Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI 2015)

Greedy Structure Search for Sum-Product Networks

Aaron Dennis and **Dan Ventura**
Computer Science Department
Brigham Young University
Provo, UT 84602
adennis@byu.edu, ventura@cs.byu.edu

Abstract

Sum-product networks (SPNs) are rooted, directed acyclic graphs (DAGs) of sum and product nodes with well-defined probabilistic semantics. Moreover, exact inference in the distribution represented by an SPN is guaranteed to take linear time in the size of the DAG. In this paper we introduce an algorithm that learns the structure of an SPN using a greedy search approach. It incorporates methods used in a previous SPN structure-learning algorithm, but, unlike the previous algorithm, is not limited to learning tree-structured SPNs. Several proven ideas from circuit complexity theory along with our experimental results provide evidence for the advantages of SPNs with less-restrictive, non-tree structures.

1 Introduction

Sum-product networks (SPNs) are a recently-proposed class of probabilistic models in which exact inference is guaranteed to take linear time in the size of the model. They can efficiently represent a larger class of distributions than some other models such as mixture models and thin junction trees [Poon and Domingos, 2011]. SPN parameters can be learned using expectation-maximization or gradient descent in both the generative and discriminative settings [Poon and Domingos, 2011; Gens and Domingos, 2012].

Results [Gens and Domingos, 2012] on twenty real-world datasets compare SPNs to Bayesian networks learned using the WinMine toolkit [Chickering, 2002] and to Markov networks learned using two other methods [Della Pietra *et al.*, 1997; Ravikumar *et al.*, 2010]. In these experiments SPNs and graphical models fit the data with comparable likelihood, but the inference accuracy of SPNs is better, as measured using conditional-likelihood. Also, SPN inference is about two orders of magnitude faster. Similar results [Rooshenas and Lowd, 2014] were found when comparing an augmented SPN to mixtures of trees [Meila and Jordan, 2001] and latent tree models [Choi *et al.*, 2011].

SPNs are represented using a directed, acyclic graph (DAG) of sum and product nodes. Recent approaches to SPN learning focus on the structure of this graph along with its parameters [Rooshenas and Lowd, 2014; Gens and Domingos, 2013; Peharz *et al.*, 2013; Dennis and Ventura, 2012]. These algorithms add nodes in either a top-down or bottom-up fashion until a complete SPN is constructed. In contrast, the algorithm introduced in this paper uses a search procedure that incrementally expands a simple, but complete, SPN to produce a series of increasingly complex SPNs.

SPNs and multilinear arithmetic circuits (MACs), a model from circuit complexity theory, are closely related. Both are represented by DAGs whose internal nodes are sums and products and both compute multilinear polynomials in their leaf nodes. Multilinear arithmetic formulas (MAFs) are MACs whose DAG is a tree. Raz has shown that MACs are, in a sense, more powerful than MAFs: he proves a certain polynomial to be computable by a MAC of polynomial-size but only computable by a MAF of super-polynomial-size [Raz, 2006].

In a way this result is irrelevant to SPNs. With SPNs we are concerned with representing probability distributions, not computing polynomials. In other words, we care that an SPN computes a certain function, not that it uses a certain polynomial to do so. Still we conjecture that DAG-structured SPNs are more powerful than tree-structured SPNs with respect to their ability to compactly represent probability distributions.

We do not attempt to prove or disprove this conjecture here. Instead we introduce a greedy structure search algorithm that learns DAG-structured SPNs and compare this with a structure learning algorithm that learns tree-structured SPNs. Empirical results provide evidence that DAG-SPNs have advantages over tree-SPNs. We also prove a theorem that helps clarify and simplify certain SPN concepts and helps us define a useful approximate likelihood function for SPNs.

2 Previous Work

Delalleau & Bengio [2011] connect SPNs to work in both circuit complexity and deep learning. They provide theoretical evidence for the utility of deep learning by proving lower bounds on the size of shallow (depth two) sum-product networks for two classes of functions $\mathcal{F}$ and $\mathcal{G}$. They show an exponential separation in the size of shallow and deep SPNs; to compute functions in $\mathcal{F}$ a shallow SPN requires at least $2^{\sqrt{n}-1}$ nodes while a deep SPN requires only $n-1$ nodes.

Darwiche [2003] introduced the idea of representing a Bayesian network using polynomials, called network polynomials, and of computing these polynomials using arithmetic circuits. He also showed how to perform inference using network polynomials and their corresponding circuits. This work was foundational to the introduction of SPNs by Poon & Domingos [2011].

Lowd & Domingos [2008] proposed the first search algorithm for learning the structure of arithmetic circuits. Since an arithmetic circuit can be converted to an equivalent SPN, this work can be thought of as the first SPN structure search algorithm. Their algorithm builds an arithmetic circuit that maintains equivalence with a Bayesian network. A result of this equivalence constraint is that a single step in the search process can dramatically increase the size of the arithmetic circuit. Our search algorithm increases the size of an SPN by a modest amount at each step.

