UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Proyecto de Fin de Máster en IA avanzada: fundamentos, métodos y aplicaciones

SUM-PRODUCT NETWORKS

Iago París Fernández

Dirigido por: Dr. Francisco Javier Díez Vegas

Dra. Raquel Sánchez Cauce

Curso: 2018-2019: 2ª Convocatoria

Dedicatory

I want to dedicate the effort behind this work, the journey, to Raquel.
For me a role model as a person, as a colleague and as a friend.

Acknowledgements

I first thank Francisco Javier Díez Vegas for the opportunity to work in this master thesis and all I have learned from it. I also thank Raquel Sánchez Cauce for her insights and her humor. I thank the CISIAD members for its support during this course. And I thank my office teammates, for being there when it mattered. Thank you, everyone!

This work would have not been possible without the predoctoral grant cofinanciated by the European Social Fund and the Comunidad de Madrid. It would have not been possible without the National University of Distance Education (UNED), who provided for the funding to extend my activity for another year. And it would have not been possible without the national project TIN2016-77206-R which financed the physical resources necessary to complete this work.

Abstract

Sum-Product Networks (SPNs) are a new model that join elements from deep learning and probabilistic graphical models. They model a dataset through a hierarchical combination of mixtures and factorizations of probability distributions. Since their appearance, SPNs have obtained state of the art results in several machine learning areas. The literature about SPNs contains several obsolete papers and has neither a survey nor an introductory work addressed to the new reader. The main contribution of this work, the first survey about SPNs, aims to fill this long-standing gap. For this work, we have thoroughly reviewed most of the existing literature. The survey presents the basic knowledge required to comprehend SPNs and then reviews how they learn their structure and parameters from data, how they perform inference, where and how they have been applied, and how they compare with other models. As an experimental contribution, the recent model of convolutional SPNs has been applied to different image classification problems and compared with convolutional neural networks.

Contents

List of Figures

List of Tables

Chapter 1

General introduction and objectives

In 2011, Poon and Domingos presented a new algorithm for the Artificial Intelligence area [53]. Inspired in the arithmetic circuits of Darwiche [12], sum-product networks (SPNs) join elements from deep learning and probabilistic graphical models, bringing onto the stage a generative algorithm arrayed as a computational graph with a probabilistic interpretation in every one of its nodes, along with the possibility of computing exact inference in linear time with respect to the number of links of the network.

Since then, SPNs have performed very well on several machine learning tasks as image completion, natural language processing, video categorization or medical image analysis [2, 7, 53, 59].

1.1 Motivation and objectives

The Research Center for Intelligent Decision-Support Systems (CISIAD) is a group with a long research trajectory about probabilistic graphical models (PGMs) such as Bayesian networks and influence diagrams. The appearance of SPNs is an opportunity to join this past experience on PGMs with the present success of neural networks. This master thesis is the first step to open this new line of research for the group. SPNs were born eight years ago so there are few published works. This made us capable to be comprehensive, to read and understand most of them and to know about a vast majority of the rest. We found that in spite of eight years of development, the literature lacked a stepping stone to introduce the new reader into SPNs. Our main objective is to remedy that lack by writing the first survey about SPNs as the main contribution of this master thesis. This is specially relevant since several statements presented in the first published works are now obsolete. Moreover, we provide in our survey original proofs that explain different characteristics and algorithms of SPNs without the help of interpretations based on other models (such as Bayesian networks or arithmetic circuits), aiming to be self-contained. As an additional contribution, we complement the theoretical survey with a practical application of one of the cutting-edge SPN-based models: convolutional SPNs (ConvSPNs). ConvSPNs translate the advantages of the convolutional layer of neural networks using

an SPN structure. They were presented at the end of February 2019 alongside an SPN python library that implements them: LibSPN, developed by researchers of the University of Washington. Our first step here was to understand the model and to get acquainted with the library. After that, we decided to test ConvSPNs on the well-known task of image classification, and compare their performance with the model that inspired them: convolutional neural networks.

The main objective of this work is to understand sum-product networks both theoretically and experimentally. The particular objectives can be summarized as follows:

1. Theoretical understanding
    a) Study the two main basis of SPNs: deep learning and probabilistic graphical models.
    b) Research the developments about SPNs in these eight years.
    c) Write the first survey about SPNs.
2. Practical application
    a) Understand and learn how to apply one of the current state of the art SPN algorithms.
    b) Use it to solve image classification tasks of different difficulty.
    c) Compare its performance with convolutional neural networks.

1.2 Structure of this work

Aside from this introduction, this work follows with two main sections. The first presents the first survey about SPNs. The second describes the application of ConvSPNs. This work is organized as follows.

This chapter introduces the work and its goals.

Chapter 2 contains the survey about sum-product networks. In the first section some preliminaries are given to the reader. The next section explains what an SPN is and introduces several relevant properties. The inference section shows the different inference queries that an SPN can answer and the works that delved on that area. The parameter learning section lay out the two main algorithms for parameter learning: Expectation-Maximization (EM) and gradient descent, and the improvements presented in the literature. The structure learning section presents the different approaches to automatic structure learning from data. Next the applications section reviews some of the most relevant tasks solved with SPNs in areas such as video classification or robotics. Finally, the last sections briefly comment the available software and different extensions to SPNs.

Chapter 3 addresses the application of ConvSPNs to image classification. The first section presents a brief introduction to both convolutional neural networks and ConvSPNs, and the concepts necessary to understand them. The methods section explains the experimental design and the

choice of datasets. Then, the results section show accuracies and times for both ConvSPNs and convolutional neural networks. Finally, the last section discusses those results.

The last chapter compares more generally SPNs with probabilistic graphical models and neural networks. Then it presents the conclusions and points to several lines of future work.

Chapter 2

Sum-product networks: a survey

2.1 Preliminaries

Here we introduce relevant concepts useful for understanding this master thesis.

In this work we assume that every variable either has a finite set of possible values (called *states*) or is continuous, i.e., takes values in $\mathbb{R}$.

2.1.1 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 $\text{conf}(\mathbf{V})$ is the set of all the configurations of $\mathbf{V}$. The empty set has only one configuration, represented by $\blacklozenge$.

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

$$\text{conf}^*(\mathbf{V}) = \{\mathbf{x} \mid \mathbf{X} \subseteq \mathbf{V}\} \,. \tag{2.1}$$

We can think of $\text{conf}^*(\mathbf{V}) \setminus \text{conf}(\mathbf{V})$ 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* 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 $\mathbf{v}$. In order to simplify the notation, when $\mathbf{X}$ has a single variable, V, we write v instead of (v) and $\mathbf{v}^{\downarrow V}$ instead of $\mathbf{v}^{\downarrow \{V\}}$.

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 $(\mathbf{xy})^{\downarrow \mathbf{Y}} = \mathbf{y}$.

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. The configuration of the empty set $\blacklozenge$ is compatible with every other configuration.

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

$$\mathbb{I}_v(\mathbf{x}) = \begin{cases} 1 & \text{if } V \notin \mathbf{X} \vee v = \mathbf{x}^{\downarrow V} \\ 0 & \text{otherwise .} \end{cases} \tag{2.2}$$

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

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

Indicators can be used to determine whether two configurations are compatible, i.e., whether $\mathbf{x} = \mathbf{v}^{\downarrow \mathbf{X}}$, as follows:

Proposition 3. *If* $\mathbf{x}$ *and* $\mathbf{v}$ *are two configurations such that* $\mathbf{X} \subseteq \mathbf{V}$*, then*

$$\prod_{V \in \mathbf{V}} \mathbb{I}_v(\mathbf{x}) = \begin{cases} 1 & \textit{if } \mathbf{x} = \mathbf{v}^{\downarrow \mathbf{X}} \\ 0 & \textit{otherwise} , \end{cases} \tag{2.3}$$

where $v = \mathbf{v}^{\downarrow V}$ *for every* $V \in \mathbf{V}$*.*

The proof of all the propositions is in the section 2.9.

2.1.2 Probability distributions and probability functions

Definition 4. A *probability distribution* defined on $\mathbf{V}$ is a function $P : \text{conf}(\mathbf{V}) \mapsto \mathbb{R}$ such that:

$$P(\mathbf{v}) \geq 0, \tag{2.4}$$

$$\sum_{\mathbf{v}} P(\mathbf{v}) = 1 . \tag{2.5}$$

This definition can be extended so that P represents not only a probability distribution but also all its marginal probabilities, as follows.

Definition 5. A *probability function* defined on $\mathbf{V}$ is a function $P : \text{conf}^*(\mathbf{V}) \mapsto \mathbb{R}$ such that the restriction of P to $\text{conf}(\mathbf{V})$ is a probability distribution and for every configuration $\mathbf{x}$ such that $\mathbf{X} \subset \mathbf{V}$,

$$P(\mathbf{x}) = \sum_{\mathbf{v} | \mathbf{v}^{\downarrow \mathbf{X}} = \mathbf{x}} P(\mathbf{v}) . \tag{2.6}$$

This equation is the definition of marginal probability: $P(\mathbf{x})$ is obtained by summing the probabilities of all the configurations of $\mathbf{V}$ compatible with $\mathbf{x}$. Because of Equation 2.3 it can be rewritten as

$$P(\mathbf{x}) = \sum_{\mathbf{v}} P(\mathbf{v}) \prod_{V \in \mathbf{V}} \mathbb{I}_v(\mathbf{x}) . \tag{2.7}$$

Proposition 6. *If P is a probability function defined on $\mathbf{V}$ and $\mathbf{X} \subset \mathbf{V}$, the restriction of P to* $\mathrm{conf}(\mathbf{X})$ *is a probability distribution* for $\mathbf{X}$.

It is possible to define a new probability function as the sum or the product of other probability functions.

Proposition 7. *Let us consider n probability functions $\{P_1, \ldots, P_n\}$ defined on the same set of variables, $\mathbf{V}$, and n weights, $\{w_1, \ldots, w_n\}$, with $w_j \geq 0$ for every j, and $\sum_{j=1}^{n} w_j = 1$. The function P : conf$^*(\mathbf{V}) \mapsto \mathbb{R}$, such that for every configuration of $\mathbf{X} \subseteq \mathbf{V}$*

$$P(\mathbf{x}) = \sum_{j=1}^{n} w_j \cdot P_j(\mathbf{x}),$$

is a probability function. It is said to be a weighted average *or a* convex combination *of probability functions.*

Proposition 8. *Let $\{P_1, \ldots, P_n\}$ be a set of probability functions defined on n disjoint sets of variables, $\{\mathbf{V}_1, \ldots, \mathbf{V}_n\}$, respectively. Let $\mathbf{V} = \mathbf{V}_1 \cup \ldots \cup \mathbf{V}_n$. The function P : conf$^*(\mathbf{V}) \mapsto \mathbb{R}$, such that for every configuration of $\mathbf{X} \subseteq \mathbf{V}$*

$$P(\mathbf{x}) = \prod_{j=1}^{n} P_j(\mathbf{x}) \ ,$$

is a probability function.

2.1.3 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 links. When there is a link $n_i \to n_j$ we say that n_i is a *parent* of n_j and n_j is a *child* of n_i; there cannot be another link 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 Figure 2.2.1, $ch(1) = \{2, 3\}$. 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. The set of descendants of n_i is denoted by desc(n_i).

A cycle of length l consists of a set of l nodes and l links $\{n_1 \to n_2, n_2 \to n_3, \ldots, n_{l-1} \to n_l, n_l \to n_1\}$. 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 work when we say "a tree" we mean "a directed tree".

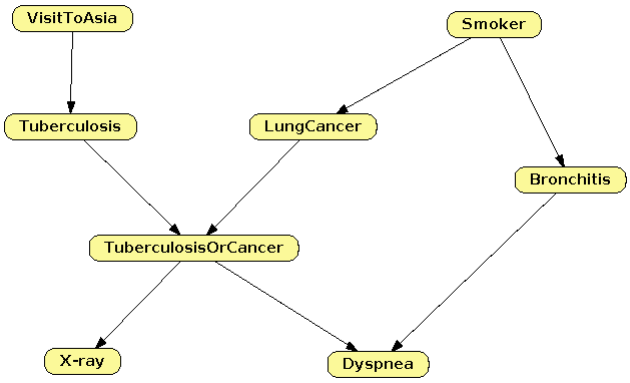

Figure 2.1.1: Example of a Bayesian net.

2.1.4 Bayesian networks

A Bayesian network models the probability distribution of a set of random variables $\mathbf{V}$ through a product of conditional distributions:

$$P(\mathbf{v}) = \prod_i P\left(v_i | pa\left(V_i\right)\right).$$

This product can be represented as a directed acyclic graph where each variable has a node assigned and an incoming directed edge for each variable that conditions it, which in graph terminology are called its parents. An example is represented on Figure 2.1.1.

2.1.5 MAP, MPE, and MAX inference

Let $\mathbf{E}$ and $\mathbf{X}$ be two disjoint subsets of $\mathbf{V}$, where $\mathbf{E}$ is the set of variables of known values, the evidence; and $\mathbf{X}$ is the set of variables whose values we want to know, the query. The probability $P(\mathbf{x} \mid \mathbf{e})$ is relevant in practice when configuration $\mathbf{e}$ denotes the evidence, i.e., the values observed (for example, the symptoms and signs of a medical examination, the pixels in an image...) and $\mathbf{X}$ contains the variables of interest (the possible diagnostics, the objects that may be present in the image...). In this context $P(\mathbf{x} \mid \mathbf{e})$ is called the *posterior probability*.

Given $\mathbf{e}$ and $\mathbf{X}$, the *maximum a-posteriori* (MAP) configuration is

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

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* (MPE) is the configuration of $\mathbf{X} = \mathbf{V} \setminus \mathbf{E}$ that maximizes the posterior probability:

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

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.

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:

$$\text{MAX}\,(\mathbf{x}) = \arg\max_{\mathbf{x}} P(\mathbf{x})\ .$$

2.2 Basic definitions of SPNs

2.2.1 Structure of an SPN

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

- every leaf node is an indicator for a variable,
- all the other nodes are either of type sum or product,
- the parents of a sum node are all product nodes, and vice versa, and
- every link $n_i \to n_j$ outgoing from a sum node has an associated weight, $w_{ij} \geq 0$.

Usually $w_{ij} > 0$. We will assume, unless otherwise stated, that all SPNs are normalized, i.e.,

$$\forall i,\ \sum_{j \in \text{ch}(i)} w_{ij} = 1\ .$$

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

If a leaf associated to variable V is a descendant of node n_i, we say that V belongs to the *scope* of n_i, which is denoted by $\text{sc}(n_i)$. Alternatively, we can define the scope recursively as follows: if n_i is an indicator node associated to variable V, then $\text{sc}(n_i) = \{V\}$; if n_i is a non-terminal node, its scope is the union of the scopes of its children:

$$\text{sc}(n_i) = \bigcup_{j \in ch(i)} \text{sc}(n_j)\ . \tag{2.1}$$

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

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 disjoint scopes. An SPN is *decomposable* if all its product nodes are decomposable.

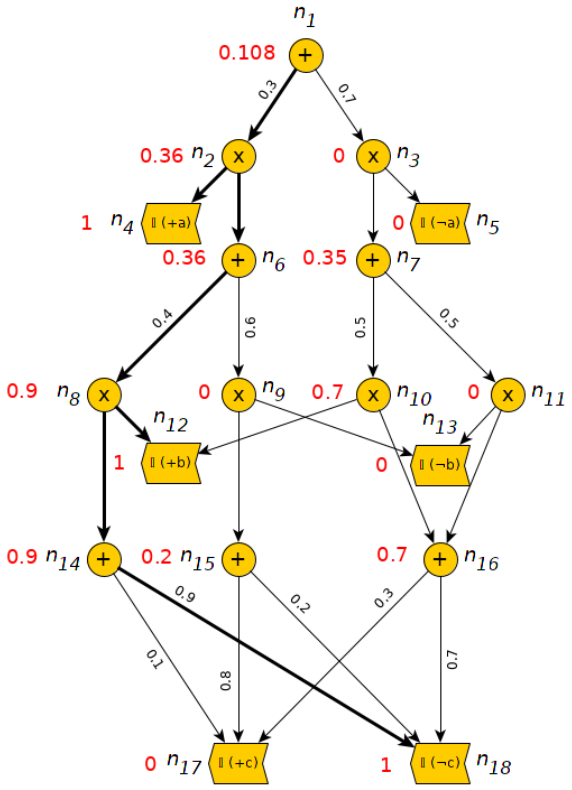

Figure 2.2.1: An SPN whose scope contains three variables: A, B and C. The 6 leaf nodes in the SPN, drawn like switches, are the indicators for these variables. They are the input of the SPN for every configuration of the variables (including partial configurations). The numbers in red are the values $S_i(\mathbf{v})$ for $\mathbf{v} = (+a, +b, \neg c)$.

Proposition 9. *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 work we will assume that all the SPNs are complete and decomposable.

2.2.2 Node values and probability distributions

Definition 10 (Value $S_i(\mathbf{x})$). Let n_i be a node of $\mathcal{S}$ and $\mathbf{x} \in \text{conf}^*(\mathcal{S})$. If n_i is an indicator $\mathbb{I}_v$, then

$$S_i(\mathbf{x}) = \mathbb{I}_v(\mathbf{x}) \; ; \tag{2.2}$$

if it is a sum node,

$$S_i(\mathbf{x}) = \sum_{j \in ch(i)} w_{ij} \cdot S_j(\mathbf{x}) \; , \tag{2.3}$$

and if it is a product node,

$$S_i(\mathbf{x}) = \prod_{j \in ch(i)} S_j(\mathbf{x}) \; . \tag{2.4}$$

Definition 11 (Value $S(\mathbf{x})$)**.** The *value* $S(\mathbf{x})$ returned by the SPN is the value of the root, $S_r(\mathbf{x})$.

Theorem 12. *For a node n_i in an SPN, the function P_i : conf($\mathcal{S}$) $\mapsto \mathbb{R}$, such that*

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

is a probability function defined on $\mathbf{V} = sc(n_i)$.

