Sum-Product Networks: A Survey

Raquel Sánchez-Cauce, Iago París, and Francisco Javier Díez

Abstract—A sum-product network (SPN) is a probabilistic model, based on a rooted acyclic directed graph, in which terminal nodes represent probability distributions and non-terminal nodes represent convex sums (weighted averages) and products of probability distributions. They are closely related to probabilistic graphical models, in particular to Bayesian networks with multiple context-specific independencies. Their main advantage is the possibility of building tractable models from data, i.e., models that can perform several inference tasks in time proportional to the number of edges in the graph. They are somewhat similar to neural networks and can address the same kinds of problems, such as image processing and natural language understanding. This paper offers a survey of SPNs, including their definition, the main algorithms for inference and learning from data, several applications, a brief review of software libraries, and a comparison with related models.

Index Terms—Sum-product networks, probabilistic graphical models, Bayesian networks, machine learning, deep neural networks

1 Introduction

Sum-product networks (SPNs) were proposed by Poon and Domingos [1] in 2011 as a modification of Darwiche's [2], [3] arithmetic circuits. An SPN is a directed graph that represents a probability distribution resulting from a hierarchy of distributions combined in the form of mixtures (sum nodes) and factorizations (product nodes), as shown in Fig. 1. SPNs, like arithmetic circuits, can be built by transforming a probabilistic graphical model [4], such as a Bayesian network or a Markov network, but they can also be learned from data. The main advantage of SPNs is that several inference tasks can be performed in time proportional to the number of edges in the graph.

In these ten years there has been great progress: numerous algorithms have been proposed for inference and learning, and SPNs have been successfully applied in several areas, including computer vision and natural language processing, where probabilistic models could not compete with neural networks. The understanding of SPNs has also improved and some aspects can now be explained more clearly than in the original publications. For example, the first two papers about SPNs [1], [5] presented them as an efficient representation of network polynomials, while most of the later references, beginning with [6], define them as the composition of probability distributions, which is, in our view, more intuitive and much easier to understand. Consistency was initially one of the defining properties of SPNs, which made them more general than arithmetic circuits, but it later became clear that decomposability, a stronger but much more intuitive property, suffices to build SPNs for practical applications. In contrast, selectivity (called determinism in arithmetic circuits), which was not mentioned in the original paper [1], proved to be relevant for some inference tasks and for parameter learning [7], [8]. Additionally, some of the algorithms for SPNs are only sketched in [1], without much detail or formal proofs, and one of them turned out to be correct only for selective SPNs. Other basic algorithms are scattered over several papers, each using a different mathematical notation.

For these reasons we decided to write a survey explaining the main concepts and algorithms for SPNs. We have intentionally avoided any reference to network polynomials, which has forced us to develop new proofs for some algorithms and propositions, alternative to those found in other references, such as [9]. We have also reviewed the literature on SPNs, with especial emphasis on their applications.

The rest of this paper is structured as follows. The two subsections of this introduction highlight the significance of SPNs by comparing them with probabilistic graphical models and neural networks, respectively. After some mathematical preliminaries (Section 2), we introduce the basic definitions of SPNs (Section 3) and the main algorithms for inference (Section 4), parameter learning (Section 5), and structural learning (Section 6). We then review some applications in several areas (Section 7), a few open-source packages (Section 8), and some extensions (Section 9). Section 10 contains the conclusions. Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPAMI.2021.3061898, compares SPNs with arithmetic circuits. Appendix B, available in the online supplemental material, analyzes the interpretation of sum nodes as weighted averages of conditional probabilities, and Appendix C, available in the online supplemental material, contains the proofs of all the propositions.

1.1 SPNs Versus Probabilistic Graphical Models (PGMs)

SPNs are similar to PGMs, such as Bayesian networks (BNs) and Markov networks (also called Markov random fields) [4], [10], in their ability to compactly represent probability distributions. The main difference is that in a PGM every

• *The authors are with the Department of Artificial Intelligence, Universidad Nacional de Educación a Distancia (UNED), 28015 Madrid, Spain. E-mail: {rsanchez, iagoparis, fjdiez}@dia.uned.es.*

Manuscript received 23 Mar. 2020; revised 6 Feb. 2021; accepted 16 Feb. 2021. Date of publication 25 Feb. 2021; date of current version 3 June 2022. (Corresponding author: Francisco Javier Díez.) Recommended for acceptance by D. Lowd. Digital Object Identifier no. 10.1109/TPAMI.2021.3061898

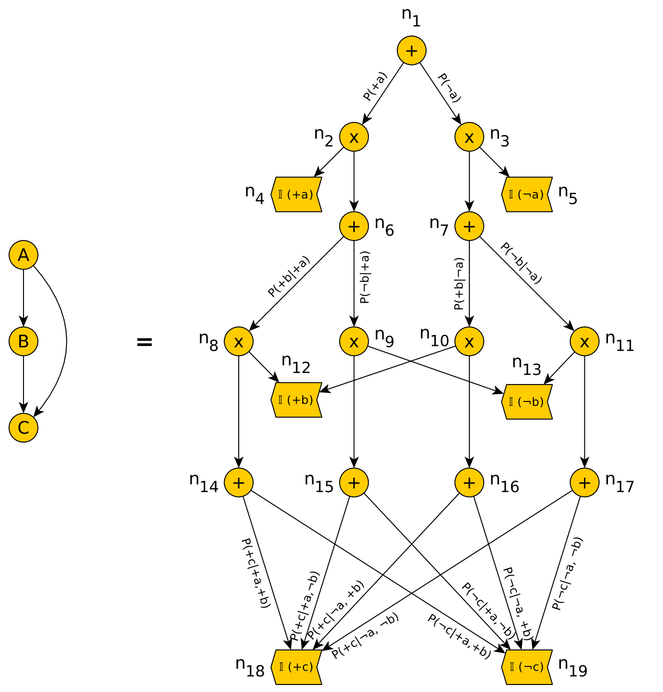

Fig. 1. A Bayesian network (left) and an equivalent SPN (right). The 6 terminal nodes in the SPN are indicators for the 3 variables in the model, A, B, and C; they are the input of the SPN for every configuration of these variables, including partial configurations. The root node, n_1, computes the joint and marginal probabilities.

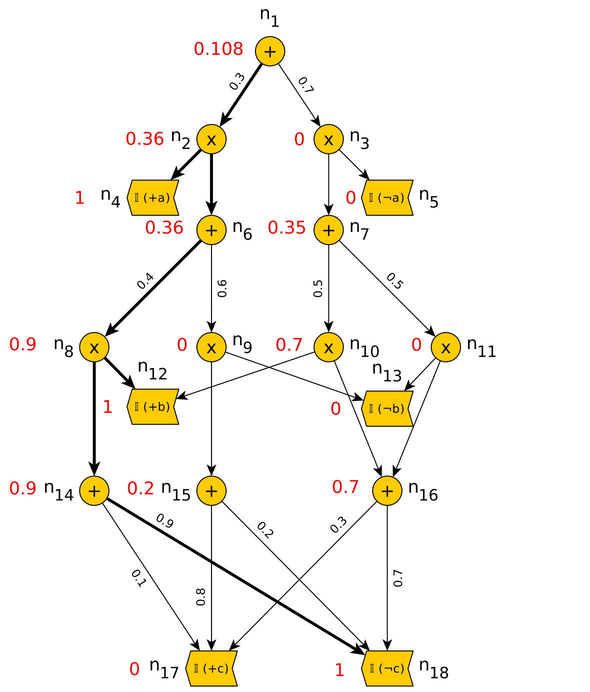

Fig. 2. If $P(c \mid \neg a, +b) = P(c \mid \neg a, \neg b)$ for every value of C (context-specific independence of B and C given $\neg a$), then nodes n_{16} and n_{17} in Fig. 1 can be coalesced into node n_{16} in this figure. The numbers in red are the values $S_i(\mathbf{v})$ for $\mathbf{v} = (+a, +b, \neg c)$.

node represents a variable and—roughly speaking—edges represent probabilistic dependencies, sometimes due to causal influences, while in an SPN every node represents a probability distribution. PGMs and other factored probability distributions can be compiled into arithmetic circuits or SPNs [11]. In general the graph of a PGM is more compact than the compiled SPN, as shown in Fig. 1. (For the "decompilation" of SPNs into BNs, see [12], [13].)

Inference in BNs in NP-hard [14], while SPNs compute marginal and conditional probabilities in time proportional to the number of edges in the graph. Therefore it is not surprising that the compilation of BNs sometimes generates very large graphs. In particular, some quadratic-size BNs (for example, $n \times n$ lattices) generate exponential-size SPNs. But if a family of BNs can be evaluated in polynomial time with a general algorithm, such as variable elimination, then the compilation generates polynomial-size SPNs [11]. Additionally, context-specific independencies in a BN [15] can significantly reduce the size of the corresponding SPN, as shown in Fig. 2.

More importantly, while PGMs learned from data are usually intractable—except for small problems or for specific types of models with limited expressiveness, such as the naïve Bayes—the algorithms presented in Section 6 can build tractable SPNs that yield excellent approximations both for generative and discriminative tasks. (An exception are latent tree models, a new type of PGM that share the same advantages of SPNs [16].)

In contrast, BNs can be built using causal knowledge elicited from human experts and there is a large body of recent research on building causal BNs from experimental and/or observational data, under certain conditions [17]. It is also possible to combine causal knowledge and data, and even to build BNs interactively [18]. All these options are currently impossible with SPNs. Additionally, the independencies in a BN or in a Markov model are easier to read than those in an SPN. On the other hand, the graph of an SPN can sometimes be built from human knowledge by representing part/subpart and class/subclass hierarchies—see [1, Sec. 5] for an example.

In conclusion, each type of model has advantages and disadvantages, and the choice for a real-world application must take into account the size of the problem, the amount of knowledge and data available, and the explanations required by the user.

1.2 SPNs Versus Neural Networks

SPNs can be seen as a particular type of feedforward neural networks because there is a flow of information from the input nodes (the leaves) to the output node (the root), but in this paper we reserve the term "neural network" (NN) for the models structured in layers connected by the standard operators: sigmoid, ReLU, softmax, etc.

The main difference is that SPNs have a probabilistic interpretation while standard NNs do not. Inference is also different: computing a posterior probability requires two passes, and the most probable explanation (MPE)—in selective SPNs, defined below—can be found by backtracking from the root to the leaves, as explained in Section 4. Additionally, SPNs can do inference with partial information (i.e., when the values of some of the variables are unknown), while in a NN it is necessary to assign a value to each input node.

From the point of view of parameter learning, NNs are usually trained with gradient descent or variations thereof, while SPNs can also be trained with several probabilistic algorithms, such as EM and Bayesian methods, which are much more efficient and have lower risk of overfitting (cf. Section 5).

When building practical applications, the main difference is the possibility of determining the structure of an SPN from data, looking for a balance between model complexity and accuracy. In contrast, NNs are usually designed by hand and it is necessary to examine different architectures of different sizes with different hyperparameters, in a trial-and-error approach, until a satisfactory model is found. For this reason, NNs that have succeeded in practical applications are usually very big, and training them requires huge computational power. There are some proposals to learn the structure of NNs using evolutionary computation, which yields more efficient graphs, but this also requires immense computational power [19], [20].

In spite of these advantages, NNs are clearly superior to SPNs in many tasks, at least at the moment. For example, in 2012 an SPN by Gens and Domingos [5] achieved a classification accuracy of 84 percent for the CIFAR-10 image dataset, one of the highest scores at the time, but deep NNs have amply surpassed that result, reaching an impressive 99.7 percent accuracy.[1]

Nevertheless, in Section 7 we mention several applications in which SPNs are competitive with NNs and superior in some aspects. For instance, some SPNs [21], [22] have recently attained a classification accuracy comparable to deep NNs for MNIST and other image datasets, with the advantages of being probabilistic generative models and being much more robust to missing features. For another example, Stelzner *et al.* [23] proved that the attend-infer-repeat (AIR) framework used for object detection and location is much more efficient when the variational autoencoders (VAEs) are replaced by SPNs: they achieved an improvement in speed of an order of magnitude, with slightly higher accuracy, as well as robustness against noise. Other examples can be found in Sections 7 and 9.

For a more detailed comparison of SPNs with NNs, VAEs, generative adversarial networks (GANs), and other models, see [21].

2 Mathematical Preliminaries

2.1 Configurations of Variables

We denote by a capital letter, V, a variable and by the corresponding lowercase letter, v, any value of V. Similarly a boldface capital letter denotes a set of variables, $\mathbf{V} = \{V_1, \ldots, V_n\}$, the corresponding lowercase letter denotes any of its configurations, $\mathbf{v} = (v_1, \ldots, v_n)$, and $\mathrm{conf}(\mathbf{V})$ is the set of all the configurations of $\mathbf{V}$. The empty set has only one configuration, represented by $\blacklozenge$.

We denote by $\mathrm{conf}^*(\mathbf{V})$ the set of all the configurations of $\mathbf{V}$ and its subsets:[2]

$$\mathrm{conf}^*(\mathbf{V}) = \bigcup_{\mathbf{X} \in \mathscr{P}(\mathbf{V})} \mathrm{conf}(\mathbf{X}) = \{\mathbf{x} \,|\, \mathbf{X} \subseteq \mathbf{V}\}. \tag{1}$$

We can think of $\mathrm{conf}^*(\mathbf{V}) \setminus \mathrm{conf}(\mathbf{V})$ as the set of *partial configurations* of $\mathbf{V}$, i.e., the configurations in which only some of the variables in $\mathbf{V}$ have an assigned value.

If $\mathbf{X} \subseteq \mathbf{V}$, the *projection* (sometimes called *restriction*) of a configuration $\mathbf{v}$ of $\mathbf{V}$ onto $\mathbf{X}$, $\mathbf{v}^{\downarrow \mathbf{X}}$, is the configuration of $\mathbf{X}$ such that every variable $V \in \mathbf{X}$ takes the same value as in $\mathbf{v}$; for example, $(+x_1, +x_2, \neg x_3)^{\downarrow \{X_1, X_3\}} = (+x_1, \neg x_3)$ and $(+x_1, +x_2, \neg x_3)^{\downarrow \varnothing} = \blacklozenge$. In order to simplify the notation, when $\mathbf{X}$ has a single variable, V, we will write v instead of (v) and $\mathbf{v}^{\downarrow V}$ instead of $\mathbf{v}^{\downarrow \{V\}}$; for example, $(+x_1, +x_2, \neg x_3)^{\downarrow X_2} = +x_2$.

Given two configurations, $\mathbf{x}$ and $\mathbf{y}$, of two disjoint sets, $\mathbf{X}$ and $\mathbf{Y}$, the *composition* of them, denoted by $\mathbf{xy}$, is the configuration of $\mathbf{X} \cup \mathbf{Y}$ such that $(\mathbf{xy})^{\downarrow \mathbf{X}} = \mathbf{x}$ and $(\mathbf{xy})^{\downarrow \mathbf{Y}} = \mathbf{y}$. For example, $(+x_1, +x_2)(\neg x_3) = (+x_1, +x_2, \neg x_3)$.

When $\mathbf{X} \subseteq \mathbf{V}$, a configuration $\mathbf{x}$ is *compatible* with configuration $\mathbf{v}$ if $\mathbf{x} = \mathbf{v}^{\downarrow \mathbf{X}}$, i.e., if every variable $V \in \mathbf{X}$ has the same value in both configurations. All configurations are compatible with $\blacklozenge$.