Several SPN structure learning algorithms have been proposed; these we denote and reference as follows: DV [Dennis and Ventura, 2012], GD [Gens and Domingos, 2013], RL [Rooshenas and Lowd, 2014], and PGP [Peharz *et al.*, 2013]. DV, GD, and RL are top-down SPN structure learning algorithms that start with a root node and recursively add children until a full SPN has been built. DV uses an ad-hoc clustering method to build several SPNs that are then merged together; GD and RL take a more principled approach that creates product node children using tests of independence and that creates sum node children by learning naive Bayes models. GD and RL construct trees, with the leaves of GD being univariate distributions and the leaves of RL being multivariate distributions [Lowd and Rooshenas, 2013]; the graphs constructed by DV and PGP are not restricted to trees. PGP uses a bottom-up approach, merging smaller SPNs into larger ones. Another approach learns an SPN structure by sampling from a prior over tree or DAG structures [Lee *et al.*, 2014]; no experimental results have been reported yet for this algorithm.

One of the difficulties in SPN structure learning has been taking advantage of the architectural flexibility of SPNs. GD and RL use well-justified algorithms but limit themselves to learning SPN trees—except at the leaf nodes in the case of RL, which can be arithmetic-circuit representations of Markov networks. One of the aims of this paper is to provide an algorithm that employs the principled approaches of GD and RL while learning DAG-structured SPN.

3 Sum-Product Networks

Indicators for a random variable X_i are defined as

$$\lambda_{X_i=j} = \begin{cases} 1 & \text{if } X_i = j \text{ or } X_i \text{ is unknown} \\ 0 & \text{otherwise} \end{cases}$$

where j is one of the values that X_i can take. An SPN can be represented as a rooted DAG whose leaf nodes are indicators and whose internal nodes are sums and products. If an indicator for X_i appears in an SPN then its *scope* contains X_i. Let $X = \{X_1, \ldots, X_m\}$ be the set of random variables in the scope of an SPN.

Definition 1. *A* sum-product network (SPN) *is:*

1. *an indicator node,*
2. *a product node whose children are SPNs with disjoint scopes, or*
3. *a sum node whose children are SPNs with the same scope, and whose edges to these children have non-negative weights.*

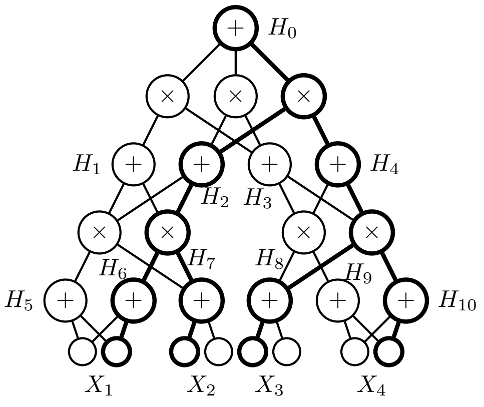

Figure 1: The bold lines show a complete sub-circuit. The circuit corresponds to variable settings $H_0{=}\,2$, $H_2{=}\,1$, $H_4{=}\,1$, $H_6{=}\,1$, $H_7{=}\,0$, $H_8{=}\,0$, $H_{10}{=}\,1$, $X_1{=}\,1$, $X_2{=}\,0$, $X_3{=}\,0$, and $X_4{=}\,1$.

We do not use the similar definition from Gens & Domingos [2013], even though it handles continuous variables more naturally, because Definition 1 simplifies the discussion in this paper. We assume without loss of generality that the children of product nodes are sum nodes and that the children of sums are either products or indicators.

Definition 2 (based on Chan & Darwiche [2006]). *A tree c embedded within an SPN is a* complete sub-circuit *if and only if it can be constructed recursively, starting from the root of c, by including all children of a product node p (and the edges connecting them to p), and exactly one child of a sum node s (and the edge connecting it to s). Let C be the set of all complete sub-circuits embedded in an SPN.*

An SPN computes a multilinear polynomial in which indicators appear as variables. For any node n in an SPN let f_n be the polynomial computed by it and let $\text{ch}(n)$ be its children. Let w_{st} be the weight on the edge between sum node s and its child t; we assume that $\sum_{t\in\text{ch}(s)} w_{st} = 1$. The root node r of an SPN computes polynomial f_r as follows. If r is the indicator node for $\lambda_{X_i=j}$ then $f_r = \lambda_{X_i=j}$; if r is a product node then $f_r = \prod_{t\in\text{ch}(r)} f_t$; if r is a sum node then $f_r = \sum_{t\in\text{ch}(r)} w_{rt} f_t$. We say that an SPN computes the polynomial computed by its root node. The polynomial computed by a complete sub-circuit is defined similarly. For input x and polynomial f_n we let $f_n(x)$ be the value of f_n at input x. For input x and g_c, the polynomial computed by complete sub-circuit c, we let $c(x) = g_c(x)$ be the value of g_c at input x.

