Sum-Product Networks: an alternative to Neural Networks?

Alex Pasquali
Technische Universität München (TUM)
alex.pasquali@tum.de

Abstract

Sum-Product Networks (SPNs) are a probabilistic framework that allows building tractable models from data, making it possible to perform various inference tasks in polynomial time. They are based on a rooted directed acyclic graph in which terminal nodes are the indicators of the input random variables (they can be seen as univariate probability distributions) and internal nodes divides themselves into sum nodes, which perform mixtures of sub-distributions, and product nodes, which perform factorizations. SPNs are related to probabilistic graphical models, such as Bayesian Networks, but the latter often suffer from tractability issues, since exact inference often requires exponential time. SPNs are also linked to Neural Networks (NNs), they share some similarities but there exist also many differences between these two classes of models. In some cases they can be used as alternatives to each other, to tackle similar kinds of problems, but they can also be applied in a complementary way, to exploit the best characteristics of both.

1 Introduction

Sum-Product Networks (SPNs) are a class of probabilistic models that was first introduced by Poon and Domingos in 2011 [1]. They are based on arithmetic circuits [2] and consist of a rooted directed acyclic graph (rooted DAG) that represents a probability distribution obtained from a hierarchy of other sub-distributions combined in the form of mixtures and factorizations. They have three kinds of nodes: the leaves are indicators for the random variables (RVs) involved in the distribution and the others divide themselves into sum and product nodes. The former perform mixtures of distributions through weighted sums of the values of their children and can be seen as a latent RVs, while the latter model complex interactions through factorizations.
SPNs have seen a growing interest because they allow to build tractable models from data, making it possible to perform various exact inference tasks in polynomial time. This is a big advantage over other probabilistic graphical models (PGMs), which are forced to rely on variational (i.e. approximated) inference or/and hand-engineered structures to model dependencies among RVs in a way that makes inference tractable. Their representations are based on a normalized product of factors as in Eq. 1, where $\mathbf{x}$ is a d-dimensional vector and each *potential function* ϕ_k is a function of a subset of the variables, indicated by $\mathbf{x}_{\{k\}}$. Z is called *partition function* and it is defined as in Eq. 2. This is oftentimes intractable because it is a sum over an exponential number of terms, i.e. every possible assignment of $\mathbf{x}$ (being $\mathbf{x}$ a d-dimensional vector).

$$P(\mathbf{X} = \mathbf{x}) = \frac{1}{Z} \prod_k \phi(\mathbf{x}_{\{k\}}), \tag{1}$$

$$Z = \sum_{\mathbf{x} \in \mathbf{X}} \prod_k \phi(\mathbf{x}_{\{k\}}), \tag{2}$$

Submitted for the seminar *Beyond Deep Learning: Selected Topics on Novel Challenges* @ TUM, January 2022

Being able to compute Z is fundamental for an accurate computation of $P(\mathbf{X} = \mathbf{x})$, since the latter contains the former in its definition (Eq. 1). Furthermore, all marginals are sums of subsets of the terms in the summation of Z, thus if it can be computed efficiently, so can they [1]. SPNs promise to accomplish exact and efficient inference by reorganizing the sums and products that appear in Z into a computation involving only a polynomial number of operations.
A more detailed description of SPNs is provided in the next section, while Section 3 describes how to infer different types of probabilities. SPNs can learn both structure and parameters from data, exploiting generative and discriminative approaches as explained in Section 4. From their introduction in 2011, they have been applied to a variety of tasks, ranging from image processing (which still remains one of the most promising areas), to natural language processing and activity recognition. An overview of different successful applications of SPNs is provided in Section 5. Finally, since in many cases SPNs can be applied alternatively to neural networks to solve similar kinds of problems, in Section 6 there is a comparison between the two classes of models, highlighting the advantages of one over the other and vice versa, as well as cases in which it is beneficial to combine their use.

2 Definition and characteristics of Sum-Product Networks

The original paper that introduced SPNs in 2011 [1] formally defines them as follows (Definition 1):

Definition 1 *A sum-product network (SPN) over variables $x_1, \ldots, x_d$ is a rooted directed acyclic graph whose leaves are the indicators $x_1, \ldots, x_d$ and $\bar{x}_1, \ldots, \bar{x}_d$ and whose internal nodes are sums and products. Each edge (i, j) emanating from a sum node i has a non-negative weight w_{ij}. The value of a product node is the product of the values of its children. The value of a sum node is $\sum_{j \in Ch(i)} w_{ij} v_j$, where $Ch(i)$ are the children of i and v_j is the value of the node j. The value of an SPN is the value of its root.*

Sum and products are assumed to be arranged in alternating layers.

An SPN S is denoted as a function of the indicators $S(x_1, \ldots, x_d, \bar{x}_1, \ldots, \bar{x}_d)$ and this notation will be often abbreviated to $S(\mathbf{x})$, where $\mathbf{x}$ indicates a complete state of the variables.
With S_n it is indicated the sub-SPN rooted in the node n of the original SPN and it is important to notice that S_n is itself an SPN. Calling the root node r, $S(\mathbf{X}) = S_r(\mathbf{X})$ is an unnormalized probability distribution over $\mathbf{X}$. The partition function of an SPN is $Z = \sum_{\mathbf{x} \in \mathbf{X}} S(\mathbf{x})$, therefore, the normalized probability of $\mathbf{x}$ is $P(\mathbf{x}) = S(\mathbf{x})/Z$. If the weights at each sum node sum to one and the leaf distributions are normalized, then $Z = 1$ and $P(\mathbf{x}) = S(\mathbf{x})$. When dealing with an evidence $\mathbf{e}$, its unnormalized probability under such distribution is defined as $\Phi_S(\mathbf{e}) = \sum_{\mathbf{x} \in \mathbf{e}} S(\mathbf{x})$, where the sum is over all states compatible with the evidence.
Before proceeding to define the properties of SPNs, it is important to first define the concept of *scope* ($sc(S)$) as the set of variables that appear in the network. Note that, since the sub-network rooted in an arbitrary node of an SPN is itself an SPN, also each individual node has its own scope, in particular, if n_i is a sum or product node, its scope is the union of the scopes of its children.

Definition 2 (Validity) *An SPN S is valid iff $\forall e$ $S(\mathbf{e}) = \Phi_S(\mathbf{e})$.*

This means that a valid SPN must always compute correctly the probability of an evidence. Also, S being valid implies that $S(x_1 = 1, \ldots, x_d = 1, \bar{x}_1 = 0, \ldots, \bar{x}_d = 0) = Z_S$, it means that, in this case, the value of S when all the indicators are set to 1 corresponds to the value of the partition function Z_S, which becomes computable with a single upward pass, in a time proportional to the number of links in the network (that will be forced to be polynomial w.r.t. the number of variables). This report, as most of the scientific literature concerning this topic, will focus only on valid SPNs.

Definition 3 (Network polynomial) *Let $\Phi(\mathbf{x}) \geq 0$ be an unnormalized probability distribution. The network polynomial of $\Phi(\mathbf{x})$ is $\sum_{\mathbf{x}} \Phi(\mathbf{x}) \prod(\mathbf{x})$, where $\prod(\mathbf{x})$ is the product of the indicators that are set to 1 in state $\mathbf{x}$.*

Definition 4 (Completeness) *A sum node is complete iff all its children have the same scope. An SPN is complete if all its sum nodes are complete.*

Definition 5 (Consistency) *An SPN is consistent iff no variable appears negated in onr child of a product node and non-negated in another.*

From the last two properties the next theorem follows:

Theorem 1 *An SPN is valid if it is complete and consistence [1].*

Since the first papers about SPNs [1, 3] were published, the understanding of these models has changed and improved. This brought a visible "change of focus" in the more recent literature (starting with [4]), where SPNs tend not to be presented anymore as an efficient way to compute partition functions or an efficient representation of network polynomials (Def. 3), but they are often defined as hierarchical compositions of probability distributions, which is, in the words of París et al. [5], *"more intuitive and much easier to understand"*. In this later literature, consistency has been set aside in favour of decomposability (Def. 6), which is a stronger but more intuitive property that suffices to build SPNs for practical applications [5]. Furthermore, selectivity (Def. 7) was not mentioned in the first paper from Poon and Domingos [1], but proved to be relevant for some inference tasks and parameters learning [6, 7].