Definition 1: *Given a value v of a variable $V \in \mathbf{V}$, we define the* indicator function, $\mathbb{I}_v : \mathrm{conf}(\mathbf{V}) \mapsto \{0, 1\}$, *as follows:*

$$\mathbb{I}_v(\mathbf{x}) = \begin{cases} 1 & \text{if } V \notin \mathbf{X} \vee v = \mathbf{x}^{\downarrow V} \\ 0 & \text{otherwise .} \end{cases} \tag{2}$$

If all the variables in $\mathbf{V}$ are binary, then there are $2n$ indicator functions.

Example 2: If $\mathbf{V} = \{V_0, V_1\}$ and the domains of these variables are $\{+v_0, \neg v_0\}$ and $\{+v_1, \neg v_1\}$ respectively, then $\mathbb{I}_{+v_0}(+v_0, +v_1) = 1$, $\mathbb{I}_{+v_0}(\neg v_0, +v_1) = 0$, $\mathbb{I}_{+v_0}(+v_1) = \mathbb{I}_{+v_0}(\neg v_1) = \mathbb{I}_{+v_0}(\blacklozenge) = 1$, etc.

2.2 Probability Distributions

In order to simplify the statement of definitions and propositions, we assume in this section that all variables have finite states, but these results can be generalized to variables defined on $\mathbb{R}$ and to more complex types of variables.

Definition 3: *A* probability distribution *defined on* $\mathbf{V}$ *is a function* $P : \mathrm{conf}(\mathbf{V}) \mapsto \mathbb{R}$ *such that*

$$P(\mathbf{v}) \geq 0, \tag{3}$$

$$\sum_{\mathbf{v}} P(\mathbf{v}) = 1. \tag{4}$$

Definition 4: *An* extended probability distribution *defined on* $\mathbf{V}$ *is a function* $P : \mathrm{conf}^*(\mathbf{V}) \mapsto \mathbb{R}$ *such that the restriction of* P *to* $\mathrm{conf}(\mathbf{V})$ *is a probability distribution and for every configuration* $\mathbf{x}$ *such that* $\mathbf{X} \subset \mathbf{V}$,

$$P(\mathbf{x}) = \sum_{\mathbf{v} \,|\, \mathbf{v}^{\downarrow \mathbf{X}} = \mathbf{x}} P(\mathbf{v}). \tag{5}$$

Every probability distribution can be extended by computing its marginal probabilities with Equation (5).

1. See https://paperswithcode.com/sota/image-classification-on-cifar-10

2. If the domains of the variables in $\mathbf{V}$ overlap, each configuration should be tagged with the subset $\mathbf{X}$ to distinguish those that consist of the same values; for example, to distinguish $(0,0)_{\{V_1,V_2\}}$ from $(0,0)_{\{V_1,V_3\}}$.

Proposition 5: *A convex combination (weighted average) of extended probability distributions defined on the same set of variables is an extended probability distribution.*

Proposition 6: *A product of extended probability distributions defined on disjoint sets of variables is an extended probability distribution.*

2.3 MAP, MPE, and MAX Inference

In some inference tasks $\mathbf{e}$ denotes the evidence, i.e., the values observed for a set of variables $\mathbf{E}$ (for example, the symptoms and signs of a medical examination or the pixels in an image), and $\mathbf{X}$ the variables of interest (for example, the possible diagnostics or the objects that may be present in the image), with $\mathbf{X} \cap \mathbf{E} = \varnothing$. In this context, $P(\mathbf{x} \,|\, \mathbf{e})$ is called the *posterior probability*.

The *maximum a-posteriori* (MAP) configuration is

$$\mathrm{MAP}(\mathbf{e}, \mathbf{X}) = \arg\max_{\mathbf{x}} P(\mathbf{x} \,|\, \mathbf{e}). \tag{6}$$

Therefore, MAP inference divides the variables into three disjoint sets: observed variables ($\mathbf{E}$), variables of interest ($\mathbf{X}$), and hidden variables ($\mathbf{H} = \mathbf{V} \setminus (\mathbf{E} \cup \mathbf{X})$).

The *most probable explanation* is the configuration of $\mathbf{X} = \mathbf{V} \setminus \mathbf{E}$ that maximizes the posterior probability

$$\mathrm{MPE}\,(\mathbf{e}) = \arg\max_{\mathbf{x}} P(\mathbf{x} \,|\, \mathbf{e}). \tag{7}$$

MPE is a special case of MAP in which $\mathbf{H} = \varnothing$, i.e., every variable that is not observed is a variable of interest. In general, MAP inference is much harder than MPE [24].

Finally, MAX is a special case of MPE in which all the variables are of interest, i.e., $\mathbf{X} = \mathbf{V}$ and $\mathbf{H} = \mathbf{E} = \varnothing$. The MAX configuration is the configuration of $\mathbf{X}$ that maximizes the probability

$$\mathrm{MAX}\,(\mathbf{x}) = \arg\max_{\mathbf{x}} P(\mathbf{x}). \tag{8}$$

MPE and MAP are relevant when we wish to know the most probable configuration of the variables of interest $\mathbf{X}$ (for example, the possible diagnostics), which is different from finding the most probable value for each variable in $\mathbf{X}$, as we will see in Example 25. MAP is relevant when some unobserved variables are not of interest; for example, $\mathbf{H}$ may represent the tests not performed: these variables are neither observed nor part of the diagnosis. See also [4, Secs. 2.1.5.2 and 2.1.5.3], where MPE and MAP are called "MAP" and "marginal MAP" respectively. The definition of MAX will be useful in Section 4.3.

2.4 Basic Definitions About Graphs

Graphs have many applications in computer science. We describe here the type of graph used to build SPNs.

A directed *graph* consists of a set of nodes and a set of directed edges. When there is an edge $n_i \rightarrow n_j$ we say that n_i is a *parent* of n_j and n_j is a *child* of n_i; there cannot be another edge from n_i to n_j. Given a node n_i, we denote by $pa(i)$ the set of indices of its parents and by $ch(i)$ the set of indices of its children. For example, in Fig. 1, $ch(1) = \{2, 3\}$. Node n_k is a *descendant* of n_i if it is a child of n_i or a child of a descendant of n_i; we also say that n_i is an *ancestor* of n_k.

A cycle of length l consists of a set of l nodes and l edges $\{n_1 \rightarrow n_2, n_2 \rightarrow n_3, \ldots, n_{l-1} \rightarrow n_l, n_l \rightarrow n_1\}$. A graph that contains no cycles, i.e., no node is a descendant of itself, is *acyclic*. An acyclic directed graph (ADG) is *rooted* if there is only one node (the *root*, denoted by n_r) having no parents. *Terminal nodes*, also called *leaves*, are those that do not have children.

A *directed tree* is a rooted ADG in which every node has one parent, except the root. In this paper when we say "a tree" we mean "a directed tree".

3 Basic Definitions of SPNs

Given that this is an introductory paper, we will assume that every leaf node represents a probability distribution for one finite-state variable. In practice, a leaf in an SPN may also represent a univariate distribution, for example, Gaussian [6], [25], Poisson [26], piecewise polynomial [27], etc., or a multivariate probability density, such as a multivariate Gaussian [28], [29] or a Chow-Liu tree [30]. Some of the results presented here generalize straightforwardly, but others require a more sophisticate mathematical treatment.

3.1 Structure of an SPN

An *SPN* $\mathcal{S}$ is a rooted acyclic directed graph such that:

- each leaf node represents a probability distribution for a finite-states variable, V,
- all the other nodes are either of type sum or product, and
- every edge $n_i \rightarrow n_j$ outgoing from a sum node has an associated weight, $w_{ij} > 0$.[3]

We will assume, unless otherwise stated, that all SPNs are normalized, i.e., for every sum node n_i,

$$\sum_{j \in ch(i)} w_{ij} = 1. \tag{9}$$

An SPN can be built bottom-up beginning with sub-SPNs of one node and joining them with sum and product nodes. All the definitions of SPNs can be established recursively, first for one-node SPNs, and then for sum and product nodes. Similarly, all the properties of SPNs can be proved by structural induction.

The *scope* of a node n_i is denoted by $\mathrm{sc}(n_i)$. For a leaf node, it is the set of variables on which the probability distribution is defined. The *scope* of a non-terminal node n_i is the union of the scopes of its children

$$\mathrm{sc}(n_i) = \bigcup_{j \in ch(i)} \mathrm{sc}(n_j). \tag{10}$$

The *scope* of an SPN $\mathcal{S}$, denoted by $\mathrm{sc}(\mathcal{S})$, is the scope of its root, $\mathrm{sc}(n_r)$. The variables in the scope of an SPN are sometimes called *model variables*—in contrast with *latent variables*, which we present below. We define $\mathrm{conf}(\mathcal{S}) = \mathrm{conf}(\mathrm{sc}(\mathcal{S}))$ and $\mathrm{conf}^*(\mathcal{S}) = \mathrm{conf}^*(\mathrm{sc}(\mathcal{S}))$.

3. SPNs are usually defined with $w_{ij} \geq 0$, but edges with $w_{ij} = 0$ should be removed to avoid unnecessary terms in Eqs. (12) and (19).

A sum node is *complete* if all its children have the same scope. An SPN is *complete* if all its sum nodes are complete. (In arithmetic circuits this property is called *smoothness*.)

A product node is *decomposable* if its children have pairwise disjoint scopes. An SPN is *decomposable* if all its product nodes are decomposable.

Proposition 7: *A product node n_i is decomposable if and only if no node in the SPN is a descendant of two different children of n_i.*

In the rest of the paper we assume that all the SPNs are complete and decomposable.

3.2 Node Values and Probability Distributions

Definition 8 (Value $S_i(\mathbf{x})$): *Let n_i be a node of $\mathcal{S}$ and $\mathbf{x} \in \mathrm{conf}^*(\mathcal{S})$. If n_i is a leaf node with extended probability distribution $P(v)$, then*

$$S_i(\mathbf{x}) = P(\mathbf{x}^{\downarrow V}); \tag{11}$$

if it is a sum node,

$$S_i(\mathbf{x}) = \sum_{j \in ch(i)} w_{ij} \cdot S_j(\mathbf{x}), \tag{12}$$

and if it is a product node,

$$S_i(\mathbf{x}) = \prod_{j \in ch(i)} S_j(\mathbf{x}). \tag{13}$$

When the probability distribution for a leaf node n_i is degenerate, i.e., there is one value v^* of V such that $P(v^*) = 1$ and $P(v) = 0$ otherwise, Equations (2) and (11) lead to $S_i(\mathbf{x}) = \mathbb{I}_{v^*}(\mathbf{x})$. This is the reason for using indicators as leaf nodes, as in Figs. 1 and 2. However, SPNs (unlike arithmetic circuits) accept more general distributions in their leaves.

Definition 9 (Value $S(\mathbf{x})$): *The* value $S(\mathbf{x})$ *returned by the SPN is the value of the root, $S_r(\mathbf{x})$.*

Theorem 10: *For every node n_i in an SPN, the function $P_i : \mathrm{conf}^*(\mathcal{S}) \mapsto \mathbb{R}$, such that*

$$P_i(\mathbf{x}) = S_i(\mathbf{x}), \tag{14}$$

is an extended probability distribution defined on $\mathbf{V} = \mathrm{sc}(n_i)$.

Please note that S_i is defined on the configurations of the scope of the whole network, $\mathrm{conf}^*(\mathcal{S})$, while P_i is defined on the configurations of the scope of the node, $\mathrm{conf}^*(\mathrm{sc}(n_i)) \subseteq \mathrm{conf}^*(\mathcal{S})$.

This theorem guarantees that $P(\mathbf{x}) = P_r(\mathbf{x})$ is an extended probability distribution, i.e., that the SPN properly computes a probability distribution and all its marginals. The proof of the theorem, in Appendix C, available in the online supplemental material, relies on the completeness and decomposability of the SPN, which are closely related with Propositions 5 and 6, respectively.

3.3 Selective SPNs

We introduce now a particular type of SPNs that have interesting properties for MPE inference and parameter learning and for the interpretation of sum nodes.

When computing $S(\mathbf{x})$ for a given $\mathbf{x} \in \mathrm{conf}^*(\mathcal{S})$, probability flows from the leaves to the root (cf. Definition 8). Equation (12) says that all the children of a sum node n_i can contribute to $S_i(\mathbf{x})$. However, n_i may have the property that for every configuration $\mathbf{v} \in \mathrm{conf}(\mathcal{S})$ at most one child makes a positive contribution, i.e., $S_j(\mathbf{x}) = 0$ for the other children of n_i. We then say that n_i is *selective* [31]. The formal definition is as follows.

Definition 11: *A sum node n_i in an SPN is* selective *if*

$$\forall \mathbf{v} \in \mathrm{conf}(\mathcal{S}), \exists j^* \in ch(i) \,|\, j \in ch(i), j \neq j^* \Rightarrow S_j(\mathbf{v}) = 0. \tag{15}$$

Please note that this definition says "conf", not "conf*". Therefore even if n_i is selective there may be a partial configuration $\mathbf{x} \in \mathrm{conf}^*(\mathcal{S}) \setminus \mathrm{conf}(\mathbf{V})$ such that several children of n_i make positive contributions to $S_i(\mathbf{x})$.

Definition 12: *An SPN is* selective *if all its sum nodes are selective.*

Whether a node is selective (or not) does not change if its weights are replaced with different positive numbers, which implies that selectivity only depends on the structure of the SPN, not on its weights.

Example 13: Given the SPN in Fig. 2, we can check that if $\mathbf{v} = (+a, +b, \neg c)$ then $S_2(\mathbf{v}) = 0.36$ and $S_3(\mathbf{v}) = 0$. Only node n_2 makes a positive contribution to $S_1(\mathbf{v})$, so Property 15 holds for this $\mathbf{v}$ with $j^* = 2$. We can make the same check for each of the 6 sum nodes and each of the 8 configurations of $\{A, B, C\}$ in order to conclude that this SPN is selective.

The main difference between arithmetic circuits and SPNs is that the former are deterministic (i.e., selective) while the latter are not necessarily so [32].

3.4 Sum Nodes That Represent Model Variables

Several papers about SPNs say that sum nodes represent latent random variables. However, in this section we show that in some cases sum nodes represent model variables.

Definition 14: *Let n_i be a sum node having m children and $V \in \mathrm{sc}(n_i)$ a variable with m states. Let σ be a one-to-one function $\sigma : \{1, \ldots, m\} \mapsto ch(i)$. If for every $j \in \{1, \ldots, m\}$ either $\mathbb{I}_{v_j}$ is a child of $n_{\sigma(j)}$ (and hence a grandchild of n_i) or $\mathbb{I}_{v_j} = n_{\sigma(j)}$ (i.e., the indicator itself is a child of n_i), we then say that n_i* represents *variable V.*

Example 15: Node n_{14} in Fig. 1 represents variable C, with $\sigma(1) = 17$ and $\sigma(2) = 18$, because $\mathbb{I}_{c_1} = \mathbb{I}_{+c} = n_{17} = n_{\sigma(1)}$, and $\mathbb{I}_{c_2} = \mathbb{I}_{\neg c} = n_{18} = n_{\sigma(2)}$. Nodes 15, 16, and 17 also represent C for the same reason.

Node n_6 represents variable B, with $\sigma(1) = 8$ and $\sigma(2) = 9$, because $\mathbb{I}_{b_1} = \mathbb{I}_{+b}$ is a child of $n_{\sigma(1)} = n_8$ and $\mathbb{I}_{b_2} = \mathbb{I}_{\neg b}$ is a child of $n_{\sigma(2)} = n_9$. For analogous reasons node n_7 also represents B and n_1 represents A.