A proof of the following theorem appears in the appendix.

Theorem 1. *For some SPN let r be its root and let g_c be the polynomial computed by $c \in C$. Then $f_r = \sum_{c\in C} g_c$.*

Using Theorem 1 we place a simple probabilistic interpretation on the computation of an SPN. For $c \in C$ the polynomial $g_c = \prod_{w\in W_c} w \prod_c(\lambda_X)$, where W_c is the multiset of weights in c and $\prod_c(\lambda_X)$ is the product of indicators in c. We interpret this by letting $\prod_{w \in W_c} w = P(Z{=}c)$ and $\prod_c(\lambda_X) = P(X|Z{=}c)$, where Z is a hidden variable whose values are the circuits in C. Therefore $g_c = P(X, Z{=}c)$ and

$$f_r = \sum_{c \in C} g_c = \sum_{c \in C} P(X, Z{=}c) = P(X).$$

Thus an SPN represents a joint distribution over X. From Definition 1 we see that each node n in an SPN is the root of its own SPN. Let Φ_n be the distribution represented by the SPN rooted at n.

SPNs compute the marginal probability of any subset of X when the indicators for variables not in the subset are all set to one. Thus marginal and, consequently, conditional inference always takes time linear in the size of the SPN [Poon and Domingos, 2011]. Inferring $\underset{X,Z}{\operatorname{argmax}}\ P(X, Z)$, called the most probable explanation (MPE), is also done in linear time. After replacing sum nodes with max nodes, the SPN is evaluated in an upward pass followed by a downward pass. The downward pass assigns the (or an) MPE circuit to Z and assigns values to any unobserved variables in X. The circuit is constructed by starting at the root, traversing to all children of product nodes and at sum nodes traversing to the (or a) child whose weighted value is a maximum [Poon and Domingos, 2011; Chan and Darwiche, 2006]. We let c^*_x be an MPE circuit for variable setting $X = x$.

Poon & Domingos [2011] associate with each sum node s a hidden variable H_s whose values are the children of s. Similar to how we see hidden variable Z as being summed out of $P(X, Z)$, they view an SPN as summing out H from $P(X, H)$, where H is the set of all H_s. The benefits of our interpretation are a simpler mathematical formulation and a clearer relationship to MPE inference. We do make use of the variables in H, however, and relate them to Z by way of complete sub-circuits as follows. A complete sub-circuit c assigns values to a subset of H and all of the observed variables. If the edge from s to $t \in \text{ch}(s)$ is in c then $H_s = t$. If the indicator node for $\lambda_{X_i=j}$ is in c then $X_i = j$. Let H_c be the set of variables to which c assigns values. See Figure 1.

4 SPN Structure Search

To learn an SPN, our greedy structure search algorithm uses a training set T that contains i.i.d. samples of the variables in X. For each product node p we use T to create another dataset T_p. We infer for each training instance $x \in T$ the MPE state $X{=}x, Z{=}c^*_x$, which in turn assigns values to the variables in $H_{c^*_x}$. If p is in c^*_x (for some $x \in T$) then all of its children are in c^*_x as well and c^*_x assigns values to all $H_s, s \in \text{ch}(p)$. Thus MPE inference gives us a sample of the hidden variables associated with children of p; this sample is added to T_p.

In brief, a step in the search is as follows. We use the datasets T_p to select a product node p^* (Section 4.2). We pass T_{p^*} to modified versions of the variable- and instance-partitioning procedures (Section 4.2) found in GD [Gens and Domingos, 2013]. The structure modification algorithm (Section 4.1) then uses the partitioning of T_{p^*} to change the

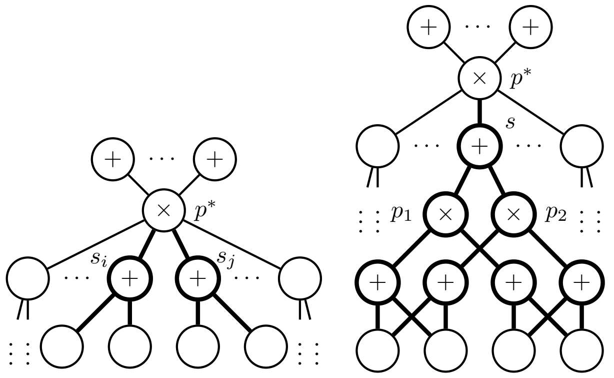

Figure 2: An example of the MixClones algorithm applied to the product node p^* and a subset of its children $S_1 = \{s_i, s_j\}$ in the network fragment on the left ($k = 2$). The algorithm replaces the bold nodes in the left with the bold nodes in the network fragment on the right.