Definition 6 (Decomposability) *A product node is decomposable iff all its children have disjoint scopes. An SPN is decomposable if all its product nodes are decomposable.*

Definition 7 (Selectivity) *A sum node is selective if at most one child makes a positive contribution. An SPN is selective if all its sum nodes are selective.*

For discrete - *non-binary* - variables, the indicators at the leaves simply assume the values that each variable can assume, therefore every leaf can be seen as a univariate distribution. This has let SPNs to be also defined recursively, as combinations of sub-SPNs. Gens and Domingos, in their work from 2013 [4], proposed Definition 8, an alternative to Definition 1 that focuses on viewing SPNs as compositions of probability distributions, highlighting the evolution of the understanding of these models through the years.

Definition 8 *A sum-product network (SPN) is defined as follows.*

1. *A tractable univariate distribution is an SPN.*
2. *A product of SPNs with disjoint scopes is an SPN.*
3. *A weighted sum of SPNs with the same scope is an SPN, provided all weights are positive.*
4. *Nothing else is an SPN.*

3 Inference in SPNs

Let us now analyze how to perform various inference tasks in Sum-Product Networks, focusing also on the time complexity required for their computation. In Theorem 1 of [4], Gens and Domingos state that the partition function and the probability of an evidence can be computed in linear time in an SPN.

3.1 Marginal and posterior probabilities

Recall that, being r the root of the SPN S, $P(\mathbf{x}) = S(\mathbf{x}) = S_r(\mathbf{x})$. The value of $S(\mathbf{x})$ can be computed by an upward pass, from the leaves to the root, in a time proportional to the number of links in S.
Calling $\mathbf{V}$ the set of variables, if $\mathbf{X}$ and $\mathbf{E}$ are two disjoint subsets of $\mathbf{V}$, then $P(\mathbf{x}|\mathbf{e}) = S(\mathbf{xe})/S(\mathbf{e})$, where $\mathbf{xe}$ is simply the composition of $\mathbf{x}$ and $\mathbf{e}$, and it can be computed in two upward passes. Therefore, any joint, marginal or conditional probability can be computed by SPNs in at most two upward passes whose complexity is linear in the number of edges and therefore polynomial. Note that this is verified because it is possible to impose constraints when deciding the initial structure of the network or when learning it (more details in Section 4).
As a further note, Buts et al. [8] proposed a partial propagation which only propagates the nodes in $\mathbf{X} \cup \mathbf{E}$ and can be significantly faster.

3.2 MPE inference

The maximum probable explanation (MPE) in an SPN is defined as follows, assuming S has normalized weights:

$$MPE(\mathbf{e}) = \arg\max_{\mathbf{x}} P(\mathbf{x}|\mathbf{e}) = \arg\max_{\mathbf{x}} P(\mathbf{x}\mathbf{e}) = \arg\max_{\mathbf{x}} S(\mathbf{x}\mathbf{e}) \tag{3}$$

where $\mathbf{x}\mathbf{e}$ is the composition of $\mathbf{x}$ and $\mathbf{e}$. This type of inference can be efficiently computed by the Best Tree algorithm (BT), which was mentioned in the very first paper on SPNs [1], even though it got its name later, in [9].
Before analyzing BT in detail, it is necessary to introduce the concept of *induced tree*:

Definition 9 (Induced tree [10, 5]) *Let S be an SPN and $\mathbf{x} \in conf(S)$, where $conf(S)$ is the set of all the configurations of $\mathbf{X}$, such that $S(\mathbf{x}) \neq 0$. The sub-SPN induced by $\mathbf{v}$, denoted by $S_{\mathbf{v}}$, is a non-normalized SPN obtained by:*

1. *removing every node n_i such that $S_i(\mathbf{x}) = 0$ and the corresponding links,*
2. *removing every link $n_i \to n_j$ such that $w_{ij} = 0$,*
3. *removing recursively all the nodes without parents (except, in case, the root).*

Proposition 1 *If S is selective, $\mathbf{x} \in conf(S)$, and $S(\mathbf{x}) \neq 0$, then $S_{\mathbf{x}}$ is a tree in which every sum node has exactly one child. In this case the notation $\mathcal{T}_{\mathbf{x}}$ will substitute $S_{\mathbf{x}}$ to remark that it is a tree [10, 5].*

3.2.1 Best Tree algorithm (BT)

Assuming S is selective[1] (Def. 7), then $\mathbf{X} \cup \mathbf{E} = sc(S) \Rightarrow \mathbf{x}\mathbf{e} \in conf(S)$ and the sub-SPN induced by $\mathbf{x}\mathbf{e}$ is a tree where every sum node has only one child. Therefore, MPE can be found by examining all the trees for the configurations $\mathbf{x}\mathbf{e}$ in which $\mathbf{e}$, denoting an evidence, is fixed and $\mathbf{x}$ varies, assuming all its possible assignments. It is possible to compare all these trees at once by performing a single upward pass, computing $S_i^{max}(\mathbf{e})$ for each node, and the backtracking from the root to the leaves. The various $S_i^{max}(\mathbf{e})$ are computed as follows:

$$S_i^{max}(\mathbf{e}) = \begin{cases} \max\limits_{j \in Ch(i)} w_{ij} \cdot S_j^{max}(\mathbf{e}) & \text{if } n_i \text{ is a sum node} \\ S_i(\mathbf{e}) & \text{otherwise} \end{cases} \tag{4}$$

The algorithm then backtracks from the root to the leaves, selecting for each sum node the child that led to $S^{max}(\mathbf{e})$, making sum nodes become *max* nodes, and for each product node all of its children are selected. When arriving at a leaf node, let V be the variable associated to that node, the algorithm selects the value $v = \arg\max_{v'} P(v')$. The values selected at the terminal nodes produce the configuration $\hat{\mathbf{x}} = MPE(\mathbf{e})$. Figure 1 shows an example of the execution of BT on an SPN.

[1] Theorem 2 in [6] proves that in selective SPNs, the Best Tree algorithm (BT) computes the true MPE, however if the SPN is not selective, the configuration $\mathbf{x}\mathbf{e}$ is not necessarily a tree, so the configuration returned by BT may be different from the true MPE. In fact, Theorem 5.3 in [11] states the MPE is NP-complete for general (thus without a selectivity assumption) SPNs.