In Appendix B, available in the online supplemental material, we prove that when the root node of an SPN represents a variable V, this node can be interpreted as the weighted average of the conditional probabilities given V, with $w_{i,\sigma(j)} = P(v_j)$. Similarly, if every ancestor of a sum

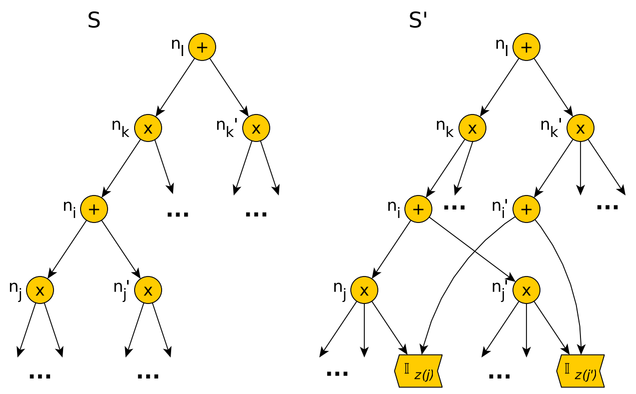

Fig. 3. Augmentation of an SPN, assuming that n_i is not selective in $\mathcal{S}$. This process adds an indicator $\mathbb{I}_{z(j)}$ for every child n_j of n_i. Node $n_{i'}$ is added to restore the completeness of n_l in $\mathcal{S}'$.

node n_i represents a variable, then n_i can be interpreted as the weighted average of the conditional probabilities given the scenario (i.e., the configuration of variables) defined by the path from the root to n_i.

Proposition 16: *Every sum node that represents a model variable is selective.*

This proposition leads to a sufficient condition for an SPN to be selective. For example, according to Definition 14, every node in Figs. 1 and 2 represents a variable, which implies that every node is selective and consequently both SPNs are selective. In the next section we will use this property to transform any non-selective SPN into selective.

3.5 Augmented SPN

The goal of augmenting a non-selective SPN $\mathcal{S}$ [7], [9] is to transform it into a selective SPN $\mathcal{S}'$ that (after marginalizing out the variables introduced in this process) represents the same probability distribution. For every non-selective node n_i in $\mathcal{S}$ the augmentation consists in adding a new variable Z, with as many states as children of n_i, so that n_i represents Z in $\mathcal{S}'$. The process is as follows. For every child n_j we add a state $z(j)$ to Z. If n_j is a product node, we add the indicator $\mathbb{I}_{z(j)}$ as a child of n_j, as shown in Fig. 3; if n_j is a terminal node, we insert a product node, make n_j a child of the new node (instead of being a child of n_i) and add $\mathbb{I}_{z(j)}$ as the second child of the new node. In the resulting SPN, $\mathcal{S}'$, n_i represents variable Z (because of Definition 14) and is therefore selective.

However this transformation of the SPN may cause an undesirable side effect. Let us assume, as shown in Fig. 3, that n_i has a parent, n_k, and n_l is a parent of both n_k and $n_{k'}$. Even though n_l was complete in $\mathcal{S}$, the addition of Z has made this node incomplete in $\mathcal{S}'$ because $Z \in \mathrm{sc}(n_i)$ and $Z \in \mathrm{sc}(n_k)$ but $Z \notin \mathrm{sc}(n_{k'})$. It is then necessary to make $Z \in \mathrm{sc}(n_{k'})$ in order to restore the completeness of n_l. So we create a new sum node, $n_{i'}$, and make it a parent of all the indicators of Z, $\{\mathbb{I}_{z_1}, \ldots, \mathbb{I}_{z_m}\}$ (see again Fig. 3); the weights for $n_{i'}$ can be chosen arbitrarily provided that they are all positive and their sum is 1. If $n_{k'}$ is a product node, then we add $n_{i'}$ as a child of $n_{k'}$. If $n_{k'}$ is a terminal node, we insert a product node, making both $n_{i'}$ and $n_{k'}$ children of this new node. If n_l has other children, such as $n_{k''}$, or ancestral sum nodes in $\mathcal{S}$, then we must make each one of them a parent or a grandparent of n'_i, as we did for n'_k.

Given that variable Z was not in the scope of the original SPN, we can say that Z was *latent* in $\mathcal{S}$ and the augmentation of n_i has made it explicit. The SPN $\mathcal{S}'$ obtained by augmenting all the non-selective nodes is said to be the *augmentedversion* of $\mathcal{S}$. Therefore, $\mathrm{sc}(\mathcal{S}') = \mathrm{sc}(\mathcal{S}) \cup \mathbf{Z}$, where $\mathbf{Z}$ contains one variable for each sum node that was not selective in $\mathcal{S}$.[4]

Proposition 17: *If $\mathcal{S}'$ is the augmented version of $\mathcal{S}$, then $\mathcal{S}'$ is complete, decomposable, and selective, and represents the same extended probability distribution for $\mathrm{sc}(\mathcal{S})$, i.e., if $\mathbf{x} \in \mathrm{conf}^*(\mathcal{S})$, then $P'(\mathbf{x}) = P(\mathbf{x})$.*

3.6 Induced Trees

The following definition is based on [9], [31].

Definition 18: *Let $\mathcal{S}$ be an SPN and $\mathbf{v} \in \mathrm{conf}(\mathcal{S})$ such that $S(\mathbf{v}) \neq 0$. The* sub-SPN induced by $\mathbf{v}$, *denoted by $\mathcal{S}_\mathbf{v}$, is a non-normalized SPN obtained by (1) removing every node n_i such that $S_i(\mathbf{v}) = 0$ and the corresponding edges, (2) removing every edge $n_i \to n_j$ such that $w_{ij} = 0$, and (3) removing recursively all the nodes without parents, except the root.*

Proposition 19: *If we denote by $S_\mathbf{v}(\mathbf{x})$ the value that $\mathcal{S}_\mathbf{v}$ returns for $\mathbf{x}$, then $S_\mathbf{v}(\mathbf{v}) = S(\mathbf{v})$.*

Proposition 20: *If $\mathcal{S}$ is selective, $\mathbf{v} \in \mathrm{conf}(\mathcal{S})$, and $S(\mathbf{v}) \neq 0$, then $\mathcal{S}_\mathbf{v}$ is a tree in which every sum node has exactly one child. (Following the literature, in this case we will write $\mathcal{T}_\mathbf{v}$ instead of $\mathcal{S}_\mathbf{v}$ to remark that it is a tree.)*

Example 21: Given the SPN in Fig. 2 and $\mathbf{v} = (+a, +b, \neg c)$, $\mathcal{T}_\mathbf{v}$ only contains the edges drawn with thick lines in that figure and the nodes connected by them.

When an SPN is selective, the set of trees obtained for all the configurations in $\mathrm{conf}(\mathcal{S})$ is similar to the set of trees obtained by recursively decomposing the SPN, beginning from the root, as proposed by Zhao *et al.* [33], which leads us to the following results.

Proposition 22: *If $\mathcal{S}$ is selective, $\mathbf{v} \in \mathrm{conf}(\mathcal{S})$, and $S(\mathbf{v}) \neq 0$, then*

$$S(\mathbf{v}) = \prod_{n_i \to n_j \in \mathcal{T}_\mathbf{v}} w_{ij} \prod_{n_k \text{ is terminal in } \mathcal{T}_\mathbf{v}} S_k(\mathbf{v}). \tag{16}$$

Corollary 23: *If all the terminal nodes in $\mathcal{S}$ are indicators, then*

$$S(\mathbf{v}) = \prod_{n_i \to n_j \in \mathcal{T}_\mathbf{v}} w_{ij}. \tag{17}$$

Example 24: For the SPN in Fig. 2, when $\mathbf{v} = (+a, +b, \neg c)$ we have $S(\mathbf{v}) = w_{1,2} \cdot w_{6,8} \cdot w_{14,18} = 0.3 \cdot 0.4 \cdot 0.9 = 0.108$.

4. The original algorithm, proposed by Peharz [9], augments every node in $\mathcal{S}$, even those that were already selective. In contrast, our algorithm only processes the nodes that were not selective in $\mathcal{S}$, so that the augmentation of a selective SPN does not modify it.

4 Inference

4.1 Marginal and Posterior Probabilities

As defined in the previous section, $P(\mathbf{x}) = S(\mathbf{x}) = S_r(\mathbf{x})$. The value $S(\mathbf{x})$ can be computed by an upward pass from the leaves to the root in time proportional to the number of edges in the SPN. If $\mathbf{X}$ and $\mathbf{E}$ are two disjoint subsets of $\mathbf{V}$, then $P(\mathbf{x}\,|\,\mathbf{e}) = S(\mathbf{xe})/S(\mathbf{e})$, where $\mathbf{xe}$ is the composition of $\mathbf{x}$ and $\mathbf{e}$. Therefore, any joint, marginal, or conditional probability can be computed with at most two upward passes. Partial propagation, which only propagates from the nodes in $\mathbf{X} \cup \mathbf{E}$, can be significantly faster [34].

4.2 MPE Inference

The MPE configuration for an SPN is (see Section 2.3)

$$\begin{aligned} MPE\,(\mathbf{e}) &= \arg\max_{\mathbf{x}} P(\mathbf{x}\,|\,\mathbf{e}) = \arg\max_{\mathbf{x}} P(\mathbf{xe})/P(\mathbf{e}) \\ &= \arg\max_{\mathbf{x}} P(\mathbf{xe}) = \arg\max_{\mathbf{x}} S(\mathbf{xe}). \end{aligned} \tag{18}$$

Let us assume that $\mathcal{S}$ is selective. Then $\mathbf{X} \cup \mathbf{E} = \mathrm{sc}(\mathcal{S})$ implies that $\mathbf{xe} \in \mathrm{conf}(\mathcal{S})$ and, because of Proposition 20, the sub-SPN induced by $\mathbf{xe}$ is a tree in which every sum node has only one child. Therefore, the MPE can be found by examining all the trees for the configurations $\mathbf{xe}$ in which $\mathbf{e}$ is fixed and $\mathbf{x}$ varies. It is possible to compare all those trees at once with a single pass in $\mathcal{S}$, by computing $S_i^{\max}(\mathbf{e})$ for each node as follows:

- if n_i is a sum node, then

$$S_i^{\max}(\mathbf{e}) = \max_{j \in ch(i)} w_{ij} \cdot S_j^{\max}(\mathbf{e}); \tag{19}$$

- otherwise $S_i^{\max}(\mathbf{e}) = S_i(\mathbf{e})$ (cf. Eqs. (11) and (13)).

Then the algorithm backtracks from the root to the leaves, selecting for each sum node the child that led to $S^{\max}(\mathbf{e})$ and for each product node all its children. When arriving at a terminal node with scope $\mathbf{Y}$, the algorithm selects the mode of P, $\hat{\mathbf{y}} = \arg\max_{\mathbf{y}'} P(\mathbf{y}')$. In particular, if the terminal node is an indicator $\mathbb{I}_v$, then $\hat{v} = \arg\max_{v'} P(v') = v$. A tie means that there are two or more configurations having the same probability $P(\mathbf{x}, \mathbf{e})$; these ties can be broken arbitrarily. The backtracking phase is equivalent to pruning the SPN in order to obtain a tree in which every sum node has only one child and there is exactly one terminal node for each variable; the composition of the $\hat{\mathbf{y}}$'s selected at the terminal nodes makes up the configuration $\hat{\mathbf{x}} = MPE\,(\mathbf{e})$.

Example 25: Fig. 4 shows the MPE inference for the SPN in Fig. 2 when $\mathbf{e} = +c$. The MPE is obtained by backtracking from the root to the leaves: $\hat{\mathbf{x}} = MPE\,(\mathbf{e}) = (+a, \neg b)$. We can check that $S_1^{\max}(\mathbf{e}) = P(\hat{\mathbf{x}}\mathbf{e}) = P(+a, \neg b, +c) = 0.144$. For any other configuration $\mathbf{x}$ of $\mathbf{X} = \mathbf{V} \setminus \mathbf{E} = \{A, B\}$, we have $S(\mathbf{xe}) < S(\hat{\mathbf{x}}\mathbf{e})$, in accordance with Equation (18). We can also check that the nodes selected by the backtracking phase are those of the tree induced by $\hat{\mathbf{x}}\mathbf{e}$—see Definition 18 and Proposition 20.

Note that, as mentioned in Section 2.3, the MPE cannot be determined by selecting the most probable value for each variable. In this example $P(\neg a\,|\,\mathbf{e}) = 0.57 > P(+a\,|\,\mathbf{e})$ and $P(\neg b\,|\,\mathbf{e}) = 0.68 > P(+b\,|\,\mathbf{e})$, so we would obtain the configuration $(\neg a, \neg b)$, which is not the MPE.

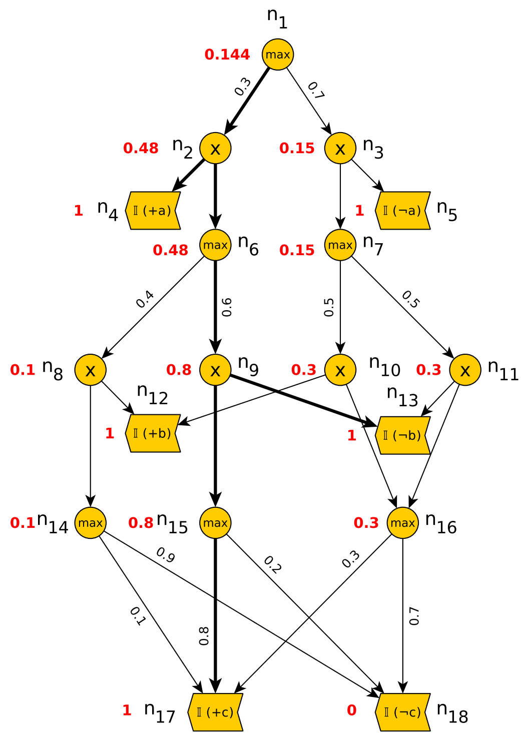

Fig. 4. MPE computation for the SPN in Fig. 2. Sum nodes turn into max nodes. The numbers in red are the values $S_i^{\max}(\mathbf{e})$ when the evidence is $\mathbf{e} = +c$. The most probable explanation, $MPE\,(\mathbf{e}) = (+a, \neg b)$, is found by backtracking from the root to the leaves (thick lines).

This algorithm was proposed for arithmetic circuits by Chan and Darwiche [35], adapted to SPNs by Poon and Domingos [1], and later called Best Tree (BT) in [36]. Peharz [7, Th. 2] proved that when an SPN is selective, BT computes the true MPE. However, when a network is not selective, the sub-SPN induced by a configuration $\mathbf{xe}$ is not necessarily a tree, so the value $S^{\max}(\mathbf{e})$ computed by BT—which only considers the probability that flows along trees with one child for each sum node—may be different from $\max_{\mathbf{x}} S(\mathbf{xe})$ and, consequently, the configuration returned by BT may be different from the true MPE. Therefore, even though the MPE can be found in time proportional to the size of the graph for selective SPNs, MPE is NP-complete for general SPNs (an incorrect proof was included in Peharz's thesis [9, Th. 5.3] and amended in an erratum; the first correct proof was offered in [37]).[5]

4.3 MAX and MAP

Exact MAP inference for SPNs is NP-hard because it includes as a particular case MPE (see Section 2.3), which is NP-complete. Conaty *et al.* [37], [38] gave an inapproximation result for this problem, and showed that an extension

5. When an SPN $\mathcal{S}$ is not selective, it is possible to find an approximation to the MPE by augmenting it and then finding the MPE for $\mathcal{S}'$ given $\mathbf{e}$. The result is a configuration $\mathbf{y}$ of $\mathbf{Y} = (\mathbf{V} \cup \mathbf{Z}) \setminus \mathbf{E} = \mathbf{X} \cup \mathbf{Z}$, which we can then project onto $\mathbf{X}$. However, Park [24] proved that in general this method does not find good approximations, i.e., the posterior probability of the configuration found by this method may differ significantly from that of the true MPE.

of BT for MAP can obtain exponentially better solutions at the expense of quadratic runtime in the size of the network.