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

The probability function P for the SPN is $P(\mathbf{x}) = P_r(\mathbf{x})$. The above theorem guarantees that the SPN properly computes a probability distribution and all its marginal probabilities.

The proof of the theorem relies on the completeness and decomposability of the SPN. We explain in the section 2.9 that the theorem would still hold if we replaced decomposability with a weaker condition, consistency [53], but this would complicate the definition of SPN without contributing any advantage in practice.

2.2.3 Selective SPNs

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

When computing $S(\mathbf{x})$ for a given $\mathbf{x} \in$ conf*($\mathcal{S}$), probability flows from the leaves to the root (cf. Def. 10). Equation 2.3 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 $\mathbf{v} \in$ 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 [49]. The formal definition is as follows.

Definition 13. A sum node n_i in an SPN is *selective* if

$$\forall \mathbf{v} \in \text{conf}(\mathcal{S}), \exists j^* \in ch(i) \mid j \in ch(i), j \neq j^* \Rightarrow S_j(\mathbf{v}) = 0 \ . \tag{2.5}$$

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

Definition 14. An SPN is *selective* if all its sum nodes are selective.

Even though this property might seem odd, many SPNs built in practice are selective. (Arithmetic circuits satisfying this property are said to be *deterministic*.)

Example 15. Given the SPN in Figure 2.2.1, we can check that if $\mathbf{v} = (+a, +b, \neg c)$ then $P_2(\mathbf{v}) = 0.36$ and $P_3(\mathbf{v}) = 0$; i.e., Property 2.5 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. However, instead of making these 48 checks, we can apply the following proposition, which offers a sufficient condition for an SPN to be selective.

Definition 16. Let n_i be a sum node having m children and $V \in \text{sc}(n_i)$ a variable with m states. Let σ be a one-to-one function $\sigma : \{1, \ldots, m\} \mapsto ch(i)$. If for every $m \in \{1, \ldots, m\}$ either $\mathbb{I}_{v_j} = n_{\sigma(j)}$ or $\mathbb{I}_{v_j}$ is a child of $n_{\sigma(j)}$, we then say that n_i *represents* variable V.

Example 17. Node n_{14} in Figure 2.2.1 represents variable C, with $\sigma(1) = 17$, $\sigma(2) = 18$, 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 and 16 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.

Proposition 18. *If a sum node n_i represents a model variable V, then n_i is selective.*

2.2.4 Induced trees

Definition 19. Let $\mathcal{S}$ be an SPN and $\mathbf{v} \in \text{conf}(\mathcal{S})$ 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 removing every node n_i such that $S_i(\mathbf{v}) = 0$ (and the corresponding links), every link $n_i \to n_j$ such that $w_{ij} = 0$, and then recursively all the nodes without parents, except the root.

We denote by $S_\mathbf{v}(\mathbf{v})$ the value that $\mathcal{S}_\mathbf{v}$ returns for $\mathbf{v}$. Given that the "induction" of $\mathcal{S}_\mathbf{v}$ only removes the nodes that do not contribute to $S(\mathbf{v})$, we have $S_\mathbf{v}(\mathbf{v}) = S(\mathbf{v}) = P(\mathbf{v})$.

Proposition 20. *If $\mathcal{S}$ is selective, $\mathbf{v} \in$ 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.*

Example 21. Given the SPN in Figure 2.2.1 and $\mathbf{v} = (+a, +b, \neg c)$, $\mathcal{S}_\mathbf{v}$ only contains the links drawn with thick lines in that figure and the nodes connected by them. This graph is a tree because the SPN is selective.

When an SPN is selective the set of trees obtained for all the configurations in conf$(\mathcal{S})$ is similar to the the set induced trees obtained by recursively decomposing the SPN, beginning from the root, as proposed by Zhao et al. [72]. The next proposition is analogous to Theorem 2 in that paper.

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

$$S(\mathbf{v}) = \prod_{(i,j) \in \mathcal{S}_\mathbf{v}} w_{ij} \,, \tag{2.6}$$

where (i, j) denotes a link.

Example 23. For the SPN in Figure 2.2.1, when $\mathbf{v} = (+a, +b, \neg c)$ we have $S(\mathbf{v}) = w_{1,2} \cdot w_{6,8} \cdot w_{14,18} = 0.3 \cdot 0.4 \cdot 0.9 = 0.108$.

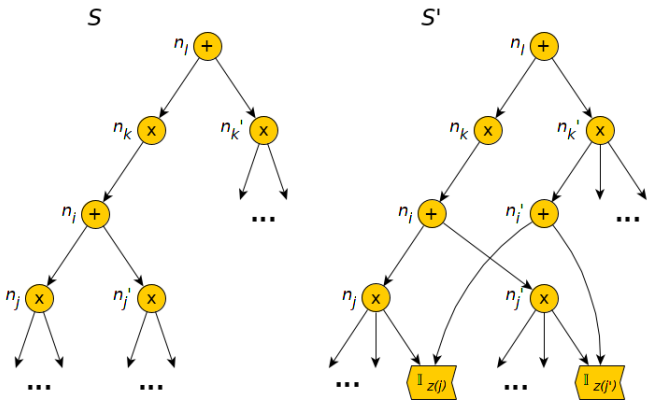

Figure 2.2.2: Augmentation of an SPN, assuming that n_i is not selective in $\mathcal{S}$. This process adds an indicator $\mathbb{I}_{z(j)}$ for every child n_j of n_i. Node $n_{i'}$ is added to restore the completeness of n_l in $\mathcal{S}'$.

2.2.5 Augmented SPN

The augmentation of a non-selective SPN $\mathcal{S}$ [47, 50] consists in adding new variables, new nodes, and new links in order to create a selective SPN $\mathcal{S}'$ that represents the same probability function. It proceeds as follows.

Let n_i be a non-selective sum node in $\mathcal{S}$. In order to make it selective, we define a new finite-states variable, Z. For every child n_j we add a state, $z(j)$, to Z; then if n_j is a product node, we add the indicator $\mathbb{I}_{z(j)}$ as a child of n_j, as shown in Figure 2.2.2; 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, denoted by $\mathcal{S}'$, n_i represents the variable Z and is therefore selective.

However this transformation of the SPN may have caused an undesirable side effect. Let us assume, as shown in Figure 2.2.2, 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 $Z \in \mathrm{sc}(n_i)$, $Z \in \mathrm{sc}(n_k)$, and $Z \in \mathrm{sc}(n_l)$, 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. 2.2.2); the weights for $n_{i'}$ can be chosen arbitrarily provided that they are all non-negative 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 $n_{k''}$, we must repeat the same process as for $n_{k'}$, as well as for the other sum nodes that are ancestors of n_i in $\mathcal{S}$.

After processing in the same way all the non-selective nodes, we obtain $\mathcal{S}'$, the *augmented version* of $\mathcal{S}$.[1] Therefore, $\mathrm{sc}(\mathcal{S}') = \mathrm{sc}(\mathcal{S}) \cup \mathbf{Z}$, where $\mathbf{Z}$ contains one variable for each sum node that was

[1] In Peharz's [47] algorithm, the augmented version of an SPN the process of adding a new variable is performed for every node, even for those that were already selective. In our definition of augmented network, it suffices to add a variable that was not selective in $\mathcal{S}$; therefore the augmentation of a selective SPN does not add any node.

not selective in $\mathcal{S}$.

Proposition 24. *If $\mathcal{S}'$ is the augmented version of $\mathcal{S}$, then $\mathcal{S}'$ is selective and represents the same probability function for sc($\mathcal{S}$), i.e., if $\mathbf{x} \in conf^*(\mathcal{S})$, then $P'(\mathbf{x}) = P(\mathbf{x})$.*

2.2.6 Weighted sums of conditional probabilities. Latent variables

Proposition 25. *Let $\mathcal{S}$ be an SPN whose scope contains at least two variables, such that its root is a sum node that represents variable V. We define $\widetilde{\mathbf{V}} = sc(n_i) \setminus \{V\}$, which is not empty. Then for every $j \in \{1, \ldots, m\}$ $n_{\sigma(j)}$ is a product node, $\mathbb{I}_{v_j}$ is a child of $n_{\sigma(j)}$,*

$$P(v_j) = w_{i,\sigma(j)} \ , \tag{2.7}$$

if $\widetilde{\mathbf{x}} \in conf^*(\widetilde{\mathbf{V}})$ and $w_{i,\sigma(j)} \neq 0$ then

$$P(\widetilde{\mathbf{x}}|v_j) = \prod_{k \in ch(\sigma(j)) \setminus n_k \neq \mathbb{I}_{v_j}} S_k(\mathbf{x}) \ , \tag{2.8}$$

where σ is the function that associates each value v_j with child $n_{\sigma(j)}$—cf. Def. 16. Given that $V \notin sc(n_k)$,

$$P(\widetilde{\mathbf{x}}|v_j) = \prod_{k \in ch(\sigma(j)) \setminus n_k \neq \mathbb{I}_{v_j}} S_k(\widetilde{\mathbf{x}}) \ , \tag{2.9}$$

This proposition is especially interesting when n_r represents a variable V and each child $n_{\sigma(j)}$ only has two children: $\mathbb{I}_{v_j}$ and another node, say $n_{k(j)}$. Then Equation 2.3 can be rewritten as

$$P(\widetilde{\mathbf{x}}) = \sum_j \underbrace{P(\widetilde{\mathbf{x}}|v_j)}_{S_{k(j)}(\widetilde{\mathbf{x}})} \cdot \underbrace{P(v_j)}_{w_{i,\sigma(j)}} . \tag{2.10}$$

In summary, when the root is a sum node and each of the children, $n_{\sigma(j)}$, has two children, $\mathbb{I}_{v_j}$ and $n_{k(j)}$, then $S_{k(j)}(\widetilde{\mathbf{x}})$ represents the conditional probability $P(\widetilde{\mathbf{x}}|v_j)$, where sc$(n_k) = \widetilde{\mathbf{V}}$—which is coherent with Theorem 12—and $w_{i,\sigma(j)} = P(v_j)$.

Example 26. For the SPN in Figure 2.2.1 we have

$$P(a_1) = P(+a) = w_{1,\sigma(1)} = w_{1,2} = 0.3$$
$$P(a_2) = P(\neg a) = w_{1,\sigma(2)} = w_{1,3} = 0.7 \ ;$$

for every $\widetilde{\mathbf{x}} \in \text{conf}^*(\widetilde{\mathbf{V}}) = \{B, C\}$—for example, $(+b, +c)$, $(+b)$, or $(+c)$—

$$P(\widetilde{\mathbf{x}}|+a) = \prod_{k \in ch(2) \setminus n_k \neq \mathbb{I}_{+a}} S_k(\mathbf{x}) = S_6(\widetilde{\mathbf{x}})$$
$$P(\widetilde{\mathbf{x}}|\neg a) = \prod_{k \in ch(3) \setminus n_k \neq \mathbb{I}_{\neg a}} S_k(\mathbf{x}) = S_7(\widetilde{\mathbf{x}})$$

and

$$P(\widetilde{\mathbf{x}}) = \underbrace{P(\widetilde{\mathbf{x}}|+a)}_{S_6(\widetilde{\mathbf{x}})} \cdot \underbrace{P(+a)}_{w_{1,2}} + \underbrace{P(\widetilde{\mathbf{x}}|\neg a)}_{S_7(\widetilde{\mathbf{x}})} \cdot \underbrace{P(\neg a)}_{w_{1,3}} .$$

If every ancestor of a sum node n_i represents a variable, then the above interpretation is still valid for the context defined by the ancestors of n_i, $\mathbf{A}$. In this case Equation 2.11 becomes

$$P(\widetilde{\mathbf{x}}|\mathbf{a}) = \sum_{j=1}^{m} \underbrace{P(\widetilde{\mathbf{x}}|v_j, \mathbf{a})}_{S_{k(j)}(\widetilde{\mathbf{x}})} \cdot \underbrace{P(v_j|\mathbf{a})}_{w_{i,\sigma(j)}} . \tag{2.11}$$

Example 27. Node n_6 in Figure 2.2.1 represents variable B. Its only ancestor sum node, n_1, represents variable A. The path from the root to n_6 defines the scenario $\{A = +a\}$. We have $\mathsf{sc}(n_6) = \{B, C\}$ and $\widetilde{\mathbf{V}} = \mathsf{sc}(n_6) \setminus \{B\} = \{C\}$. In this example Equation 2.11 instantiates into

$$P(c|+a) = \underbrace{P(c|+b,+a)}_{S_{14}(\widetilde{\mathbf{x}})} \cdot \underbrace{P(+b|+a)}_{w_{6,8}} + \underbrace{P(c|\neg b,+a)}_{S_{15}(\widetilde{\mathbf{x}})} \cdot \underbrace{P(\neg b|+a)}_{w_{6,9}} .$$

In this section we have discussed the case in which every sum node represents a variable in $\mathsf{sc}(\mathcal{S})$. In the previous one we showed how the addition of a new variable (its indicators, sometimes auxiliary nodes, and some links) can turn a non-selective node into a selective; in the new SPN, $\mathcal{S}'$, this node represents Z. The same procedure can be applied to any node that did not represent any variable in the original network, $\mathcal{S}$. Given that the two SPNs represent the same probability distribution—for $\mathsf{sc}(\mathcal{S})$—we can say that the variables added, $\mathbf{Z}$, were *latent* in $\mathcal{S}$ and that the augmentation of the SPN has just made them explicit variables (i.e., model variables) in $\mathcal{S}'$ [47, 50].

2.2.7 SPNs defined on continuous variables

It is possible to generalize SPNs by allowing each terminal node to represent a univariate probability density $p(v)$ for a continuous variable, V [50]. In this case,

$$S_i(\mathbf{x}) = \begin{cases} 1 & \text{if } V \notin \mathbf{X} \\ p(\mathbf{x}^{\downarrow V}) & \text{otherwise} , \end{cases} \tag{2.12}$$

where $\mathbf{x} \in \mathsf{conf}^*(\mathcal{S})$. This terminal node plays a similar role to an indicator, $\mathbb{I}_v$, for a finite-states variable V (cf. Eqs. 2.2 and 2.2). In both cases, the scope of the terminal node is $\{V\}$ and $S_i(\mathbf{x})$ is a non-negative real number. The main differences are that a terminal node for a continuous variable is not associated to any particular value of V, and $0 \leq S_i(\mathbf{x}) < +\infty$, while $\mathbb{I}_v(\mathbf{x})$ is either 0 or 1.

The distribution p can be a Gaussian [21, 60], Poisson [40] piecewise polynomial [41], etc.

SPNs can be further generalized by allowing each terminal node to represent a multivariate probability density—for example, a multivariate Gaussian [16, 23].

2.3 Inference

2.3.1 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 links in the SPN. If $\mathbf{X}$ and $\mathbf{E}$ are two disjoint subsets of $\mathbf{V}$, then$P(\mathbf{x} \mid \mathbf{e}) = S(\mathbf{xe})/S(\mathbf{e})$, 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 [6].

2.3.2 MPE inference

The MPE configuration for an SPN is (see Sec. 2.1.5)

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

MPE inference, i.e., finding the MPE for an SPN, is much harder than initially thought, except for selective networks.

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 those trees all 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 $S_i^{\max}(\mathbf{x}) = \max_{j \in ch(i)} w_{ij} \cdot S_j^{\max}(\mathbf{e})$ [instead of Eq. 2.3];
- otherwise $S_i^{\max}$ is computed as in the ordinary case (Eqs. 2.2, 2.4, and 2.12).

Then the algorithm backtracks from the root to the leaves, selecting for each sum node the child that led to $S^{\max}(\mathbf{e})$; if a terminal node $\mathbb{I}_v$ is selected, the value of V in $\mathit{MPE}\,(\mathbf{e})$ is v; if a terminal node for a probability density function $p(v)$ is selected (cf. Sec.2.2.7), then the value of V is $\arg\max_v p(v)$. This way the algorithm returns $\mathit{MPE}\,(\mathbf{e}) \in \mathbf{X}$. This algorithm was proposed by Poon and Domingos [53] and later called Best Tree (BT) in [37].

Peharz [50, Theorem 2] proved that when a network is selective, BT computes the true MPE. However, when a network is not selective, the sub-SPN induced by $\mathbf{xe} \in \mathrm{conf}(\mathcal{S})$ is not necessarily a tree, so the value $S^{\max}(\mathbf{e})$ computed by BT may be different from $\max_{\mathbf{x}} P(\mathbf{x} \,|\, \mathbf{e})$ and, consequently, the configuration returned by BT—which only considers the probability that flows along trees with one child for each sum node—may be different from the true MPE. Therefore, even though the

MPE can be found in time proportional to the size of the graph when the SPN is selective, MPE is NP-complete for general SPNs [47, Theorem 5.3].

2.3.3 MAX and MAP

Exact MAP inference for SPNs is NP-hard because it includes as a particular case MPE (see Sec. 2.1.5), which is NP-complete. Nevertheless, Mei et al. [37] have recently proposed several algorithms that are very efficient in practice. First they presented an algorithm for the MAX problem in general SPNs. Then they proved that every MAP problem for SPNs can be reduced to a MAX problem for a new SPN obtained in linear time. This way they were able to exactly solve MAP problems for SPNs with up to $1,000$ variables and $150,000$ links.

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 top 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.

2.4 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 given a dataset, while in *discriminative* learning the goal is to maximize the conditional likelihood for each value of a variable C, called the *class*.

