Figure 2. Visualization of DGC-SPN for 1D: Sum nodes operate on channels of a cell while product nodes act on pairs of cells. While taking the product, the stride is 1, the dilation rate starts with value 1 at layer 1, and at each layer, it is doubled. Padding nodes have a constant probability of 1. Red lines are used to emphasize that the product node is linked to the upper channel of the cell from the previous layer. Blue lines show that the product is linked to the lower channel of the cell from the previous layer. (Taken from [3].)

The authors define modified versions of SPN's completeness and decomposability properties, namely conditional completeness and conditional decomposability, and introduce one additional condition called conditional soundness. Under these conditions, they prove that their model retains the validity and tractable inference property of SPNs. Furthermore, they show that the modified conditions are relaxed versions of the original ones. As a result, they indicate that the class of distributions that SPNs can represent efficiently is the subset of the class of distributions efficiently representable by SPQNs. However, they note that because of conditional decomposability, SPQNs create a partial ordering of the input variables. Therefore, tractable marginalization is restricted to the subsets of the input variables which comply with this order.

4. Inference

In this section, based on [6], the different ways to perform inference with SPNs are discussed.

4.1. Inference with Joint, Marginal & Conditional

As mentioned, an SPN defines a distribution over its inputs and one can perform inference with joint, marginal & conditional densities exactly and efficiently with bottom-up passes from leaves to the root. (See Figure 5) More precisely, the complexity of such inference tasks in SPNs is linear in the number of edges in the SPN graph. [6]

Joint probabilities can be directly computed in a single

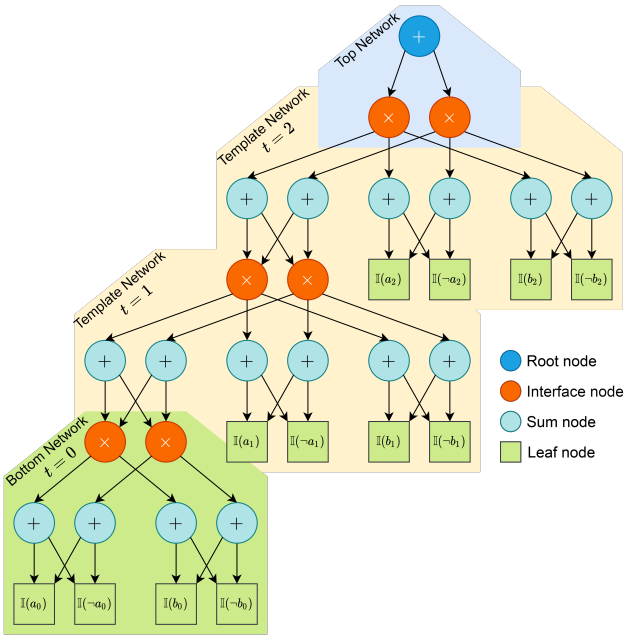

Figure 3. DSPN with 2 Binary Variables Unrolled for a Sequence of Length 3: Interface nodes, which are product nodes in this case, connect each two consecutive networks. For this example, there are 4 networks in total, one bottom network, followed by two template networks, followed by one top network. (Adapted from [4].)

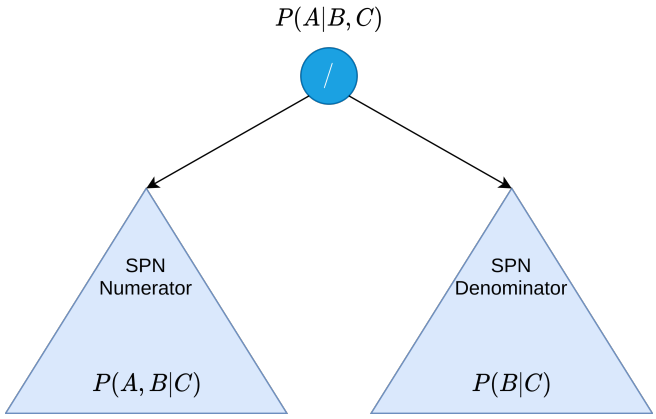

Figure 4. An Illustration for SPQN: The root quotient node takes the values of two sub-SPNs which represent two different conditional distributions and computes yet another conditional distribution using Bayes' Rule. [7]

bottom-up pass. As leaves correspond to all possible configurations of variables of the model, it suffices to set the variables to the values of the configuration of the joint distribution to be computed and perform a pass in the upward direction. [6]