Mei *et al*. [36] proposed several algorithms that are very efficient in practice. First, they presented an algorithm for the MAX problem in general SPNs. Second, they proved that every MAP problem for SPNs can be reduced to a MAX problem for a new SPN built in linear time. This way they were able to exactly solve MAP problems for SPNs with up to 1,000 variables and 150,000 edges.

Third, they proposed several approximate MAP solvers that trade accuracy for speed, obtaining excellent results. In particular, they extended the BT method to the MAX problem for non-selective SPNs. This extension, called K-Best Tree (KBT), selects the K trees with the largest output. Then, the corresponding configurations are obtained (by backtracking) and evaluated in the SPN. The one with the largest output is the approximate solution to the MAX problem. Note that, for $K=1$, KBT reduces to BT.

Most of these algorithms have been developed for the case of discrete variables and are difficult to adapt to the continuous case.

5 Parameter Learning

Parameter learning consists in finding the optimal parameters for an SPN given its graph and a dataset. In *generative* learning the most common optimality criterion is to maximize the likelihood of the parameters of the model given a dataset, while in *discriminative* learning the goal is to maximize the conditional likelihood for each value of a distinguished variable C, called the *class*.

5.1 Maximum Likelihood Estimation (MLE)

Let $\mathcal{D}=\{\mathbf{v}^1,\mathbf{v}^2,\ldots,\mathbf{v}^T\}$ be a dataset of T independent and identically distributed (i.i.d.) instances. We denote by $\mathbf{W}$ the set of weights of the SPN and by $\boldsymbol{\Theta}$ the parameters of the probability distributions in the terminal nodes; both of them act as conditioning variables for the probability of the instances in the dataset, $P(\mathcal{D}\,|\,\mathbf{w},\boldsymbol{\theta})$. We define $L_{\mathcal{D}}(\mathbf{w},\boldsymbol{\theta})$ as the logarithm of the likelihood

$$L_{\mathcal{D}}(\mathbf{w},\boldsymbol{\theta}) = \log P(\mathcal{D}\,|\,\mathbf{w},\boldsymbol{\theta}) = \sum_{t=1}^{T} \log S(\mathbf{v}^t\,|\,\mathbf{w},\boldsymbol{\theta}). \tag{20}$$

The goal is to find the configuration of $\mathbf{W}\cup\boldsymbol{\Theta}$ that maximizes $L_{\mathcal{D}}(\mathbf{w},\boldsymbol{\theta})$. Given that there is no restriction linking the parameters of one node (either sum, product, or terminal) with those of others, the optimization can be done independently for each node. For terminal nodes, the estimation of the parameters that maximize the likelihood depends on the type of distribution. In particular, indicator nodes have no parameters, so no estimation is necessary. In this section we will focus on the optimization of the weights, $\mathbf{W}$, so we omit $\boldsymbol{\Theta}$ in the equations. The configuration that maximizes the likelihood is

$$\widehat{\mathbf{w}} = \arg\max_{\mathbf{w}}\ P(\mathcal{D}\,|\,\mathbf{w}) = \arg\max_{\mathbf{w}}\ L_{\mathcal{D}}(\mathbf{w}), \tag{21}$$

subject to $w_{ij}\geq 0$ and $\sum_{j\in ch(i)} w_{ij}=1$.

5.1.1 MLE for Selective SPNs

When the SPN is selective and $S(\mathbf{v})\neq 0$, then the weights of the sum nodes can be estimated in closed form by applying MLE as follows [31].

Proposition 26: *When an SPN is selective,*

$$L_{\mathcal{D}}(\mathbf{w}) = \sum_i \sum_{j\in \mathrm{ch}(i)} n_{ij}\cdot \log w_{ij} + c, \tag{22}$$

where n_{ij} is the number of instances in the dataset for which $n_i \rightarrow n_j \in \mathcal{T}_{\mathbf{v}^t}$ and c is a value that does not depend on $\mathbf{w}$.

The n_{ij}'s can be computed by having a counter for every edge $n_i\rightarrow n_j$ in the SPN. For each instance $\mathbf{v}^t$ in the dataset, we compute $S(\mathbf{v}^t)$ and then backtrack from the root to the leaves: for each product node we select all its children; for each sum node n_i we select the only child for which $S_j(\mathbf{v}^t)>0$, and increase by 1 the counter n_{ij}.

It is then necessary to obtain the configuration $\widehat{\mathbf{w}}$ that maximizes the likelihood—cf. Eq. (21). The only constraint is $\sum_{j\in ch(i)} w_{ij}=1$ for every i, which implies that the parameters for one node can be optimized independently of those for other nodes. The values that maximize the ith term in Equation (22) are

$$\widehat{w}_{ij} = \frac{n_{ij}}{\sum_{j'\in ch(i)} n_{ij'}}. \tag{23}$$

Alternatively, it is possible to use a Laplace-like smoothing parameter α, so that

$$\widehat{w}_{ij} = \frac{n_{ij}+\alpha}{\sum_{j'\in ch(i)}(n_{ij'}+\alpha)}\ , \tag{24}$$

with $0<\alpha\leq 1$. When $S_i(\mathbf{v}^t)=0$ for every t, i.e., when none of the instances in the dataset propagates through the sum node n_i, then $n_{ij}=0$ for every child n_j, and the weights are set uniformly: $\widehat{w}_{ij}=1/|ch(i)|$.

5.1.2 Partial Derivatives of S

In this section we assume that all the parents of a sum node (if any) are product nodes, and vice versa. If there were an edge connecting two nodes of the same type, the child could be absorbed into the parent, without modifying the properties of the SPN.

For every node n_i and every instance $\mathbf{v}^t$ we define

$$S_i^{\partial}(\mathbf{v}^t) = \frac{1}{S(\mathbf{v}^t)}\cdot\frac{\partial S}{\partial S_i}(\mathbf{v}^t). \tag{25}$$

For the root node,

$$S_r^{\partial}(\mathbf{v}^t) = \frac{1}{S(\mathbf{v}^t)}\cdot\frac{\partial S}{\partial S_r}(\mathbf{v}^t) = \frac{1}{S(\mathbf{v}^t)}. \tag{26}$$

For the other nodes,

$$\begin{aligned} S_i^{\partial}(\mathbf{v}^t) &= \frac{1}{S(\mathbf{v}^t)} \cdot \frac{\partial S}{\partial S_i}(\mathbf{v}^t) \\ &= \frac{1}{S(\mathbf{v}^t)} \cdot \sum_{k \in pa(i)} \frac{\partial S}{\partial S_k}(\mathbf{v}^t) \cdot \frac{\partial S_k}{\partial S_i}(\mathbf{v}^t) \\ &= \sum_{k \in pa(i)} S_k^{\partial}(\mathbf{v}^t) \cdot \frac{\partial S_k}{\partial S_i}(\mathbf{v}^t), \end{aligned}$$

where $pa(i)$ is the set of indices for the parents of n_i. If n_i is a sum node,

$$S_i^{\partial}(\mathbf{v}^t) = \sum_{k \in pa(i)} S_k^{\partial}(\mathbf{v}^t) \cdot \prod_{i' \in ch(k) \setminus \{i\}} S_{i'}(\mathbf{v}^t); \tag{27}$$

if it is a product node,

$$S_i^{\partial}(\mathbf{v}^t) = \sum_{k \in pa(i)} w_{ki} \cdot S_k^{\partial}(\mathbf{v}^t). \tag{28}$$

These equations mean that, for every node n_i, $S_i^{\partial}(\mathbf{v}^t)$ can be computed once we have the $S_k^{\partial}(\mathbf{v}^t)$'s of its parents and the $S_{i'}(\mathbf{v}^t)$'s of its siblings. Therefore, after computing $S_i(\mathbf{v}^t)$ for every node with an upward pass, $S_i^{\partial}(\mathbf{v}^t)$ can be computed with a downward pass, both in linear time. This algorithm is similar to backpropagation for neural networks and can be implemented using software packages that support automatic differentiation.

5.1.3 Gradient Descent (GD)

Standard GD. This well-known optimization method was proposed for SPNs for both generative and discriminative models in [1] and [5], respectively.[6] The algorithm is initialized by assigning an arbitrary value to each parameter, $\widehat{w}_{ij}^{(0)}$, and in every iteration, s, this value is updated, in order to increase the likelihood of the model

$$\widehat{w}_{ij}^{(s+1)} = \widehat{w}_{ij}^{(s)} + \eta \frac{\partial L_{\mathcal{D}}(\mathbf{w})}{\partial w_{ij}}, \tag{29}$$

where $\eta > 0$ is the learning rate (a hyperparameter). Given that the parameters computed with this expression are usually non-normalized and occasionally negative, it is necessary to project them back to the feasible set after each update, as explained in [33]—see also [9, Sec. 6.2]. This method is called *projected GD* [39].

Proposition 27: *In an SPN,*

$$\frac{\partial L_{\mathcal{D}}(\mathbf{w})}{\partial w_{ij}} = \sum_{t=1}^{T} S_i^{\partial}(\mathbf{v}^t) \cdot S_j(\mathbf{v}^t). \tag{30}$$

As the $S_i(\mathbf{v}^t)$'s and $S_i^{\partial}(\mathbf{v}^t)$'s for the whole network can be computed in linear time for each instance $\mathbf{v}^t$, the time required for each iteration of GD is proportional to the size of the SPN and the number of instances in the dataset.

6. The method is commonly called "gradient descent" when its goal is to minimize a quantity—for example, the classification error in neural networks. In our case it would be more appropriate to call it "gradient *ascent*" because the goal is to maximize the likelihood. However, in this paper we follow the standard terminology for SPNs.

Stochastic GD and mini-batch GD. In the stochastic version of GD, each iteration s randomly chooses one instance of the dataset (as if $\mathcal{D}$ in Eq. (30) consisted only of that instance, i.e., $T = 1$), until the algorithm converges.

Another possibility is to use in each iteration a mini-batch, i.e., a subset L randomly drawn instances, where $L < T$ (usually $L \ll T$). This version is the most popular when applying GD.

Hard GD. The application of GD to deep networks, either neural networks or SPNs, suffers from the vanishing gradients problem: the deeper the layer, the lower the contribution of its weights to the model output, so the influence of the parameters in the deepest layers may be imperceptible.

As mentioned above, the goal of discriminative learning is to maximize the conditional likelihood, $P(\mathbf{y} \,|\, \mathbf{x})$, where $\mathbf{X}$ are the observed variables (input) and $\mathbf{Y}$ is the set of labels (output). In an SPN, the conditional likelihood of each instance in the dataset is $\log P(\mathbf{y} \,|\, \mathbf{x}) = \log S(\mathbf{xy}) - \log S(\mathbf{x})$. The *hard* version of GD for learning discriminative SPNs proposed in [5], instead of maximizing the (overall) conditional likelihood, which is $\log P(\mathbf{y} \,|\, \mathbf{x}) = \log S(\mathbf{xy}) - \log S(\mathbf{x})$, aims to maximize (the overall value of) $\log S^{\max}(\mathbf{xy}) - \log S^{\max}(\mathbf{x})$. Their algorithm computes $S^{\max}$ as the product of the weights in the tree obtained by the BT algorithm when backtracking (see Eq. (17)), and then takes its partial derivative.

5.1.4 Expectation-Maximization (EM)

Standard EM. We have seen how to learn the parameters of a selective SPN from a complete dataset. However, many real-world problems have missing values. We denote by $\mathbf{H}^t$ the variables missing (hidden) in the tth instance of the dataset. Additionally, when learning the parameters of $\mathcal{S}'$, an augmented SPN, the dataset is always incomplete, even if it contains all the values for the the model variables in $\mathcal{S}$, because it does not contain the latent variables $\mathbf{Z}$, added when augmenting the SPN, so $\mathbf{Z} \subseteq \mathbf{H}^t$ for every t.

In this situation we can apply the expectation-maximization (EM) algorithm, designed to estimate the parameters of probabilistic models from incomplete datasets. The problem is as follows. If we had a complete dataset, we would be able to estimate its parameters as explained in the previous section. Alternatively, if we knew the parameters, we would be able to generate a complete dataset by sampling from the probability distribution.

The EM algorithm proceeds by iteratively applying two steps. The E-step (expectation) computes the probability $P(\mathbf{h}^t \,|\, \mathbf{v}^t)$ for each configuration of the variables missing in $\mathbf{v}^t$ in order to impute the missing values. More precisely, instead of assigning a single value to each missing cell, we create a virtual dataset in which all the configurations of $\mathbf{H}^t$ are present, each with probability $P(\mathbf{h}^t \,|\, \mathbf{v}^t)$. The M-step (maximization) uses this virtual complete dataset to adjust the parameters of the model by MLE, as in Section 5.1.1. The two steps are repeated until the parameters (the weights) converge.

The problem is that initially we have neither a complete dataset nor parameters for sampling the values of the missing variables. The algorithm can be initialized by assigning arbitrary values to the parameters or by assigning arbitrary values to the variables in $\mathbf{Z}$. Unfortunately, like in GD, a bad choice of the initial values may cause the algorithm to converge to a local maximum of the likelihood, which may be quite different from the global maximum.

The n_{ij}'s required by the M-step are obtained by counting the number of instances in the dataset for which the edge (i, j) belongs to the tree induced by $\mathbf{v}^t\mathbf{h}^t$. These are the n_{ij}'s introduced in Equation (22), which in the case of the virtual dataset are

$$n_{ij} = \sum_{t=1}^{T} \sum_{\mathbf{h}^t \,|\, (i,j)\in\mathcal{T}'_{\mathbf{v}^t\mathbf{h}^t}} P'(\mathbf{h}^t \,|\, \mathbf{v}^t), \tag{31}$$

and can be efficiently computed by applying the following result:

Proposition 28: *Given a dataset $\mathcal{D}$ with T instances and an SPN, the n_{ij}'s in Equation (31) are*

$$n_{ij} = \sum_{t=1}^{T} w_{ij} \cdot S_i^{\partial}(\mathbf{v}^t) \cdot S_j(\mathbf{v}^t). \tag{32}$$

Please note that this proposition is valid even for non-selective SPNs (see the proof in Appendix C, available in the online supplemental material). Once we have the n_{ij}'s, the weights can be updated using Equations (23) or (24). The time required by each iteration of EM is, like in GD, proportional to the size of the network and the number of instances in the dataset.

Hard EM. The EM algorithm needs the value of S_i^{∂}, which is proportional to $\partial S/\partial w_{ij}$ and may thus be very small when the edge (n_i, n_j) is in a deep position, i.e., far from the root. Therefore this algorithm may suffer from the vanishing gradients problem in the same way as GD. To avoid it, Poon and Domingos [1] proposed a *hard* version of EM for SPNs that selects for each hidden variable $H \in \mathbf{H}^t$ the most probable state. Thus, in the E-step of each iteration, every instance of the dataset contributes to the update of just one weight per sum node.