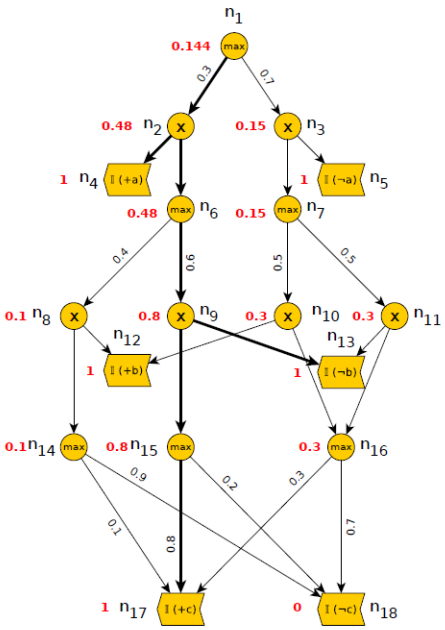

Figure 1: Image showing the MPE computation for an SPN. The numbers in red are the values $S_i^{max}(\mathbf{e})$ for $\mathbf{e} = +c$. In this case $MPE(\mathbf{e}) = (+a, \neg b)$ and it is found backtracking from the root to the leaves following the thick lines. Source: [5].

4 Learning SPNs

The structure and parameters (i.e. the weights of the sum nodes) of an SPN can be learnt jointly, as illustrated in Algorithm 1 (taken from [1]). Starting with a generic, but valid, densely connected architecture, each example $\mathbf{d}$ in the dataset $\mathcal{D}$ is fed to the model, which runs inference on it, computing $S(\mathbf{d})$, and updates the weights. This process is repeated until convergence and the final SPN is obtained by pruning the edges with zero weight and recursively removing parentless nodes (root excluded).

Algorithm 1 Learn parameters and structure of an SPN

Input: Set D of instances over variables $\mathbf{X}$.
Output: An SPN with learnt structure and parameters.
$S \leftarrow$ GenerateDenseSPN($\mathbf{X}$)
InitializeWeights(S)
repeat
  for all $\mathbf{d} \in D$ **do**
    UpdateWeights(S, Inference($S, \mathbf{d}$))
  end for
until convergence
$S \leftarrow$ PruneZeroWeights(S)
return S

It has to be noted that Algorithm 1 is a general procedure and does not specify every step in detail. For example, every time there is a "function call", it assumes that there exists some procedure to accomplish such step, but this procedure is not specified and can be implemented in many ways, keeping the general learning algorithm unchanged.
Nevertheless, the authors [1] provide a way to generate the initial structure of the network (Section 4.3.1) and some alternatives to learn the weights (Section 4), giving a possible realization of `GenerateDenseSPN` and `UpdateWeights`. As for `InitializeWeights`, in their experiments

they start with all weights equal to zero. Concerning the remaining "function calls", `Inference` consists of simply computing $S(\mathbf{d})$ as explained in Section 3 of this report and `PruneZeroWeights` is straightforward to understand.

4.1 (Generative) parameters learning

Parameters learning consists of finding the optimal parameters for an SPN given its stucture and a dataset. The optimality criterion in this case is to maximize the likelihood of the weights.
Let $\mathcal{D} = \{\mathbf{x}_1, \ldots, \mathbf{x}_T\}$ be the dataset, $\mathbf{W}$ the set of weights and $\mathbf{w} \in \mathbf{W}$ a particular realization of the parameters. $\mathcal{L}_\mathcal{D}(\mathbf{w})$ is defined in Eq. 5 as the logarithm of the likelihood:

$$\mathcal{L}_\mathcal{D}(\mathbf{w}) = \log P(\mathcal{D}|\mathbf{w}) = \sum_{t=1}^{T} \log S(\mathbf{x}_t|\mathbf{w}) \tag{5}$$

4.1.1 Gradient descent (GD)

The weight update in Algorithm 1 can be performed by gradient descent. In this case it is possible to define it as follows:

$$w_{ij}^{(s+1)} = w_{ij}^{(s)} + \gamma \frac{\partial \mathcal{L}_\mathcal{D}(\mathbf{w})}{\partial w_{ij}^{(s)}} \tag{6}$$

where s is the iteration index and γ is the learning rate.
Before proceeding in the analysis of the derivative of the log-likelihood $\mathcal{L}_\mathcal{D}(\mathbf{w})$ w.r.t any weight w_{ij}, it is appropriate to introduce the equations for the partial derivatives of S. For every node n_j, let's define:

$$S_j^\partial(\mathbf{x}) = \frac{1}{S(x)} \cdot \frac{\partial S(\mathbf{x})}{\partial S_j(\mathbf{x})} = \begin{cases} S_r^\partial(\mathbf{x}) = \frac{1}{S(\mathbf{x})} \cdot \frac{\partial S(\mathbf{x})}{\partial S_r(\mathbf{x})} = \frac{1}{S(\mathbf{x})} & \text{if } n_j \text{ is the root } r \\ \frac{1}{S(\mathbf{x})} \cdot \sum\limits_{i \in Pa(j)} S_i^\partial(\mathbf{x}) \cdot \frac{\partial S_i(\mathbf{x})}{\partial S_j(\mathbf{x})} & \text{if } n_j \text{ is not the root} \end{cases} \tag{7}$$

Now the partial derivative that appears in the second case of Eq. 7 is defined as follows:

$$\frac{\partial S_i(\mathbf{x})}{\partial S_j(\mathbf{x})} = \begin{cases} \prod_{j' \in Ch_{-j}(i)} S_{j'}(\mathbf{x}) & \text{if } n_i \text{ is a product node} \\ w_{ij} & \text{if } n_i \text{ is a sum node} \end{cases} \tag{8}$$

where $Ch_{-j}(i)$ indicates the set of children of n_i except n_j. At this point, putting together the Equations 7 and 8, we can state that, if n_i is a sum node,

$$S_j^\partial(\mathbf{x}) = \sum_{i \in Pa(j)} w_{ij} \cdot S_i^\partial(\mathbf{x}) \,, \tag{9}$$

instead, if n_i is a product node,

$$S_j^\partial(\mathbf{x}) = \sum_{i \in Pa(j)} S_i^\partial(\mathbf{x}) \cdot \prod_{j' \in Ch_{-j}(i)} S_{j'}(\mathbf{x}) \,. \tag{10}$$

Finally, the partial derivative of $\mathcal{L}_\mathcal{D}$ that appears in Eq. 6 can be computed using the S_i^∂ defined above:

$$\frac{\partial \mathcal{L}_\mathcal{D}(\mathbf{w})}{\partial w_{ij}} = \sum_{t=1}^{T} \frac{\partial \log S(\mathbf{x}_t)}{\partial w_{ij}} \tag{11}$$

where

$$\begin{aligned}
\frac{\partial \log S(\mathbf{x}_t)}{\partial w_{ij}} &= \frac{1}{S(\mathbf{x}_t)} \cdot \frac{\partial S(\mathbf{x}_t)}{\partial w_{ij}} && \text{(derivative of the } \log \text{ times the derivative of its argument)} \\
&= \frac{1}{S(\mathbf{x}_t)} \cdot \frac{\partial S(\mathbf{x}_t)}{\partial S_i(\mathbf{x}_t)} \cdot \frac{\partial S_i(\mathbf{x}_t)}{w_{ij}} && \text{(applying the chain rule)} \\
&= S_i^{\partial}(\mathbf{x}_t) \cdot S_j(\mathbf{x}_t) && \text{(using Eq. 7, Def. 1 and basic derivation rules)}
\end{aligned}$$

To conclude, it is now possible to assert the final form of the partial derivative of the log-likelihood:

$$\frac{\partial \mathcal{L}_{\mathcal{D}}(\mathbf{w})}{\partial w_{ij}} = \sum_{t=1}^{T} S_i^{\partial}(\mathbf{x}_t) \cdot S_j(\mathbf{x}_t) \; . \tag{12}$$

Since $S_j(\mathbf{x}_t)$ can be computed in a single upward pass and $S_i^{\partial}(\mathbf{x}_t)$ in a single downward pass, Equation 12 allows to perform each step of GD (i.e. each weight update) in a time proportional to the size of the SPN and the number of instances in the dataset.
Note that this automatically ensures that the weights are normalized through the $\frac{1}{S(\mathbf{x})}$ introduced in Eq. 7, therefore it is not necessary to perform a normalization after each weight update. If this is not desired, it is simply possible to remove $\frac{1}{S(x)}$ from Eq. 7 and the rest of the derivation can stay the same. Anyway normalized weights tend to provide better results.