For computing the marginals directly, a single bottom-up pass is again sufficient. However, in this case, one has to take in all possible configurations of the missing variables to marginalize them out. In the discrete case with indicator leaves, this simply means setting all leaves belonging to a variable to be marginalized to 1. Then, a forward pass yields the desired marginal probability at the root. [1]

Conditional probability can be computed indirectly with Bayes' Rule by taking the division of the results from separate bottom-up passes. [4, 11]

4.2. Most Probable Explanation (MPE) Inference

Following the notation in [6], $\mathbf{V}$ denotes all variables, $\mathbf{X}$ variables of interest, $\mathbf{x}$ values for variables of interest, $\mathbf{E}$ evidence variables, $\mathbf{e}$ values for evidence variables respectively. In MPE, all variables are either evidence variables or variables of interest, equivalently $\mathbf{V} = \mathbf{X} \cup \mathbf{E}$, meaning there are no hidden variables $\mathbf{H}$. The goal of MPE is to maximize the posterior for $\mathbf{X}$:

$$\text{MPE}(\mathbf{e}) = \arg\max_{\mathbf{x}} P(\mathbf{x} \mid \mathbf{e}) \tag{1}$$

In [1], a linear-time procedure similar to the Viterbi algorithm is outlined to compute MPE. However, the exact MPE inference for regular complete and decomposable SPNs in general is reported to be NP-hard. [10] Therefore, the result of the outlined procedure is not necessarily the actual MPE solution. On the other hand, when the SPN is selective, it computes the MPE exactly. [9]

According to [6], when an SPN is selective, the MPE inference is computed as follows:

- Sum nodes are replaced with max node to compute:

$$S_i^{max}(\mathbf{e}) = \max_{j \in ch(i)} w_{ij} \cdot S_j^{max}(\mathbf{e}) \tag{2}$$

  where S_i, S_j are the values of the sum node i and its child j respectively, w_{ij} is the weight of link between sum node i and its child i, $\mathbf{e}$ denotes values of evidence variables and $ch(i)$ is the children nodes of sum node i.

- Product nodes compute values in the same way as regular SPNs.

- A forward pass in bottom-up direction is performed to compute values for product and max nodes as described. For evidence variables, only the leaf nodes corresponding to assigned values are used. On the other hand, for variables of interest, all possible leaves are taken into account.

- After the forward pass, a backward pass in the top-down direction is performed. Along the way, for each sum node i, the child yielding S_i is pursued and for each product node, all children are followed.