2.4.1 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, acting as conditioning variables, and by $L_{\mathcal{D}}(\mathbf{w})$ the logarithm of the likelihood, i.e., the probability of the database given $\mathbf{w}$:

$$\begin{aligned} L_{\mathcal{D}}(\mathbf{w}) &= \log P(\mathcal{D}|\mathbf{w}) \\ &= \sum_{t=1}^{T} \log S(\mathbf{v}^t|\mathbf{w}) \ . \end{aligned} \tag{2.1}$$

Therefore the configuration of parameters that maximizes the likelihood is

$$\begin{aligned} \arg\max_{\mathbf{w}} P(\mathcal{D}|\mathbf{w}) &= \arg\max_{\mathbf{w}} L_{\mathcal{D}}(\mathbf{w}) \\ &= \arg\max_{\mathbf{w}} \sum_{t=1}^{T} \log S(\mathbf{v}^t|\mathbf{w}) \ , \end{aligned} \tag{2.2}$$

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

2.4.1.1 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 [49]. In this case, Equations 2.6 and 2.1 imply that

$$\log S(\mathbf{v}^t|\mathbf{w}) = \sum_{(i,j) \in \mathcal{S}_{\mathbf{v}^t}} \log w_{ij} \ , \tag{2.3}$$

and

$$L_{\mathcal{D}}(\mathbf{w}) = \sum_{t=1}^{T} \log S(\mathbf{v}^t|\mathbf{w}) = \sum_{w_{ij} \in \mathbf{W}} n_{ij} \cdot \log w_{ij} \ , \tag{2.4}$$

where n_{ij} is the number of instances in the dataset for which $(i,j) \in \mathcal{S}_{\mathbf{v}^t}$. 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 solution to this problem is:

$$\widehat{w}_{ij} = \frac{n_{ij}}{\sum_{j' \in ch(i)} n_{ij'}} \ . \tag{2.5}$$

There is a special case in which $\sum_{j \in ch(i)} n_{ij} = 0$. This occurs 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. In this case, the weights of this node can be set uniformly:

$$\forall j \in ch(i), \ \widehat{w}_{ij} = \frac{1}{|ch(i)|} \ .$$

Alternatively, it is possible to use a Laplace-like smoothing parameter α in all cases:

$$\widehat{w}_{ij} = \frac{n_{ij} + \alpha}{\sum_{j' \in ch(i)} (n_{ij'} + \alpha)} \ . \tag{2.6}$$

Usually $0 < \alpha \leq 1$.

The n_{ij}'s can be computed by having a counter for every weight. 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 and 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}.

2.4.1.2 Partial derivatives of S

We explain now how to compute the value $S_i^{\partial}(\mathbf{x})$ for every node, which will be used by the GD and EM algorithms described below. It is defined as follows:

$$S_i^{\partial}(\mathbf{x}) = \frac{1}{S(\mathbf{x})} \cdot \frac{\partial S}{\partial S_i}(\mathbf{x}) \ . \tag{2.7}$$

For the root node we have

$$S_r^{\partial}(\mathbf{x}) = \frac{1}{S(\mathbf{x})} \cdot \frac{\partial S}{\partial S_r}(\mathbf{x}) = \frac{1}{S(\mathbf{x})} \ . \tag{2.8}$$

If n_j is not the root,

$$\begin{aligned} S_j^{\partial}(\mathbf{x}) &= \frac{1}{S(\mathbf{x})} \cdot \frac{\partial S}{\partial S_j}(\mathbf{x}) \\ &= \frac{1}{S(\mathbf{x})} \cdot \sum_{i \in pa(j)} \frac{\partial S}{\partial S_i}(\mathbf{x}) \cdot \frac{\partial S_i}{\partial S_j}(\mathbf{x}) \\ &= \sum_{i \in pa(j)} S_i^{\partial}(\mathbf{x}) \cdot \frac{\partial S_i}{\partial S_j}(\mathbf{x}), \end{aligned} \tag{2.9}$$

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

$$\frac{\partial S_i}{\partial S_j}(\mathbf{x}) = w_{ij} \ ; \tag{2.10}$$

if it is a product node,

$$\frac{\partial S_i}{\partial S_j}(\mathbf{x}) = \prod_{j' \in ch(i) \setminus \{j\}} S_{j'}(\mathbf{x}) \ . \tag{2.11}$$

Therefore, after computing the value $S_i(\mathbf{x})$ for every node with an upward pass, the $S_i^{\partial}(\mathbf{x})$ values can be computed by a downward pass—also in linear time—which is similar to backpropagation for neural networks.

2.4.1.3 Gradient descent

Standard gradient descent Gradient descent (GD), a well known optimization method, was proposed for SPNs for both generative and discriminative models in [53] and [20], respectively.[2] In the first case, the algorithm is initialized by assigning an arbitrary value to each parameter, $\widehat{w}_{ij}^{(0)}$ and in every iteration this value is updated as follows, in order to increase the likelihood of the model:

$$\widehat{w}_{ij}^{(s+1)} = \widehat{w}_{ij}^{(s)} + \gamma \frac{\partial L_{\mathcal{D}}(\mathbf{w})}{\partial w_{ij}} \ , \tag{2.12}$$

where γ is the learning rate (a hyperparameter).

Because of the definition of $L_{\mathcal{D}}(\mathbf{w})$,

$$\frac{\partial L_{\mathcal{D}}(\mathbf{w})}{\partial w_{ij}} = \sum_{t=1}^{T} \frac{\partial \log S}{\partial w_{ij}}(\mathbf{v}^t) \ ,$$

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

and

$$\frac{\partial \log S}{\partial w_{ij}}(\mathbf{v}^t) = \frac{1}{S(\mathbf{v}^t)} \cdot \frac{\partial S}{\partial w_{ij}}(\mathbf{v}^t)$$
$$= \frac{1}{S(\mathbf{v}^t)} \cdot \frac{\partial S}{\partial S_i}(\mathbf{v}^t) \cdot \frac{\partial S_i}{\partial w_{ij}}(\mathbf{v}^t) \ ,$$

which together with Equations 2.7 and 2.3 leads to

$$\frac{\partial \log S}{\partial w_{ij}}(\mathbf{v}^t) = S_i^{\partial}(\mathbf{v}^t) \cdot S_j(\mathbf{v}^t) \tag{2.13}$$

and, finally,

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

This equation allows us to perform each iteration of GD in time proportional to the size of the SPN and the number of instances in the database.

Stochastic gradient descent In this version of GD the parameters are updated for each instance of the dataset, i.e.,

$$\widehat{w}_{ij}^{(s+1)} = w_{ij}^{(s)} + \gamma \frac{\partial \log S}{\partial w_{ij}}(\mathbf{v}^t) \tag{2.14}$$

for every instance, until the algorithm converges.

Hard gradient descent One typical problem of GD for deep networks is that it 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. The *hard* version of GD solves this problem by replacing the sum nodes of the SPN with max nodes and reparametrizing the weights so that the gradient of the log-likelihood function remains constant. This method was introduced for SPNs by Gens and Domingos [20] for discriminative learning.

2.4.1.4 Expectation-Maximization (EM)

Standard EM The EM algorithm was designed to estimate the parameters of a statistical model when the dataset is incomplete, i.e., when there are missing values. In particular, it can be used to learn the parameters of $\mathcal{S}'$, the augmented version of $\mathcal{S}$. In this case the dataset is incomplete because it only contains the model variables, i.e., those in $\mathrm{sc}(\mathcal{S})$, not the latent variables $\mathbf{Z}$. Additionally, the dataset may have other missing values. We denote by $\mathbf{H}^t$ the variables missing (hidden) in the t-th instance of the database, i.e., $\mathbf{H}^t = \mathrm{sc}(\mathcal{S}') \setminus \mathbf{V}^t$, with $\mathbf{Z} \subseteq \mathbf{H}^t$.

In this case, if we had a complete database we would be able to estimate the parameters of $\mathcal{S}'$, as in Section 2.4.1.1. Alternatively, if we knew the parameters, we might generate a complete database.

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. In practice, instead of assigning a single value to each missing cell, we create a virtual database 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 complete virtual database to adjust the parameters of the model by MLE, as in Section 2.4.1.1. The two steps are repeated until the parameters (the weights) converge.

The problem is that initially we have neither a complete database 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, a bad choice of the initial values may cause the algorithm to converge at a local maximum of the likelihood, which may be quite different from the global maximum.

The n_{ij} required by the M-step is computed by counting the number of cases in the database for which the link (i, j) belongs to the tree induced by $\mathbf{v}^t\mathbf{h}^t$ (cf. Sec. 2.4.1.1):

$$n_{ij} = \sum_{t=1}^{T} \sum_{\mathbf{h}^t|(i,j)\in\mathcal{S}'_{\mathbf{v}^t\mathbf{h}^t}} P'(\mathbf{h}^t|\mathbf{v}^t) \ . \tag{2.15}$$

We prove in the section 2.9 that

$$n_{ij} = \sum_{t=1}^{T} w_{ij} \cdot S_i^{\partial}(\mathbf{v}^t) \cdot S_j(\mathbf{v}^t) \ . \tag{2.16}$$

The weights can then be computed with Equation 2.5 or 2.6. The time required by each iteration of EM is proportional to the size of the network and the number of instances in the dataset.

Hard EM EM algorithm needs the value of $\partial S/\partial w_{ij}$, which may be very small when the link (n_i, n_j) is in a deep position, i.e., far from the root, which implies that this algorithm may suffer from the vanishing gradients problem in the same way as GD. To avoid it, Poon and Domingos [53] 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, instead of contributing to all of them proportionally.

Hsu et al. [23] proposed a variant of hard EM for SPNs with Gaussian leaves. This method proceeds top-down and decides which child gets the contribution from the sum node. For that, it departs from the top sum node and distributes the instances among its children by maximum likelihood. The next sum nodes receives only the data from their parents and distributes it in the same fashion. This way it updates the weights of the sum nodes locally. The process is similar to the automatic parameter learning in LearnSPN (cf. Sec. 2.5.2). They also provide formulas to update the parameters of Gaussian leaves.

2.4.1.5 Comparison of MLE algorithms

The application of the EM to SPNs has been justified with different mathematical arguments. Peharz [47] exploited the interpretation of the sum nodes in the augmented network as the sum of conditional

probability functions (cf. Sec. 2.2.6). Zhao et al. [72], 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, and in the case of SPNs, CCCP leads to the same algorithm as EM, in spite of being very different in general. Their experiments proved that EM/CCCP converges much faster that the other algorithms, including GD. In turn, Desana and Schnörr [16] 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 [20]. Rashwan et al. [57] addressed this problem with the extended Baum-Welch (EBW) algorithm, which transforms the parameters of the network using a transform 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.

2.4.2 Semi-supervised learning

Trapp et al. [63] 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 the 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 [33], for learning safe semi-supervised SPNs. Their results we competitive with those of purely supervised algorithms.

2.4.3 Approximate Bayesian learning

There are alternative methods for learning the parameters of an SPN based on approximate Bayesian techniques, such as Bayesian moment matching [58] and collapsed variational inference [69], 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 $P(w_{ij} \mid \mathcal{D})$ 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. [58] 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. [26].

In the same vein, Zhao and Gordon [70] presented an optimal linear time algorithm for computing the moments in SPNs with general directed acyclic 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. [69] 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 data of each instance are not independent of the missing data of the 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 maximize some evidence lower bound of the log-likelihood function of the dataset (the "variational inference" step). The experiments showed that the online version of CVB-SPN outperforms oBMM in many datasets.

2.4.4 Deep learning approach

Peharz et al. [52] considered a special class of SPNs, which they called random SPNs, and trained them with automatic differentiation, stochastic GD, and dropout, using GPU parallelization. The resulting model was called RAT-SPN. Its classification accuracy, measured on the MNIST images and other databases, was comparable to that of deep neural networks, with the advantages of being a probabilistic generative model, such as interpretability and robustness to missing features.

2.5 Structural learning

Structural learning consists in finding the optimal (or near-optimal) graph of an SPN. Most of the algorithms for this task require some computation of probabilities during the process.

2.5.1 First structure learners

BuildSPN, by Dennis and Ventura [13] was the first algorithm of this kind. It looks for subsets of highly correlated variables; introduces latent variables to account for those dependences; 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 [53], both designed for image processing, assumed neighborhood dependence. In order to overcome that limitation, Peharz et al. [48] proposed an algorithm that subsequently combines SPNs of few variables into larger ones applying a statistical dependence test.

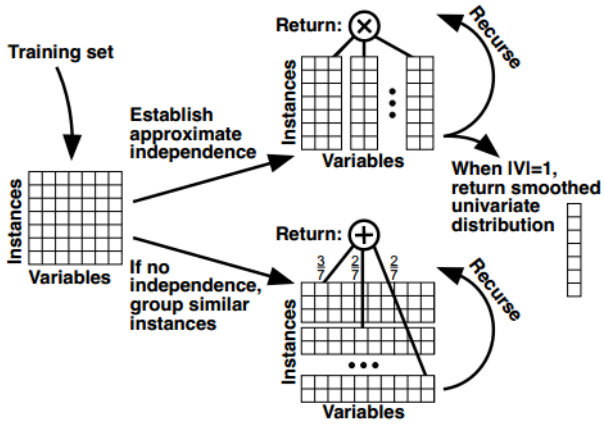

Figure 2.5.1: The LearnSPN algorithm recursively creates a product node when there are subsets of (approximately) independent variables and a sum node otherwise, grouping similar instances. (Reproduced from [21] with the authors' permission).

BuildSPN was also critiqued by Gens and Domingos [21] 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.

2.5.2 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 [21] recursively splits the variables into independent subsets (thus "chopping" the data matrix, as shown in Figure 2.5.1) 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 2.1. 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 2.1) and clustering similar instances (*clusterInstances* in that algorithm). Originally Gens and Domingos [21] chose the G-Test for splitting and hard incremental EM for clustering.

Splitting the variables ("chopping") only considers pair-wise independences. The process departs from a graph containing a node for each variables and no links. It randomly selects one variable

Algoritmo 2.1 LearnSPN$(T, \mathbf{V}, \alpha, m)$

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

and adds an edge to the first other variable deemed dependent by the G-test, then moves to that variable, and iterates until no new variable can be linked to this component of the graph. At the end, if this component has gathered all variables only one component is generated; then the clustering concludes and the algorithm clusters instances instead.

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 $\mathbf{C}_i$. Formally:

$$P(\mathbf{v}) = \sum_i P(\mathbf{c}_i) \prod_j P(v_j|\mathbf{c}_i) .$$

This particular model produces a clustering that can be chopped on the next recursion. This version of LearnSPN forces a clustering in the first step, without trying a split.

2.5.3 ID-SPN and other algorithms

Later, Peharz et al. [49] proposed a structure learner that searches for structures within the space of selective SPNs and showed that it is competitive with LearnSPN.

Rooshenas and Lowd [60] 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 learn arithmetic circuit Markov networks (ACMN) [34], 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 outperforms 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 grid search.

Adel et al. [1] pointed out that previous work had only compared algorithms on binary datasets. They designed SVD-SPN, which proceeds by finding rank-1 matrices. This allows the algorithm to cluster and split at the same time, producing optimal data matrix pieces. It operates recursively, like LearnSPN, but constructing the SPN from the rank-1 submatrices extracted. It also considers a multivariate base case when the variables in the pieces of the data matrix are highly correlated. In this case a sum node is created with as many children as instances in the piece of the matrix; each child is a product node of all the variables in the matrix. In their experiments, the results of SVD-SPN were similar to those of LearnSPN an ID-SPN for binary datasets, but it outperformed them in multiple-category datasets, such as Caltech-101, and is 5 times faster.

2.5.4 Improvements to LearnSPN

Even though LearnSPN is not the best performing algorithm, it is still widely used for its simplicity and modularity [6] and has led to several variants.

2.5.4.1 The algorithm of Vergari et al.

Vergari et al. [66] proposed three modifications to LearnSPN:

1. *Binary splits*. Every split cuts the data matrix into only two pieces. This avoids creating too complex structures at early stages (common 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.

2. *Chow-Liu trees (CLTs) in the leaves*. The naïve Bayes factorization used as base case of LearnSPN (see Algorithm 2.1) can be replaced by the creation of Chow-Liu trees [9], 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 naïve Bayes factorization (which is a particular case of CLT) without adding computational complexity. 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.

3. *Bagging*. "Bagging", a technique used to build random forests, consists in taking several random samples from a dataset, each consisting of several instances, and building a classifier for each

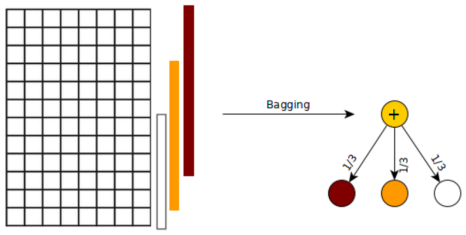

Figure 2.5.2: Bagging in SPNs.

sample. The overall classification can be the average of the outputs of the individual classifiers (in the continuous case) or the mode (in the discrete case). In SPN learning, it extracts—with replacement—n samples of the dataset and produces a sum node with n children, setting every weight to $1/n$, as shown in Figure 2.5.2. Each child represents each individual classifier and the sum node averages the n results. Since the network size would grow exponentially if bagging were applied before every clustering, it is only applied before the first LearnSPN operation, which is a clustering, in order to achieve the widest effect on the resulting structure.

The experiments with these modifications showed that binary splits yield deeper and simpler SPNs and generally reduces the number of edges and parameters. Using Chow-Liu trees attains the same effect and generally increases the likelihood. Bagging also increases the likelihood, specially in datasets with a low number of instances. Using Chow-Liu trees and bagging, LearnSPN achieved the same performance as ID-SPN.

2.5.4.2 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 efficient SPNs, Dennis and Ventura [14] 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 searches greedily 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 [56] 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 higher likelihood for the combined data. This modification of LearnSPN increases the likelihood and reduces the number of parameters of the SPN; additionally, it dramatically increases the learning time for

some datasets. In combinations with bagging, it outperformed other algorithms—including ID-SPN [60]—for high-dimensional datasets.

2.5.4.3 Further improvements to LearnSPN

As mentioned in Section 2.4.1.5, Zhao et al. [72] showed that learning the parameters with the CCCP algorithm improves the performance of LearnSPN.

Di Mauro et al. [18] proposed approximate splitting methods to accelerate LearnSPN, thus trading speed for quality (likelihood).

Butz et al. [6] studied the different combinations of algorithms for LearnSPN. They compared mutual information and the G-test for splitting and with 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 generating CLTs in the leaves.

Liu et al. [32] 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 with standard LearnSPN, LearnSPN with binary splits [66], and LearnSPN with approximate splitting [18], their method achieved better likelihood in 20 binary datasets and generated deeper networks (i.e., more expressive SPNs) while maintaining a reasonable size.