4.1.2 Expectation Maximization (EM)

Expectation Maximization (EM) is an iterative algorithm that can be used to find the maximum likelihood of the parameters in a probabilistic model when this model depends also on some hidden variables. Each iteration alternates between an E-step (expectation), which estimates the expected value of the log-likelihood using the current parameters, and an M-step (maximization), which updates the parameters maximizing the expected log-likelihood found in the E-step.
SPNs can also learn their parameters using EM, by viewing each sum node n_i as the result of summing out a corresponding hidden variable Y_i whose values corresponds to the children of n_i. Now the inference in Algorithm 1 can be seen as the E-step, which computes the marginal probabilities of the various hidden variables Y_i. The weight update, instead, is the M-step, which adds each hidden variable's marginal to its sum from the previous iteration. The weights are then obtained through a normalization.

4.1.3 Hard EM

Unfortunately, both gradient descent (Sec. 4.1.1) and Expectation Maximization (Sec. 4.1.2) do not work well when applied to SPNs. The key problem for both algorithms is the well-known *vanishing gradient* (a.k.a. *gradient diffusion*), which is a common phenomenon in deep learning, where adding more and more layers makes the gradient signal become weaker and weaker until it is so close to zero that the weights stop updating, blocking the learning process. Since in GD there is a direct link between the magnitude of the gradient and the intensity of the weight update (see Eq. 6), it is straight forward to see why this algorithm is strongly affected by the problem. Concerning EM, instead, the reason why it suffers from the same phenomenon is because, for every node n_i that is child of a sum node n_k, $P(Y_k|\mathbf{e}) \propto \partial S(\mathbf{e})/\partial S_k(\mathbf{e})$ [1], and therefore the updates in the M-step could become very small as well.
To overcome this difficulty, Poon and Domingos [1] proposed the so-called Hard EM, where marginal inference is replaced with MPE inference (Sec. 3.2), selecting for each hidden variable the most probable state. Algorithm 1 now maintains a count for each sum child, and the M-step simply increments the count of the winning child. The weights are obtained by normalizing the counts.

4.2 Discriminative parameters learning

In Section 4.1 have been introduced some algorithms that were first proposed by Poon and Domingos [1] for generatively training SPNs. Yet, Domingos himself, in one of his later works with Robert Gens [3], argued that it is generally observed that discriminative learning fares better. This section is heavily based on [3] and the main idea that will be discussed is to optimize $P(\mathbf{X}|\mathbf{Y})$ instead of

$P(\mathbf{X}, \mathbf{Y})$. In standard PGMs, such as conditional random fields, doing this allow a join inference over the variables $\mathbf{Y}$ while allowing for flexible features over the observable inputs $\mathbf{X}$. Unfortunately, the conditional partition function is just as intractable as it was with generative training. For this reason, Gens and Domingos [3] thought about introducing discriminative training in SPNs to be able to combine flexible features with fast and exact inference, combining the advantages of SPNs with those of discriminative models. The authors define an SPN $S[\mathbf{y}, \mathbf{h}|\mathbf{x}]$ that takes as input three disjoint sets of variables $\mathbf{X}$ (given and observable), $\mathbf{Y}$ (query variables) and $\mathbf{H}$ (hidden variables). When all the indicators of a set of variables, for example $\mathbf{H}$, are set to 1, the above SPN will be indicated as $S[\mathbf{y}, \mathbf{1}|\mathbf{x}]$ (where $\mathbf{1}$ is a vector) and this is equivalent to the marginalization $\sum_{\mathbf{h}} \Phi(\mathbf{y}, \mathbf{h}|\mathbf{x})$ where Φ indicates the network polynomial (Def. 3). Therefore, when marginalizing, it is possible to compute the whole sum over states of a variable in a single upward pass by setting all its indicators to 1. The given variables $\mathbf{X}$ will be treated as constants, therefore there will not be any sum over the states of $\mathbf{X}$ and those can be ignored in the scope of a node when considering completeness and consistency, allowing for a wider variety of architectures compared to generative training. The conditional probability is modelled by discriminative SPNs as follows:

$$P(\mathbf{y}|\mathbf{x}) = \frac{\Phi(\mathbf{y}|\mathbf{x})}{\sum_{\mathbf{y}'} \Phi(\mathbf{y}'|\mathbf{x})} = \frac{\sum_{\mathbf{h}} \Phi(\mathbf{y}, \mathbf{h}|\mathbf{x})}{\sum_{\mathbf{y}', \mathbf{h}} \Phi(\mathbf{y}', \mathbf{h}|\mathbf{x})} \tag{13}$$

where Φ indicates the network polynomial (Def. 3), thus an unnormalized probability distribution.

4.2.1 Discriminative gradient descent

The weight update rule (Eq. 6) of generative gradient descent stays of course the same in the discriminative case, but this time the log-likelihood is conditional: $\mathcal{L}(\mathbf{x}|\mathbf{y}) = \log P(\mathbf{y}|\mathbf{x})$.

$$w_{ij}^{(s+1)} = w_{ij}^{(s)} + \gamma \frac{\partial \mathcal{L}(\mathbf{x}|\mathbf{y})}{\partial w_{ij}^{(s)}} \tag{14}$$

Now, the partial derivative of the conditional log-likelihood is defined as follows:

$$\frac{\partial \mathcal{L}(\mathbf{x}|\mathbf{y})}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \log P(\mathbf{y}|\mathbf{x}) = \frac{1}{S[\mathbf{y}, \mathbf{1}|\mathbf{x}]} \frac{\partial S[\mathbf{y}, \mathbf{1}|\mathbf{x}]}{\partial w_{ij}} - \frac{1}{S[\mathbf{1}, \mathbf{1}|\mathbf{x}]} \frac{\partial S[\mathbf{1}, \mathbf{1}|\mathbf{x}]}{\partial w_{ij}} \tag{15}$$

The partial derivatives of the SPN can be computed as shown in Section 4.1.1. Technically, the authors [3] did not include the normalization in their partial derivatives, but it was seen as an optional step to perform after each weight update. Their backpropagation algorithm is reported in the Appendix A.1 (Algorithm 3).

4.2.2 Hard gradient descent

Hard gradient descent is based on discriminative gradient descent (Sec. 4.2.1) but the marginal inference is replaced with MPE inference (Sec. 3.2), similarly to EM and Hard EM (Sections 4.1.2 and 4.1.3). Gens and Domingos [3] state that there are various reasons why MPE inference is appealing for discriminative training in SPNs:

- first, hard inference was crucial in tackling the vanishing gradient in generative training;
- second, for many applications the goal is to predict the most probable structure, thus it makes sense to use this also during training;
- and finally, even though summations in SPNs are efficient and exact, approximating those with maxes makes them even faster.