Hsu *et al.* [29] proposed a variant of hard EM for SPNs with Gaussian leaves. When processing an instance, it first computes the likelihood of every node and then updates the counter associated to each node, beginning from the root, so that for each sum node only the counter of the child with the highest likelihood is increased. These counters are used to update the weights, as well as the mean and the covariance of each leaf node.

5.1.5 Comparison of MLE Algorithms

The application of EM to SPNs has been justified with different mathematical arguments. Peharz [7], [9] obtained the first derivation of the EM for SPNs by exploiting the interpretation of sum nodes in the augmented network as the sum of conditional probability distributions (cf. Eqs. (38) and (39) in Appendix B, available in the online supplemental material). In turn, Zhao *et al.* [33], using a unified framework based on signomial programming, designed two algorithms for learning the parameters of SPNs: sequential monomial approximations (SMA) and the concave-convex procedure (CCCP). GD is a special case of SMA, while CCCP coincides with EM in the case of SPNs, despite being different algorithms in general. Their experiments proved that EM/CCCP converges much faster than the other algorithms, including GD. In turn, Desana and Schnörr [28] derived the EM algorithm for SPNs whose leaf nodes may represent complex probability distributions.

In discriminative learning, neither EM nor CCCP have a closed-form expression for updating the weights [5]. Rashwan *et al.* [40] addressed this problem with the extended Baum-Welch (EBW) algorithm, which updates the parameters of the network using a transformation that increases the value of the likelihood function monotonically. In the generative case, this transformation coincides with the update formula of EM/CCCP (the M-step), while in the discriminative case it provides a method to maximize the (conditional) likelihood function with a closed-form formula. They also adapted this method to SPNs with Gaussian leaves.

Both the algorithm of Desana and Schnörr and EBW outperformed GD and EM in a wide variety of datasets.

Finally, Trapp *et al.* [41] have shown that increasing the depth of tree-structured SPNs can accelerate GD, since it is equivalent to applying adaptive and time-varying learning rates and adding momentum terms, as it was first observed for deep neural networks [42].

5.2 Semi-Supervised Learning

Trapp *et al.* [43] introduced a safe semi-supervised learning algorithm for SPNs. By "safe" they mean that the model performance can be increased but never degraded by adding unlabeled data. They extended EM to generative semi-supervised learning and defined a discriminative semi-supervised learning approach. They also introduced the maximum contrastive pessimistic algorithm (MCP-SPN), based on [44], for learning safe semi-supervised SPNs. Their results were competitive with those of purely supervised algorithms trained on fully labeled datasets.

5.3 Approximate Bayesian Learning

There are alternative methods for learning the parameters of an SPN based on approximate Bayesian techniques, such as Bayesian moment matching [45] and collapsed variational inference [46], which are not as exposed to overfitting as GD or EM. Both Bayesian methods start with a product of Dirichlet distributions as a prior; the posterior distribution $P(w_{ij} \,|\, \mathcal{D})$ is a mixture of products of Dirichlets, which is computationally intractable. In both works the solution applied was to approximate that distribution with a single product of Dirichlet distributions. Rashwan *et al.* [45] applied online Bayesian moment matching (oBMM), which approximates the posterior distributions of the weights by computing a subset of their moments and finding another distribution from a tractable family that matches those moments. In this case, it sufficed to match the first and second order moments of the distribution. The experiments showed that this approach outperforms SGD and online EM. This method has also been adapted to SPNs with Gaussian leaves by Jaini *et al.* [47]. In the same vein, Zhao and Gordon [48] presented an optimal linear time algorithm for computing the moments in SPNs with general acyclic directed graph structures, based on the mixture-of-trees interpretation of SPNs. This provides an effective method to apply Bayesian moment matching to a broad family of SPNs.

As mentioned above, Zhao *et al*. [46] addressed the problem by applying collapsed variational Bayesian inference (CVB-SPN). This approach treats the dataset as partial evidence, whose missing values correspond to the latent variables of the SPN. They assumed that the missing values of each instance are not independent of those missing in other instances, and marginalized these variables out of the joint posterior distribution (the "collapse" step). Then, they approximated this distribution with the product of the Dirichlets that maximizes certain evidence lower bound (ELBO) of the log-likelihood function of the dataset (the "variational inference" step). The lower bound obtained with this method is tighter that the one obtained with standard variational inference. The experiments showed that the online version of CVB-SPN—i.e., the version that receives a stream of data in real time—outperforms oBMM in many datasets.

5.4 Deep Learning Approach

Peharz *et al*. [21] considered a special class of SPNs, which they called random SPNs. Varying a parameter, λ, they could control the trade-off between generative behavior (measured in log-likelihood) and discriminative behavior (measured in accuracy or cross-entropy), and trained them for generative and discriminative problems with automatic differentiation, stochastic GD, and dropout, using GPU parallelization. The resulting model was called RAT-SPN. Its classification accuracy, measured on several datasets, was comparable to that of deep neural networks, with the advantages of a probabilistic generative model, such as robustness to missing features.

5.5 Sample Complexity of Learning SPNs

Aden-Ali and Ashtiani [49] have recently shown that the number of samples necessary to learn tree structured SPNs with discrete or Gaussian leaves grows at most linearly (up to logarithmic factors) with the number of parameters of the SPN.

6 Structural Learning

Structural learning consists in finding the graph of an SPN that closely fits the available data. Most of the algorithms for this task require some computation of probabilities during the process.

6.1 First Structure Learners

BuildSPN, by Dennis and Ventura [50] was the first algorithm of this kind. It looks for subsets of highly correlated variables and introduces latent variables to account for those dependencies. These variables generate sum nodes and the process is repeated recursively looking for the new latent variables.

BuildSPN and the hand-coded structure of Poon and Domingos [1], both designed for image processing, assumed neighborhood dependence. In order to overcome that limitation, Peharz *et al*. [51] proposed an algorithm that

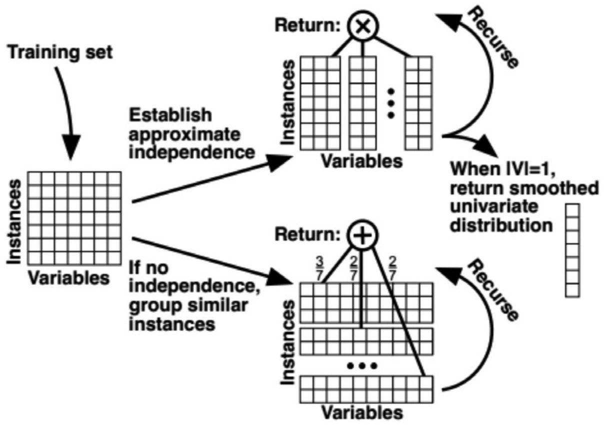
Training set
Instances
Variables
Establish approximate independence
Return: ⊗
Instances
Variables
Recurse
When |V|=1, return smoothed univariate distribution
If no independence, group similar instances
Return: ⊕
3/7
2/7
2/7
Instances
Variables
Recurse

Fig. 5. The LearnSPN algorithm recursively creates a product node when there are subsets of (approximately) independent variables and a sum node otherwise, grouping similar instances. (Reproduced from [6] with the authors' permission.)

subsequently combines SPNs of few variables into larger ones applying a statistical dependence test.

BuildSPN was also critiqued by Gens and Domingos [6] because (1) the clustering process may separate highly dependent variables, (2) the size of the SPN and the time required can grow exponentially with the number of variables, and (3) it requires an additional step to learn the weights.

Algorithm 1. LearnSPN$(\mathbf{V}, T, m, \alpha)$

```
Input: $\mathbf{V}$: a set of variables;
  $T$: a data matrix of $|T|$ instances;
  $m$: minimum number of instances to allow a split of variables;
  $\alpha$: Laplace smoothing parameter
Output: an SPN $\mathcal{S}$ with $\mathrm{sc}(\mathcal{S}) = \mathbf{V}$
if $|\mathbf{V}| = 1$ then
  $\mathcal{S} \leftarrow$ univariateDistribution$(\mathbf{V}, T, \alpha)$
else if $|T| < m$ then
  $\mathcal{S} \leftarrow$ naïveFactorization$(\mathbf{V}, T, \alpha)$
else
  $\{\mathbf{V}_j\}_{j=1}^{C} \leftarrow$ splitVariables$(\mathbf{V}, T, \alpha)$
  if $C > 1$ then
    $\mathcal{S} \leftarrow \prod_{j=1}^{C}$ LearnSPN$(\mathbf{V}_j, T, \alpha, m)$
  else
    $\{T_i\}_{i=1}^{R} \leftarrow$ clusterInstances$(\mathbf{V}, T)$
    $\mathcal{S} \leftarrow \sum_{i=1}^{R} \frac{|T_i|}{|T|}$ LearnSPN$(T_i, \mathbf{V}, \alpha, m)$
return $\mathcal{S}$
```

6.2 LearnSPN

It is common in machine learning to see a dataset as a *data matrix* whose columns are attributes or variables and whose rows are observations or instances. The LearnSPN algorithm [6] recursively splits the variables into independent subsets (thus "chopping" the data matrix, as shown in Fig. 5) and then clusters the instances (thus "slicing" the matrix). Every "chopping" creates a product node and every "slicing" a sum node, as indicated in Algorithm 1. There are two base cases:

1) When the piece of the data matrix produced by "chopping" contains a single column (i.e., one

variable), the algorithm creates a terminal node with a univariate distribution using MLE.

2) When the piece of the data matrix produced by "slicing" contains several columns with relatively few rows, the algorithm applies a naïve Bayes factorization over those variables. This is like "chopping" that piece into individual columns, which will be processed as in the base case 1.

LearnSPN can be seen as a framework algorithm in the sense that it does not specify the procedures for splitting independent subsets of variables (*splitVariables* in Algorithm 1) and clustering similar instances (*clusterInstances* in that algorithm). Originally Gens and Domingos [6] chose the G-Test for splitting and hard incremental EM for clustering.

One possibility for splitting the variables ("chopping") is to create a graph with an edge between each pair of variables found to be dependent according to the G-test. Each connected component determines a subset of variables, $\mathbf{V}_j$.

Clustering similar instances ("slicing") is achieved by the hard EM algorithm assuming a naïve Bayes mixture model, where the variables are independent given the cluster $\mathbf{C}_i$: $P(\mathbf{v}) = \sum_i P(\mathbf{c}_i) \prod_j P(v_j \mid \mathbf{c}_i)$. This particular model produces a clustering that can be chopped in the next recursion. This version of LearnSPN forces a clustering in the first step, without attempting a split.

6.3 ID-SPN

Rooshenas and Lowd [25] observed that PGM learners usually analyze direct interactions (dependencies) between variables while previous SPN learners analyze indirect interactions (dependencies through a latent variable). The indirect-direct SPN (ID-SPN) structure learner combines both methods. Their initial idea is that any tractable multivariate distribution that can be represented as an arithmetic circuit or an SPN can be the leaf of an SPN without losing tractability. With this idea they learned arithmetic circuit Markov networks (ACMN) [52], which are roughly Markov networks learned as arithmetic circuits. ID-SPN begins with a singular ACMN node and tries to replace it with a mixture (yielding a sum node) or a product (yielding a product node), similar to the cluster and split operations in LearnSPN. If a replacement increases the likelihood, it is saved and the algorithm recurs on the new ACMN leaves, until the likelihood does not increase. This top-down process represents the learning of indirect interactions, while the creation of ACMN leaves represents the learning of direct interactions. This algorithm outperformed all previous algorithms and is currently the state of the art. However, ID-SPN is slower and more complex than LearnSPN, and has many more hyperparameters to tune, which requires a random search in the space of hyperparameters instead of a grid search.

6.4 Other Algorithms

Peharz *et al*. [31] proposed a structure learner that searches within the space of selective SPNs and showed that it is competitive with LearnSPN.

Adel *et al*. [53] designed SVD-SPN, an algorithm for learning both generative and discriminative SPNs. It operates recursively, like LearnSPN, but instead of searching for independences, it searches for highly correlated variables. More precisely, it decomposes the data matrix by extracting approximate rank-1 submatrices, which allows the algorithm to cluster and split at the same time, as shown in Fig. 2 in their paper. In their experiments, SVD-SPN obtained results similar to those of LearnSPN and ID-SPN for binary datasets, but outperformed them in multiple-category datasets, such as Caltech-101 and MNIST, and is 5 times faster.

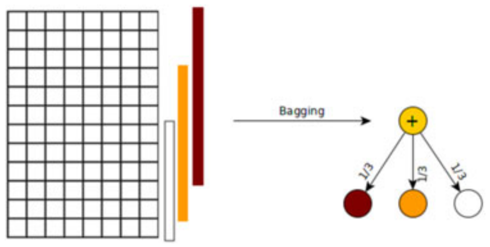

Fig. 6. Learning SPNs with bagging.

6.5 Improvements to LearnSPN

Even though LearnSPN is not the best performing algorithm, it is still widely used owing to its simplicity and modularity [34] and has led to several variants.

6.5.1 *Algorithm of Vergari* et al. *[30]*

It consists of three modifications to LearnSPN:

— *Binary splits*. Every split cuts the data matrix into only two pieces. This avoids creating too complex structures at early stages (which often occurs when learning from noisy data) and favors deep structures over shallow ones. This is not a limitation in the number of children of product nodes because consecutive splits can be applied if necessary.

— *Chow-Liu trees (CLTs) in the leaves*. The naïve Bayes factorization used as the base case of LearnSPN (see Algorithm 1) can be replaced by the creation of Chow-Liu trees [54], which are equivalent to tree-shaped Bayesian networks or Markov networks. Every tree is built by linking the variables with higher mutual information until there is a path between every pair of variables. CLTs are more expressive than the naïve Bayes factorization (which is a particular case of CLT) without increasing the cost of inference. LearnSPN stops earlier when using CLTs as leaves because each tree can accommodate more instances, thus yielding simpler SPN structures (with fewer edges) with lower risk of overfitting.

— *Bagging*. This technique, originally used to build random forests [55], consists in taking several random subsets from a dataset, each consisting of several instances, and building a model for each subset. In SPN learning, it extracts—with replacement—n subsets and produces n models, joined by a sum node with uniform weights, as shown in Fig. 6. Since the network size would grow exponentially if bagging were applied before every clustering, it is applied only before the first LearnSPN operation—which is a clustering—in order to achieve the widest effect on the resulting structure.

The experiments showed that (1) binary splits yield deeper and simpler SPNs and generally reduce the number of edges and parameters, (2) using Chow-Liu trees attains the same effect and generally increases the likelihood, and (3) bagging also increases the likelihood, especially in datasets with a low number of instances. With these modifications, LearnSPN achieved the same performance as ID-SPN.

6.5.2 Beyond Tree SPNs

One of the main disadvantages of both LearnSPN and ID-SPN is that they always produce trees (except when the leaves are Markov networks). In order to generate more compact SPNs, Dennis and Ventura [56] designed SearchSPN, an algorithm that produces SPNs in which nodes may have several parents. It selects the product node that contributes less to the likelihood and greedily searches for candidate structures using modified versions of the clustering methods of LearnSPN. The resulting likelihood is significantly better than that of LearnSPN for the majority of datasets and comparable with that of ID-SPN, but on average the execution is 7 times faster and the number of nodes 10 times smaller.