2.5.4.4 LearnSPN with piecewise polynomial distributions

In Section 2.2.7 we mentioned that SPNs can be integrated into SPNs by having leave nodes that represent probability density functions. If a function belongs to a family of probability distributions (Gaussian, Poisson, etc.), its parameters can be estimated with standard statistical techniques. However, there are at least two variants of LearnSPN in which the user does not need to specify a parametric family for each leave representing a numeric variable: they used instead piecewise polynomial distributions. These algorithms can be applied to mixed datasets, i.e., those containing both discrete and continuous data.

Molina et al. [41] proposed an algorithm for learning mixed SPN (MSPNs), whose leaves can represent not only indicators (for finite-states variables) but also piecewise polynomial distributions (for numeric variables). The operations of decomposition (splitting) and conditioning (clustering) are based on the Hirschfeld-Gebelein-Rényi maximum correlation coefficient.

Independently, Bueff et al. [4] developed LearnWMISPN, an algorithm that combines LearnSPN with weighted model integration (WMI) in order to build SPNs whose leave nodes can also represent piecewise polynomial distributions. The order of each polynomial is determined using the Bayesian information criterion (BIC). A preprocessing step transforms finite-states, 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.

2.5.5 Online structural learning

The algorithms presented so far need the entire dataset to produce a structure. However, it may happen that the dataset is so big that the computer has not enough memory to store it at once, or the data arrive constantly (for example, information for product recommendation). 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. [31] 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 [15] extended their SearchSPN algorithm [14] to the online setting. The online version is as fast as the offline version working only on the current batch and the quality of the resulting SPN is the same.

Hsu et al. [23] created oSLRAU, an online structure learner for Gaussian leaves (oSLRAU) which begins with a completely uncorrelated SPN structure that is updated 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. [25] proposed an algorithm, Prometheus, whose first concern is to avoid the correlation threshold parameter, for that instead of creating a product node it creates a mixture of them representing different subset partitions. The way the partitions are created allows them to share subsets, which in the structure is reflected as sharing children, overcoming the restriction to trees on the way. This is in some sense similar to bagging in sum nodes (cf. Sec. 2.5.4) and makes the algorithm robust in low data regimes. 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% of the data.

2.5.6 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*.

With this setting in mind, Melibari et al. [39] proposed dynamic SPNs (DSPNs), which extend SPNs in the same way as dynamic Bayesian networks extend Bayesian network, as well as a local-search structure learner. It generates an initial template SPN and searches for neighboring structures trying to maximize the likelihood. The neighbors come from replacing product nodes. These nodes represent a specific choice of factorization of the variables in its scope; so the algorithm searches over other choices of factorizations and updates the structure if a better one is found. This algorithm outperforms non-dynamic algorithms, such as LearnSPN, and other models, such as dynamic Bayesian networks and recurrent neural networks.

Later, Kalra et al. [27] 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 that of Melibari et al. [39] and hidden Markov models in 5 sequential datasets, and recurrent neural networks in 4 out of those datasets.

2.5.7 Relational data learning

Nath and Domingos [43] introduced relational SPNs (RSPNs), which generalize SPNs by modeling a set of instances jointly, allowing them to influence each other's probability distributions, as well as modeling probabilities of relations between objects. Their LearnRSPN outperformed Markov Logic Networks in both running time and predictive accuracy on three datasets.

2.5.8 Bayesian structure learning

We mentioned in Section 2.2.4 that every SPN can be represented as sum of induced trees [72]. With this idea in mind, Trapp et al. [64] designed a Bayesian non-parametric extension of SPNs based on *infinite sum-product trees* and showed that it yields higher likelihood than infinite Gaussian mixture models.

Later, Trapp et al. have proposed another Bayesian approach that avoids the main criticism that these authors cast over the other learning algorithms: that the global goal of structure learning is not declared—put another way, the lack of a principled criteria for deciding what a "good" structure is. The solution they propose decomposes the problem into two phases: finding a graph and learning the scope-function, ψ, which assigns to each node its scope. The function ψ and the parameters of the model are learned jointly using Gibbs sampling. The Bayesian nature of this approach reduces the risk of overfitting, waives the necessity of a separate validation set to adjust the hyperparameters of the algorithm, and allows the learning SPN structures robustly under missing data.

2.6 Applications

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

2.6.1 Image processing

2.6.1.1 Image reconstruction and classification

Poon and Domingos, in their seminal paper about SPNs [53], 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. Then Gens and Domingos [20] used a different hand-made structure for image classification on the datasets CIFAR-10 and STL-10.

2.6.1.2 Image segmentation

Image segmentation consists in labeling every pixel with the object it belongs to. Yuan et al. [68] 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. [59] have done *medical image segmentation* of OCT scans of retinal tissue. They first built a segmentation model for the health model and for every pathology and then added to the list typical shape variations of the retina tissue for some pathology-specific regions. The resulting SPN selects candidate regions (either healthy or unhealthy) and finds the combination of them that maximizes the likelihood. After a smoothing step a complete segmentation of the retina tissue is obtained, as well as the diagnosis and the affected regions. This method achieved state-of-the-art performance without needing images labeled by pathologies.

2.6.1.3 Activity recognition

Wang and Wang [67] 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 only consider 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 different partitions, while the bottom layers represent the parts included in each sub-image and their relative position using the spatial indicator nodes. In this sense the SPN first learns spatial relations of isolated parts in sub-images and then learns correlations between sub-images. They compared it with other current activity recognition algorithms and found spatial SPNs to be the best among them. Spatial SPNs also allow to discover discriminant pairs of parts for a class.

Amer and Todorovic [2] worked on *activity localization and recognition* in videos. They combined SPNs with a counting grid model that treats a video as a three-dimensional grid (height-width-time) where *visual words* lie; a visual word is meaningful piece of an image, previously extracted with a neural network. Every grid position represents a cube on the multidimensional space and has a histogram of visual words associated. This is called a *bag of words*. To construct the SPN, each bag of words is treated as a variable with two states: foreground and background, which means two indicator nodes per variable. 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. Parameter learning consists in iteratively learning with gradient descent the weights of the SPN from the parameters of the *bag of words* and then variational learning of the parameters of the bag of words from the weights of the SPN. The weakly supervised setting achieved precision of only 1.6 to 3 % worse than the supervised setting. This approach in general achieved better performance than state-of-the-art algorithms on several action-recognition datasets.

2.6.1.4 Robotics

Pronobis et al. [55] designed a probabilistic representation of spatial knowledge called DASH (Deep Spatial Affordance Hierarchy), which encodes several levels of abstractions using a deep model of spatial concepts and can model knowledge gaps and affordances. Knowledge is modeled 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 laser-range sensor.

Zheng et al. [73] designed Graph-Structured Sum-Product Networks (GraphSPNs) for structured prediction. Their algorithm learns template SPNs and makes a mixture over those templates (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, thus surpassing the classical approach based on undirected graphical models (Markov networks).

The authors pointed as future direction that both works can be joined into a *complete robotic*

hierarchical model.

2.6.2 NLP and sequence data analysis

Peharz et al. [51] applied SPNs to *modeling speech* by retrieving the lost frequencies of telephonic communication (artificial bandwidth extension). In this problem tractable and quick (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. [7] used a discriminative SPN [20] which takes vectors with the information of previous words as leaf nodes and computes the probability of the next word. They have obtained empirically better performance than classic methods for language modeling like feedforward neural networks or recurrent neural networks.

Later, Melibari et al. [39] used dynamic SPNs (DSPNs) 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. They compared their approach with five other methods, including HMMs and neural networks with long short-term memory (LSTM). DSPNs were superior in four out of the five datasets examined.

2.6.3 Other applications

Butz et al. [5] used Bayesian networks to recognize independences through 3,500 datasets of *soil bacteria* and combined them into an SPN in order to efficient compute conditional probabilities and the MPE.

Nath and Domingos [44] used relational SPNs for *fault localization*, i.e. finding the most probable location of bugs in computer source code. The networks were trained on a corpus of previously diagnosed buggy programs and learned to identify recurring patterns of bugs. They could also accept clues about bug suspicion from other bug detectors, such as Tarantula.

2.7 Software for SPNs

Every publication about SPNs contains some experiments, and in many cases the source code is publicly available. The web page `https://github.com/arranger1044/awesome-spn` contains many references about SPNs, classified by year and by topic; the section "Resources" includes links 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, written in Python and using TensorFlow as a backend for speeding up some operations. One of them is LibSPN,[3] initiated by Andrzej Pronobis at the University of Washington, Seattle, WA [54]. It implements methods for inference (marginal and conditional probabilities, and approximate MPE), parameter learning (batch and online, with gradient descent and hard EM), and visualization of SPNs. It lacks algorithms for structural learning, but it allows building convolutional SPNs with a layer-oriented interface [65]. 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 fulfilled 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 [55, 65, 73].

The other library is SPFlow[4], whose development is led by Alejandro Molina at the University of Darmstadt, in Germany. Most of the contributors belong to different universities in Germany, Italy, United Kingdom, and Canada [42]. It implements methods for inference (marginal and conditional probabilities, and approximate MPE), parameter learning (with gradient descent) and several structural learning algorithms, can be extended and customized to implement new algorithms. SPNs are usually compiled into TensorFlow for fast computation, but they can also be compiled to C, CUDA, or FPGA code.

There also exist some smaller libraries of interest, such as SumProductNetworks.jl for Julia,[5] which implements inference and parameter learning, and The Libra Toolkit [35],[6] 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 [60].

2.8 Extensions of SPNs

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

Sum-product-max networks (SPMNs) [38] generalize SPNs to the class of decision making problems by including two new types of nodes: max nodes and utility nodes. The solution of these networks provides a decision rule that maximizes the expected utility in linear time.

In credal sum-product networks (CSPNs) [36] the weights of each sum node have not a fixed value, but they can vary in some set (product of probability simplexes) in such a way that each choice of the weights defines an SPN.

[3] https://www.libspn.org.
[4] https://github.com/SPFlow/SPFlow.
[5] https://github.com/trappmartin/SumProductNetworks.jl
[6] http://libra.cs.uoregon.edu

Sum-product graphical models (SPGMs) [17] join the semantics of 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, like usual nodes in graphical models.

Sum-product-quotient networks (SPQNs) [62] introduce quotient nodes, which take two inputs and output their quotient, allowing these models to represent conditional probabilities explicitly.

Tensor SPNs (tSPNs) [28] provides a tensor approach to SPNs. This model has generally many fewer parameters and allows faster inference and a deeper and more narrow neural-network architecture with little loss of modeling accuracy.

Convolutional sum-product networks (ConvSPNs) [65] exploit the inherent structure of spatial data in a similar way to convolutional neural networks by using the sum and product operations of SPNs.

Submodular sum-product networks (SSPNs) [19] are an extension of SPNs for scene understanding in which the weights can be defined by submodular energy functions.

Compositional kernel machines (CKMs) [22] are a model closely related with SPNs which have been successfully applied to image processing tasks, mainly object recognition.

Conditional sum-product networks (CSPNs) [61] extend SPNs to conditional probability distributions. For that, their leaf nodes represent normalized univariate conditional distributions and they include a new type of node, called gating node, which computes a convex combination of the conditional probability of its child nodes with non-fixed weights.

2.9 Proofs

This section contains the proofs of all the propositions and the theorem. Recall that we denote by ⧫ the only configuration of the empty set.

Proof of Proposition 3. Let $\mathbf{x} = \mathbf{v}^{\downarrow \mathbf{X}}$. If $V \in \mathbf{X}$, then $\mathbf{x}^{\downarrow V} = (\mathbf{v}^{\downarrow \mathbf{X}})^{\downarrow V} = \mathbf{v}^{\downarrow V} = v$ and $\mathbb{I}_v(\mathbf{x}) = 1$. If $V \notin \mathbf{X}$, then $\mathbb{I}_v(\mathbf{x}) = 1$. Therefore $\prod_{V \in \mathbf{V}} \mathbb{I}_v(\mathbf{x}) = 1$.

Let $\mathbf{x} \neq \mathbf{v}^{\downarrow \mathbf{X}}$. Then there exists a $V \in \mathbf{X}$ such that $\mathbf{x}^{\downarrow V} \neq (\mathbf{v}^{\downarrow \mathbf{X}})^{\downarrow V} = \mathbf{v}^{\downarrow V} = v$, which implies that $\mathbb{I}_v(\mathbf{x}) = 0$ and $\prod_{V \in \mathbf{V}} \mathbb{I}_v(\mathbf{x}) = 0$. □

Proof of Proposition 6. It is clear that $P(\mathbf{x}) \geq 0$. Because of the definition of $P(\mathbf{x})$,

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

Given that each configuration of $\mathbf{V}$ is compatible with one configuration of $\mathbf{X}$, we have

$$\sum_{\mathbf{x}} P(\mathbf{x}) = \sum_{\mathbf{v}} P(\mathbf{v}) = 1 \, .$$

□

Before proving Propositions 7 and, we introduce a new proposition.

Proposition 28. *Let P be a function $P : conf^*(\mathbf{V}) \mapsto \mathbb{R}$ that satisfies Equation 2.6. If $P(\blacklozenge) = 1$, then $\sum_{\mathbf{v}} P(\mathbf{v}) = 1$.*

Proof. Because of Equation 2.6, with $\mathbf{X} = \varnothing$, we have

$$P(\blacklozenge) = \sum_{\mathbf{v}|\mathbf{v}^{\downarrow\varnothing}=\blacklozenge} P(\mathbf{v}) \ .$$

Taking into account that the empty configuration is compatible with every configuration of every set,

$$P(\blacklozenge) = \sum_{\mathbf{v}} P(\mathbf{v}) \ ,$$

which proves the proposition. □

Proof of Proposition 7. We also have

$$\begin{aligned} P(\mathbf{x}) &= \sum_{j=1}^{n} w_j \cdot P_j(\mathbf{x}) = \sum_{j=1}^{n} w_j \cdot \sum_{\mathbf{v}|\mathbf{v}^{\downarrow\mathbf{X}}=\mathbf{x}} P_j(\mathbf{v}) \\ &= \sum_{\mathbf{v}|\mathbf{v}^{\downarrow\mathbf{X}}=\mathbf{x}} \sum_{j=1}^{n} w_j \cdot P_j(\mathbf{v}) = \sum_{\mathbf{v}|\mathbf{v}^{\downarrow\mathbf{X}}=\mathbf{x}} P(\mathbf{v}) \ , \end{aligned}$$

which proves that P satisfies Equation 2.6. It is clear that $P(\mathbf{v}) \geq 0$ for all $\mathbf{v} \in \text{conf}(\mathbf{V})$ and, because of Proposition 28,

$$\sum_{\mathbf{v}} P(\mathbf{v}) = P(\blacklozenge) = \sum_{j=1}^{n} w_j \cdot P_j(\blacklozenge) = \sum_{j=1}^{n} w_j = 1 \ ,$$

which completes the proof. □

Proof of Proposition 8. When $n = 1$ the proof is trivial because $P = P_1$. When $n = 2$ we have, for every configuration $\mathbf{x}$, with $\mathbf{X} \subseteq \mathbf{V}$,

$$\sum_{\mathbf{v}|\mathbf{v}^{\downarrow\mathbf{X}}=\mathbf{x}} P(\mathbf{v}) = \sum_{\mathbf{v}|\mathbf{v}^{\downarrow\mathbf{X}}=\mathbf{x}} P_1(\mathbf{v}^{\downarrow\mathbf{V}_1}) \cdot P_2(\mathbf{v}^{\downarrow\mathbf{V}_2}) \ .$$

Given that $\mathbf{V}_1 \cup \mathbf{V}_2 = \mathbf{V}$ and $\mathbf{V}_1 \cap \mathbf{V}_2 = \varnothing$, every configuration of $\mathbf{V}$ can be obtained from a combination of two configurations, $\mathbf{v} = \mathbf{v}_1\mathbf{v}_2$, where$\mathbf{v}_1 = \mathbf{v}^{\downarrow\mathbf{V}_1}$ and $\mathbf{v}_2 = \mathbf{v}^{\downarrow\mathbf{V}_1}$. The condition$\mathbf{v}^{\downarrow\mathbf{X}} = \mathbf{x}$—the compatibility of $\mathbf{x}$ with $\mathbf{v}$—can be decomposed into two conditions: $\mathbf{v}_1^{\downarrow\mathbf{X}_1} = \mathbf{x}^{\downarrow\mathbf{X}_1}$ and $\mathbf{v}_2^{\downarrow\mathbf{X}_2} = \mathbf{x}^{\downarrow\mathbf{X}_2}$, where $\mathbf{X}_1 = \mathbf{V}_1 \cap \mathbf{X}$ and $\mathbf{X}_2 = \mathbf{V}_2 \cap \mathbf{X}$. Therefore,

$$\sum_{\mathbf{v}|\mathbf{v}^{\downarrow\mathbf{X}}=\mathbf{x}} P(\mathbf{v}) = \sum_{\mathbf{v}_1|\mathbf{v}^{\downarrow\mathbf{X}_1}=\mathbf{x}^{\downarrow\mathbf{X}_1}} \sum_{\mathbf{v}_2|\mathbf{v}^{\downarrow\mathbf{X}_2}=\mathbf{x}^{\downarrow\mathbf{X}_2}} P_1(\mathbf{v}_1) \cdot P_2(\mathbf{v}_2) \ .$$

The property $\mathbf{V}_1 \cap \mathbf{V}_2 = \varnothing$ also implies that $P_1(\mathbf{v}_1)$ does not depend on $\mathbf{v}_2$ and vice versa, so

$$\sum_{\mathbf{v}|\mathbf{v}^{\downarrow\mathbf{X}}=\mathbf{x}} P(\mathbf{v}) = \left(\sum_{\mathbf{v}_1|\mathbf{v}_1=\mathbf{x}^{\downarrow\mathbf{X}_1}} P_1(\mathbf{v}_1) \right) \cdot \left(\sum_{\mathbf{v}_2|\mathbf{v}_2=\mathbf{x}^{\downarrow\mathbf{X}_2}} P_2(\mathbf{v}_2) \right) .$$

The fact that P_1 is a probability function implies that

$$P_1(\mathbf{x}^{\downarrow\mathbf{X}_1}) = \sum_{\mathbf{v}_1|\mathbf{v}_1=\mathbf{x}^{\downarrow\mathbf{X}_1}} P_1(\mathbf{v}_1)$$

and the definition $\mathbf{X}_1 = \mathbf{V}_1 \cap \mathbf{X}$ implies that $\mathbf{x}^{\downarrow\mathbf{X}_1} = \mathbf{x}^{\downarrow\mathbf{V}_1}$. Therefore

$$\sum_{\mathbf{v}|\mathbf{v}^{\downarrow\mathbf{X}}=\mathbf{x}} P(\mathbf{v}) = P_1(\mathbf{x}^{\downarrow\mathbf{V}_1}) \cdot P_2(\mathbf{x}^{\downarrow\mathbf{V}_2}) = P(\mathbf{x}) ,$$

which proves that P satisfies Equation 2.6. It is clear that $P(\mathbf{v}) \geq 0$ for all $\mathbf{v} \in \text{conf}(\mathbf{V})$ and, because of Proposition 28,

$$\sum_{\mathbf{v}} P(\mathbf{v}) = P(\blacklozenge) = P_1(\blacklozenge) \cdot P_2(\blacklozenge) = 1 ,$$

which completes the proof for $n = 2$. As a consequence, if $P_1(\mathbf{x}^{\downarrow\mathbf{V}_1}) \cdot \ldots \cdot P_{n-1}(\mathbf{x}^{\downarrow\mathbf{V}_{n-1}})$ and $P_n(\mathbf{x}^{\downarrow\mathbf{V}_n})$ are probability functions for disjoint $\mathbf{V}_j$'s, then $P_1(\mathbf{x}^{\downarrow\mathbf{V}_1}) \cdot \ldots \cdot P_n(\mathbf{x}^{\downarrow\mathbf{V}_n})$ is also a probability function, which proves Proposition 8 for any value of n. □

Please note that if two probability functions are defined over non-disjoint sets of variables, their product is not necessarily a probability function. Consider the following counterexample: $\mathbf{V}_1 = \mathbf{V}_2 = \{V\}$, $P_1(+v) = P_1(\neg v) = P_2(+v) = P_2(\neg v) = 0.5$. Then $P(+v) = P_1(+v) \cdot P_2(+v) = 0.25$, $P(\neg v) = 0.25$, and $P(+v) + P(\neg v) \neq 1$.