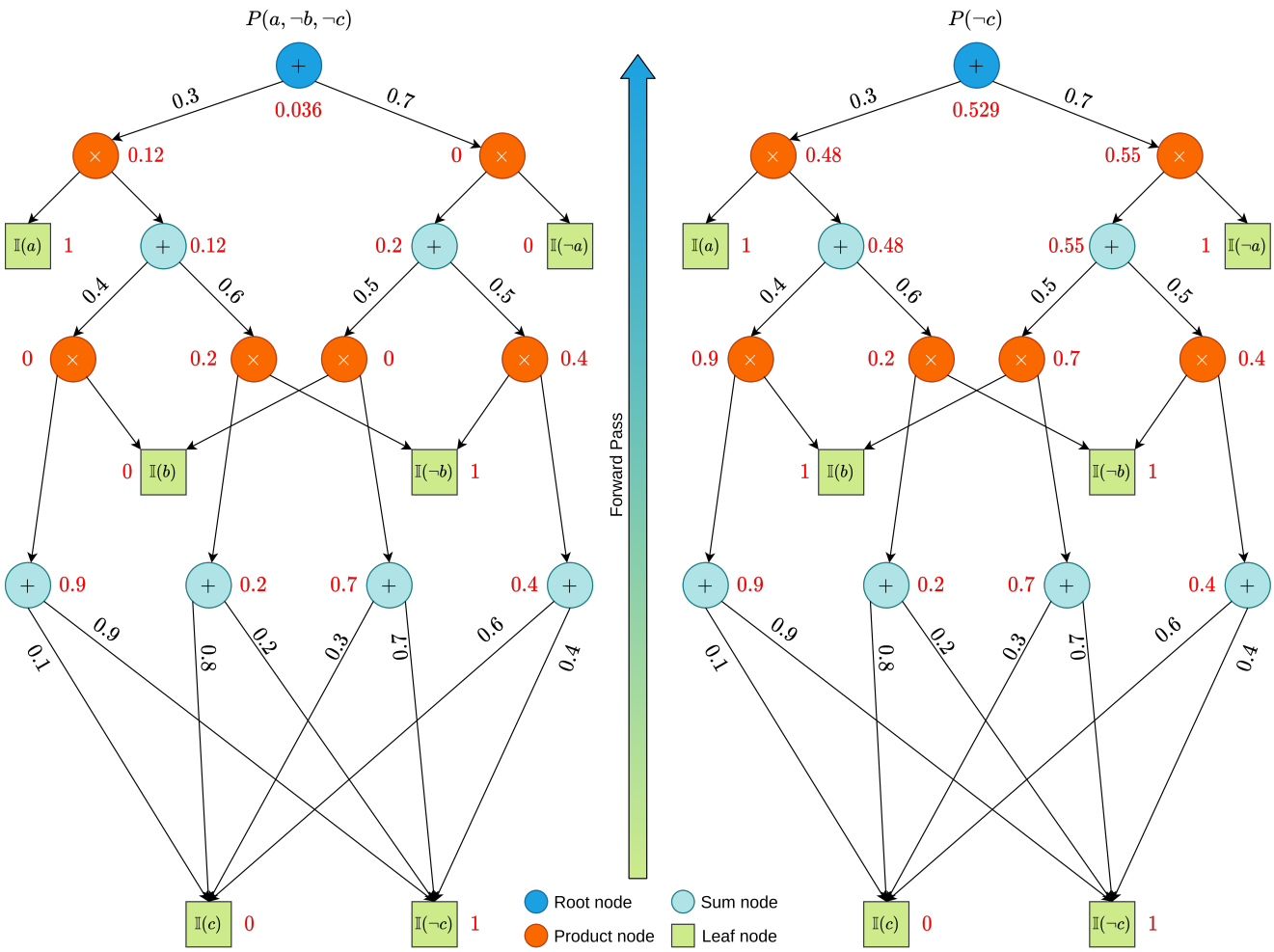

Figure 5. Computation of Joint and Marginal Probabilities: Left figure shows the SPN configured to compute joint probability for a particular assignment $(a, \neg b, \neg c)$ of variables A, B & C. Right figure shows the SPN configured to compute a marginal probability for the variable C taking the value $\neg c$. Using the result of these two separate forward passes, one can compute the conditional probability $P(a, \neg b \mid \neg c)$ indirectly: $P(a, \neg b \mid \neg c) = \dfrac{P(a, \neg b, \neg c)}{P(\neg c)} = 0.068$. (Adapted from [6].)

- At each leaf node, where the backward pass ended, the value v which satisfies $v = \arg\max_{v^*} P(v^*)$ is returned. When leaves hold continuous distributions, the return value is equivalent to the mode of the distribution at the arrived leaf. In the discrete case with indicators, the return value is simply the discrete label of the arrived node.

In Figure 6, MPE inference computation is illustrated for a complete and decomposable SPN.