In the same vein, Rahman and Gogate [57] created a post-processing algorithm that, after applying LearnSPN with CLTs in the leaves, merges similar sub-SPNs. Similarity is measured with a Manhattan distance; if two sub-SPNs are closer than a certain threshold, the pieces of the data matrix from which they come are combined and the algorithm chooses the sub-SPN with the higher likelihood for the combined data. This modification of LearnSPN increases the likelihood and reduces the number of parameters of the SPN, but it also increases dramatically the learning time for some datasets. In combination with bagging, it outperformed other algorithms—including ID-SPN [25]—for high-dimensional datasets.

6.5.3 Further Improvements to LearnSPN

As mentioned in Section 5.1.5, Zhao *et al*. [33] showed that learning the parameters with the CCCP algorithm improves the performance of LearnSPN.

Di Mauro *et al*. [58] proposed approximate splitting methods to accelerate LearnSPN, thus trading speed for quality (likelihood).

Butz *et al*. [34] studied the different combinations of algorithms for LearnSPN. They compared mutual information and the G-test for splitting, and k-means and Gaussian mixture models for clustering. The best results were obtained when using the G-test and either k-means or Gaussian mixture models, both for the standard LearnSPN and for the version that generates CLTs in the leaves.

Liu *et al*. [59] proposed a clustering method that decides the number of instance clusters adaptively, i.e., depending on each piece of data matrix evaluated. Their goal was to generate more expressive SPNs, in particular deeper ones with controlled widths. When compared previous algorithms (namely, standard LearnSPN [6], LearnSPN with binary splits [30], and LearnSPN with approximate splitting [58]), their method achieved higher likelihood in 20 binary datasets and generated deeper networks (i.e., more expressive SPNs) while maintaining a reasonable size.

6.5.4 LearnSPN With Piecewise Polynomial Distributions

Most learning algorithms assume that each terminal node represents either a discrete probability distribution or a univariate distribution belonging to a known family (Poisson, Gaussian, etc.), so that only the parameters need to be optimized. However, there are at least two variants of LearnSPN that, in addition to having indicators for finite-state variables, use a piecewise polynomial distribution for each leaf node representing a numeric variable, instead of requiring the user to specify a parametric family [27], [60].

In LearnWMISPN [60], which combines LearnSPN with weighted model integration (WMI), the order of each polynomial is determined using the Bayesian information criterion (BIC) [61]. A preprocessing step transforms finite-state, categorical, and continuous features into a binary representation before applying LearnSPN. The corresponding inference algorithm can answer complex conditional queries involving both intervals for continuous variables and values for discrete variables.

In mixed SPN (MSPNs) [27] the operations of decomposition (splitting) and conditioning (clustering) are based on the Hirschfeld-Gebelein-Rényi maximum correlation coefficient. These models extend SPNs to hybrid domains, and are competitive with parameterized distributions and mixed graphical models.

6.6 Online Structural Learning

The algorithms presented so far need the complete dataset to produce a structure. However, sometimes the dataset is so big that the computer does not have enough memory to store it at once. In other situations, e.g., in recommender systems, the data arrive constantly. In these cases the learning algorithm must be able to update the structure instead of learning it from scratch every time new data arrives.

In this context, Lee *et al*. [62] designed a version of LearnSPN where clustering (slicing) is replaced by online clustering, so that new sum children can be added when new data arrive, while product nodes are unmodified.

Later Dennis and Ventura [63] extended their SearchSPN algorithm [56] to the online setting. This online version is as fast as the offline version that works only on the current batch, and the quality of the resulting SPN is the same.

Hsu *et al*. [29] created oSLRAU, an online structure learner for Gaussian leaves (oSLRAU) that begins with a completely uncorrelated SPN structure and updates it when the arriving data reveals a new correlation. The update consists in replacing a leaf with a multivariate Gaussian leaf or a mixture over its scope.

Jaini *et al*. [64] proposed an algorithm, Prometheus, whose first concern is to avoid the parameter that decides when two subsets of variables are independent in order to perform a LearnSPN split. So instead of creating a product node, it creates a mixture of them, representing different subset partitions. The partitions created may share subsets, which overcomes the restriction to trees. However, the complexity of the algorithm grows with the square of the number of variables. In order to extend it to high-dimensional datasets, the authors created a version that samples in each step from the set of variables instead of using all of them. This algorithm can treat discrete, continuous, and mixed datasets. Their experiments showed that this algorithm surpasses both LearnSPN and ID-SPN in the three types of datasets. It is also robust in low data regimes, achieving the same performance as oSLRAU with only 30-40 percent of the data. These authors have recently proposed a more efficient version of Prometheus, which runs in sub-quadratic time w.r.t. the number of features [22].

6.7 Learning With Dynamic Data

Data are said to be *dynamic* when all the variables (or at least some of them) have different values in different time points—for example, *Income-at-year-1*, *Income-at-year-2*, etc. The set of variables for a specific time point is usually called a *slice*. The slice structure, called *template*, is replicated and chained to accommodate as many time points as necessary. The length of the chain is called the *horizon*.

For this problem Melibari *et al*. [65] proposed dynamic SPNs (DSPNs), which extend SPNs in the same way that dynamic Bayesian networks (DBNs) [66] extend BNs. A local-search structure learner generates an initial template SPN and searches for neighboring structures, trying to maximize the likelihood. Every neighbor, which stems from replacing a product node, represents a specific choice of factorization of the variables in its scope. The algorithm searches over several factorizations and updates the structure if a better one is found. In their experiments this method outperformed non-dynamic algorithms, such as LearnSPN, and other models, such as dynamic Bayesian networks and recurrent neural networks.

Later, Kalra *et al*. [67] extended oSLRAU to the dynamic setting by unrolling the SPN to match the length of the chain to the horizon, with shared weights and a shared covariance matrix, to decide when a new correlation requires a change in the template. This algorithm surpassed DSPNs [65] and hidden Markov models in 5 sequential datasets, and recurrent neural networks in 4 of those datasets.

6.8 Relational Data Learning

Nath and Domingos [68] introduced relational SPNs (RSPNs), which generalize SPNs by modeling a set of instances jointly, allowing them to influence each other's probability distributions, and modeling probabilities of relations between objects. Their LearnRSPN algorithm outperformed Markov logic networks in both running time and predictive accuracy when tested on three datasets.

6.9 Bayesian Structure Learning

Lee *et al*. [69] designed a Bayesian non-parametric extension of SPNs. Trapp *et al*. [70] criticized this work for neglecting induced trees in their posterior construction; they corrected it by introducing *infinite sum-product trees* and showed that it yields higher likelihood than infinite Gaussian mixture models.

A common problem of structural learning algorithms is the lack of a principled criterion for deciding what a "good" structure is. For this reason, Trapp *et al*. [71] proposed an alternative Bayesian approach that decomposes the problem into two phases: first finding a graph and then learning a function that assigns a scope to each node. This function and the parameters of the model are learned jointly using Gibbs sampling. The Bayesian nature of this approach reduces the risk of overfitting, avoids the need for a separate validation set to adjust the hyperparameters of the algorithm, and enables robust learning of SPN structures under missing data.

7 Applications

SPNs have been used for a wide variety of applications, from toy problems to real-world challenges.

7.1 Image Processing

7.1.1 Image Reconstruction and Classification

Poon and Domingos, in their seminal paper about SPNs [1], applied them to image reconstruction, using a hand-designed structure that took into account the local structure of the image data. They tested their method on the datasets Caltech-101 and Olivetti. As mentioned in Section 1.1, Gens and Domingos [5] used a different hand-built structure for image classification on the datasets CIFAR-10 and STL-10, obtaining excellent results for that time. Hartmann [72] and van de Wolfshaar and Pronobis [73] used convolutional SPNs for classification, and Butz *et al*. [74] for image completion. RAT-SPNs [21] and Prometheus [22] have achieved the highest accuracy scores so far (98.29 and 98.37 percent respectively) for MNIST image classification with SPNs.

7.1.2 Image Segmentation

Image segmentation consists in labeling every pixel with the object it belongs to. Yuan *et al*. [75] developed an algorithm that scales down every image recursively to different sizes and generates object tags and unary potentials for every scale. Then, it builds a multi-stacked SPN where every stack has a bottom and a top SPN. The bottom SPN works on a pixel and its vicinity, going from the pixel to bigger patches. Product nodes model correlations between patches while sum nodes combine them into a feature of a bigger patch. When the patch is as big as the pixel in the next scaled image, the results are introduced in the top SPN alongside the unary potentials and the tags of that scale. This process is stacked until the "patch" treated is the whole image. Multi-stacked SPNs have been especially effective for handling occlusions in scenes.

Rathke *et al*. [76] applied SPNs to segment OCT scans of retinal tissue. They first built a segmentation model for the health model and for every pathology and then added to those models typical shape variations of the retina tissue for some pathology-specific regions. The resulting SPN extracts candidate regions (either healthy or unhealthy) and selects the combination that maximizes the likelihood. After a smoothing step, they obtain a complete segmentation of the retina tissue, as well as the diagnosis and the affected regions. This method achieved state-of-the-art performance without needing images labeled by pathologies.

7.1.3 Activity Recognition

Wang and Wang [77] addressed activity recognition on still images. They used unsupervised learning and a convolutional neural network to isolate parts of the images, such as a hand or a glass, and designed a spatial SPN including the spatial indicator nodes "above", "below", "left", and "right" for the product nodes to encode spatial relations between pairs of these parts. They first partitioned the image to consider only local part configurations. Its SPN structure has two components: the top layers represent a partitioning of the image into sub-images, where product nodes act as partitions and sum nodes as combinations of partitions, while the bottom layers represent the parts included in each sub-image and their relative position using the spatial indicator nodes. This way the SPN first learns spatial relations of isolated parts in sub-images and then learns correlations between sub-images. Spatial SPNs outperformed other activity recognition algorithms and were able to discover discriminant pairs of parts for each class.

Amer and Todorovic [78] worked on activity localization and recognition in videos. In their work, a *visual word* is a meaningful piece of an image, previously extracted with a neural network. Visual words lie in a grid with three dimensions: height, width, and time. Every grid position has a histogram of associated visual words, called *bag of words*. To construct an SPN, each bag of words is treated as a variable with two states: foreground and background. Product nodes represent a combination of sub-activities into a more complex activity (for example, “join hands + separate hands = clap”) and sum nodes represent variations of the same activity. An SPN is trained for every activity in a supervised context, in which the foreground and the background values are known, and in a weakly supervised context, in which only the activity is known. The structure is a near-completely connected graph, pruned after parameter learning, which proceeds iteratively: it learns—with GD—the weights of the SPN from the parameters of the bag of words and then learns—with variational methods—the parameters of the bag of words from the weights of the SPN. The accuracy of this weakly supervised setting was only 1.6 to 3 percent worse than that of the supervised setting. This approach in general achieved better performance than state-of-the-art algorithms on several action-recognition datasets.

7.1.4 Object Detection

Stelzner *et al*. [23] demonstrated empirically that the attend-infer-repeat framework used for object detection and location is much more efficient when the variational autoencoders that model individual objects are replaced by SPNs: they achieved an improvement in speed of an order of magnitude, with slightly higher accuracy, as well as robustness against noise.

7.2 Robotics

Sguerra and Cozman [79] used SPNs for *aerial robots navigation*. Micro aerial vehicles need a set of sensors that must comply with two criteria: light weight and real-time response. Optical recognition with cameras satisfies the former while fast inference with SPNs ensures the latter, as they were able to classify in real time what the camera sees into pilot commands, such as “turn right”, obtaining 75 percent of accuracy with just 66 images.

Pronobis *et al*. [80] designed a probabilistic representation of spatial knowledge called “deep spatial affordance hierarchy” (DASH), which encodes several levels of abstractions using a deep model of spatial concepts. It models knowledge gaps and affordances by a deep generative spatial model (DGSM) which uses SPNs for inference across different levels of abstractions. SPNs fit naturally with DGSM because latent variables of the former are internal descriptors in the latter. The authors tested it in a robot equipped with a laser-range sensor.

Zheng *et al*. [81] designed graph-structured SPNs (GraphSPNs) for structured prediction. Their algorithm learns template SPNs and makes a mixture over them (a template distribution), which can be applied to graphs of varying size re-using the same templates. The authors applied them to model large-scale global semantic maps of office environments with a exploring robot, obtaining better results than with the classical approach based on undirected graphical models (Markov networks).

These authors later joined both models into an end-to-end deep model for semantic mapping in large-scale environments with multiple levels of abstraction, called TopoNets [82], which can perform real-time inference, with novelty detection, for unknown spatial information.

7.3 NLP and Sequence Data Analysis

Peharz *et al*. [83] applied SPNs to *modeling speech* by retrieving the lost frequencies of telephone communications (artificial bandwidth extension). In this problem tractable and real-time inference is essential. They used a hidden Markov model (HMM) to represent the temporal evolution of the log-spectrum, clustered the data using the Linde–Buzo–Gray algorithm and trained an SPN for each cluster. The SPNs model each cluster and can be used to retrieve the lost frequencies by MPE inference. This model has achieved better results than state-of-the-art algorithms, both objectively, with a measure of log-spectral distortion, and subjectively, through listening tests.

In language modeling, Cheng *et al*. [84] used a discriminative SPN [5] whose leaves represent vectors with information about previous words. This SPN was able to compute the probability of the next word more accurately than classic methods for language modeling, such as feedforward neural networks and recurrent neural networks.

Melibari *et al*. [65] used dynamic SPNs—see Section 6.7—to analyze different sequence datasets. Unlike dynamic Bayesian networks, for which inference is generally exponential in the number of variables per time slice, inference in DSPNs has linear complexity. In a comparative study with five other methods, including HMMs and neural networks with long short-term memory (LSTM), DSPNs were superior in 4 of the 5 datasets examined.

Ratajczak *et al*. [85] replaced the local factors of higher-order linear-chain conditional random fields (HO-LC-CRFs) and maximum entropy Markov models (MEMMs) with SPNs. These outperformed other state-of-the-art methods in optical character recognition and achieved competitive results in phoneme classification and handwriting recognition.

7.4 Other Applications

Vergari *et al*. [86] used SPNs as autoencoders (SPAEs) for feature extraction. They trained the SPNs with LearnSPN and used the values of the internal nodes or the states of the latent variables associated to sum nodes as the codification variables. Although this model was not trained to reconstruct its inputs, experiments showed that SPAEs are competitive with state-of-the-art autoencoder architectures for several multilabel classification problems.

Butz *et al*. [87] used Bayesian networks to recognize independencies in 3,500 datasets of *soil bacteria* and combined them into an SPN in order to efficiently compute conditional probabilities and the MPE.

Nath and Domingos [88] used relational SPNs (cf. Section 6.8) for *fault localization*, i.e., finding the most probable location of bugs in computer source code. The networks, trained on a corpus of previously diagnosed buggy programs, learned to identify recurring patterns of bugs. They could also receive clues about bug suspicion from other bug detectors, such as TARANTULA.

Hilprecht *et al*. [89] proposed learning database management systems from data instead of queries, using ensembles of relational SPNs. This approach provides better accuracy and better generalization to unseen queries than different state-of-the-art methods.

Vergari *et al*. [90] also evaluated SPNs for representation learning [91]. Their SPNs encode a hierarchy of part-based representations which can be ordered by scope length. When compared with other representation learners, such as VAEs or random Boltzmann machines (RBMs), they provided competitive results both in supervised and semi-supervised settings. Moreover, the model trained for extracting representations can be used as-is for inference.

Vergari *et al*. [92] designed a tool for automatic exploratory data analysis without the need for expert statistical knowledge. It leverages Gibbs sampling and a modification of mixed SPNs [27] to model the data, and provides functionalities such as data type recognition, missing values imputation, anomaly detection, and others.