SPN structure at p^*. As described, variable- and instance-partitioning in GD applies at the leaves of half-formed SPNs. The use of the datasets T_p broadens the applicability of these procedures to the internal nodes of SPNs.

4.1 Search Operator

The structure modification algorithm, or structure operator, we use is called MixClones and is outlined in Algorithm 1. An example of applying this operator to an SPN is shown in Figure 2. By analyzing the scopes of the newly added nodes, we can see that the operator transforms one SPN into another SPN. The new SPN has more parameters, making it more flexible in approximating a target distribution.

Product node p^*, a subset of its children S_1, and an integer k are given as input to MixClones. The set S_1 identifies the sub-distribution $\prod_{s' \in S_1} \Phi_{s'}$, which is assumed to not fit the training data well enough. MixClones replaces it with a more expressive distribution: a mixture of k clones of the sub-distribution (where the mixing coefficients and parameters of each sub-distribution are subsequently changed, with the goal of better fitting the training data).

MixClones builds the mixture of clones as follows. It removes the connections between p^* and the nodes in S_1. It then creates $k-1$ clone sets, $S_2, \dots, S_k$, of the set S_1; a clone set contains, for each $s' \in S_1$, a corresponding sum node with the same set of children as s'. MixClones creates product node p_i for each set S_i (including p_1 for S_1), and adds p_i as the parent of the nodes in S_i. The product nodes p_i now represent the k clones of the sub-distribution $\prod_{s' \in S_1} \Phi_{s'}$. Adding a new sum node s as parent to the nodes p_i creates the mixture of clones: $\Phi_s = \sum_i w_{sp_i} \prod_{s' \in S_i} \Phi_{s'}$. The mixture is tied back in to the SPN by adding p^* as the parent of s. Sum node s is returned.

The weights w_{sp_i} of s are the mixing coefficients of the mixture model Φ_s and the weights of the sum nodes in $S_1, \dots, S_k$ are the sub-distribution parameters. Outside of MixClones these parameters are chosen to (indirectly) increase the training set likelihood.

4.2 Search Algorithm

This section describes and justifies Algorithm 2 (SEARCHSPN), our SPN structure search procedure. At a high level it is simply a repeated application of MIXCLONES followed by some parameter updating. This requires selecting p^*, S_1, and k at each search iteration. We now explain how these selections are made and how parameter updating is done.

Selecting p^*.

We think of product node p^* as identifying the weakest point in the SPN structure, where we say a node is weaker than another node if it contributes less to the likelihood. An approximate likelihood function $\tilde{\mathcal{L}}$ gives us a simple method for measuring weakness. Using Theorem 1, the true likelihood can be written as $\mathcal{L}(f_r|T) = \prod_{x \in T} \sum_{c \in C} c(x)$. The approximation is derived from $\mathcal{L}$ by replacing the sum with a max operator. Thus

$$\tilde{\mathcal{L}}(f_r|T) = \prod_{x \in T} c^*_x(x), \qquad (1)$$

where c^*_x is, as defined before, the circuit whose output is maximal for input x.

Remember that $c^*_x(x) = \prod_{w \in W_{c^*_x}} w$, where here we assume that when $X = x$ the indicators in c^*_x take the value one. Substituting this expression into Equation 1 we see that $\tilde{\mathcal{L}} = \prod_{x \in T} \prod_{w \in W_{c^*_x}} w$ is a product of weights in the SPN; weights in the product may appear more than once. We will reorder the product of weights to assign partial responsibility for the value of $\tilde{\mathcal{L}}$ to each product node.

We define the multiset W_r for root node r and, letting P be the set of product nodes, define the multiset W_p for every $p \in P$. For some $x \in T$, if product node p and weight w_{st} are both in c^*_x, and p is a parent of s, then we add w_{st} to W_p; if w_{st} is in c^*_x and $s = r$ then we add w_{st} to W_r. With these definitions it is possible to rewrite $\tilde{\mathcal{L}}$ as

$$\tilde{\mathcal{L}}(f_r|T) = \left(\prod_{w' \in W_r} w' \right) \prod_{p \in P} \prod_{w \in W_p} w.$$

Product node p is said to be responsible for multiplying into $\tilde{\mathcal{L}}$ the value $\prod_{w \in W_p} w$. Therefore $p^* = \underset{p}{\operatorname{argmin}} \prod_{w \in W_p} w$ is the product node that contributes least to a high value of $\tilde{\mathcal{L}}$.