4.3. Maximum A Posterior (MAP) Inference

Similar to MPE, the objective of MAP is:

$$\text{MAP}(\mathbf{X}, \mathbf{e}) = \arg\max_{\mathbf{x}} P(\mathbf{x} \mid \mathbf{e}) \tag{3}$$

However, MAP is more general in MPE because it additionally considers the existence of hidden variables $\mathbf{H}$. Therefore, now $\mathbf{V} = \mathbf{X} \cup \mathbf{E} \cup \mathbf{H}$. Due to the presence of hidden variables, MAP is naturally a harder problem than MPE. As a result, the MAP is also NP-hard. However, some algorithms to efficiently approximate MAP for SPNs are available. [6]

5. Learning

5.1. Parameter Learning

Learnable parameters of the SPNs are weights of sum nodes and parameters of the leaf distributions. The most prevalent parameter learning techniques for SPNs are based on Maximum Likelihood Estimation (MLE). Following the notation in [6], $\mathcal{D} = \{\mathbf{v}^1, \mathbf{v}^2, \ldots, \mathbf{v}^T\}$ denotes dataset con-

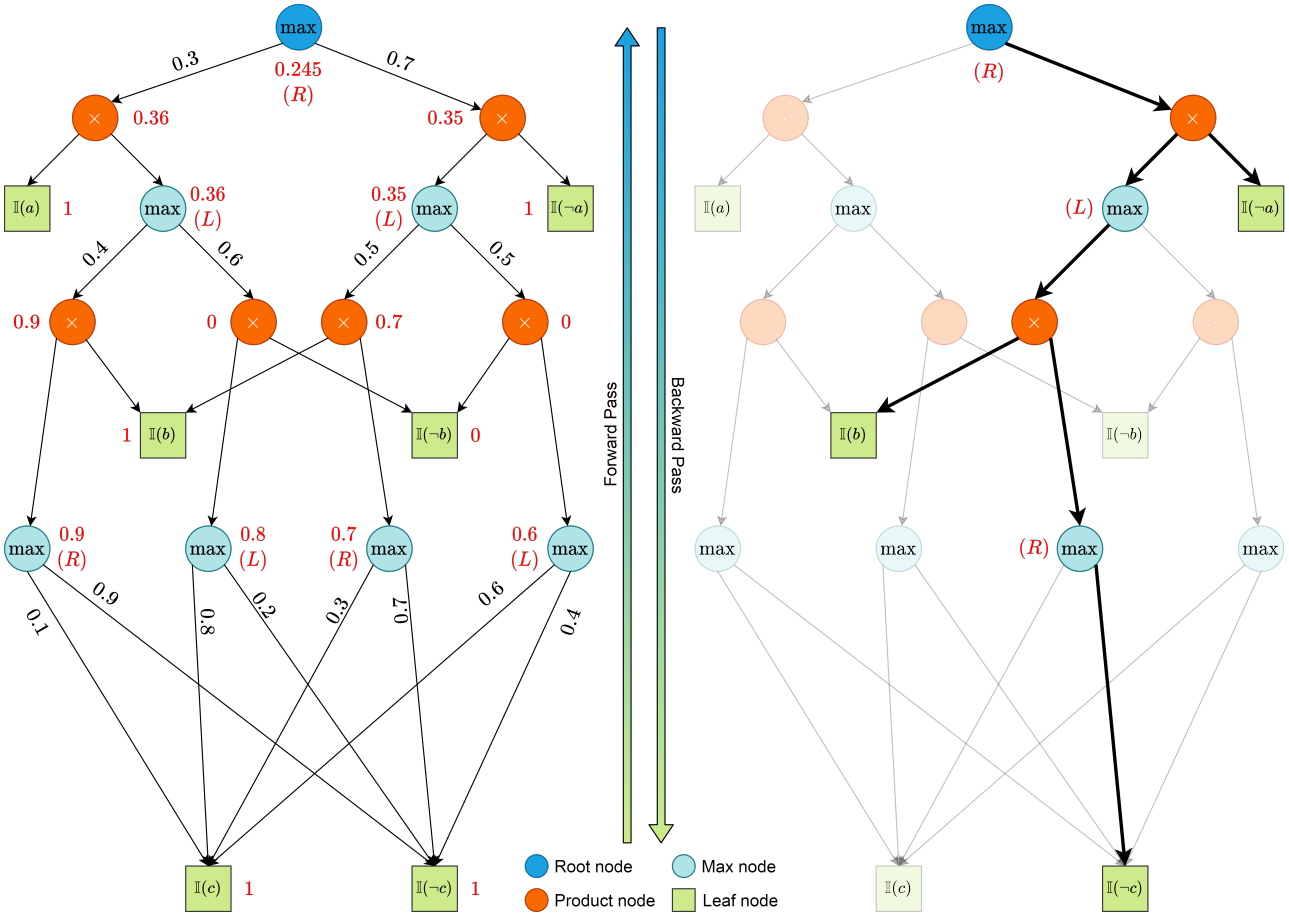

Figure 6. MPE Inference for Variables of Interest A & C with Evidence Variable $B = b$: Left figure shows forward pass made on the Max-Product Network obtained by replacing sum nodes of the original SPN with max nodes. During the forward pass, max nodes keep track of which link was previously followed. As shown in right figure, during the backward pass, stored links are followed to find the MPE values for variables A & C, which is $(\neg a, \neg c)$ in this example. (Adapted from [6].)

taining T independent and identically distributed samples, $\mathbf{W}$ is the sum node weights and $\mathbf{\Theta}$ is the parameters for distributions at the leaves. Then log-likelihood is defined as:

$$\begin{aligned} L_{\mathcal{D}}(\mathbf{w}, \boldsymbol{\theta}) &= \log P(\mathcal{D} \mid \mathbf{w}, \boldsymbol{\theta}) \\ &= \sum_{t=1}^{T} \log S(\mathbf{v}^t \mid \mathbf{w}, \boldsymbol{\theta}) \end{aligned} \tag{4}$$

And, Maximum (Log) Likelihood Estimate for parameters is then found by the following objective:

$$\begin{aligned} \hat{\mathbf{w}}, \hat{\boldsymbol{\theta}} &= \arg\max_{\mathbf{w}, \boldsymbol{\theta}} \; L_{\mathcal{D}}(\mathbf{w}, \boldsymbol{\theta}) \\ \text{subject to } w_{ij} \geq 0 \text{ and } &\sum_{j \in ch(i)} w_{ij} = 1 \end{aligned} \tag{5}$$