Roy *et al*. [93] addressed the explanation of activity recognition and localization in videos. A deep convolutional neural network is used for localization and its output is introduced to an SPN. Both models are learned jointly. The explainability of the system was evaluated by the user's subjective trust in the explanations that the SPN provides about the criteria of the neural net.

8 SOFTWARE FOR SPNS

Every publication about SPNs presents some experiments, and in many cases the source code is publicly available. The web page https://github.com/arranger1044/awesome-spncontains many references about SPNs, classified by year and by topic; the section "Resources" includes edges to talks and tutorials, the source code for some of those publications, and several datasets commonly used for the experiments. Most of the software is written in Python or C++.

In particular, there are two projects that aim to develop comprehensive, simple, and extensible libraries for SPNs. Both of them are written in Python and use TensorFlow as a backend for speeding up some operations. LibSPN,[7] initiated by Andrzej Pronobis at the University of Washington, Seattle, WA [94], implements several methods for inference (marginal and conditional probabilities, and approximate MPE), parameter learning (batch and online, with GD and hard EM), and visualization of SPNs. It lacks algorithms for structural learning, but it allows building convolutional SPNs with a layer-oriented interface [73]. The SPNs, stored as Python structures, are compiled into TensorFlow graphs for parameter learning and inference; for this purpose LibSPN has implemented in C++ and CUDA some operations that cannot be performed efficiently with native TensorFlow operations. Several tutorials in Jupyter Notebook are available at its website. It has been used mainly for computer vision and robotics [73], [80], [81].

The other library, SPFlow,[8] is developed by Alejandro Molina at the University of Darmstadt, Germany, with contributors from several countries [95]. It implements methods for inference (marginal and conditional probabilities, and approximate MPE), parameter learning (with GD) and several structural learning algorithms, and can be extended and customized to implement new algorithms. SPNs are usually compiled into TensorFlow for fast computation, but they can also be compiled into C, CUDA, or FPGA code.

Additionally, Peharz *et al*. [96] have implemented SPNs in Pytorch as a special case of *einsum networks*,[9] obtaining speedups and memory savings of up to two orders of magnitude. They intend to incorporate and further develop their code in SPFlow.

There are also some smaller libraries of interest, such as SumProductNetworks.jl for Julia,[10] which implements inference and parameter learning, and the LibraToolkit [97],[11] a collection of algorithms written in OCaml for learning several types of probabilistic models, such as BNs, SPNs, and others, including the ID-SPN algorithm [25].

9 EXTENSIONS OF SPNS

In recent years there have been several extensions of SPNs to more general models. In this section we briefly comment on some of them.

Sum-product-max networks (SPMNs) [98] generalize SPNs to the class of decision making problems by including two new types of nodes, max and utility, like in influence diagrams. These networks can compute the expected utility and the optimal policy in time proportional to the number of edges.

Autoencoder SPNs (AESPNs) [99] combine two SPNs with an autoencoder between them. This model produces better samples than SPNs by themselves.

Tan and Peharz [100] designed a mixture model of VAE pieces over local subsets of variables combined via an SPN. This combination yields better density estimates, smaller models, and improved data efficiency with respect to VAEs.

In credal sum-product networks (CSPNs) [101], [102], [103] the weights of each sum node do not have a fixed value, but vary inside a set (a product of probability simplexes) in such a way that each choice of the weights defines an SPN.

Sum-product graphical models (SPGMs) [104] join the semantics of probabilistic graphical models with the evaluation efficiency and expressiveness of SPNs by allowing the nodes associated to variables to appear in any part of the network, not only in the leaf nodes. Their LearnSPGM algorithm outperformed both LearnSPN and ID-SPN on the 20 real-world datasets previously used in [105].

Sum-product-quotient networks (SPQNs) [106] introduce quotient nodes, which take two inputs and output their quotient, allowing these models to represent conditional probabilities explicitly.

Tensor SPNs (tSPNs) [107] are an alternative representation of SPNs. Their main advantage is an important

7. https://www.libspn.org

8. https://github.com/SPFlow/SPFlow

9. https://github.com/cambridge-mlg/EinsumNetworks

10. https://github.com/trappmartin/SumProductNetworks.jl

11. http://libra.cs.uoregon.edu

reduction in the number of parameters—between 24 and 569 times in the experiments—with little loss of modeling accuracy. Additionally, tSPNs allow for faster inference and a deeper and more narrow neural-network architecture.

Submodular SPNs (SSPNs) [108] are an extension of SPNs for scene understanding, whose weights can be defined by submodular energy functions.

Compositional kernel machines (CKMs) [109] are an instance-based method closely related to SPNs. They have been successfully applied to image processing tasks, mainly to object recognition.

Conditional SPNs [110] extend SPNs to conditional probability distributions. They include a new type of node, called a *gating node*, which computes a convex combination of the conditional probability of its children with non-fixed weights.

Deep Structured Mixtures of Gaussian Processes (DSMGPs) [111], which have Gaussian processes (GPs) as leaf distributions, capture predictive uncertainties consistently better than previous expert-based GPs approximations.

The tutorial by Vergari, Choi, Peharz and Van den Broeck at AAAI-2020 offers an excellent review of *probabilistic circuits*, which include arithmetic circuits, SPNs, cutset networks (CNets), and probabilistic sentential decision diagrams (PSDDs), with many references about inference, learning, applications, hardware implementations, etc.[12]

10 Conclusion

SPNs are closely related to probabilistic graphical models (PGMs), such as Bayesian networks and Markov networks, but have the advantage of allowing the construction of tractable models from data. SPNs have been applied to the same tasks as neural networks, mainly computer vision and natural language processing, which exceeded by far the capabilities of PGMs. Even though deep neural networks yield in general better results, SPNs have the possibility of automatically building the structure from data and learning the parameters with either gradient descent or some of the algorithms developed for probabilistic models.

In this paper we have tried to offer a gentle introduction to SPNs, collecting information that is spread in many publications and presenting it in a coherent framework, trying to keep the mathematical complexity to the minimum necessary for describing with rigor the main properties of SPNs, from their definition to the algorithms for parameter and structural learning. We have intentionally avoided presenting SPNs as representations of network polynomials; readers interested in them can consult [9] and references therein. We have then reviewed several applications of SPNs in different domains, some extensions, and the main software libraries for SPNs. Given the rapid growth of the literature about SPNs, some sections of this paper might become obsolete soon, but we still hope it will be useful for those researchers who wish to get acquainted with this fascinating topic.

Acknowledgments

The authors would like to thank Pascal Poupart for convincing us about the advantages of SPNs, and Concha Bielza, Adnan Darwiche, Pedro Domingos, Pedro Larrañaga, Alejandro Molina, Andrzej Pronobis, Martin Trapp, Jos van de Wolfshaar, Antonio Vergari, Kaiyu Zheng, and the anonymous reviewers for many useful comments. However, all the possible mistakes and omissions are our sole responsibility. This work was supported by Grants TIN2016-77206-R and PID2019-110686RB-I00 of the Spanish Government, co-financed by the European Regional Development Fund. RSC received a postdoctoral grant and IP a predoctoral grant and from UNED, both co-financed by the Regional Government of Madrid and the Youth Employment Initiative (YEI) of the European Union.

12. http://starai.cs.ucla.edu/slides/AAAI20.pdf

References

[1] H. Poon and P. Domingos, "Sum-product networks: A new deep architecture," in *Proc. 12th Conf. Uncertainty Artif. Intell.*, 2011, pp. 337–346.
[2] A. Darwiche, "A logical approach to factoring belief networks," in *Proc. 8th Int. Conf. Princ. Knowl. Representation Reasoning*, 2002, pp. 409–420.
[3] A. Darwiche, "A differential approach to inference in Bayesian networks," *J. ACM*, vol. 50, pp. 280–305, 2003.
[4] D. Koller and N. Friedman, *Probabilistic Graphical Models: Principles and Techniques*. Cambridge, MA, USA: The MIT Press, 2009.
[5] R. Gens and P. Domingos, "Discriminative learning of sum-product networks," in *Proc. 26th Int. Conf. Neural Inf. Process. Syst.*, 2012, pp. 3239–3247.
[6] R. Gens and P. Domingos, "Learning the structure of sum-product networks," in *Proc. 30th Int. Conf. Mach. Learn.*, 2013, pp. 873–880.
[7] R. Peharz, R. Gens, F. Pernkopf, and P. M. Domingos, "On the latent variable interpretation in sum-product networks," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 39, no. 10, pp. 2030–2044, Oct. 2017.
[8] R. Peharz, S. Tschiatschek, F. Pernkopf, and P. Domingos, "On theoretical properties of sum-product networks," in *Proc. 18th Int. Conf. Artif. Intell. Statist.*, 2015, pp. 744–752.
[9] R. Peharz, "Foundations of sum-product networks for probabilistic modeling," PhD thesis, Graz Univ. Technol., Austria, 2015. [Online]. Available: https://www.researchgate.net/publication/273000973_Foundations_of_Sum-Product_Networks_for_Probabilistic_Modeling
[10] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. San Mateo, CA, USA: Morgan Kaufmann, 1988.
[11] M. Chavira and A. Darwiche, "Compiling Bayesian networks using variable elimination," in *Proc. 20th Int. Joint Conf. Artif. Intell.*, 2007, pp. 2443–2449.
[12] H. Zhao, M. Melibari, and P. Poupart, "On the relationship between sum-product networks and Bayesian networks," in *Proc. 32nd Int. Conf. Mach. Learn.*, 2015, pp. 116–124.
[13] C. Butz, J. Oliveira, and R. Peharz, "Sum-product network decompilation," *Proc. Mach. Learn. Res.*, vol. 138, pp. 53–64, 2020. [Online]. Available: http://proceedings.mlr.press/v138/butz20a.html
[14] G. F. Cooper, "The computational complexity of probabilistic inference using Bayesian belief networks," *Artif. Intell.*, vol. 42, pp. 393–405, 1990.
[15] C. Boutilier, N. Friedman, M. Goldszmidt, and D. Koller, "Context-specific independence in Bayesian networks," in *Proc. 12th Conf. Uncertainty Artif. Intell.*, 1996, pp. 115–123.
[16] R. Mourad *et al.*, "A survey on latent tree models and applications," *J. Artif. Intell. Res.*, vol. 47, pp. 157–203, 2013.
[17] J. Pearl, *Causality. Models, Reasoning, and Inference*. Cambridge, U.K.: Cambridge Univ. Press, 2000.
[18] I. Bermejo, J. Oliva, F. J. Díez, and M. Arias, "Interactive learning of Bayesian networks with OpenMarkov," in *Proc. 6th Conf. Probabilistic Graphical Models*, 2012, pp. 27–34.
[19] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evol. Comput.*, vol. 10, pp. 99–127, 2002.
[20] S. Ding *et al.*, "Evolutionary artificial neural networks: A review," *Artif. Intell. Rev.*, vol. 39, pp. 251–260, 2013.
[21] R. Peharz *et al.*, "Random sum-product networks: Simple and effective approach to probabilistic deep learning," in *Proc. 35th Conf. Uncertainty Artif. Intell.*, 2020, pp. 334–344.
[22] A. Ghose, P. Jaini, and P. Poupart, "Learning directed acyclic graph SPNs in sub-quadratic time," *Int. J. Approx. Reasoning*, vol. 120, pp. 48–73, 2020.

[23] K. Stelzner, R. Peharz, and K. Kersting, “Faster attend-infer-repeat with tractable probabilistic models,” in *Proc. 36th Int. Conf. Mach. Learn.*, 2019, pp. 5966–5975.

[24] J. D. Park, “MAP complexity results and approximation methods,” in *Proc. 9th Conf. Uncertainty Artif. Intell.*, 2002, pp. 388–396.

[25] A. Rooshenas and D. Lowd, “Learning sum-product networks with direct and indirect variable interactions,” in *Proc. 31st Int. Conf. Mach. Learn.*, 2014, pp. 710–718.

[26] A. Molina, S. Natarajan, and K. Kersting, “Poisson sum-product networks: A deep architecture for tractable multivariate poisson distributions,” in *Proc. 31st AAAI Conf. Artif. Intell.*, 2017, pp. 2357–2363.

[27] A. Molina *et al.*, “Mixed sum-product networks: A deep architecture for hybrid domains,” in *Proc. 32nd AAAI Conf. Artif. Intell.*, 2018, pp. 3828–3835.

[28] M. Desana and C. Schnörr, “Learning arbitrary sum-product network leaves with Expectation-Maximization,” 2017, *arXiv:1604.07243*.

[29] W. Hsu, A. Kalra, and P. Poupart, “Online structure learning for sum-product networks with Gaussian leaves,” in *Proc. 5th Int. Conf. Learn. Representations*, 2017. [Online]. Available: https://iclr.cc/archive/www/doku.php%3Fid=iclr2017:main.html

[30] A. Vergari, N. Di Mauro, and F. Esposito, “Simplifying, regularizing and strengthening sum-product network structure learning,” in *Proc. Eur. Conf. Mach. Learn. Princ. Pract. Knowl. Discov. Databases*, 2015, pp. 343–358.

[31] R. Peharz, R. Gens, and P. Domingos, “Learning selective sum-product networks,” in *Proc. 31st Int. Conf. Mach. Learn. Learn. Tractable Probabilistic Models Workshop*, 2014. [Online]. Available: https://sites.google.com/site/ltpm2014/accepted-papers

[32] A. Choi and A. Darwiche, “On relaxing determinism in arithmetic circuits,” in *Proc. 34th Int. Conf. Mach. Learn.*, 2017, pp. 825–833.

[33] H. Zhao, P. Poupart, and G. J. Gordon, “A unified approach for learning the parameters of sum-product networks,” in *Proc. 30th Conf. Neural Inf. Process. Syst.*, 2016, pp. 433–441.

[34] C. J. Butz *et al.*, “An empirical study of methods for SPN learning and inference,” in *Proc. 9th Int. Conf. Probabilistic Graphical Models*, 2018, pp. 49–60.

[35] H. Chan and A. Darwiche, “On the robustness of most probable explanations,” in *Proc. 22nd Conf. Uncertainty Artif. Intell.*, 2006, pp. 63–71.

[36] J. Mei, Y. Jiang, and K. Tu, “Maximum a posteriori inference in sum-product networks,” in *Proc. 32nd AAAI Conf. Artif. Intell.*, 2018, pp. 1923–1930.

[37] D. Conaty, D. D. Mauá, and C. P. de Campos, “Approximation complexity of maximum a posteriori in sum-product networks,” 2017, *arXiv: 1703.06045*.

[38] D. Conaty, D. D. Mauá, and C. P. de Campos, “Approximation complexity of maximum a posteriori in sum-product networks,” in *Proc. 33rd Conf. Uncertainty Artif. Intell.*, 2017, pp. 322–331.

[39] J. Duchi, S. Shalev-Shwartz , Y. Singer, and T. Chandra, “Efficient projections onto the l_1-ball for learning in high dimensions,” in *Proc. 25th Int. Conf. Mach. Learn.*, 2008, pp. 272–279.

[40] A. Rashwan, P. Poupart, and C. Zhitang, “Discriminative training of sum-product networks by extended Baum-Welch,” in *Proc. 9th Int. Conf. Probabilistic Graphical Models*, 2018, pp. 356–367.