Algorithm 1 MIXCLONES(p^*, S_1, k)

```
Input: product node $p^*$, $S_1 \subseteq \text{ch}(p^*)$, $k > 1$
Output: sum node $s$
remove the edge between $p^*$ and each $s \in S_1$
$s \leftarrow$ new sum node
set $p^*$ as the parent of $s$
$S_2, \ldots, S_k \leftarrow k-1$ clones of $S_1$
for all $S_i \in \{S_1, \ldots, S_k\}$ do
    $p_i \leftarrow$ new product node
    set $s$ as the parent of $p_i$
    set $p_i$ as the parent of each $s' \in S_i$
end for
return $s$
```

Algorithm 2 SEARCHSPN(N, T)

```
Input: SPN $N$, training instances $T$
while stop criteria not met do
    $p^* \leftarrow \underset{p}{\operatorname{argmin}} \prod_{w \in W_p} w$
    use $T_{p^*}$ to partition $\{H_s | s \in \text{ch}(p^*)\}$ into
        approximately independent subsets $V_j$
    for all $S_j = \{s | H_s \in V_j\}$ where $|S_j| > 1$ do
        partition $T_{p^*}$ into $k$ groups $T_i$ of similar
            instances, ignoring variables not in $V_j$
        $s \leftarrow$ MIXCLONES($p^*$, $S_j$, $k$)
        update weights of $s$ and its grandchildren
    end for
end while
```

Selecting S_1 and k.

With p^* selected, we would like to change the structure of the SPN at p^* such that the distribution $\Phi_{p^*} = \prod_{s \in \text{ch}(p^*)} \Phi_s$ better fits the training data T. Since it is unclear how to do this directly, SEARCHSPN instead solves a simpler, related problem that admits the use of the variable- and instance-partitioning methods of GD. More specifically, it changes the structure of the SPN such that the distribution $\Psi_{p^*} = \prod_{s \in \text{ch}(p^*)} \Psi_s$ better fits the training set T_{p^*}, where Ψ_s, for each $s \in \text{ch}(p^*)$, is a categorical distribution over the variable H_s and its parameters are the weights of s.

If the variables in $V_{p^*} = \{H_s | s \in \text{ch}(p^*)\}$ are mutually independent then Ψ_{p^*} is an appropriate model. If, however, dependencies exist amongst the variables in some subset $V_j \subseteq V_{p^*}$ then there will be a loss of likelihood. SEARCHSPN attempts to detect such subsets V_j that also have no dependencies with the other variables $V_{p^*} \setminus V_j$. Calling MIXCLONES with S_1 set to $\{s | H_s \in V_j\}$ changes the structure of the SPN to better model the dependencies in V_j. We now describe how SEARCHSPN selects subsets V_j and how it fits the model created by MIXCLONES to the data T_{p^*}.

Variable-partitioning, or selecting subsets V_j, can be done empirically by analyzing T_{p^*} as follows. SEARCHSPN detects pairwise dependencies among the variables in V_{p^*}, builds a graph representation of these dependencies, and partitions V_{p^*} by finding the connected components in the graph. It uses a normalized mutual information measure (and tunable threshold τ) to test for pairwise dependencies. MIXCLONES is called once for each non-singleton subset in the partition of V_{p^*}. If the partition only contains singleton subsets then p^* is added to a blacklist so that it is not selected again and a new search iteration is started.

After selecting $V_j \subseteq V_{p^*}$, SEARCHSPN builds a mixture model to explain the dependencies amongst the variables in V_j. It does this by fitting the model to T_{V_j}, the portion of the dataset T_{p^*} that involves only variables in V_j. The mixture model is defined as $\sum_{i=1}^{k} w_i \prod_{H_s \in V_j} \Psi_s^{(i)}$, where the w_i are mixing coefficients and each component $\prod_{H_s \in V_j} \Psi_s^{(i)}$ is a product of categorical distributions $\Psi_s^{(i)}$ over the variables in V_j. We use K-means to partition T_{V_j} into k datasets $T_1, \ldots, T_k$ and, assigning T_i to the ith component, set the mixture model parameters to their maximum likelihood estimate. The parameters of the i^{th} categorical distribution over H_s, $\Psi_s^{(i)}$, are set to maximize its likelihood given the dataset T_i, where we ignore all variables in T_i except H_s. Mixture coefficient w_i is set to $|T_i|/|T_{V_j}|$.

We could use hard EM instead of K-means, but Rooshenas & Lowd report little difference between the two methods [Rooshenas and Lowd, 2014]. We run K-means several times, increasing k on each run, and select the k that leads to the mixture model with highest penalized likelihood. Like GD we penalize the likelihood by placing an exponential prior on k, $P(k) \propto \exp(-\gamma k|V_j|)$, where γ is a tunable parameter.

Updating Parameters and Efficiency.

SEARCHSPN calls MIXCLONES with S_1 set to $\{s|H_s \in V_j\}$ and this returns a sum node that we denote s_{V_j}. The weights of s_{V_j} are set to the mixing coefficients w_i and the weights of the sum nodes in $S_1, \ldots, S_k$ (the grandchildren of s_{V_j}) are set using the distributions $\Psi_s^{(i)}$ as follows. Let S_i be the grandchildren of s_{V_j} from its i^{th} child. We set the weights of each $s' \in S_i$ to the parameters of $\Psi_s^{(i)}$, where s is chosen as follows. If $S_i = S_1$ then we set $s = s'$; otherwise we set s to be the node in S_1 that s' is a clone of.