In the generative settings, parameters learned with the MLE model the full joint distribution of the samples while in discriminative settings, learned parameters model conditional distributions of samples given certain classes. [6]

Since there is no dependence between parameters of different nodes, each can be optimized individually. For leaf node distribution parameters, existing statistical methods can be applied. [6] Therefore, following subsections concentrate on learning the sum node weights only.

5.1.1 MLE in Closed Form

As indicated in [6, 9], for a selective SPN, MLE for weights w_{ij} of sum node i is given in closed form:

$$\hat{w}_{ij} = \begin{cases} \dfrac{n_{ij}}{\sum_{j' \in ch(i)} n_{ij'}}, & \text{if } \sum_{j \in ch(i)} w_{ij} = 1 \\ \dfrac{1}{|ch(i)|} \; \forall i \in ch(i), & \text{otherwise} \end{cases} \tag{6}$$

The n_{ij} corresponds to the number of times the link $i \rightarrow j$ is observed from sum node i to its child j according to the samples in the dataset. More precisely, for each sample, two passes are made in the following order: a forward pass and a backward pass. Forward pass computes the output for all nodes. Backward pass determines the used links between sum nodes and their children. For the links observed, the respective counts are incremented. In the backward pass, all links are followed for product nodes while for sum nodes, only the child with positive value is followed and the counts corresponding to the followed links are incremented. [6]

5.1.2 Expectation Maximization (EM)

This section summarizes the description of Expectation Maximization in [6].

Samples in actual datasets often do not contain values for all variables. Variables without a value should be treated as hidden variables. The Expectation Maximization algorithm is well-suited for incomplete data. EM algorithm alternates between two steps named E-step and M-step.

In the E-step, parameters of the model are fixed and the probability for each possible configuration of hidden variables is computed given the state of the model and variables whose values are already known. After this step, the original dataset is augmented to include all possible hidden state configurations and their respective probabilities given model parameters and variables with already known values.