Proof of Proposition 9. Let n_j and $n_{j'}$ be two different children of n_i. If $\mathit{desc}(n_j) \cap \mathit{desc}(n_{j'}) = \varnothing$, then $\mathit{sc}(n_j) \cap \mathit{sc}(n_{j'}) = \varnothing$, i.e., their scopes are disjoint.

Reciprocally, if a node n_k is a descendant of both n_j and $n_{j'}$ then $\mathit{sc}(n_k) \subseteq \mathit{sc}(n_j) \cap \mathit{sc}(n_{j'})$, which implies that the scopes are not disjoint because, by the definition of scope, $\mathit{sc}(n_k) \neq \varnothing$. □

Proof of Theorem 12. Let n_i be a terminal node, i.e., the indicator associated to value v_0 of a finite-states variable V; it is then the indicator $\mathbb{I}_{v_0}$. Then $\text{sc}(n_i) = \{V\}$, $P_i(v) = S_i(v) = \mathbb{I}_{v_0}(v)$ and $P_i(\blacklozenge) = S_i(\blacklozenge) = 1$. P_i is a probability distribution defined on $\text{sc}(n_i) = \{V\}$ because $P_i(v) = S_i(v) = \mathbb{I}_{v_0}(v) \geq 0$ for every value v and $\sum_v P_i(v) = \sum_v \mathbb{I}_{v_0}(v) = \mathbb{I}_{v_0}(v_0) = 1$. When $\mathbf{X} \subset \mathbf{V}$ then $\mathbf{X} = \varnothing$; Equation 2.6 holds because $P_i(\blacklozenge) = S_i(\blacklozenge) = 1$ and

$$\sum_{\mathbf{v}|\mathbf{v}^{\downarrow\varnothing}=\blacklozenge} P_i(\mathbf{v}) = \sum_v P_i(v) = P_i(v_0) = 1 .$$

Therefore P_i is a probability function in this case.

Let n_i be a non-terminal node. We assume that P_j is a probability distribution for each of its children, n_j. If n_i is a sum node then P_i is a probability function because of Proposition 7, with $\mathbf{V} = \text{sc}(n_i) = \text{sc}(n_j)$. The completeness of the SPN guarantees that n_i and all its children have the same scope. If n_i is a product node then P_i is a probability function because of Proposition 8, with $\mathbf{V}_j = \text{sc}(n_j)$. The decomposability of the SPN guarantees that the $\mathbf{V}_j$'s are disjoint and the definition of $\text{sc}(n_i)$ ensures that $\bigcup_j \mathbf{V}_j = \mathbf{V} = \text{sc}(n_i)$. □

Proof of Proposition 18. Let $\mathbf{v} \in \text{conf}(\mathcal{S})$. There is an integer j^* such that $v_{j^*} = \mathbf{v}^{\downarrow V}$. Let $j \in ch(i)$, $j \neq j^*$. If $\mathbb{I}_{v_j} = n_j$ then $S_j(\mathbf{v}) = \mathbb{I}_{v_j}(\mathbf{v}) = 0$. If $\mathbb{I}_{v_j}$ is a child of n_j, then n_j is a product node and the contributions of the other children of n_j are multiplied by 0, which implies that $S_j(\mathbf{v}) = 0$. □

Proof of Proposition 20. Selectivity implies that if n_i is a sum node in $\mathcal{S}$ and $S_i(\mathbf{v}) \neq 0$, then it has exactly one child n_j such that $S_j(\mathbf{v}) \neq 0$ and $w_{ij} > 0$ (which implies that this link has not been removed), so in $\mathcal{S}_\mathbf{v}$ every sum node n_i has exactly one child. Every node in $S_i(\mathbf{v})$ other than the root has at least one parent—otherwise, it would have been removed. A node cannot have more than two or more parents because, being in a rooted graph, they should have a common ancestor having two or more children, but this is impossible because each sum node has exactly one child and two different product nodes cannot have a common descendant (cf. Proposition 9). □

Proof of Proposition 22. We begin by considering the case in which $\mathcal{S}$ only has one terminal node, $n_r = \mathbb{I}_v$. The scope of $\mathcal{S}$ is $\{V\}$. If $S(\mathbf{v}) \neq 0$, there are only two possibilities: either $\mathbf{v} = \blacklozenge$ or $\mathbf{v} = (v)$. In both cases, $S(\mathbf{v}) = 1$. Given that $\mathcal{S}_\mathbf{v}$ contains one node and no links, the right hand side of Equation 2.6 is 1, so that equation holds. Let us assume that the proposition holds for all the descendants of a node n_k. If $S_k(\mathbf{v}) \neq 0$, Equation 2.6 holds when n_k is a sum node (with only one child, because of Proposition 20) as well as when it is a product node. □

Proof of Proposition 24. Every node n_i that was not selective has been made selective by the addition of variable Z and its indicators. If a node $n_{i'}$ was added to restore the completeness of the SPN, it is also selective because all its children, being indicators of Z, have the same scope (see Fig. 2.2.2). If $\mathbf{x} \in \text{conf}^*(\mathcal{S})$, then $\mathbb{I}_z(\mathbf{x}) = 1$ because $V \notin \mathbf{X}$ and $S'_{i'}(\mathbf{x}) = 1$, which implies that every node existing in $\mathcal{S}$ has the same value in $\mathcal{S}'$. □

Proof of Proposition 25. Let $j \in \{1, \ldots m\}$. Since n_r represents variable V, either $\mathbb{I}_{v_j} = n_{\sigma(j)}$ or $\mathbb{I}_{v_j}$ is a child of $n_{\sigma(j)}$—see Definition 16. In the first case, we would have $\text{sc}(n_{\sigma(j)}) = \{V\}$ and the completeness of $\mathcal{S}$ would imply that $\text{sc}(n_r) = \text{sc}(n_r) = \{V\}$, in contradiction with the assumption that $\text{sc}(\mathcal{S})$ has at least to variables. Therefore $\mathbb{I}_{v_j}$ must be a child of $n_{\sigma(j)}$, a product node, and

$$S_{\sigma(j)}(\mathbf{x}) = \mathbb{I}_{v_j}(\mathbf{x}) \cdot \prod_{k \in ch(\sigma(j)) \setminus n_k \neq \mathbb{I}_{v_j}} S_k(\mathbf{x}) \ ,$$

where $\mathbf{x} \in \text{conf}^*(\mathcal{S})$. Let $\mathbf{x}$ be the composition of v_j and any $\widetilde{\mathbf{x}} \in \text{conf}^*(\widetilde{\mathbf{V}})$. Then

$$\begin{aligned} S_{\sigma(j)}(v_j\widetilde{\mathbf{x}}) &= \mathbb{I}_{v_j}(v_j) \cdot \prod_{k \in ch(\sigma(j)) \setminus n_k \neq \mathbb{I}_{v_j}} S_k(\widetilde{\mathbf{x}}) \ , \\ &= \prod_{k \in ch(\sigma(j)) \setminus n_k \neq \mathbb{I}_{v_j}} S_k(\widetilde{\mathbf{x}}) \ . \end{aligned}$$

In contrast, when $j' \neq j$ we have $\mathbb{I}_{v'_j}(v_j) = 0$ and $S_{\sigma(j')}(v_j\widetilde{\mathbf{x}})$. Since n_r is a sum node,

$$\begin{aligned} P(v_j\widetilde{\mathbf{x}}) &= \sum_{j'=1}^{m} w_{i,\sigma(j')} \cdot S_{\sigma(j')}(v_j\widetilde{\mathbf{x}}) \\ &= w_{i,\sigma(j)} \cdot \prod_{k \in ch(\sigma(j)) \setminus n_k \neq \mathbb{I}_{v_j}} S_k(\widetilde{\mathbf{x}}) \ . \end{aligned}$$

In particular, if $\widetilde{\mathbf{x}} = \blacklozenge$ then $S_k(\blacklozenge) = 1$ for every k and

$$P(v_j) = w_{i,\sigma(j)} \ .$$

If $w_{i,\sigma(j)} \neq 0$,

$$P(v_j|\widetilde{\mathbf{x}}) = \frac{P(v_j\widetilde{\mathbf{x}})}{P(v_j)} = \prod_{k \in ch(\sigma(j)) \setminus n_k \neq \mathbb{I}_{v_j}} S_k(\widetilde{\mathbf{x}}) \ .$$

□

Now, before proving Equation 2.16, we introduce an auxiliary proposition.

Proposition 29. *If $\mathcal{S}$ is selective, $\mathbf{v} \in$ conf$(\mathcal{S})$, $S(\mathbf{v}) \neq 0$, and $(i,j) \in \mathcal{S}_{\mathbf{v}}$, then*

$$S(\mathbf{v}) = w_{ij} \cdot \frac{\partial S(\mathbf{v})}{\partial w_{ij}} \ . \tag{2.1}$$

Proof. Equation 2.6 implies $w_{ij} \neq 0$ (because $S(\mathbf{v}) \neq 0$) and

$$\frac{\partial S(\mathbf{v})}{\partial w_{ij}} = \frac{S(\mathbf{v})}{w_{ij}} \ .$$

□

Proof of Equation 2.16. Given that $\mathbf{v}^t\mathbf{h}^t$ is a complete configuration of sc$(\mathcal{S}')$, i.e., $\mathbf{v}^t\mathbf{h}^t \in$ conf$(\mathcal{S}')$, and $\mathcal{S}'$ is selective, Proposition 29 implies that

$$S'(\mathbf{v}^t\mathbf{h}^t) = w_{ij} \cdot \frac{\partial S'(\mathbf{v}^t\mathbf{h}^t)}{\partial w_{ij}}$$

and

$$P'(\mathbf{h}^t|\mathbf{v}^t) = \frac{S'(\mathbf{v}^t\mathbf{h}^t)}{S'(\mathbf{v}^t)} = w_{ij} \cdot \frac{1}{S'(\mathbf{v}^t)} \cdot \frac{\partial S'(\mathbf{v}^t\mathbf{h}^t)}{\partial w_{ij}} \ .$$

So Equation 2.15 can be rewritten as

$$n_{ij} = \sum_{t=1}^{T} w_{ij} \cdot \frac{1}{S'(\mathbf{v}^t)} \cdot \sum_{\mathbf{h}^t | (i,j) \in \mathcal{S}'_{\mathbf{v}^t \mathbf{h}^t}} \frac{\partial S'(\mathbf{v}^t \mathbf{h}^t)}{\partial w_{ij}} \, .$$

If link (i, j) does not belong to the tree induced by $\mathbf{v}^t\mathbf{h}^t$ then $\partial S'(\mathbf{v}^t\mathbf{h}^t)/\partial w_{ij} = 0$, so the inner summation in the previous expression can be extend to all the configurations of $\mathbf{H}^t$:

$$n_{ij} = \sum_{t=1}^{T} w_{ij} \cdot \frac{1}{S'(\mathbf{v}^t)} \cdot \sum_{\mathbf{h}^t} \frac{\partial S'(\mathbf{v}^t \mathbf{h}^t)}{\partial w_{ij}} \, .$$

Given that $S'(\mathbf{v}^t) = \sum_{\mathbf{h}^t} S'(\mathbf{v}^t\mathbf{h}^t)$, we have

$$n_{ij} = \sum_{t=1}^{T} w_{ij} \cdot \frac{1}{S'(\mathbf{v}^t)} \cdot \frac{\partial S'(\mathbf{v}^t)}{\partial w_{ij}} \, .$$

Because of Proposition 24, $S'(\mathbf{v}^t) = S(\mathbf{v}^t)$ and

$$\begin{aligned} n_{ij} &= \sum_{t=1}^{T} w_{ij} \cdot \frac{1}{S(\mathbf{v}^t)} \cdot \frac{\partial S(\mathbf{v}^t)}{\partial w_{ij}} \\ &= \sum_{t=1}^{T} w_{ij} \cdot \frac{1}{S(\mathbf{v}^t)} \cdot \frac{\partial S(\mathbf{v}^t)}{\partial S_i} \cdot \frac{\partial S_i(\mathbf{v}^t)}{\partial w_{ij}} \, . \end{aligned}$$

This result, together with Equations 2.7 and 2.3, leads to Equation 2.16. □

Chapter 3

Image classification with convolutional SPNs

Convolutional SPNs (ConvSPNs) are an extension of SPNs (cf. Section 2.8) that translate the convolutional layers of the neural networks into an SPN valid structure. The application proposed here aims to extend the work of Wolfshaar et al. [65] to different image classification problems and compare its performance with the model from which they are born: convolutional neural networks (CNNs). We chose to work with ConvSPNs since they have shown better performance than any other pure SPN approach to image classification tasks [65]. In the next section we provide a background of both CNNs and ConvSPNs.

3.1 Background

3.1.1 Convolutional neural networks

The challenges of artificial vision gave rise to a special class of neural network inspired in the processing that happens at the visual cortex: CNNs. The biologic inspiration is found in Hubel and Wiesel's work [24] and is based on primate visual cortex structure. CNNs first came to the scene in 1989 in the work of LeCuN to process grid-like topological data (image and time series) [30]. CNNs are currently one of the best models for understanding images and have obtained state-of-the-art results on image detection, recognition, segmentation and retrieval [10]. At present, most of the front-runners of image processing competitions are employing deep CNN-based models.

In a structural sense, CNNs are neural networks that leverage through special layers the fact that the input is an image. The base intuition of deep learning algorithms is to solve machine learning tasks by constructing complex features from simpler features in a layered fashion. CNNs do the same using image subsections as features, and solve image tasks by constructing complex subsections (such as faces or doors) from simpler and smaller ones (such as edges, corners or color gradients). This is done through **local connectivity**, which will be explained below.

The special layers that provide convolutional networks with this power are **convolutional layers** and **pooling layers**.

3.1.1.1 Convolutional layers

The intuition of convolution is a moving window passing over an image and computing the overlap between itself and the image at each point. This is done by element-wise multiplication of two matrices. In CNNs, the moving window is called **filter** and the area it passes over is the image. Both of them are represented by three-dimensional matrices, being the dimensions width, height and channels. The channels are used to encode the color of a pixel into numbers. For example, the famous RGB encoding stands for three channels: red, green and blue. The transparency of an image is also a channel, commonly called "alpha channel". So the dimension of a full HD color image is 1920x1080x3. The filter is a matrix with typically smaller width and height but the same number of channels as the image it passes over. When the filter is applied to a section of the image (the **receptive field**), it outputs a value. Then it is applied to adjacent sections of the image and the outputs of every application can be ordered into a new image, see Figure 3.1.1. It is called filter since it outputs a new image from the initial one, like filters of image edition programs. The value the filter outputs says how much the filter is activated on each image subsection. Because of this, the image generated by the filter is called **activation map** since it measures spatially how the filter is activated.

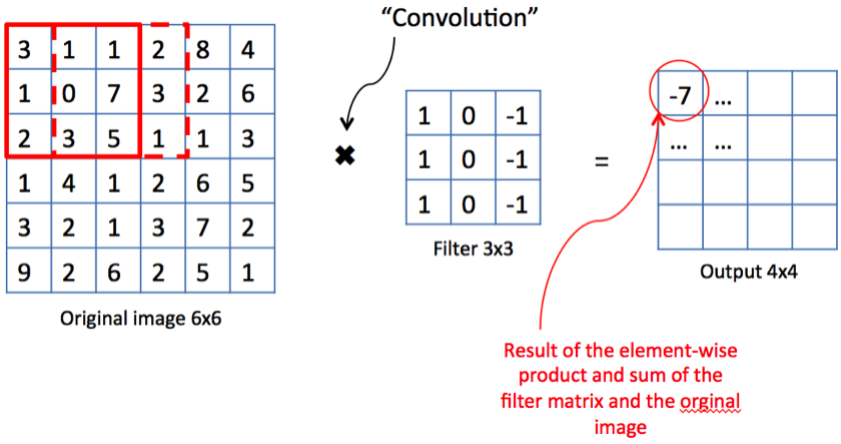

Figure 3.1.1: A convolutional application of a filter.