A key technical problem in implementing our search algorithm is maintaining the training sets T_p as the SPN graph structure changes. The obvious method is to re-build these sets from scratch after each search step, but this requires a full pass through the dataset, evaluating the SPN for each instance. We avoid this (to almost the same effect) by updating only those sets directly affected by the search step.

Our structure search algorithm uses data likelihood as its scoring function. We stop when the likelihood of a validation set reaches a maximum. To reduce the computational burden we take many steps in the search space before computing the likelihood. If the likelihood begins to drop we intelligently re-trace our steps to find an SPN with high likelihood.

5 Experiments

We compare SEARCHSPN and LEARNSPN (from GD) on twenty datasets that were recently used in Rooshenas and Lowd [2014] and Gens and Domingos [2013]. We also compare the algorithms on a set of artificially-generated datasets based on the permanent of an $n \times n$ matrix. For each dataset we run a grid search over hyperparameter values $\gamma \in \{0.1, 0.3, 1.0, 3.0, 10.0\}$ and $\tau \in \{0.003, 0.01, 0.03, 0.1, 0.3\}$, the cluster penalty and pairwise dependency threshold, respectively. Chosen models are those with the highest likelihood on a validation set. Table 1 and Table 2 show the mean test set likelihood over ten runs.

We implement LEARNSPN using the same variable- and instance-partitioning code that we use in SEARCHSPN. We do this to make the algorithms as similar as possible so that result differences in our experiments can be attributed as much as possible to the different classes of SPN structure that the algorithms are able to learn (DAG vs. tree).

Table 1: The left two columns show log-likelihoods on 20 datasets for the DAG-SPN (learned using SEARCHSPN) and tree-SPN (learned using LEARNSPN from GD) models. Bold numbers indicate statistically significant results with $p = 0.05$.

Dataset	DAG-SPN	Tree-SPN
NLTCS	-6.072	**-6.058**
MSNBC	-6.057	**-6.044**
KDDCup 2k	-2.159	-2.160
Plants	-13.127	**-12.868**
Audio	**-40.128**	-40.486
Jester	**-53.076**	-53.595
Netflix	**-56.807**	-57.515
Accidents	**-29.017**	-29.363
Retail	-10.971	**-10.970**
Pumbs-start	-28.692	**-25.501**
DNA	**-81.760**	-81.993
Kosarek	-10.999	**-10.933**
MSWeb	**-9.972**	-10.300
Book	**-34.911**	-36.288
EachMovie	**-53.279**	-54.627
WebKB	**-157.883**	-164.615
Reuters-52	**-86.375**	-92.796
20 Newsgrp.	**-153.626**	-164.188
BBC	**-252.129**	-261.778
Ad	**-16.967**	-18.613

5.1 Permanent Distribution

A result from circuit complexity theory shows that a MAF cannot compute the permanent of an $n \times n$ matrix unless it is super-polynomial in size [Raz, 2009]. We also assume it is a difficult problem for MACs since computing the permanent is #P-complete [Valiant, 1979].

We build MNPerm, a set of artificial datasets based on the permanent. Let a_{ij} be the entries in an $n \times n$ matrix. Let S_n be the set of all permutations of $\{1, \ldots, n\}$. Then the permanent is defined as $\sum_{\sigma \in S_n} \prod_{i=1}^{n} a_{i\sigma(i)}$. Viewing the entries a_{ij} as variables we see that this expression is a multilinear polynomial. It defines an unnormalized probability distribution over variables $X_1, \ldots, X_n$ if we view the entries a_{ij} as indicator variables, where $a_{ij} = \lambda_{X_i=j}$. Thus each variable X_i is discrete and can take one of n values. This distribution evaluates to zero unless each variable takes a value that is different from the value taken by every other variable. The partition function is $n!$.

For each $n \in \{2, \ldots, 9\}$ we construct a fully-connected pairwise Markov network whose distribution is a softened version of the permanent distribution. We do this by defining the factor for each pair of variables X_i, X_j to take the value one if $X_i = X_j$ and take the value ten otherwise. The datasets used to produce the results in Table 2 were generated by sampling from the constructed Markov networks using Gibbs sampling.