Let's define the *max-product network* (MPN) $M[\mathbf{y}, \mathbf{h}|\mathbf{x}]$ that compactly represents the maximizer polynomial $\max_{\mathbf{x}} \Phi(\mathbf{x}) \prod(\mathbf{x})$, which computes the MPE. To convert an SPN to an MPN it is necessary to simply replace sum nodes with max nodes. The gradient of the conditional log-likelihood with MPE w.r.t a weight w_{ij} is then shown in Equation 16:

$$\begin{aligned}\frac{\partial}{\partial w_{ij}} \log \tilde{P}(\mathbf{y}|\mathbf{x}) &= \frac{\partial}{\partial w_{ij}} \log \max_{\mathbf{h}} \Phi(\mathbf{y}, \mathbf{h}|\mathbf{x}) - \frac{\partial}{\partial w_{ij}} \log \max_{\mathbf{y}', \mathbf{h}} \Phi(\mathbf{y}', \mathbf{h}|\mathbf{x}) \\ &= \frac{\partial}{\partial w_{ij}} \log M[\mathbf{y}, \mathbf{1}|\mathbf{x}] - \frac{\partial}{\partial w_{ij}} \log M[\mathbf{1}, \mathbf{1}|\mathbf{x}]\end{aligned} \tag{16}$$

and the partial derivative of the logarithm on an MPN w.r.t. a weight w_{ij} takes the form

$$\frac{\partial \log M}{\partial w_{ij}} = \frac{\partial \log M}{\partial M} \frac{\partial M}{\partial w_{ij}} = \frac{1}{M} \frac{\partial M}{\partial w_{ij}} = \frac{c_{ij}}{w_{ij}} \tag{17}$$

where c_{ij} is the number of times w_{ij} connects a sum/max node n_i with its winning child[2] n_j. Finally, the gradient of the conditional log-likelihood with MPE inference is

$$\frac{\partial}{\partial w_{ij}} \log \tilde{P}(\mathbf{y}|\mathbf{x}) = \frac{\Delta c_{ij}}{w_{ij}} = \frac{c'_{ij} - c''_{ij}}{w_{ij}} \tag{18}$$

where c'_{ij} and c''_{ij} are the number of times w_{ij} is traversed by the two MPE inference paths in $M[\mathbf{y}, \mathbf{1}|\mathbf{x}]$ and $M[\mathbf{1}, \mathbf{1}|\mathbf{x}]$. A graphical example is provided in Figure 2.

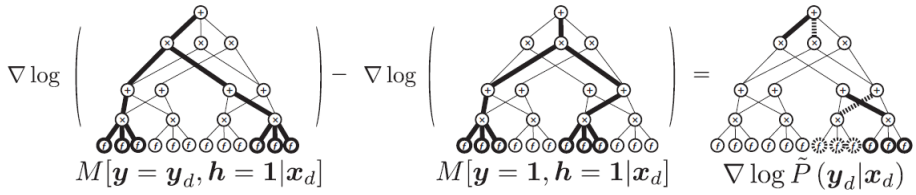

Figure 2: Example of the computation of the gradient of the conditional log-likelihood with MPE inference (a.k.a. hard gradient). Source: [3].

4.3 Structure learning

Structure learning consists in finding the optimal (or near-optimal) architecture/graph for an SPN. Poon and Domingos [1] proposed a way to generate the initial structure for a network, this was hand-coded and designed for image data, assuming neighborhood dependence, and it explained more in depth in Section 4.3.1. This method does not properly count as "learning", since it simply generates a graph as a first action, before feeding the network with any instance from a dataset. This process is what is called `GenerateDenseSPN` in Algorithm 1, and if we really want to see a "structure learning" in it, we might consider the combination of `GenerateDenseSPN` and `PruneZeroWeights` as such. Subsequently, more advanced methods were proposed in the literature, and one of the most popular and used, especially if we count the many variants that came out over the years, is *LearnSPN*, and it is explained in the following Section 4.3.2.

4.3.1 Generate initial structure for SPN learning

Poon and Domingos, in the paper that introduced SPNs [1], propose a general scheme for producing the initial architecture. It is designed for image processing and leverages the local structure in image data assuming neighborhood dependence. It works as follows:

1. Select a set of subsets of variables.
2. For each subset R, create k sum nodes $S_1^R, \ldots, S_k^R$ and select a **set** of ways to decompose R into selected subsets $R_1, \ldots, R_l$ ($l \leq k$).
3. For each of these decompositions, and $\forall R_i$, create a product node with parents S_j^R and children $S_1^{R_1}, \ldots, S_l^{R_l}$.

[2]The meaning of "winning child" is better explained in the the section about MPE inference (Sec. 3.2).

It is required that only a polynomial number of subsets is selected, and for each subset only a polynomial number of decompositions is chosen.
A practical example is given in the *Experiments* section of the original paper [1] and it is also summarized in Section 5 of this report.

4.3.2 LearnSPN

This method was proposed by Gens and Domingos in their work *Learning the Structure of Sum-Product Networks* from 2013 [4] and the main procedure is reported in Algorithm 2.
The input is supposed to be in the classic matrix form, where the columns are the variables and the rows are the various instances, expressed as vectors where each item contains a realization/value for a specific variable. If these vectors are of unit length, the algorithm returns the corresponding univariate distribution using MLE (i.e. creates a terminal node), otherwise it recurses on sub-matrices. If it is able to split the variables into mutually independent subsets, it recurses on those and returns the product of the resulting SPNs. If this splitting fails, the algorithm clusters similar instances into subsets, recurses on those and returns the weighted[3] sum of the resulting SPNs. These splittings and clusterings are made more clear in Figure 3.
Like Algorithm 1, LearnSPN is more of a schema that can be realized in many variants, rather than a single specific algorithm. In fact, there may be many ways to split the variables and to cluster the instances in the dataset. For the former, the authors propose to use the pairwise independence between variables as splitting criterion, while for the clustering of similar instances they use Expectation Maximization (EM).

Algorithm 2 LearnSPN

Input: Set T of instances over variables $\mathbf{X}$.
Output: An SPN representing a distribution over $\mathbf{X}$ learnt from T
if $|\mathbf{X}| = 1$ **then**
  return univariate distribution from the variables' values in T
else
  partition $\mathbf{X}$ into approximately independent subsets $\mathbf{X}_j$
  if success **then**
    return $\prod_j$ LearnSPN$(T, \mathbf{X}_j)$
  else
    partition T into subsets of similar instances T_i
    return $\sum_i \frac{|T_i|}{|T|} \cdot$ LearnSPN$(T_i, \mathbf{X})$
  end if
end if

Splitting the variables This sub-task can be accomplished in a number of ways as well, but the one proposed in the original paper of LearnSPN is the following: apply an independence test[4] to each pair of variables, build a graph with a node for each variable and a link between each couple of variables that are found to be dependent, finally recurse on each connected component. If the graph has only a single connected component, the split fails, and the algorithm proceeds clustering the instances as explained in the next paragraph.

Clustering the instances Clustering instances into subsets depending on their similarity with one another can be done through the EM algorithm. At each splitting it is assumed a naive Bayes mixture model where all variables are independent conditioned on the cluster: $P(\mathbf{X}) = \sum_i P(C_i) \prod_j P(X_j|C_i)$, where C_i is the i-th cluster and X_j the j-th variable.
For soft EM, since the instances could be assigned to different clusters with different probabilities, T would need to be extended with a weight for each instance, however this is considerably less efficient than hard EM [4]. The authors, in fact, use hard incremental EM, which automatically determines the number of clusters by assigning each instance to its most likely cluster, possibly a new one.

[3] Weighted by the fraction of instances in the corresponding subset.

[4] G-Test of pairwise independence: $G(x_1, x_2) = 2 \sum_{x_1} \sum_{x_2} c(x_1, x_2) \cdot \log \frac{c(x_1,x_2) \cdot |T|}{c(x_1)c(x_2)}$, where $c(\cdot)$ counts the occurrences of a setting of a variable pair or singleton.

Overfitting is avoided by placing an exponential prior[5] on the number of clusters as a regularization method.