In the M-step, the augmented dataset obtained after E-step is used to compute the MLE for model parameters. If SPN is selective, then the closed-form MLE solution in Section 5.1.1 is again applicable at M-step because the augmented dataset is already complete.

The EM algorithm alternates between these steps until convergence. Initialization of the algorithm is crucial to find a good solution.

Lastly, because the EM algorithm depends on partial derivatives flowing from the root to the leaves, the performance degrades as the network gets deeper. The reason is the deeper the network becomes, the smaller the gradients and their products get. This problem is known as the vanishing gradient problem, which is also very common in deep NNs. In SPNs, this problem can be circumvented with Hard EM, which assigns the most probable value to each hidden variable at the E-step, instead of augmenting the dataset with all possible configurations of hidden variables. [1, 6]

5.1.3 Gradient Ascent

Similar to NNs, maximization of likelihood can also be achieved with Gradient Ascent (GA), which is an iterative optimization technique based on gradients and backpropagation algorithm. SPNs can be trained with GA both generatively [1] and discriminatively [12]. Stochastic and mini-batch versions of GA are also applicable. Similar to EM, GA is also prone to vanishing gradient issue. Therefore, Hard GA is introduced. SPN training follows the same standard recipe as NNs. More detailed explanation is available in [6].

5.2. Structure Learning

In the first paper for SPN [1], authors outline a procedure which starts from a generic and dense SPN structure which is later refined by removing links with zero weights as the dataset is processed. Although this procedure partially learns a structure from data, the first algorithm that builds SPN completely from data is the BuildSPN algorithm presented by [13]. Later, it is followed by the LearnSPN algorithm of [14]. Although LearnSPN is not the current state-of-the-art, it remains to be an important structure learning algorithm for SPNs. [6] In this section, these two algorithms will be explained.

5.2.1 BuildSPN

According to [13], the main idea of BuildSPN is clustering dependent variables under a set of sum nodes which can be interpreted as latent variables. Each of these latent variables models the dependency between its children. Authors of [13] indicate that the initial architecture which uses rectangular regions on image data to utilize spatial relationships in [1] creates artifacts like region boundaries. The structure learned with BuildSPN overcomes such problems because such a structure can learn to put sum nodes on top of spatially local or non-local regions with arbitrary shapes. Therefore, regions processed by learned SPNs are not biased towards any particular shape, and learned SPNs can also model data that does not have spatially local features. As a result, architectures generated by BuildSPN can be seen as a generalization over those obtained in [1].

Briefly, the BuildSPN algorithm starts with partitioning the scope of SPN's root node S. A new partition node P is created under S. The generated partition node is then given two region nodes as children. By partitioning and introducing extra nodes, S effectively distributes its responsibilities to the created region nodes through partition nodes. Then, scope partitioning continues recursively with each subgraph defined by the added region nodes. In reality, the graph created in this fashion is not exactly an SPN. A further conversion from the region graph to an SPN is required but fundamentally, partition nodes can be viewed as product nodes while region nodes can be seen as sum nodes.

5.2.2 LearnSPN

As noted in [14], BuildSPN has some limitations. First of all, variables taking similar values in the dataset are clustered. Therefore, some variables exhibiting strong depen-

dence but taking different values might be separated during the clustering process (For instance, when two variables are negations of one another.). The separation of strongly dependent variables leads to a huge loss in likelihood. Secondly, parameter learning can only be done after the structure is learned, resulting in extra overhead for learning parameters and a possibly sub-optimal structure and parameter estimation. Lastly, the number of nodes in the SPN and the cost of learning are exponential in the number of variables in the worst-case. The authors propose the LearnSPN algorithm to overcome these limitations.

Briefly, the steps in the algorithm are as follows: [6, 14]

- Algorithm takes a matrix as input where each row represents an instance from the dataset and each column corresponds to a variable.
- If the input matrix has a single column, the algorithm returns a leaf containing a univariate distribution whose parameters are estimated with MLE.
- If it has more than one column, LearnSPN performs an independence test on variables:
  - If variables can be partitioned into mutually independent subsets, the algorithm proceeds with the subsets after creating a product node on top of them.
  - Else, when they are not mutually dependent, the algorithm partitions the instances into clusters and proceeds with each cluster. In this case, a sum node is generated on top of them and the weights of sum node links are computed with the ratio of instances in the cluster to the total number of instances before clustering.