Table 2: Log-likelihood of the MNPerm datasets for the DAG-SPN and tree-SPN models. Bold numbers indicate statistically significant results with $p = 0.05$. The right column indicates the ratio of the average size of the two models.

n	DAG-SPN	Tree-SPN	|DAG|/|Tree|
2	**-0.192**	-0.216	1.14
3	-1.656	-1.646	1.13
4	**-3.309**	-3.651	1.05
5	**-5.030**	-6.129	1.23
6	**-6.867**	-8.834	1.04
7	**-8.821**	-11.809	1.09
8	**-11.650**	-15.392	1.07
9	**-14.297**	-18.811	1.06

5.2 Observations

The performance of SEARCHSPN is better than LEARNSPN on thirteen of the twenty datasets and worse on six of them. A similar outcome is seen when comparing the SEARCHSPN results with the results reported in [Gens and Domingos, 2013], although statistical significance cannot be determined in this comparison.

Comparing to the results reported in [Rooshenas and Lowd, 2014]—again without any significance claim—we see that SEARCHSPN only gets a higher likelihood on the Ad dataset. One explanation for the success of the RL models is that, like SEARCHSPN models, they are not restricted to being tree-structured. Leaf nodes in RL models are multivariate Markov networks modeled using arithmetic circuits that are not restricted to having a tree structure.

SEARCHSPN arguably has an advantage over RL models in that it seems to produce smaller networks in shorter training times. We have model-size and training-time data for RL models [Rooshenas, 2014] on six of the datasets (NLTCS, KDDCup 2k, Book, 20 Newsgrp., and Ad). The models learned by RL range in size from 385k to 1.2M nodes and the DAG-SPNs ranged in size from 2k to 114k nodes. Learning times for RL ranged from 19m to 15.7h and the DAG-SPNs ranged from 3m to 1.4h. For any of these datasets the DAG-SPN has at least 10 times fewer nodes and its learning time is at least 7 times faster. The learning-time results are less definitive than the model-size results since some or all of the difference reported here could be due to differences in such factors as the hardware and programming language used in the experiments, and not due to differences in the algorithms. While further investigation is warranted, SEARCHSPN seems to produce compact models quickly.

The results in Table 2 for the MNPerm datasets show a clear separation in likelihood between DAG-SPNs and tree-SPNs as n increases. And the difference does not seem due to a difference in the size of the learned SPNs since the DAG-structured SPNs are only marginally larger. These results support the idea that DAG-structured SPNs have a distinct advantage over tree-structured SPNs.

6 Conclusion

SEARCHSPN is the first algorithm designed for SPNs that takes a search approach to structure-learning. In contrast with previous work it does not dramatically increase the size of the SPN at any point in the search and it uses principled methods without restricting the class of learned structure to trees.

We have linked SPNs to the MAC and MAF models from circuit complexity theory and highlighted some interesting connections to that field that suggest tree-structured models may be less powerful than DAG-structured models. Our experiments indicate that being able to learn a wider class of SPN structures can be advantageous. Future work includes better understanding what types of datasets and distributions benefit from a DAG-structured SPN and what types can be well-modeled with tree-structured SPNs.

A Proof of Theorem 1

Proof. The proof is by induction from the leaf nodes to the root node; thus if r has children we assume that for any $t \in \text{ch}(r)$ $f_t = \sum_{c \in C_t} g_c$. The proof is also broken into the cases from Definition 1. Let C_n be the set of complete sub-circuits in the SPN rooted at node n (thus $C = C_r$).

Case 1. Let r be an indicator node for $\lambda_{X=i}$. Then $f_r = \lambda_{X=i}$. Since $C = \{c'\}$, where c' consists of the node r, $g_{c'} = \lambda_{X=i}$. Thus $f_r = g_{c'} = \sum_{c \in C} g_c$.

Case 2. Let r be a product node whose children are SPNs with disjoint scopes. Let $\text{ch}(r) = \{t_1, \ldots, t_m\}$, $C^\times$ be the Cartesian product $\prod_{i=1}^m C_{t_i}$, and $c^\times = (c_1, \ldots, c_m) \in C^\times$. The scopes of the children of r are disjoint so any two circuits $c_i, c_j, c_i \neq c_j$ from $c^\times$ have no common nodes or edges. Thus every $c^\times$ yields a unique $c \in C$ using the following construction. Add r, add r's edges, and add the nodes and edges in each c_i; then $g_c = \prod_{i=1}^m g_{c_i}$. Every circuit in C can be constructed in this manner. Thus the construction is a one-to-one and onto mapping from $C^\times$ to C. By definition $f_r = \prod_{i=1}^m f_{t_i}$ and by the inductive hypothesis $f_r = \prod_{i=1}^m \sum_{c' \in C_{t_i}} g_{c'}$. Multiplying out the right-hand side yields $f_r = \sum_{c^\times \in C^\times} \prod_{i=1}^m g_{c_i}$ and applying the mapping yields $f_r = \sum_{c \in C} g_c$.