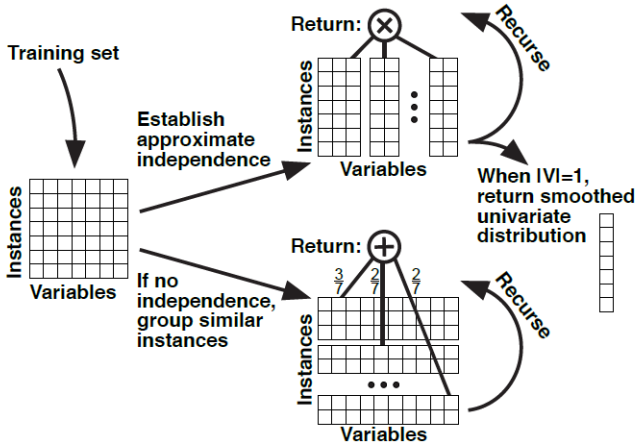

Figure 3: Explanatory image of Algorithm 2 (LearnSPN).

5 Applications of SPNs

In this section some meaningful and successful applications of SPNs will be presented. These models were first applied to image processing tasks such as image completion or classification, but subsequently their potential has been exploited in fields like natural language processing, activity recognition and more.

5.1 Image completion

Poon and Domingos, in the first SPNs paper [1], tested these models on the problem of completing images. This is a challenging benchmark in computer vision, being a difficult task where detecting deep structure is fundamental. To make the task even more difficult, in their experiments, the authors occluded half of the image, instead of the usual small area that is common practice to do in many experiments of this kind.

Experiments setup The designated datasets are Caltech-101 [12] and the Olivetti face dataset (introduced in [13]). They normalized the intensities of the gray-scale input images to have zero mean and unit variance, and treated each pixel's variable X_i as a sample from a Gaussian mixture with k unit-variance components. For each pixel, the intensities of training examples were divided into k equal quantiles and the mean of each component is set to that of the corresponding quantile. k was set to 4, but increasing its value yielded no improvements.
To generate the initial structure of the network, they used the method explained in Section 4, obtaining an architecture with 36 layers (in general there are $2(d-1)$ layers for dxd images). In all these images, all rectangular regions are selected, with the smallest ones being a single pixel, and for each of these rectangular regions all the possible ways to decompose them into two rectangular sub-regions are considered . They have also noted that SPNs can adopt multiple levels of resolution, where larger regions adopt coarser decompositions. This can make learning much faster with a negligible loss in accuracy. Specifically, they used a coarse resolution of 4x4 and a finer one only within each 4x4 region.
To learn the weights, they used minibatch hard EM. They noticed that running soft EM before hard EM yields no improvement and the best results were obtained using sums in the upward passes and maxes in the downward passes.

[5] $P(S) \propto e^{-\lambda C|\mathbf{X}|}$, where C is the number of clusters, λ the cluster penalty and $|\mathbf{X}|$ the number of variables.

Comparison with other models SPNs have been compared on this task with other state-of-the-art deep architectures with a probabilistic interpretation such as Deep Belief Networks (DBN) [14] and Deep Boltzmann Machines (DBMs) [15]. Furthermore, as a baseline, the authors made a comparison with some classical methods like nearest neighbor and PCA (with 100 principal components, results with a higher or lower number were similar). The metric of evaluation was the mean squared error (MSE) of the completed pixels and the main results are reported in Table 1 (note that the DBN results are not directly comparable with the others[6]). Furthermore, in Figure 4 are depicted some sample reconstructions of images taken from the Olivetti dataset.
Overall, SPNs outperforms all other methods by a wide margin. PCA performs surprisingly well in terms of MSE, but its completions are often quite blurred since they are a linear combination of prototypical images. Nearest neighbor can give good completions if there is a similar image in the training set, but in general their completions can be quite poor. Compared to DBNs and DBMs, Poon and Domingos [1] state the following main advantages of SPNs:

- SPNs are simpler and potentially more powerful. They can perform efficient exact inference, while DBNs and DBMs require approximations. Furthermore, the problem of vanishing gradient limits most DBNs and DBMs to few layers, whereas with hard EM, SPNs can become quite deep. In substance, DBNs and DBMs need much more engineering: they require a careful choice of the hyperparameters - while SPNs are much more flexible[7] - and the number of iterations during training must be deterined empirically using an apposite development set - while in SPNs the learning simply terminates when the log-likelihood stops improving.

- SPNs are an order of magnitude faster in both training and inference, and all the results are exact. In contrast, in DBNs and DBMs estimating marginals requires many Gibbs sampling steps and the results are approximate anyway.

LEFT	DPN	DBM	DBN	PCA	NN
Caltech (ALL)	3551	9043	4778	4234	4887
Face	1992	2998	4960	2851	2327
Heicopter	3284	5935	3353	4056	4265
Dolphin	2983	6898	4757	4232	4227
Olivetti	942	1866	2386	1076	1527

BOTTOM	DPN	DBM	DBN	PCA	NN
Caltech (ALL)	3270	9792	4492	4465	5505
Face	1828	2656	3447	1944	2575
Heicopter	2801	7325	4389	4432	7156
Dolphin	2300	7433	4514	4707	4673
Olivetti	918	2401	1931	1265	1793

Table 1: Mean squared errors on completed image pixels in the left or bottom half. NN is nearest neighbor. The upper table contains the results when the left half of the image was occluded, while in the lower one the bottom half was.

[6] Using the original images and without additional preprocessing, the learned DBN gave very poor results. In the original paper of DBNs, Hinton and Salakhutdinov [14] used reduced-scale images (25×25) compared to the original size of 64×64. By converting to the reduced scale and initializing with their learned model, the results improve significantly, so Poon and Domingos [1] reported these results instead.

[7] Poon and Domingos [1] state that the hyperparameters configurations that they used in SPNs in preliminary analyses were already providing good results for all datasets, and they also used the same initial architecture for all the experiments.

Figure 4: Sample face completions. Top to bottom: original, SPN, DBM, DBN, PCA, nearest neighbor. The first three images are from Caltech-101, the rest from Olivetti.

5.2 Activity recognition

Amer and Todorovic [16] developed some PSN-based methods that were able to perform activity recognition in videos. They call each meaningful piece of an image a *visual word* and they extract these using a neural network. These visual words lie in a 3D grid with dimensions width, height and time, and every window in the grid is modeled as a *bag of words* (BoW), i.e. a histogram on the associated visual words. Each bag of words is treated as a random variable that can assume two states: foreground and background. Product nodes in the SPN represent combinations of sub-activities into more complex ones (e.g. "join hand" + "separate hands" = "clapping") and sum nodes represent variations of the same activity. A schema of the procedure is illustrated in Figure 5.
An SPN is trained for each activity in a supervised way, where foreground and background values are known, and in a weakly supervised way, where only the activity is known. The initial structure of the SPN is an almost completely connected graph, which then gets pruned after learning the parameters, following the same general schema of Algorithm 1. The parameters learning is an iterative process where the following two actions are repeated:

- learn SPN's weights through GD from the parameters of the BoW
- learn the parameters of the BoW from the weights of the SPN (using variational methods)

This approach in general led to better results on a number of datasets when compared to state-of-the-art methods that do not use SPNs.