When no dependencies between the variables are found, the algorithm returns a completely factorized distribution. When no independent subsets of variables are found, the algorithm returns the kernel density estimate of the distribution.

Note that LearnSPN is a framework algorithm. Choices of methods for variable splitting and instance clustering might be different. In [14], authors chose to use the EM algorithm for clustering instances and G-test as the independence test for splitting the variables.

6. Comparison to Neural Networks

This section explains the similarities, differences between SPNs and NNs as well as the pros and cons of each model.

The authors of [1] compare SPNs with CNNs due to similarity in terms of alternating sum-product operations. They state that the average pooling operation of CNNs resembles marginal inference of SPNs while max-pooling operation in CNNs is similar to MPE inference in SPNs. However, while SPNs have a probabilistic interpretation and a general-purpose usage, CNNs do not have a probabilistic meaning and they are mainly used in tasks requiring utilization of spatial structure in the data.

In [6], SPNs are compared with NNs in general. According to the authors, SPNs show a similarity to NNs in terms of information flow. Therefore, SPNs can be thought of as a form of a feed-forward deep network without special nonlinearity functions. The prevailing training algorithm for NNs is gradient descent. SPNs, on the other hand, can be additionally trained with expectation maximization or other probabilistic methods with better generalization and efficiency. Moreover, there are also several structure learning algorithms available for SPNs. Although there are structure learning algorithms for NNs too, they demand huge computational resources. Therefore, the structure of NNs is often handcrafted. As manually-designed structures are not learned from data, they tend to be huge and computationally expensive. While inference tasks can be done by SPNs exactly and efficiently with a few forward passes, NNs require only one pass but the inference is not exact. SPNs can compute probabilities via marginalization even when some input values are missing. However, NNs require a complete assignment to input values to perform inference.

A variant of SPN is introduced in [11] in order to observe the performance of SPNs when they are trained in a similar way to NNs. This variant selects an SPN structure at random, learns its parameters in the NN style and regularizes it with the probabilistic dropout technique introduced in the same work. This SPN variant is called Random Tensorized SPN (RAT-SPN). Authors use a hybrid learning objective combining cross-entropy and log-likelihood based on a control parameter, which switches the training schema of RAT-SPN between discriminative (with cross-entropy) and generative (with log-likelihood) learning. Training RAT-SPNs in a generative fashion makes them more robust against missing features and uncertainties because of the nature of generative models. As generative models represent full joint distribution, one can compute the likelihood of a sample with respect to the modeled distribution and can assess whether a sample is an inlier or outlier. Similarly, RAT-SPNs can handle missing inputs by marginalization when they learn the full joint distribution. For SPNs, these kinds of probabilistic computations are tractable. On the other hand, standard MLPs do not already represent a joint distribution and even current generative NN models such as Generative Adversarial Networks (GAN) and Variational Autoencoders (VAE) can not evaluate the likelihood for data exactly. Yet, authors count the constrained structural properties of SPNs, the prevalence of PGM-style learning algo-

rithms for SPN training, the inferior representation power of SPNs compared to NNs as the reasons for why NNs are primarily used instead of SPNs for solving various tasks.

As mentioned in Section 3.2, DSPNs are presented by [4] to process sequential data with SPNs. The authors report that DSPNs are inferior to RNNs in terms of expressiveness because DSPNs are limited to sum and product nodes while RNNs benefit non-linearities to increase expressive power. However, they note that DSPNs are easier to train than RNNs due to highly non-convex objective function and vanishing or exploding gradient issues of RNNs.

The superiority of NNs to SPNs in terms of expressiveness is also mentioned in [3, 8]. As noted in [3], SPNs can perform discriminative and generative tasks with a single network definition while CNNs are dedicated to discriminative problems and GANs are assigned to generative tasks. However, GANs only allow efficient sampling despite being generative models, and contrary to SPNs, exact tractable inference is not possible with GANs, but the inference is approximated through samples generated by the network.