Case 3. Assume r is a sum node whose children are SPNs with the same scope. Let $C^\cup = \bigcup_{t \in \text{ch}(r)} C_t$. Every $c' \in C^\cup$ yields a unique $c \in C$ using the following construction. Add r, add the edge from r to the root t of c', and add the nodes and edges in c'; then $g_c = w_{rt} g_{c'}$. Every circuit in C can be constructed in this manner. Thus the construction is a one-to-one and onto mapping from $C^\cup$ to C. By definition $f_r = \sum_{t \in \text{ch}(r)} w_{rt} f_t$ and by the inductive hypothesis $f_r = \sum_{t \in \text{ch}(r)} w_{rt} \sum_{c' \in C_t} g_{c'}$. The double summation ranges over $C^\cup$ so $f_r = \sum_{c' \in C^\cup} w_{rt} g_{c'}$. Applying the mapping yields $f_r = \sum_{c \in C} g_c$. $\square$

References

[Chan and Darwiche, 2006] Hei Chan and Adnan Darwiche. On the robustness of most probable explanations. In *Proceedings of the Twenty-Second Annual Conference on Uncertainty in Artificial Intelligence*, pages 63–71. AUAI Press, 2006.

[Chickering, 2002] David Maxwell Chickering. The WinMine toolkit. Technical Report MSR-TR-2002-103, Microsoft, Redmond, WA, 2002.

[Choi *et al.*, 2011] Myung Jin Choi, Vincent YF Tan, Animashree Anandkumar, and Alan S Willsky. Learning latent tree graphical models. *The Journal of Machine Learning Research*, 12:1771–1812, 2011.

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

[Delalleau and Bengio, 2011] Olivier Delalleau and Yoshua Bengio. Shallow vs. deep sum-product networks. In *Advances in Neural Information Processing Systems 24*, pages 666–674, 2011.

[Della Pietra *et al.*, 1997] Stephen Della Pietra, Vincent Della Pietra, and John Lafferty. Inducing features of random fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(4):380–393, 1997.

[Dennis and Ventura, 2012] Aaron Dennis and Dan Ventura. Learning the architecture of sum-product networks using clustering on variables. In *Advances in Neural Information Processing Systems 25*, pages 2042–2050, 2012.

[Gens and Domingos, 2012] Robert Gens and Pedro Domingos. Discriminative learning of sum-product networks. In *Advances in Neural Information Processing Systems 25*, pages 3248–3256, 2012.

[Gens and Domingos, 2013] Robert Gens and Pedro Domingos. Learning the structure of sum-product networks. In *Proceedings of the 30th International Conference on Machine Learning*, pages 873–880, 2013.

[Lee *et al.*, 2014] Sang-Woo Lee, Christopher Watkins, and Byoung-Tak Zhang. Non-parametric bayesian sum-product networks. In *Workshop on Learning Tractable Probabilistic Models*, 2014.

[Lowd and Domingos, 2008] Daniel Lowd and Pedro Domingos. Learning arithmetic circuits. In *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence*, pages 383–392. AUAI Press, 2008.

[Lowd and Rooshenas, 2013] Daniel Lowd and Amirmohammad Rooshenas. Learning Markov networks with arithmetic circuits. In *Proceedings of the Sixteenth International Conference on Artificial Intelligence and Statistics*, pages 406–414, 2013.

[Meila and Jordan, 2001] Marina Meila and Michael I Jordan. Learning with mixtures of trees. *The Journal of Machine Learning Research*, 1:1–48, 2001.

[Peharz *et al.*, 2013] Robert Peharz, Bernhard C Geiger, and Franz Pernkopf. Greedy part-wise learning of sum-product networks. In *Machine Learning and Knowledge Discovery in Databases*, pages 612–627. Springer, 2013.

[Poon and Domingos, 2011] Hoifung Poon and Pedro Domingos. Sum-product networks: A new deep architecture. In *Proceedings of the Twenty-Seventh Annual Conference on Uncertainty in Artificial Intelligence*, pages 337–346. AUAI Press, 2011.

[Ravikumar *et al.*, 2010] Pradeep Ravikumar, Martin J Wainwright, John D Lafferty, et al. High-dimensional Ising model selection using 1-regularized logistic regression. *The Annals of Statistics*, 38(3):1287–1319, 2010.

[Raz, 2006] Ran Raz. Separation of multilinear circuit and formula size. *Theory of Computing*, 2(6):121–135, 2006.

[Raz, 2009] Ran Raz. Multi-linear formulas for permanent and determinant are of super-polynomial size. *Journal of the ACM*, 56(2):8:1–8:17, April 2009.

[Rooshenas and Lowd, 2014] Amirmohammad Rooshenas and Daniel Lowd. Learning sum-product networks with direct and indirect variable interactions. In *Proceedings of The 31st International Conference on Machine Learning*, pages 710–718, 2014.

[Rooshenas, 2014] Pedram Rooshenas. Private communication, 2014.

[Valiant, 1979] Leslie G Valiant. The complexity of computing the permanent. *Theoretical computer science*, 8(2):189–201, 1979.