[41] M. Trapp, R. Peharz, and F. Pernkopf, “Optimisation of overparametrized sum-product networks,” in *Proc. Workshop Tractable Probabilistic Modeling ICML*, 2019. [Online]. Available: https://sites.google.com/view/icmltpm2019/accepted-papers

[42] S. Arora, N. Cohen, and E. Hazan, “On the optimization of deep networks: Implicit acceleration by overparameterization,” in *Proc. 35th Int. Conf. Mach. Learn.*, 2018, pp. 244–253.

[43] M. Trapp *et al.*, “Safe semi-supervised learning of sum-product networks,” in *Proc. 33rd Conf. Uncertainty Artif. Intell.*, 2017. [Online]. Available: http://auai.org/uai2017/

[44] M. Loog, “Contrastive pessimistic likelihood estimation for semi-supervised classification,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 38, no. 3, pp. 462–475, Mar. 2016.

[45] A. Rashwan, H. Zhao, and P. Poupart, “Online and distributed Bayesian moment matching for parameter learning in sum-product networks,” in *Proc. 19th Int. Conf. Artif. Intell. Statist.*, 2016, pp. 1469–1477.

[46] H. Zhao, T. Adel, G. Gordon, and B. Amos, “Collapsed variational inference for sum-product networks,” in *Proc. 33rd Int. Conf. Mach. Learn.*, 2016, pp. 1310–1318.

[47] P. Jaini *et al.*, “Online algorithms for sum-product networks with continuous variables,” in *Proc. 8th Int. Conf. Probabilistic Graphical Models*, 2016, pp. 228–239.

[48] H. Zhao and G. J. Gordon, “Linear time computation of moments in sum-product networks,” in *Proc. 26th Conf. Neural Inf. Process. Syst.*, 2017, pp. 6894–6903.

[49] I. Aden-Ali and H. Ashtiani, “On the sample complexity of learning sum-product networks,” in *Proc. 23rd Int. Conf. Artif. Intell. Statist.*, 2020, vol. 108, pp. 4508–4518.

[50] A. Dennis and D. Ventura, “Learning the architecture of sum-product networks using clustering on variables,” in *Proc. 26th Conf. Neural Inf. Process. Syst.*, 2012, pp. 3239–3247.

[51] R. Peharz, B. C. Geiger, and F. Pernkopf, “Greedy part-wise learning of sum-product networks,” in *Proc. Eur. Conf. Mach. Learn. Princ. Pract. Knowl. Discov. Databases*, 2013, pp. 612–627.

[52] D. Lowd and A. Rooshenas, “Learning Markov networks with arithmetic circuits,” in *Proc. 16th Int. Conf. Artif. Intell. Statist.*, 2013, pp. 406–414.

[53] T. Adel, D. Balduzzi, and A. Ghodsi, “Learning the structure of sum-product networks via an SVD-based algorithm,” in *Proc. 31st Conf. Uncertainty Artif. Intell.*, 2015, pp. 32–41.

[54] C. Chow and C. Liu, “Approximating discrete probability distributions with dependence trees,” *IEEE Trans. Inf. Theory*, vol. IT-14, no. 3, pp. 462–467, May 1968.

[55] L. Breiman, “Random forests,” *Mach. Learn.*, vol. 45, pp. 5–32, 2001.

[56] A. Dennis and D. Ventura, “Greedy structure search for sum-product networks,” in *Proc. 24th Int. Joint Conf. Artif. Intell.*, 2015, pp. 932–938.

[57] T. Rahman and V. Gogate, “Merging strategies for sum-product networks: From trees to graphs,” in *Proc. 32nd Conf. Uncertainty Artif. Intell.*, 2016, pp. 617–626.

[58] N. Di Mauro, F. Esposito, F. Ventola, and A. Vergari, “Alternative variable splitting methods to learn sum-product networks,” in *Proc. 16th Int. Conf. Italian Assoc. Artif. Intell.*, 2017, pp. 334–346.

[59] Y. Liu and T. Luo, “The optimization of sum-product network structure learning,” *J. Vis. Commun. Image Representation*, vol. 60, pp. 391–397, 2019.

[60] A. Bueff, S. Speichert, and V. Belle, “Tractable querying and learning in hybrid domains via sum-product networks,” 2018, *arXiv: 1807.05464*.

[61] G. Schwarz, “Estimating the dimension of a model,” *Ann. Statist.*, vol. 17, pp. 461–464, 1978.

[62] S. Lee, M. Heo, and B. Zhang, “Online incremental structure learning of sum-product networks,” in *Proc. 20th Int. Conf. Neural Inf. Process.*, 2013, pp. 220–227.

[63] A. Dennis and D. Ventura, “Online structure-search for sum-product networks,” in *Proc. 16th IEEE Int. Conf. Mach. Learn. Appl.*, 2017, pp. 155–160.

[64] P. Jaini, A. Ghose, and P. Poupart, “Prometheus: Directly learning acyclic directed graph structures for sum-product networks,” in *Proc. 9th Int. Conf. Probabilistic Graphical Models*, 2018, pp. 181–192.

[65] M. Melibari, P. Poupart, P. Doshi, and G. Trimponias, “Dynamic sum-product networks for tractable inference on sequence data,” in *Proc. 8th Conf. Probabilistic Graphical Models*, 2016, pp. 345–356.

[66] T. Dean and K. Kanazawa, “A model for reasoning about persistence and causation,” *Comput. Intell.*, vol. 5, pp. 142–150, 1989.

[67] A. Kalra *et al.*, “Online structure learning for feed-forward and recurrent sum-product networks,” in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2018, pp. 6944–6954.

[68] A. Nath and P. Domingos, “Learning relational sum-product networks,” in *Proc. 29th AAAI Conf. Artif. Intell.*, 2015, pp. 2878–2886.

[69] C. Lee, S. Watkins and B. Zhang, “Non-parametric Bayesian sum-product networks,” in *Proc. Int. Conf. Mach. Learn.*, 2014.

[70] M. Trapp *et al.*, “Structure inference in sum-product networks using infinite sum-product trees,” in *Proc. 29th Conf. Neural Inf. Process. Syst.*, 2016.

[71] M. Trapp *et al.*, “Bayesian learning of sum-product networks,” in *Proc. 33rd Conf. Neural Inf. Process. Syst.*, 2019, pp. 6344–6355. [Online]. Available: https://sites.google.com/site/nipsbnp2016/accepted-papers. There are no page numbers for this workshop

[72] T. Hartmann, “Discriminative convolutional sum-product networks on GPU,” M.S. thesis, Dept. Auton. Intell. Syst., Rheinische Friedrich-Wilhelms-Universität Bonn, Germany, 2014.

[73] J. van de Wolfshaar and A. Pronobis, "Deep generalized convolutional sum-product networks," in *Proc. 10th Int. Conf. Probabilistic Graphical Models*, 2020, pp. 533–544.

[74] C. J. Butz, J. S. Oliveira, A. E. dos Santos, and A. L. Teixeira, "Deep convolutional sum-product networks," in *Proc. 33rd AAAI Conf. Artif. Intell.*, 2019, pp. 3248–3255.

[75] Z. Yuan *et al.*, "Modeling spatial layout for scene image understanding via a novel multiscale sum-product network," *Expert Syst. Appl.*, vol. 63, pp. 231–240, 2016.

[76] F. Rathke, M. Desana, and C. Schnörr, "Locally adaptive probabilistic models for global segmentation of pathological OCT scans," in *Proc. 20th Int. Conf. Med. Image Comput. Comput. Assisted Intervention*, 2017, pp. 177–184.

[77] J. Wang and G. Wang, "Hierarchical spatial sum-product networks for action recognition in still images," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 28, no. 1, pp. 90–100, Jan. 2018.

[78] M. R. Amer and S. Todorovic, "Sum product networks for activity recognition," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 38, no. 4, pp. 800–813, Apr. 2016.

[79] B. M. Sguerra and F. G. Cozman, "Image classification using sum-product networks for autonomous flight of micro aerial vehicles," in *Proc. 5th Brazilian Conf. Intell. Syst.*, 2016, pp. 139–144.

[80] A. Pronobis, F. Riccio, and R. P. N. Rao, "Deep spatial affordance hierarchy: Spatial knowledge representation for planning in large-scale environments," in *Proc. 27th Int. Conf. Autom. Planning Scheduling Workshop Planning Robot.*, 2017, pp. 68–79.

[81] K. Zheng, A. Pronobis, and R. P. N. Rao, "Learning semantic maps with topological spatial relations using graph-structured sum-product networks," in *Proc. AAAI Conf. Artif. Intell.*, 2018, pp. 4547–4555.

[82] K. Zheng and A. Pronobis, "From pixels to buildings: End-to-end probabilistic deep networks for large-scale semantic mapping," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2019, pp. 3511–3518.

[83] R. Peharz, G. Kapeller, P. Mowlaee, and F. Pernkopf, "Modeling speech with sum-product networks: Application to bandwidth extension," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, 2014, pp. 3699–3703.

[84] W.-C. Cheng *et al.*, "Language modeling with sum-product networks," in *Proc. 15th Annu. Conf. Int. Speech Commun. Assoc.*, 2014, pp. 2098–2102.

[85] M. Ratajczak, S. Tschiatschek, and F. Pernkopf, "Sum-product networks for sequence labeling," 2018, *arXiv: 1807.02324*.

[86] A. Vergari, R. Peharz, N. Di Mauro , A. Molina, K. Kersting, and F. Esposito, "Sum-product autoencoding: Encoding and decoding representations using sum-product networks," in *Proc. 32nd AAAI Conf. Artif. Intell.*, 2018, pp. 4163–4170.

[87] C. J. Butz, A. E. dos Santos, J. S. Oliveira, and J. Stavrinides, "Efficient examination of soil bacteria using probabilistic graphical models," in *Proc. 31st Int. Conf. Ind. Eng. Other Appl. Appl. Intell. Syst.*, 2018, pp. 315–326.

[88] A. Nath and P. Domingos, "Learning tractable probabilistic models for fault localization," in *Proc. 30th AAAI Conf. Artif. Intell.*, 2016, pp. 1294–1301.

[89] B. Hilprecht *et al.*, "DeepDB: Learn from data, not from queries!," in *Proc. VLDB Endowment*, vol. 13, pp. 992–1005, 2020.

[90] A. Vergari, N. Di Mauro, and F. Esposito, "Visualizing and understanding sum-product networks," *Mach. Learn.*, vol. 108, pp. 551–573, 2019.

[91] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 35, no. 8, pp. 1798–1828, Aug. 2013.

[92] A. Vergari *et al.*, "Automatic Bayesian density analysis," in *Proc. 33rd AAAI Conf. Artif. Intell.*, 2019, vol. 33, pp. 5207–5215.

[93] C. Roy *et al.*, "Explainable activity recognition in videos," in *Proc. 24th Int. Conf. Intell. User Interfaces Workshops*, 2019. [Online]. Available: http://ceur-ws.org/Vol-2327/

[94] A. Pronobis, A. Ranganath, and R. P. N. Rao, "LibSPN: A library for learning and inference with sum-product networks and TensorFlow," in *Proc. 34th Int. Conf. Mach. Learn. Principled Approaches Deep Learn. Workshop*, 2017. [Online]. Available: https://icml.cc/Conferences/2017/ScheduleMultitrack?event=13

[95] A. Molina *et al.*, "SPFlow: An easy and extensible library for deep probabilistic learning using sum-product networks," 2019, *arXiv: 1901.03704*.

[96] R. Peharz *et al.*, "Einsum networks: Fast and scalable learning of tractable probabilistic circuits," in *Proc. 29th AAAI Conf. Artif. Intell.*, 2015, pp. 2878–2886.

[97] D. Lowd and A. Rooshenas, "The Libra toolkit for probabilistic models," *J. Mach. Learn. Res.*, vol. 16, pp. 2459–2463, 2015.

[98] M. Melibari, P. Poupart, and P. Doshi, "Sum-product-max networks for tractable decision making," in *Proc. 25th Int. Joint Conf. Artif. Intell.*, 2016, pp. 1846–1852.

[99] A. Dennis and D. Ventura, "Autoencoder-enhanced sum-product networks," in *Proc. 16th IEEE Int. Conf. Mach. Learn. Appl.*, 2017, pp. 1041–1044.

[100] P. L. Tan and R. Peharz, "Hierarchical decompositional mixtures of variational autoencoders," in *Proc. 36th Int. Conf. Mach. Learn.*, 2019, pp. 6115–6124.

[101] D. D. Mauá, F. G. Cozman, D. Conaty, and C. P. de Campos, "Credal sum-product networks," in *Proc. 10th Int. Symp. Imprecise Probability: Theories Appl.*, 2017, pp. 205–216.

[102] D. D. Mauá *et al.*, "Robustifying sum-product networks," *Int. J. Approx. Reasoning*, vol. 101, pp. 163–180, 2018.

[103] A. Levray and V. V. Belle, "Learning credal sum-product networks," in *Proc. Conf. Autom. Knowl. Base Construction*, 2020. [Online]. Available: https://www.akbc.ws/2020/papers/

[104] M. Desana and C. Schnörr, "Sum-product graphical models," *Mach. Learn.*, vol. 109, pp. 135–173, 2020.

[105] D. Lowd and P. Domingos, "Learning arithmetic circuits," in *Proc. 24th Conf. Uncertainty Artif. Intell.*, 2008, pp. 383–392.

[106] O. Sharir and A. Shashua, "Sum-product-quotient networks," in *Proc. 21st Int. Conf. Artif. Intell. Statist.*, 2018, pp. 529–537.

[107] C. Y. Ko *et al.*, "Deep model compression and inference speedup of sum–product networks on tensor trains," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 31, no. 7, pp. 2665–2671, Jul. 2020.

[108] A. L. Friesen and P. Domingos, "Unifying sum-product networks and submodular fields," in *Proc. 34th Int. Conf. Mach. Learn.*, 2017. [Online]. Available: https://icml.cc/Conferences/2017/ScheduleMultitrack?event=13

[109] R. Gens and P. Domingos, "Compositional kernel machines," in *Proc. 5th Int. Conf. Learn. Representations*, 2017.

[110] X. Shao *et al.*, "Conditional sum-product networks: Imposing structure on deep probabilistic architectures," in *Proc. Work. Notes ICML 2019 Workshop Tractable Probabilistic Models*, 2019. [Online]. Available: https://sites.google.com/view/icmltpm2019/accepted-papers

[111] M. Trapp, R. Peharz, F. Pernkopf, and C. E. Rasmussen, "Deep structured mixtures of Gaussian processes," in *Proc. 23rd Int. Conf. Artif. Intell. Statist.*, 2020, vol. 108, pp. 2251–2261.

Raquel Sánchez Cauce received the graduate degree in mathematics and the master's degree in mathematics and applications from the Autonomous University of Madrid (UAM), Spain, and the PhD degree in mathematics from the Autonomous University of Madrid (UAM), Spain, in 2018. She is a postdoctoral researcher with the Department of Artificial Intelligence, Spanish National University for Distance Education (UNED), Madrid, Spain. Her research interests include deep learning, artificial vision, differential Galois theory, and integrable systems.

Iago París received the master's degree in physics from the Complutense University of Madrid, Spain and the master's degree in artificial intelligence from UNED, Spain, in 2019, where he also worked at the Department of Artificial Intelligence, researching about deep learning algorithms, especially SPNs.

Francisco Javier Díez received the master's degree in theoretical physics from the Autonomous University of Madrid (UAM), Spain, in 1988 and the PhD degree from UNED, Spain, in 1994. He is currently a professor at the Department of Artificial Intelligence, UNED, Spain. His research interests include probabilistic graphical models and their application to medicine. He has recently become interested in deep learning applied to medical imaging.