5.3 Natural language processing

Bandwidth extension Peharz et al. [18] applied SPNs to the task of retrieving the audio frequencies that gets lost through the restricted-bandwidth telephone communications. Note that in this problem real-time inference is essential. The authors used a hidden Markov model (HMM) to represent the temporal evolution of the spectrum of the audio signals, then clustered the data and finally trained an SPN for each cluster. These were used to retrieve (actually generate) the lost frequencies through MPE inference (Sec. 3.2). They obtained better results than state-of-the-art algorithms in terms of spectral distortion between the ground truth audio tracks and the same tracks whose bandwidth was recovered after passing through a telephone communication. Furthermore, a subjective listening test confirmed the better performance of this SPN-based method.

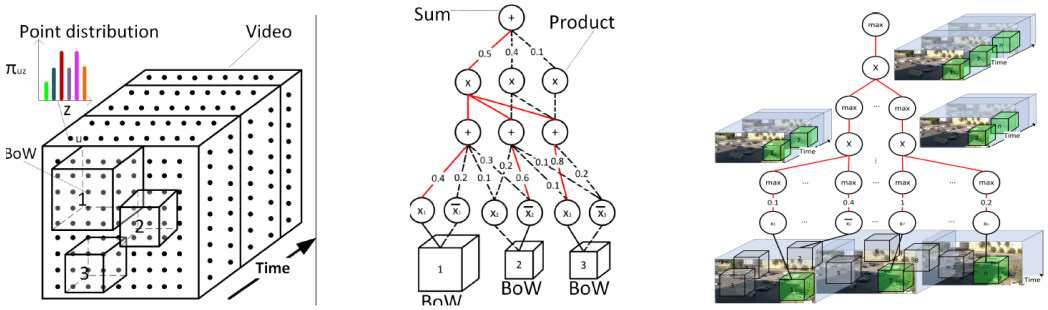

Figure 5: General schema of the procedure for activity recognition described in [16]. The first image shows a video represented as the counting grid of visual words, where 3 bags of words are depicted as boxes.ùThe second figure represents the SPN over the bags of words. The third figure shows the localization of the activity "unloading of the trunk" in an example sequence from the VIRAT dataset [17].

Language modelling Typically, the probability distribution over a sequence of words is modelled as $P(\mathbf{w}_{1:m}) \approx \prod_{k=1}^{m} P(w_k|\mathbf{w}_{k-n+1\,:\,k-1})$, where $\mathbf{w}_{i:j}$ denotes a sequence of words $w_i, \ldots, w_j$. The hard part in language modelling is this conditional probability of a word given the previous ones. Cheng et al. [19] used a discriminative SPN [3] (Sec. 4.2) to exactly and efficiently model this distribution. The leaf nodes are vectors representing the N previous words in the sequence, they are therefore evidence variables and they are expressed in 1-hot encoding. The next layer contains sum nodes and it can be interpreted as compressing each 1-hot encoded vector into a smaller one with continuous values, similarly to what a feed-forward NN would do. The last layer before the root contains product nodes, each of them is connected to one node in the previous layer and one indicator variable y_k representing the i-th word we are predicting: $y_k = 1$ if $i = k$, otherwise $y_k = 0$. In the end, the root is connected to all the nodes in the previous layer and has normalized weights to output an actual conditional probability. Given this structure (shown in Figure 6 in the Appendix A.2), the parameters learning is performed with discriminative GD (Sec. 4.2.1). This method was able to predict the next word better than classic NLP models such as feed-forward and recurrent NNs. However, since it has been introduced earlier, it has not been tested against transformers [20], which are currently the state-of-the-art in NLP.

6 Sum-Product Networks vs. Neural Networks

SPNs can be seen as a particular type of feed-forward neural networks (NNs) because there is a flow of information from the input units (leaf nodes) to the output one (root). Furthermore, in various tasks they can be used as an alternative to each other (see Sec. 5). The main difference is that SPNs have a strong probabilistic interpretation, while NNs do not really have an obvious one: SPNs take random variables[8] as inputs and give probabilities as outputs, while in NNs the output might indeed be framed probabilistically (e.g. sigmoid as output activation function for binary classification), but the inputs are often just raw numbers and the hidden units lack a direct probabilistic interpretation. On the other hand, in SPNs, being defined recursively (Def. 8), each node - not only the root - computes a probability distribution.

6.1 Inference

Inference is also different: while in feed-forward NNs it always consists of a single forward pass (that could be seen as the equivalent of an upward pass in SPNs), in SPNs there are different scenarios depending on what type of probability distribution is being computed. Specifically, computing marginals $P(X_i)$ requires as well a single upward pass, but for posteriors $P(\mathbf{X}|\mathbf{E})$ two passes are necessary, while MPE (Sec. 3.2) requires a backtracking from the root to the leaves. Furthermore, SPNs can perform inference with partial information, i.e. when the values of some variables are

[8] Actually they take indicators or components of random variables.

unknown (by setting all their indicators to 1), while NNs always require that each input unit has an "observable" value.

6.2 Training

NNs are usually trained with gradient descent-based approaches, while SPNs not only can do the same [1, 3, 21] [9], but can also learn through probabilistic algorithms such as EM (Sec. 4.1.2).
Another major difference between these two models is the fact that SPNs can learn their structure from data (Sec. 4.3.2), looking for a balance between complexity and accuracy. NNs, instead, are usually designed by hand and it is necessary to evaluate different architectures with a trial-and-error approach, which is extremely inefficient. Furthermore, this is probably the reason why the NNs that have succeeded in practical applications are usually very big and training them requires a huge computational power (this for every trial that has been done in order to determine the final architecture and hyperparameters configuration).

6.3 Performance

Despite these advantages, NNs tend still to be superior in various tasks. For example, in 2012 Poon and Domingos achieved an accuracy of 84% on CIFAR-10, which is impressive for the time, especially if we consider that SPNs were (and arguably still are) a less mature model than NNs, benefitting from less well-established learning practices and fewer heuristics and tricks to improve the performances. However, deep NNs have surpassed 99% accuracy[10]. Nevertheless, RAT-SPNs [21], a particular type of SPN with randomized architecture trained with NN-style methods (auto-differentiation, SGD and GPU parallelization), achieved comparable results on MNIST [22], Fashion-MNIST [23] and 20-NG [24]. Furthermore, SPNs and NNs could also be used complementary, like in the case of Wang and Wang [25], where a convolutional NN was used to isolate parts of images before applying an SPN for the task of activity recognition from images. In a similar domain, also Amer and Todorovic [16], in the method that was explained also in Section 5.2 of this report, extracted the visual words with a NN.

7 Conclusion

Sum-Product Networks (SPNs) are a class of probabilistic models that make exact inference tractable, unlike probabilistic graphical models (PGMs), such as Bayesian networks, that require approximations (like variational inference) or specific ways of modelling the dependencies among the random variables. SPNs are also related to neural networks (NNs) and have been applied to similar tasks, especially image and natural language processing. Despite a noticeable performance advantage of SPNs over PGMs (e.g. Section 5.1), deep NNs tend still to be better in many cases, especially in tasks such as image classification, where convolutional NNs are the state of the art (Section 6.3). However, SPNs offer the possibility of learning their structure automatically from data and learning the weights through a wider variety of algorithms: both generative and discriminative gradient descent, as well as other probabilistic algorithms like Expectation-Maximization. Furthermore, recent advances in the field, like RAT-SPNs [21], are getting closer to the performance of neural networks.
In conclusion, SPNs have many advantages over other classes of models and are theoretically more grounded. They have shown a great potential, even though their introduction is relatively recent. Still, for now their adoption has been somewhat limited, probably also due to the constraints they impose on their architecture. In addition they tend to be much bigger (more nodes and links) when compared to an equivalent PGM, which might result to be more comprehensible (e.g. Figure 1 in [5]). The understanding of SPNs has evolved and improved since their introduction, and with time they are probably going to be used more and more, as learning practices and heuristics become more well-established and new properties and techniques get descovered.