Since the same filter passes over many different places of the image, it can find a discriminative feature independently of its position in the image. As an example, if a filter is looking for the shape of an ear, it would not mind where that ear is on the image. This is the **translation invariance**

property of CNNs and it is achieved through **weight sharing.** The numbers in the filter matrix are the weights of the filter. An application of a filter is represented by a neuron (with the receptive field as input and the result of the convolution as output) so all the applications of the same filter should share weights. The fact that a neuron is connected only to a specific subsection of neurons (the ones representing the receptive field) and not the whole previous layer is called **local connectivity.**

Since a filter outputs only one number per application, the output image of a filter has only one channel. However, it is common for convolutional layers to apply several filters to its input. In this case each filter output represents a channel and the complete output has as many channels as filters in the previous convolutional layer. Channels in the initial input image represent a specific encoding of the image information, while in the subsequent convolutional layers they represent the number of filters applied by that layer.

Note in Figure 3.1.1 that the output is 2 pixels shorter and 2 pixels more narrow. This happens because the filter cannot be applied to the corners of the image. Sometimes that reduction is not desirable. To solve that, **padding** is added to the borders. The most usual padding is adding *filter size - 1* pixels to every border so that the output image preserves the size of the input image.

3.1.1.2 Pooling layers

The other most important operation is **pooling**. Pooling is basically down-sampling. As an example, an image of 200x200 pixels is reduced to 100x100. To achieve that, one possibility is to take every 2x2 square and set a 1x1 value. The criterion for choosing the value defines the type of pooling. Max pooling chooses the maximum value and is the most widely used criterion. Average pooling takes the average of the pooled values. Pooling layers comes from the intuition that once we know a specific feature is in the image (the filter outputs a high value) only its location relative to other high-valued features is important. Size reduction, therefore, does not affect the result.

A convolutional neural network consists typically of alternating convolution and pooling layers.

3.1.1.3 Additional concepts

There are some extra concepts which are important for CNNs and ConvSPN. One of them is the **stride**, the stride is the distance between filter applications. In Figure 3.1.1 the stride used is 1. A pixel in the center will contribute to nine filter outputs, each time in a different relative position with respect to the filter. If the goal is to have each pixel contributing to one and only one output in Figure 3.1.1 four filter applications are needed. This means the filter will move three pixels from one application to the next so the stride will be 3. However, stride is rarely increased since it returns fairly less information. In the example proposed, a stride of 3 will return a 2x2 image instead of the 4x4 one showed in Figure 3.1.1.

The other relevant concept is **dilation**. Dilation can be used to cover a larger receptive field without enlarging the filter. This is showed in Figure 3.1.2.

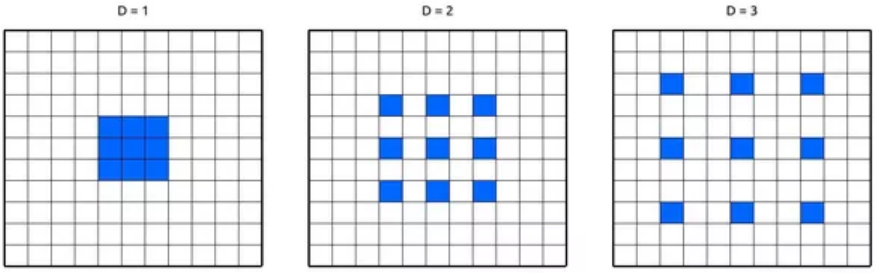

Figure 3.1.2: Dilation in CNNs. An example showing the different receptive fields for dilation values of 1, 2 and 3.

A dilation of 2 will cover an area of 5x5 with a filter of 3x3 by setting every pixel-wise application two pixels away from its neighbors.

A convolutional neural network also includes the typical layers and operations of general neural networks. In particular, the **dropout** operation and the **batch normalization** operation. The dropout operation goes through the neurons of a fully-connected or dense layer and chooses to remove each neuron with a probability called **dropout rate**. This is a regularization technique that decreases overfitting. The batch normalization subtracts the mean of each batch from each instance in that batch and then divides it by the variance of the batch. This speeds up learning.

3.1.2 Convolutional sum-product networks

Convolutional sum-product networks (ConvSPNs) aim to encode the spatial properties of CNNs in SPN architectures. Those properties are **translation invariance** through weight sharing and **local connectivity**, which provides the layered organization of the visual information typical of convolutional deep learning.

In a ConvSPN, an SPN **node** represents a pixel in one channel, while a **cell** refers to a pixel in all its channels, a pixel itself. Nodes of the same cell have the same scope, i.e., leaf nodes of the same cell represent the same variable or variables. This means that the scope does not change when the channel is different but it does change when we move to another pixel either in the vertical or the horizontal axis.

The objective of ConvSPNs is to translate the convolution operation to SPN sum and product nodes satisfying completeness and decomposability. This produces three types of layers: *local sum layers, convolutional product layers,* and *depth-wise convolutional depth layers*.

For sum layers to satisfy completeness the children of a sum node should come from the same pixel but from different channels. Taking nodes from different pixels will mean having children with different scopes, which rends the sum node incomplete. In order to avoid it, sum nodes are chosen to take as children every node in the pixel cell. This is like a convolutional layer with a 1x1 filter, which we will call a *local sum layer*. Weight sharing can be optionally used here.

A convolution operation is a weighted sum in CNNs. For product layers to compute a weighted sum they are made to propagate log-probabilities. Using logarithnms, the products are converted into sums and a product node will be equivalent to a weighted sum with weights being either one or zero. Since a product now represents a convolution, a product node represents an application of one filter. That means that a filter is represented by height$*$width product nodes. To satisfy decomposability, a product node can only have as children one node per pixel (i.e. one channel per pixel). The number of possible product nodes accounting for that restriction are height$*$width$*$(pixels in filter)$^{\text{input channels}}$. Since a filter is represented by height$*$width products, the number of filters (and the name of output channels) of a product layer is (pixels in filter)$^{\text{input channels}}$. We want to use the higher number of products to give the model the maximum capacity, but that will make channels grow exponentially layer after layer, demanding a lot of computational resources. The authors of ConvSPNs then propose two different types of convolutional product layers: *normal convolutional product layers,* which use the maximum number of products, and *depth-wise convolutional product layers,* with as many input channels as output channels. This means height$*$width$*$input channels product nodes in the layer. The first one is typically used only with few input channels.

At this point, we have three different types of layers. Now, we should be careful when stacking layers. Sum layers do not change the scope of the output pixels with respect to the scope of the input pixels, so they can be stacked without additional considerations, but product layers do, so we should take into account decomposability when stacking layers.

The authors here proposed two stacking processes [65] for product layers:

1. *Non-overlapping product layers.* This stacking process consists on setting the stride of product layers as big as the filter size. This way, in any output image every pixel will have a unique scope, so the next filters will not be able to select pixels with shared scope, and decomposability will be preserved. This method will drastically reduce the size of the input, since width and height will be divided by the filter size. For that reason another stacking process is proposed.

2. *Wicker layers.* The wicker stacking process is created to allow a stride of 1. Using a stride of 1 means adjacent output pixels will share part of their scope. The next layer should select a further pixel to preserve decomposability. This is achieved with a exponentially growing dilation rate. This dilation selects the nearest nodes that do not overlap in scope.

In any process, layers will be stacked until all product nodes have all variables in its scope. The last product layer stacked may need special padding to guarantee that.

These two architectures are actually the two extremes of several possibilities. From the wicker architecture with stride 1 and dilation growing exponentially, the stride can be increased and the dilation changed accordingly. This can be done for every possible stride between 1 and the filter size, and then the dilation will be 1 every layer and we are in the case of non-overlapping products.

Also, mixed structures beginning with non-overlapping layers and changing to wicker layers anytime are also valid. An example of a structure of this type is represented in Figure 3.1.3. However,

a non-overlapping layer cannot follow a wicker layer since the array of scopes produced by a wicker layer will produce undecomposable products in the next non-overlapping layer.

The final architecture of a ConvSPN will consist of alternating sum and product layers until all nodes in a product layer have the same scope, then a layer of sums with as many sums as classes to classify and a final sum root which will represent a latent variable that selects the class.

While the number of layers is a free parameter in CNNs, in ConvSPNs there is far less flexibility. The number of product layers in a complete wicker architecture is the whole part of the logarithm in base $kernel\,size$ of the maximum of the width and the height. The total number of layers is the number of product layers, plus one sum layer for every product layer plus the root. For example, an image of 28x28 processed with a filter size of 2 in every product layer will have $\lfloor \log_2 28 \rfloor = 5$ product layers and a total of 11 layers. Changing from wicker layers to non-overlapping layers does not modify the number of total layers of the net.

To sum up:

- There are three type of layers: *local sum layers, convolutional product layers,* and *depth-wise convolutional depth layers.*
- A ConvSPN structure alternates sums and products, beginning preferably with a product layer.
- The number of layers is fixed given the image resolution and the filter size.
- The first product layer can be a non-overlapping layer, a half-wicker, or a wicker layer.
- Once a wicker or half-wicker layer has been added, the next product layers can only be wicker layers.
- When every product of a product layer has the full scope, any structure is completed by stacking a last sum layer with as many sums as classes and the a sum node as root.

3.2 Methods

The goal of these experiments is to apply ConvSPNs to several datasets for image classification and compare their results with the state of the art algorithm on the area: CNNs.

The chosen datasets aim to cover different problems ranging in difficulty, data availability, number of classes and type of image. Here I present a summary of the datasets:

- Infected malaria cells discrimination. This dataset presents a medical problem with a sufficient amount of data and the most simple classification: infected or non-infected. Extracted from Kaggle.
- Intel scenery classification. This dataset proposes a classification of different types of places under 6 categories: buildings, forest, glacier, mountain, sea and street. Extracted from Kaggle.

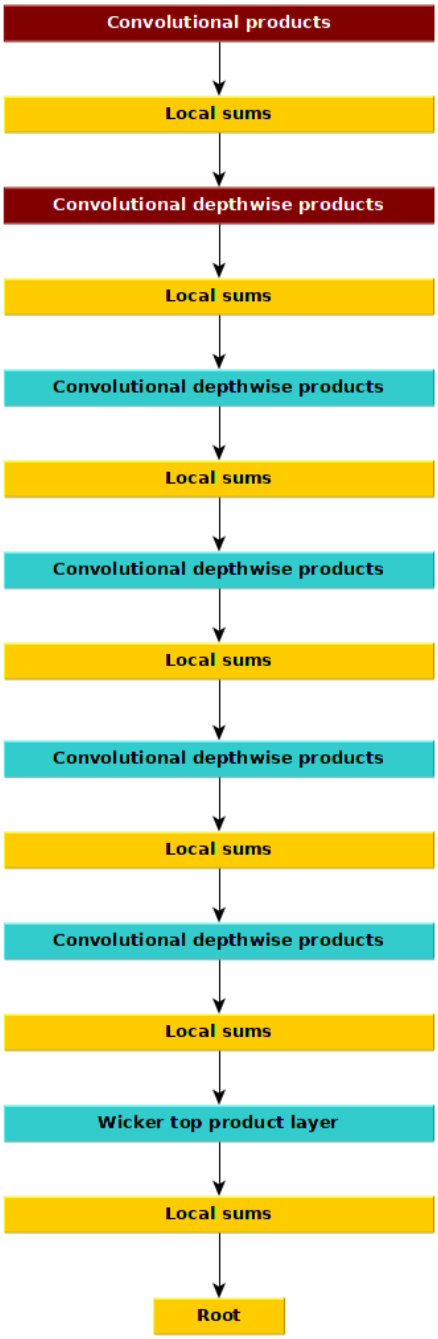

Figure 3.1.3: An example of a ConvSPN structure. Each block represents a layer. This is a mixed structure where the first two product layers are stacked in a non-overlapping fashion (scarlet or darker if black and white) and the rest of them in a wicker fashion (blue or lighter if dark and white). The only possible modification to this structure is stacking more non-overlapping layers that replace the upper wicker layers.

- Emotion generation dataset. This dataset is made to classify photos into the emotions they generate on the viewer, with eight possibilities: awe, anger, amusement, contentment, excitement, fear, disgust and sadness. It is a difficult problem with a small number of instances. Extracted from `http://www.imageemotion.org`.

In Table 3.1 we show the characteristics of every dataset and the train-validation-test split chosen.

Table 3.1: Characteristics of the datasets used in the experiments. When the image size is preceded by "around", it means the images where of different size and the dimensions provided are the means by dimension.

	Classes	Image size	Instances	Train	Validation	Test
Infected malaria cells	2	around 150x150x3	27.558	20.000	3.500	4.058
Intel scenery classification	6	150x150x3	17.034	14.034	1.685	3.000
Emotion generation	8	around 440x440x3	806	646	80	80

The Intel scenery classification already provided a train and a test set. We have used the test set as is, and extracted the validation set from the provided train set. Since the emotion generation dataset is very small and the representation of the different emotions is unbalanced, we create manually the validation and test sets so that each emotion appears the same number of times. The size of the validation and test sets are between 10% and 20% of the whole dataset.

Taking into account the limitations of our hardware, the images of every dataset had been downscaled to 64x64x3.

We have classified the three datasets using CNNs implemented in Keras and the ConvSPNs implemented in LibSPN (cf. Section 2.7). LibSPN allows to construct the structure of ConvSPNs layer by layer with the knowledge of the scope restrictions mentioned above.

Some images from the malaria dataset are shown in Figure 3.2.1.

Figure 3.2.1: Some images from the malaria dataset. The two on the left are infected cells and the one on the right is uninfected.

Some images from the Intel scenery dataset are shown in Figure 3.2.2.

Some images from the emotion generation dataset are shown in Figure 3.2.3.

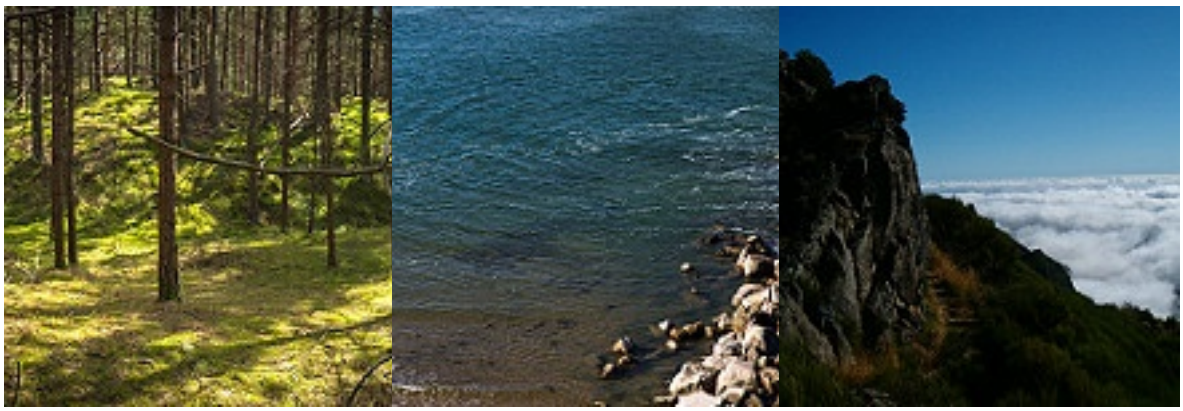

Figure 3.2.2: Some images from the Intel scenery dataset. Tags from left to right: a forest, a sea and a mountain.

Figure 3.2.3: Some images from the emotion generation dataset. Tags from left to right: awe, fear and amusement.

3.2.1 Structure and hyperparameter tuning for convolutional SPNs

As was explained in Section 3.1.2, a ConvSPN structure must satisfy the scope restrictions of validity when choosing and stacking layers. For every experiment we choose the filter size of every layer as 2x2 since it is the one that uses the most information of the images. With that set, for 64x64 images every structure will have 15 layers. Four different structures are tested for each of the three datasets, the mixed structures with one, two, three, and four non-overlapping layers followed by as many pure wicker layers as necessary. The one with two non-overlapping layers is shown in Figure 3.1.3.

The optimizer used is always AMSGrad since Wolfshaar et al. found it works better in general [65].

For hyperparameter tuning, we performed a random search of 50 configurations over the parameters organized in Table 3.2. Every configuration results are averaged over ten runs.

Table 3.2: Grid of hyperparameters for ConvSPNs. The random search selects a number of channels and use it for every layer of that type. Scale initialization is the initial mean value of the weights.

Parameter	Search space
Learning rate	$\{0.05, 0.01, 0.005, 0.001, 0.0005\}$
Channels of product wicker layers	$\{16, 32, 64\}$
Channels of product non-overlapping layers	$\{16, 32, 64\}$
Batch size	$\{16, 32, 64, 128\}$
Scale initialization	$\{0.5, 0.1, 0.05\}$

3.2.2 Structure and hyperparameter tuning for CNNs

To classify the three datasets with CNNs, we use the Keras python library. For hyperparameter tuning we use Hyperas and Hyperopt libraries.

The procedure to choose the structures was the following: we started with the CNN structure given in [8, Listing 5.5] and tested some variations of it with extra convolutional layers and fully connected layers. An example of these structures is shown in Figure 3.3.1. We keep the best performing structure from them. Then, we replaced the convolutional part of that structure with the VGG16 network with freezed weights and compared the performance of both networks on each dataset. We kept the best of both. We use VGG16 to test the performance of a pretrained model on these datasets. More powerful models are not necessary since the results obtained with these structures are conclusive.

For every structure, the filter size is set to 2x2, the activation function used in the dense layers is the ReLU and the stride of every convolutional layer is 1.

For each structure, we performed hyperparameter tuning through a random search of 50 configurations over the values of the parameters organized in Table 3.3. Every configuration results are averaged over ten runs.

Table 3.3: Grid of hyperparameters for CNNs. The random search selects a number of channels per layer. The same happens with dropout rates. The square brackets means that the value is selected from the continuous interval between those values.