7. Conclusion

In this report, Sum-Product Networks (SPNs), their definition and fundamental properties, different architectures, some of the inference, parameter, and structure learning algorithms were discussed, and they are compared with NNs.

SPNs are a type of deep network formed as a directed acyclic graph which contains sum and product nodes in an alternating fashion as well as leaf nodes representing distributions. When completeness and decomposability properties hold, SPNs can compute mixture and product of base distributions represented by the leaves. As a result, complete and decomposable SPNs and their sub-SPNs all represent different distributions. Therefore, completeness and decomposability properties attribute a probabilistic interpretation to SPNs. Moreover, this particular subset of SPNs can perform probabilistic inference tasks such as joint, marginal, and conditional probability computations in an exact and tractable way. This property is one of the most promising features of SPNs because probabilistic inference cannot be done efficiently or exactly with PGMs and NNs.

SPNs are related to both PGMs and NNs. The representation power of SPNs lies between these two classes where the largest and smallest set of representable tractable functions belong to NNs and PGMs respectively. Although NNs currently deliver state-of-the-art performance in different tasks from various domains such as spatial and temporal data, researchers introduced specialized SPN architectures, which maintains the probabilistic nature of SPNs, and tried to address many of such problems. The performance of SPNs at the moment follows NNs from behind. However, SPNs have some remarkable features that are currently unavailable for NNs, such as providing efficient and exact inference, having an inherent probabilistic construction, accepting missing features, and being able to learn the structure from the data. Because of these advantages of SPNs over NNs, SPNs are considered as a possible alternative to NNs by researchers.

References

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

[2] Cory J Butz, Jhonatan S Oliveira, André E dos Santos, and André L Teixeira. Deep convolutional sum-product networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 3248–3255, 2019.

[3] Jos van de Wolfshaar and Andrzej Pronobis. Deep generalized convolutional sum-product networks, 2020.

[4] Mazen Melibari, Pascal Poupart, Prashant Doshi, and George Trimponias. Dynamic sum product networks for tractable inference on sequence data. In *Conference on Probabilistic Graphical Models*, pages 345–355, 2016.

[5] James Martens and Venkatesh Medabalimi. On the expressive efficiency of sum product networks. *arXiv preprint arXiv:1411.7717*, 2014.

[6] Iago París, Raquel Sánchez-Cauce, and Francisco Javier Díez. Sum-product networks: A survey. *arXiv preprint arXiv:2004.01167*, 2020.

[7] Or Sharir and Amnon Shashua. Sum-product-quotient networks. In *International Conference on Artificial Intelligence and Statistics*, pages 529–537. PMLR, 2018.

[8] Xiaoting Shao, Alejandro Molina, Antonio Vergari, Karl Stelzner, Robert Peharz, Thomas Liebig, and Kristian Kersting. Conditional sum-product networks: Imposing structure on deep probabilistic architectures. *arXiv preprint arXiv:1905.08550*, 2019.

[9] Robert Peharz, Robert Gens, and Pedro Domingos. Learning selective sum-product networks. In *LTPM workshop*, 2014.

[10] 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.

[11] Robert Peharz, Antonio Vergari, Karl Stelzner, Alejandro Molina, Martin Trapp, Kristian Kersting, and Zoubin Ghahramani. Probabilistic deep learning using random sum-product networks. *arXiv preprint arXiv:1806.01910*, 2018.

[12] Robert Gens and Pedro Domingos. Discriminative learning of sum-product networks. In *Advances in Neural Information Processing Systems*, pages 3239–3247, 2012.

[13] Aaron Dennis and Dan Ventura. Learning the architecture of sum-product networks using clustering on variables. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'12, page 2033–2041, Red Hook, NY, USA, 2012. Curran Associates Inc.

[14] Robert Gens and Domingos Pedro. Learning the structure of sum-product networks. In *International conference on machine learning*, pages 873–880, 2013.