[9] Even though in [1] SGD suffered heavily from vanishing gradient, prompting the authors to rely on hard EM to get decent results, Gens and Domingos [3] proposed hard gradient descent for SPNs (Sec. 4.2.2) and Peharz et al. [21] proposed a particular type of SPN (RAT-SPN) trained with SGD. Both obtained good results.

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

References

[1] Hoifung Poon and Pedro Domingos. Sum-product networks: A new deep architecture. In *Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence*, UAI'11, page 337–346, Arlington, Virginia, USA, 2011. AUAI Press.

[2] Adnan Darwiche. A differential approach to inference in bayesian networks. *J. ACM*, 50(3):280–305, may 2003.

[3] Robert Gens and Pedro Domingos. Discriminative learning of sum-product networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[4] Robert Gens and Domingos Pedro. Learning the structure of sum-product networks. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 873–880, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.

[5] I. París, R. Sanchez-Cauce, and F. Diez. Sum-product networks: A survey. *IEEE Transactions on Pattern Analysis Machine Intelligence*, (01), feb 2020.

[6] Robert Peharz, Robert Gens, Franz Pernkopf, and Pedro Domingos. On the latent variable interpretation in sum-product networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(10):2030–2044, 2017.

[7] Robert Peharz, Sebastian Tschiatschek, Franz Pernkopf, and Pedro Domingos. On Theoretical Properties of Sum-Product Networks. In Guy Lebanon and S. V. N. Vishwanathan, editors, *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics*, volume 38 of *Proceedings of Machine Learning Research*, pages 744–752, San Diego, California, USA, 09–12 May 2015. PMLR.

[8] Cory J. Butz, Jhonatan S. Oliveira, André E. dos Santos, André L. Teixeira, Pascal Poupart, and Agastya Kalra. An empirical study of methods for spn learning and inference. In Václav Kratochvíl and Milan Studený, editors, *Proceedings of the Ninth International Conference on Probabilistic Graphical Models*, volume 72 of *Proceedings of Machine Learning Research*, pages 49–60. PMLR, 11–14 Sep 2018.

[9] Jun Mei, Yong Jiang, and Kewei Tu. Maximum a posteriori inference in sum-product networks. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence*, AAAI'18/IAAI'18/EAAI'18. AAAI Press, 2018.

[10] Robert Peharz, Robert Gens, and Pedro Domingos. Learning selective sum-product networks. In *31st International Conference on Machine Learning (ICML) - Learning Tractable Probabilistic Models (LTPM) Workshop*, 2014.

[11] Robert Peharz. *Foundations of Sum-Product Networks for Probabilistic Modeling*. PhD thesis, 2015.

[12] Caltec-101 dataset. `http://www.vision.caltech.edu/Image_Datasets/Caltech101/`. Accessed: 2022-01.

[13] F.S. Samaria and A.C. Harter. Parameterisation of a stochastic model for human face identification. In *Proceedings of 1994 IEEE Workshop on Applications of Computer Vision*, pages 138–142, 1994.

[14] G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[15] Ruslan Salakhutdinov and Geoffrey Hinton. Deep boltzmann machines. In David van Dyk and Max Welling, editors, *Proceedings of the Twelth International Conference on Artificial Intelligence and Statistics*, volume 5 of *Proceedings of Machine Learning Research*, pages 448–455, Hilton Clearwater Beach Resort, Clearwater Beach, Florida USA, 16–18 Apr 2009. PMLR.

[16] Mohamed R. Amer and Sinisa Todorovic. Sum product networks for activity recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 38(4):800–813, 2016.

[17] Sangmin Oh, Anthony Hoogs, Amitha Perera, Naresh Cuntoor, Chia-Chih Chen, Jong Taek Lee, Saurajit Mukherjee, J. K. Aggarwal, Hyungtae Lee, Larry Davis, Eran Swears, Xioyang Wang, Qiang Ji, Kishore Reddy, Mubarak Shah, Carl Vondrick, Hamed Pirsiavash, Deva Ramanan, Jenny Yuen, Antonio Torralba, Bi Song, Anesco Fong, Amit Roy-Chowdhury, and Mita Desai. A large-scale benchmark dataset for event recognition in surveillance video. In *CVPR 2011*, pages 3153–3160, 2011.

[18] Robert Peharz, Georg Kapeller, Pejman Mowlaee, and Franz Pernkopf. Modeling speech with sum-product networks: Application to bandwidth extension. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3699–3703, 2014.

[19] Wei Cheng, Stanley Kok, Hoai Vu Pham, Hai Leong Chieu, and Kian Ming Adam Chai. Language modeling with sum-product networks. In *INTERSPEECH*, 2014.

[20] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

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

[22] Yann LeCun and Corinna Cortes. The mnist database of handwritten digits. 2005.

[23] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017.

[24] 20 news groups (20-ng). `https://archive.ics.uci.edu/ml/datasets/Twenty+Newsgroups`. Accessed: 2022-01.

[25] 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, 11 2015.

A Appendix

A.1 BackpropSPN

The following is the algorithm to perform backpropagation in SPNs as reported in the paper from Gend and Domingos [3]:

Algorithm 3 BackpropSPN

Input: A valid SPN S, where S_n demptes the value of node n after bottom-up evaluation.
Output: Partial derivatives of the SPN with respect to every node $\frac{\partial S}{\partial S_n}$ and weight $\frac{\partial S}{\partial w_{i,j}}$

Initialize all $\frac{\partial S}{\partial S_n} = 0$ except $\frac{\partial S}{\partial S} = 1$
for all $n \in S$ in top-down order **do**
  if n is a sum node **then**
    for all $j \in Ch(n)$ **do**
      $\frac{\partial S}{\partial S_j} \leftarrow \frac{\partial S}{\partial S_j} + w_{n,j} \frac{\partial S}{\partial S_n}$
      $\frac{\partial S}{\partial w_{n,j}} \leftarrow S_j \frac{\partial S}{\partial S_n}$
    end for
  else
    for all $j \in Ch(n)$ **do**
      $\frac{\partial S}{\partial S_j} \leftarrow \frac{\partial S}{\partial S_j} + \frac{\partial S}{\partial S_n} \prod_{k \in Ch(n) \setminus \{j\}} S_k$
    end for
  end if
end for

A.2 Language modelling with SPNs

The following is the architecture used by Cheng et al. [19] to perform language modelling. Note that, after the leaves, there are two consecutive layers of sum nodes (instead of alternating sums and products), this because those leaves are evidence variables and discriminative SPNs do not consider completeness and consistency over those.

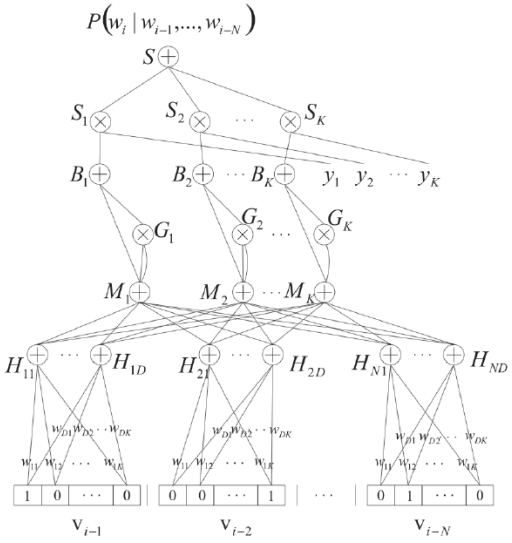

Figure 6: Architecture of an SPN for language modelling. Source: [19].