Parameter	Search space
Optimizer	{SGD, RMSprop, Adam}
Learning rate	$[0.001, 0.0001]$
Channels of convolutional layers	$\{32, 64, 128, 256\}$
Neurons of dense layers	$\{512, 256, 128, 64, 32\}$
Dropout rates	$[0, 1]$
Batch size	$\{32, 64\}$

3.3 Results

We compare CNNs and ConvSPNs through the accuracy and time elapsed of the best structures found for each model with their respective configuration of hyperparameters.

3.3.1 CNN structures

The convolutional neural network structure chosen for the malaria dataset is represented in Figure 3.3.1.

The convolutional neural network structure chosen for the Intel scenery classification dataset is the one leveraging VGG16 and is represented in Figure 3.3.2.

The convolutional neural network structure chosen for the emotion generation dataset is represented in Figure 3.3.3.

3.3.2 ConvSPN structures

As was mentioned in Section 3.2.1, the structures texted are mixed ConvSPN structures with one, two, three, and four non-overlapping layers followed by as many pure wicker layers as necessary. The structure who provided better results for each dataset is indicated by its number of non-overlapping layers in Table 3.4.

3.3.3 Hyperparameter tuning

The best configurations of parameters and its associated ConvSPN structure found for each dataset by the random search are organized in Table 3.4.

Parameter	Malaria	Intel scenery	Emotion generation
Number of non-overlapping layers	2	2	1
Learning rate	0.005	0.01	0.01
Channels of product non-overlapping layers	16	64	32
Channels of product wicker layers	32	16	16
Batch size	64	64	32
Scale initialization	0.1	0.1	0.05
Epochs	20	30	40

Table 3.4: Best hyperparameters for ConvSPNs.

The best configurations of parameters found for each dataset by the random search are organized in Table 3.5.

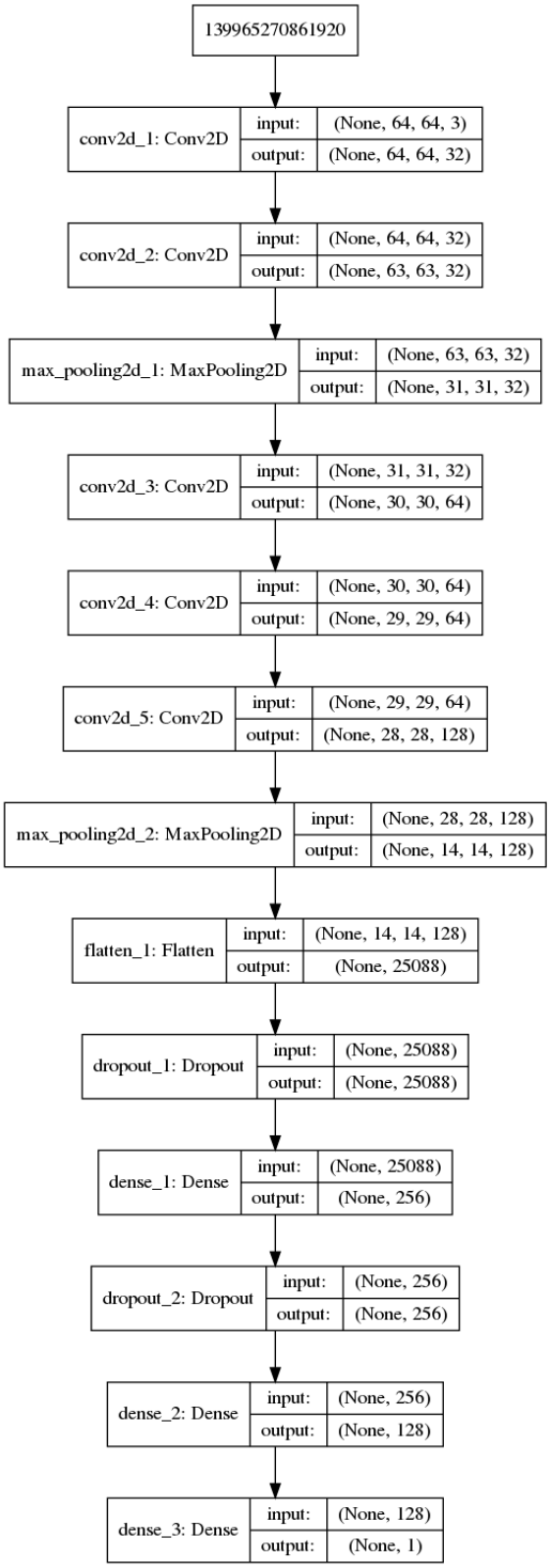

Figure 3.3.1: Convolutional neural network structure found for the malaria cell dataset represented with Keras. *Conv2D* represents a typical convolutional layer, *MaxPooling2D represents*a typical max pooling layer, *Flatten* converts a multidimensional output to a unidimensional input apt for a fully connected neural network. *Dense* represents a fully-connected layer. *Dropout* represents a dropout operation applied to the next dense layer. The last number inside each parenthesis is the input or output channels respectively.

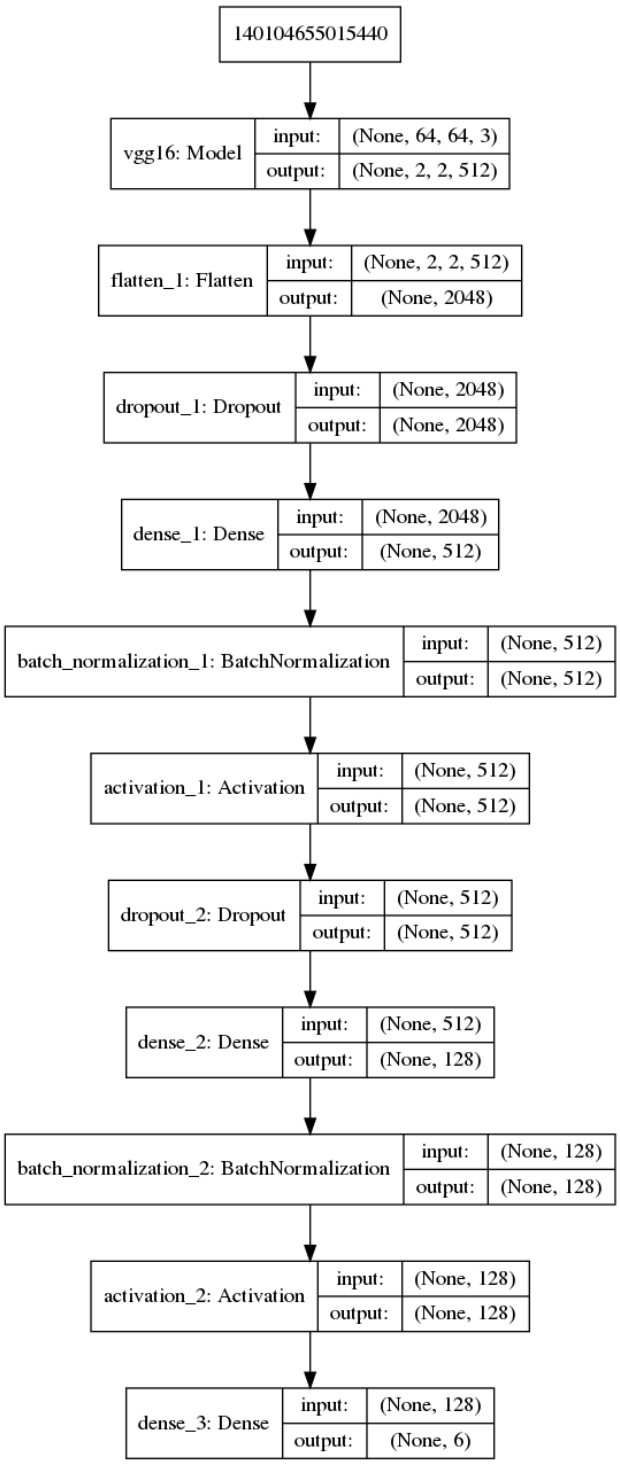

Figure 3.3.2: Convolutional neural network structure found for the Intel scenery classification dataset represented with Keras. VGG16 is the pretrained network. *Flatten* converts a multidimensional output to a unidimensional input apt for a fully connected neural network. *Dense* represents a fully-connected layer. *Dropout* represents a dropout operation applied to the next dense layer. *BatchNormalization* standarizes (dividing by the variance instead of the standard deviation) the dataset batch-wise. *Activation* represents the activation function of the dense layer, separated from it to introduce batch normalization. The last number inside each parenthesis is the input or output channels respectively.

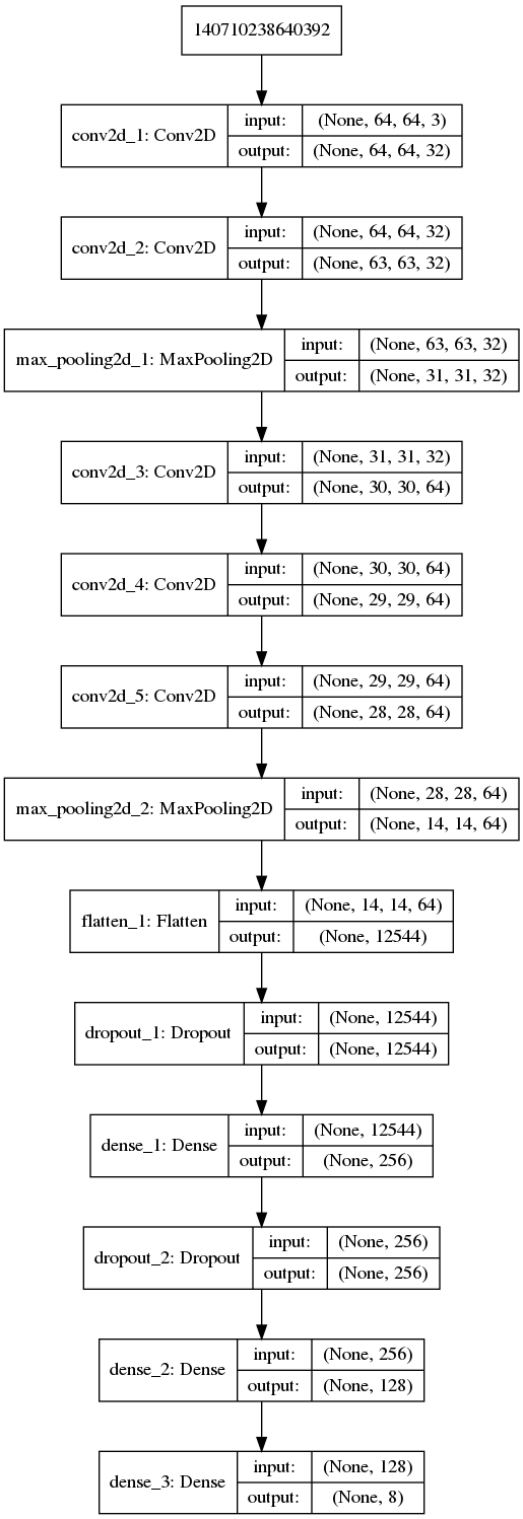

Figure 3.3.3: Convolutional neural network structure found for the emotion generation dataset represented with Keras. *Conv2D* represents a typical convolutional layer, *MaxPooling2D* a typical max pooling layer, *Flatten* converts a multidimensional output to a unidimensional input apt for a fully connected neural network. *Dense* represents a fully-connected layer. *Dropout* represents a dropout operation applied to the next dense layer. The last number inside each parenthesis is the input or output channels respectively.

Parameter	Malaria	Intel scenery	Emotion generation
Optimizer	Adam	Adam	Adam
Learning rate	0.0008	0.001	0.0009
First dropout rate	0.35	0.2	0.65
Second dropout rate	0.25	0.3	0.31
Batch size	64	32	32
Epochs	20	30	40

Table 3.5: Best hyperparameters for CNNs.

3.3.4 Accuracies and times

The results of both models for the malaria dataset are shown in Table 3.6. We observe that CNN results are notably better and that the ConvSPN overfits. The CNN is also faster.

Table 3.6: Accuracies and times for the malaria dataset.

	CNN accuracy	ConvSPN accuracy
Train	0.9886 ± 0.0061	0.9944 ± 0.0037
Validation	0.9563 ± 0.0019	0.8866 ± 0.0084
Test	0.9561 ± 0.0014	0.8895 ± 0.0077
Time per run (s)	322 ± 3	1597 ± 12

The results of both models are shown in Table 3.7. In this case CNN results are also notably better. Both models overfit. The CNN is also faster.

Table 3.7: Accuracies and times for the Intel classification dataset.

	CNN accuracy	ConvSPN accuracy
Train	0.9911 ± 0.0018	0.9967 ± 0.0007
Validation	0.8433 ± 0.0032	0.6794 ± 0.0067
Test	0.8429 ± 0.0049	0.6951 ± 0.0088
Time per run (s)	322 ± 3	2599 ± 13

The results of both models are shown in Table 3.8. We observe that the accuracies are very low and dispersed for both models. Both models have similar accuracies so there is no winner this time.

Table 3.8: Accuracies and times for the emotion generation dataset.

	CNN accuracy	ConvSPN accuracy
Train	0.9851 ± 0.0133	0.8550 ± 0.0822
Validation	0.2387 ± 0.0407	0.2388 ± 0.0308
Test	0.2037 ± 0.0252	0.2063 ± 0.0430
Time per run (s)	27 ± 0	225 ± 3

3.4 Discussion

From the results we see that CNNs clearly surpass ConvSPNs in two of the three classification tasks, while in the emotion generation dataset neither of them achieves satisfying results. This was expected, since ConvSPNs are a subclass of CNNs that lose flexibility to accommodate the completeness and decomposability properties of SPNs. Unfortunately, the advantages of imposing these restrictions (namely, to get a generative convolutional model) do not have any positive effect in a classification task.

In train set accuaracy both models are similar (the difference in train accuracy in the emotion dataset vanished if the ConvSPN ran for more epochs). That means that ConvSPNs can model without problems the train set, however, they do not generalize well and overfit notably. During hyperparameter tuning, it has been observed that configurations with less model capacity (fewer channels, for example) improved slightly the performance of the ConvSPNs on both the validation and the test set.

The emotion generation dataset is a special case where the results are neither decisive nor satisfactory. Both models struggle to deliver a result better than the one obtained when classifying two of the eight labels correctly. This is because the problem is fairly difficult, tagging photos by emotions and the dataset provided is fairly small (around 100 images per emotion).

Aside from the accuracy, CNNs work a lot faster (around 5 to 8 times faster) than ConvSPNs in spite of using structures far more complex than those of the ConvSPN, both in depth and width.

In depth, all the structures of ConvSPNs counted with seven convolutional layers (a convolutional layer in a neural network is equivalent to a product and a sum layer in a ConvSPN) while the convolutional neural network used five convolutional layers plus three dense layers.

In width, ConvSPNs only used up to 64 channels while CNNs of 256 channels where tested. The limit on this was set by the hardware we were using. The memory available was able to allocate 256 channels for a convolutional neural network but not for a ConvSPN.

This is also expected since Keras is a widely used and supported library. It has years of development and has been specially designed to work with neural network. It also counts with additional libraries that automatically run a random hyperparameter search. LibSPN, on the other hand, has been available only for months and it is yet a work in progress. At the same time, it does not count with the same human and monetary resources.

Chapter 4

Conclusions and future work

In this chapter we compare SPNs with models from the two areas they originate: probabilistic graphical models and deep learning. Then we conclude and point to some lines of future work.

4.1 Comparison with related models

4.1.1 SPNs vs. probabilistic graphical models

SPNs are similar to probabilistic graphical models (PGMs), such as Bayesian networks (BNs) and Markov networks (also called Markov random fields) [29, 45], in their ability to compactly represent probability distributions. The main difference is that in a PGM every node represents a variable and, roughly speaking, links represent probabilistic dependencies, while in an SPN every node represents a probability function. Every BN can be decomposed into an SPN, as shown in Figure 4.1.1. The reader may wonder: if inference in BNs in NP-complete [11] and all known algorithms have worst-case exponential complexity, how can SPNs do inference in linear time? The answer to this paradox is that the size of an SPNs obtained from a BN may grow exponentially with the number of variables, so that from the point of view of inference is in general worthless to convert a BN into an SPN.

Conversely, an SPN can be converted into a BN in time and space proportional to the network size using algebraic decision diagrams (ADDs) [71] and the variable elimination algorithm can be used to recover the original SPN.

Even though SPNs are not more tractable than PGMs in general, a great difference arises when learning the models from data: while learned PGMs 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 2.5 can build tractable SPNs that yield excellent approximations both for generative and discriminative tasks.

In contrast, BNs can be built from causal knowledge elicited from human experts and there is a lot of recent research on building causal BNs from experimental and/or observational data, under certain conditions [46]. It is also possible to combine knowledge and data, and even to build BNs

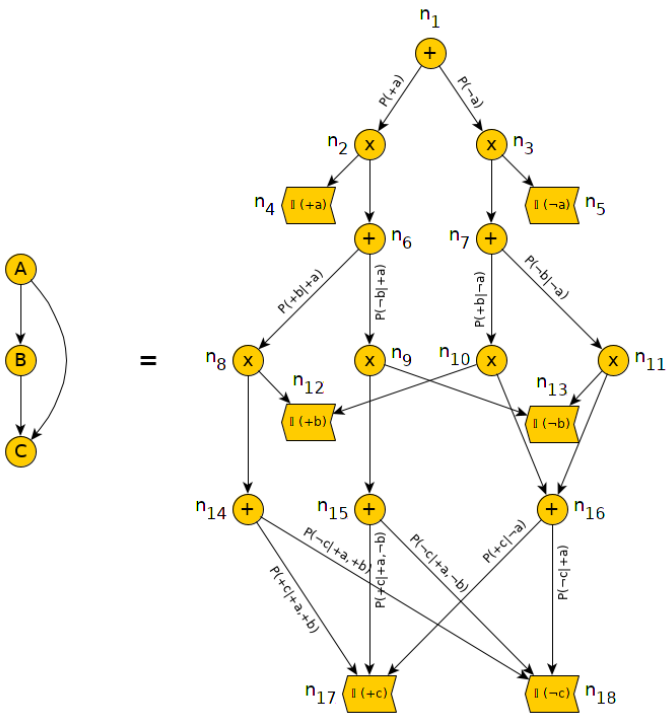

Figure 4.1.1: The weights in Figure 2.2.1 can be understood as conditional probabilities. This SPN represents a probability distribution with context-specific independence: B and C are conditionally independent given $\neg a$. The Bayesian network on the left represents the same probability distribution, but its graph does not show the context-specific independence.

interactively [3]. Additionally, the independences in a BN or in a Markov model are easier to read than those in an SPN.

Therefore 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 that are required.

4.1.2 SPNs vs. neural networks

As mentioned in the introduction, 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 we have reserved the term "neural network" (NN) for those models composed of layers of standard artificial neurons.

The main difference is that SPNs have a probabilistic interpretation as a hierarchical combination of mixtures and factorizations of probability distributions, while standard NNs do not. Inference is also different, in SPNs computing a posterior probability requires two passes and finding an MPE requires a backtrack from the root to the leaves. 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 (cf. Sec. 2.4).

When building practical applications the main difference is the possibility of determining the structure of an SPN from the available data algorithmically. In contrast, NN are usually designed by hand, and it is then necessary to try different architectures of different sizes and learn different hyperparameters for them, in a trial-and-error approach, until a satisfactory model is found. For this reason, the NNs that have succeeded in practical applications are usually very big and training them requires huge computational power. There are some recent proposals to learn the structure of NNs using evolutionary computation, which yields more efficient NNs, but this also requires tremendous computational power.

In spite of these advantages, NNs are still superior to SPNs in many tasks. For example, in 2012 an SPN by Gens and Domingos achieved a classification accuracy of 84% for the CIFAR-10 image dataset, one of the highest scores so far, but after that year, several deep NNs have improved that result by far, reaching an impressive 99% accuracy.[1]

4.2 General conclusions

SPNs have been applied to the same tasks as neural networks, mainly image and natural language processing, which exceeded by far the capabilities of PGMs, sometimes showing superior results [39]. The progress made in this subarea since 2011 make them a very promising tool for addressing some of the current challenges of artificial intelligence, but they are yet a young model with a long way to go.

In the survey part of this work 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 mathematical complexity to the minimum necessary for describing with rigor the main properties of SPNs, from their definition to the algorithms for parametric and structural learning. After comparing them with related models, such as PGMs and neural networks, we have reviewed several applications of SPNs in different domains, some extensions, and the main software libraries for SPNs. It also provides original proofs and interpretations that complement the original ones and provide the reader with different approaches to the same concepts.

We intend to submit the survey contained in Chapter 2 to the journal IEEE Transactions on Pattern Analysis and Machine Intelligence in a few weeks. We sent a draft to several experts on the field which had a great reception among them. Pedro Domingos (University of Washington), the father of SPNs, provided detailed comment beginning with: "The survey looks great". Robert

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

Peharz, IP of a research proyect about SPNs, (University of Cambridge) is yet to comment on it but says: "Interesting, and a good idea to write a survey". Andrezj Pronobis (University of Washington), cocreator of ConvSPNs and LibSPN, also says: "I'm really happy to see an up to date survey about SPNs being developed", Pascal Poupart (University of Waterloo, Canada), IP of a research proyect about SPNs and collaborator in SPFlow, says: "Writing a survey about SPNs is a great idea!" and Alejandro Molina, co-creator of SPFlow, (Technical University of Darmstadt) says: "It looks pretty good. Really nice work!".

4.3 Future work

General lines for future research are developing better learning algorithms, applying SPNs to new problems in different domains, generating didactic content to bring SPNs to a wider public, building causal SPNs from data or further developing robust and user-friendly software to work with SPNs.

Future work to bring SPNs to a wider public can be the creation of practical content like Jupyter notebooks to learn about SPNs through using them.

In the context of the practical application there exist some other lines of future work.

Image classification is a task where convolutional neural networks excel and the only one we tried. A possibility is to extend this comparison to different image tasks like object segmentation, object detection or image style transfer.

Extending this analysis to generative tasks like image generation or completion is even more interesting. In the discussion of the experimental work we mentioned that restricting convolutional layers because of the scope was not beneficial. In this case, these restrictions transform a discriminative model (CNNs) into a explicit generative model (SPNs). On one hand, we expect improved performance on generative tasks. On the other hand, ConvSPNs will compete with implicit generative models like GANs.

We aimed to test bigger databases like Food101 during our experiments, but the hardware at our disposal, in particular, the memory resources did not allow us to. Another future prospect is to improve in that aspect and be able to run bigger and longer experiments.

Finally, testing other promising extensions of SPNs is another possibility, an example is compositional kernel machines (CKM). Gens and Domingos [22] claim that "we present results [...] that show a CKM trained on a CPU can be competitive with convnets trained for much longer on a GPU".

Bibliography

[1] T. Adel, D. Balduzzi, and A. Ghodsi. Learning the structure of sum-product networks via an SVD-based algorithm. In *Proceedings of the Thirty-First Conference on Uncertainty in Artificial Intelligence*, UAI'15, pages 32–41, Arlington, Virginia, United States, 2015. AUAI Press.

[2] M. R. Amer and S. Todorovic. Sum product networks for activity recognition. *IEEE Transaction on Pattern Analysis and Machine Intelligence*, 38:800–813, 2016.

[3] I. Bermejo, J. Oliva, F. J. Díez, and M. Arias. Interactive learning of Bayesian networks with OpenMarkov. In A. Cano, M. Gómez, and T. D. Nielsen, editors, *Proceedings of the Sixth European Workshop on Probabilistic Graphical Models (PGM'12)*, pages 27–34, Granada, Spain, 2012.

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

[5] C. J. Butz, A. E. dos Santos, J. S. Oliveira, and J. Stavrinides. Efficient examination of soil bacteria using probabilistic graphical models. In *Recent Trends and Future Technology in Applied Intelligence*, pages 315–326. Springer International Publishing, 2018.

[6] C. J. Butz, J. S. Oliveira, A. E. dos Santos, A. L. Teixeira, P. Poupart, and A. Kalra. An empirical study of methods for SPN learning and inference. In *Proceedings of the Ninth International Conference on Probabilistic Graphical Models*, volume 72 of *Proceedings of Machine Learning Research*, pages 49–60. PMLR, 2018.

[7] W.-C. Cheng, S. Kok, H. V. Pham, H. L. Chieu, and K. M. A. Chai. Language modeling with sum-product networks. *Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH*, pages 2098–2102, 2014.

[8] F. Chollet. *Deep learning with Python*. Manning Publications, 2018.

[9] C. Chow and C. Liu. Approximating discrete probability distributions with dependence trees. *IEEE transactions on Information Theory*, 14(3):462–467, 1968.

[10] D. Ciresan, A. Giusti, L. M. Gambardella, and J. Schmidhuber. Deep neural networks segment neuronal membranes in electron microscopy images. In *Advances in neural information processing systems*, pages 2843–2851, 2012.

[11] G. F. Cooper. The computational complexity of probabilistic inference using Bayesian belief networks. *Artificial Intelligence*, 42:393–405, 1990.

[12] A. Darwiche. A differential approach to inference in Bayesian networks. *Journal of the ACM*, 50:280–305, 2003.

[13] A. Dennis and D. Ventura. Learning the architecture of sum-product networks using clustering on variables. In *Advances in Neural Information Processing Systems*, pages 2033–2041, 2012.

[14] A. Dennis and D. Ventura. Greedy structure search for sum-product networks. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, pages 932–938. AAAI Press, 2015.

[15] A. Dennis and D. Ventura. Online structure-search for sum-product networks. In *16th IEEE International Conference on Machine Learning and Applications*, pages 155–160, 2017.

[16] M. Desana and C. Schnörr. Learning arbitrary sum-product network leaves with expectation-maximization. *arXiv:1604.07243*, 2017.

[17] M. Desana and C. Schnörr. Sum-product graphical models. *Machine Learning*, 2019.

[18] N. Di Mauro, F. Esposito, F. Ventola, and A. Vergari. Alternative variable splitting methods to learn sum-product networks. In *AI*IA Advances in Artificial Intelligence: XVIth International Conference of the Italian Association for Artificial Intelligence*, 2017.

[19] A. L. Friesen and P. Domingos. Unifying sum-product networks and submodular fields. In *Proceedings of the Workshop on Principled Approaches to Deep Learning at ICML*, 2017.

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

[21] R. Gens and P. Domingos. Learning the structure of sum-product networks. In *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 873–880. PMLR, 2013.

[22] R. Gens and P. Domingos. Compositional kernel machines. In *5th International Conference on Learning Representations - Workshop Track*, 2017.

[23] W. Hsu, A. Kalra, and P. Poupart. Online structure learning for sum-product networks with Gaussian leaves. In *5th International Conference on Learning Representations - Workshop Track*, 2017.

[24] D. H. Hubel and T. N. Wiesel. Receptive fields and functional architecture of monkey striate cortex. *The Journal of physiology*, 195(1):215–243, 1968.

[25] P. Jaini, A. Ghose, and P. Poupart. Prometheus: directly learning acyclic directed graph structures for sum-product networks. In *Proceedings of the Ninth International Conference on Probabilistic Graphical Models*, volume 72 of *Proceedings of Machine Learning Research*, pages 181–192. PMLR, 2018.

[26] P. Jaini, A. Rashwan, H. Zhao, Y. Liu, E. Banijamali, Z. Chen, and P. Poupart. Online algorithms for sum-product networks with continuous variables. In *Conference on Probabilistic Graphical Models*, volume 52, pages 228–239, 2016.

[27] A. Kalra, A. Rashwan, W.-S. Hsu, P. Poupart, P. Doshi, and G. Trimponias. Online structure learning for feed-forward and recurrent sum-product networks. In *Advances in Neural Information Processing Systems 31*, pages 6944–6954. 2018.

[28] C.-Y. Ko, C. Chen, Y. Zhang, K. Batselier, and N Wong. Deep compression of sum-product networks on tensor networks. *arXiv:1811.03963*, 2018.

[29] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. The MIT Press, Cambridge, MA, 2009.

[30] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[31] S. Lee, M. Heo, and B. Zhang. Online incremental structure learning of sum-product networks. In *International Conference on Neural Information Processing*, pages 220–227. Springer, 2013.

[32] Y. Liu and T. Luo. The optimization of sum-product network structure learning. *Journal of Visual Communication and Image Representation*, 2019.

[33] M. Loog. Contrastive pessimistic likelihood estimation for semi-supervised classification. *IEEE Transactions on Pattern Analysis and Maching Intelligence*, 38:462–475, 2016.

[34] D. Lowd and A. Rooshenas. Learning markov networks with arithmetic circuits. In *Artificial Intelligence and Statistics*, pages 406–414, 2013.

[35] D. Lowd and A. Rooshenas. The libra toolkit for probabilistic models. *The Journal of Machine Learning Research*, 16(1):2459–2463, 2015.

[36] D. D. Mauá, F. G. Cozman, Diarmaid Conaty, and C. P. Campos. Credal sum-product networks. In *Proceedings of the Tenth International Symposium on Imprecise Probability: Theories and Applications*, pages 205–216, 2017.

[37] J. Mei, Y. Jiang, and K. Tu. Maximum a posteriori inference in sum-product networks. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[38] M. Melibari, P. Poupart, and P. Doshi. Sum-product-max networks for tractable decision making. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, pages 1846–1852, 2016.

[39] M. Melibari, P. Poupart, P. Doshi, and G. Trimponias. Dynamic sum-product networks for tractable inference on sequence data. In *Conference on Probabilistic Graphical Models (PGM-2016)*, pages 345–356, 2016.

[40] A. Molina, S. Natarajan, and K. Kersting. Poisson sum-product networks: A deep architecture for tractable multivariate poisson distributions. In *AAAI Conference on Artificial Intelligence*, pages 2357–2363, 2017.

[41] A. Molina, A. Vergari, N. Di Mauro, S. Natarajan, F. Esposito, and K. Kersting. Mixed sum-product networks: A deep architecture for hybrid domains. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[42] A. Molina, A. Vergari, K. Stelzner, R. Peharz, P. Subramani, N. Di Mauro, P. Poupart, and K. Kersting. SPFlow: an easy and extensible library for deep probabilistic learning using sum-product networks. *arXiv:1901.03704*, 2019.

[43] A. Nath and P. Domingos. Learning relational sum-product networks. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, pages 2878–2886, 2015.

[44] A. Nath and P. Domingos. Learning tractable probabilistic models for fault localization. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 1294–1301, 2016.

[45] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, San Mateo, CA, 1988.

[46] J. Pearl. *Causality. Models, Reasoning, and Inference*. Cambridge University Press, Cambridge, UK, 2000.

[47] R. Peharz. *Foundations of sum-product networks for probabilistic modeling*. PhD thesis, 2015.

[48] R. Peharz, B. C. Geiger, and F. Pernkopf. Greedy part-wise learning of sum-product networks. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 612–627. Springer, 2013.

[49] R. Peharz, R. Gens, and P. Domingos. Learning selective sum-product networks. In *LTPM workshop*, volume 32, 2014.

[50] R. Peharz, R. Gens, F. Pernkopf, and P. M. Domingos. On the latent variable interpretation in sum-product networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39:2030–2044, 2017.

[51] R. Peharz, G. Kapeller, P. Mowlaee, and F. Pernkopf. Modeling speech with sum-product networks: application to bandwidth extension. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 3699–3703, 2014.

[52] R. Peharz, A. Vergari, K. Stelzner, A. Molina, M. Trapp, K. Kersting, and Z. Ghahramani. Probabilistic deep learning using random sum-product networks. *arXiv:1806.01910*, 2018.

[53] H. Poon and P. Domingos. Sum-product networks: a new deep architecture. In *12th Conference on Uncertainty in Artificial Intelligence (UAI-2011)*, pages 337–346, 2011.

[54] A. Pronobis, A. Ranganath, and R. P. N. Rao. LibSPN: A library for learning and inference with sum-product networks and TensorFlow. In *Principled Approaches to Deep Learning Workshop*, 2017.

[55] A. Pronobis, F. Riccio, and R. P. N. Rao. Deep spatial affordance hierarchy: spatial knowledge representation for planning in large-scale environments. In *ICAPS Workshop on Planning and Robotics*, 2017.

[56] T. Rahman and V. Gogate. Merging strategies for sum-product networks: from trees to graphs. In *Proceedings of the Thirty-Second Conference on Uncertainty in Artificial Intelligence*, UAI'16, pages 617–626, Arlington, Virginia, United States, 2016. AUAI Press.

[57] A. Rashwan, P. Poupart, and C. Zhitang. Discriminative training of sum-product networks by extended Baum-Welch. In *International Conference on Probabilistic Graphical Models (PGM-2018)*, pages 356–367, 2018.

[58] A. Rashwan, H. Zhao, and P. Poupart. Online and distributed Bayesian moment matching for parameter learning in sum-product networks. In *Artificial Intelligence and Statistics*, pages 1469–1477, 2016.

[59] F. Rathke, M. Desana, and C. Schnörr. Locally adaptive probabilistic models for global segmentation of pathological OCT scans. In *Medical Image Computing and Computer Assisted Intervention*, pages 177–184, 2017.

[60] A. Rooshenas and D. Lowd. Learning sum-product networks with direct and indirect variable interactions. In *International Conference on Machine Learning*, pages 710–718, 2014.

[61] X. Shao, A. Molina, A. Vergari, K. Stelzner, R. Peharz, T. Liebig, and K. Kersting. Conditional sum-product networks: Imposing structure on deep probabilistic architectures. *arXiv:1905.08550*, 2019.

[62] O. Sharir and A. Shashua. Sum-product-quotient networks. In *International Conference on Artificial Intelligence and Statistics*, pages 529–537, 2018.

[63] M. Trapp, T. Madl, R. Peharz, F. Pernkopf, and R. Trappl. Safe semi-supervised learning of sum-product networks. In *Uncertainty in Artificial Intelligence-Proceedings of the 33rd Conference*, 2017.

[64] M. Trapp, R. Peharz, M. Skowron, T. Madl, F. Pernkopf, and R. Trappl. Structure inference in sum-product networks using infinite sum-product trees. In *Neural Information Processing Systems (NIPS) Workshop*, 2016.

[65] J. van de Wolfshaar and A. Pronobis. Deep convolutional sum-product networks for probabilistic image representations. *arXiv:1902.06155*, 2019.

[66] A. Vergari, N. Di Mauro, and F. Esposito. Simplifying, regularizing and strengthening sum-product network structure learning. In *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases - Volume Part II*, ECMLPKDD'15, pages 343–358, Switzerland, 2015. Springer.

[67] J. Wang, G. Wang, Jinghua Wang, and Gang Wang. Hierarchical spatial sum–product networks for action recognition in still images. *IEEE Transactions on Circuits and Systems for Video Technology*, 28:90–100, 2016.

[68] Z. Yuan, H. Wang, L. Wang, T. Lu, S. Palaiahnakote, and C. L. Tan. Modeling spatial layout for scene image understanding via a novel multiscale sum-product network. *Expert Systems with Applications*, 63:231–240, 2016.

[69] H. Zhao, T. Adel, G. Gordon, and B. Amos. Collapsed variational inference for sum-product networks. In *International Conference on Machine Learning*, pages 1310–1318, 2016.

[70] H. Zhao and G. J. Gordon. Linear time computation of moments in sum-product networks. In *Advances in Neural Information Processing Systems*, pages 6894–6903, 2017.

[71] H. Zhao, M. Melibari, and P. Poupart. On the relationship between sum-product networks and Bayesian networks. In *International Conference on Machine Learning*, pages 116–124, 2015.

[72] H. Zhao, P Poupart, and G. J. Gordon. A unified approach for learning the parameters of sum-product networks. In *Advances in neural information processing systems*, pages 433–441, 2016.

[73] K. Zheng, A. Pronobis, and R. P. N. Rao. Learning semantic maps with topological spatial relations using graph-structured sum-product networks. In *AAAI Conference on Artificial Intelligence*, 2018.