Learning Sum-Product Networks with Direct and Indirect Variable Interactions

Amirmohammad Rooshenas PEDRAM@CS.UOREGON.EDU
Daniel Lowd LOWD@CS.UOREGON.EDU
Department of Computer and Information Science, University of Oregon

Abstract

Sum-product networks (SPNs) are a deep probabilistic representation that allows for efficient, exact inference. SPNs generalize many other tractable models, including thin junction trees, latent tree models, and many types of mixtures. Previous work on learning SPN structure has mainly focused on using top-down or bottom-up clustering to find mixtures, which capture variable interactions indirectly through implicit latent variables. In contrast, most work on learning graphical models, thin junction trees, and arithmetic circuits has focused on finding direct interactions among variables. In this paper, we present ID-SPN, a new algorithm for learning SPN structure that unifies the two approaches. In experiments on 20 benchmark datasets, we find that the combination of direct and indirect interactions leads to significantly better accuracy than several state-of-the-art algorithms for learning SPNs and other tractable models.

1. Introduction

Probabilistic graphical models such as Bayesian and Markov networks have been widely used in many areas of AI, including computer vision, natural language processing, robotics, and more. Their biggest limitation is that inference is intractable: in general, computing the exact probability of a marginal or conditional query is #P-complete (Roth, 1996). However, there are many special cases of graphical models where inference is tractable. For problems where the quality and efficiency of inference are important, performing exact inference in a tractable model may be better than performing approximate inference in an intractable model, even if the intractable model fits the data slightly better. Tractable models are also useful for approximating intractable models in variational inference. In this paper, we focus on the problem of learning a tractable probabilistic model over a set of discrete random variables, with the goal of answering probability queries accurately and efficiently.

Previous work has considered several different approaches to representing and learning tractable models, such as learning thin junction trees (Bach & Jordan, 2001; Chechetka & Guestrin, 2008; Elidan & Gould, 2008), exploiting context-specific independencies (Gogate et al., 2010; Lowd & Domingos, 2008; Lowd & Rooshenas, 2013), and using latent variables (Choi et al., 2011; Lowd & Domingos, 2005; Meila & Jordan, 2000). Sum-product networks (SPNs) (Poon & Domingos, 2011) are a class of tractable models with the capacity to generalize all of these approaches.

However, previous work with SPNs has focused exclusively on the latent variable approach, using a complex hierarchy of mixtures to represent all interactions among the observable variables (Gens & Domingos, 2013; Dennis & Ventura, 2012). These SPN structures can be learned from data by recursively clustering instances and variables. Clustering over the instances is done to create a mixture, represented with a sum node. Clustering over the variables is done to find independencies within the cluster, represented with a product node. We refer to this approach as creating *indirect interactions* among the variables, since all dependencies among the observable variables are mitigated by the latent variables implicit in the mixtures.

This type of top-down clustering is good at representing clusters, but may have difficulty with discovering *direct interactions* among variables. For example, suppose the data in a domain is generated by a 6-by-6 grid-structured Markov network with binary-valued variables. This Markov network can be represented as a junction tree with treewidth 6, which is small enough to allow for exact inference. This can also be represented as an SPN that sums out 6 variables at a time in each sum node, represent-

Proceedings of the 31st International Conference on Machine Learning, Beijing, China, 2014. JMLR: W&CP volume 32. Copyright 2014 by the author(s).

ing each of the separator sets in the junction tree. However, learning this from data requires discovering the right set of 64 (2^6) clusters that happen to render the other regions of the grid independent from each other. Of all the possible clusterings that could be found, happening to find one of the separator sets is extremely unlikely. Learning a good structure for the next level of the SPN is even less likely, since it consists of 64 clustering problems, each working with 1/64th of the data.

In contrast, Markov network structure learning algorithms can easily learn a simple grid structure, but may do poorly if the data has natural clusters that require latent variables or a mixture. For example, consider a naive Bayes mixture model where the variables in each cluster are independent given the latent cluster variable. Representing this as a Markov network with no latent variables would require an exponential number of parameters.

In order to get the best of both worlds, we propose ID-SPN, a new method for learning SPN structures that can learn both indirect and direct interactions, including conditional and context-specific independencies. This unifies previous work on learning SPNs through top-down clustering (Dennis & Ventura, 2012; Gens & Domingos, 2013) with previous work on learning tractable Markov networks through greedy search (Lowd & Rooshenas, 2013).

On a set of 20 standard structure learning benchmarks, our ID-SPN method consistently outperforms state-of-the-art methods for learning both SPNs and tractable Markov networks. ID-SPN has better test-set log-likelihood than LearnSPN (Gens & Domingos, 2013) on every single dataset, and a majority of these differences are statistically significant. ID-SPN is more accurate than ACMN (Lowd & Rooshenas, 2013) on 17 datasets. Furthermore, ID-SPN was always more accurate than two other algorithms for learning tractable models, mixtures of trees (Meila & Jordan, 2000) and latent tree models (Choi et al., 2011). To fully demonstrate the effectiveness of ID-SPN at learning accurate models, we compared it to intractable Bayesian networks learned by the WinMine Toolkit (Chickering, 2002), and found that ID-SPN obtained significantly better test-set log-likelihood on 13 out of 20 datasets with significantly worse log-likelihood on only 4 datasets. This suggests that the combination of latent variables and direct variable interactions is a good approach to modeling probability distributions over discrete data, even when tractable inference is not a primary goal.

The remainder of our paper is organized as follows. In Section 2, we present background on graphical models and SPNs. In Section 3, we present ID-SPN, our proposed method for learning an SPN. We present experiments on 20 datasets in Section 4 and conclude in Section 5.

2. Background

A *sum-product network* (SPN) (Poon & Domingos, 2011) is a deep probabilistic model for representing a tractable probability distribution. SPNs are attractive because they can represent many other types of tractable probability distributions, including thin junction trees, latent tree models, and mixtures of tractable distributions. They have also achieved impressive results on several computer vision problems (Poon & Domingos, 2011; Gens & Domingos, 2012).

An SPN consists of a rooted, directed, acyclic graph representing a probability distribution over a set of random variables. Each *leaf* in the SPN graph is a tractable distribution over a single random variable. Each interior node is either a *sum node*, which computes a weighted sum of its children in the graph, or a *product node*, which computes the product of its children. The *scope* of a node is defined as the set of variables appearing in the univariate distributions of its descendants. In order to be valid, the children of every sum node must have identical scopes, and the children of every product node must have disjoint scopes (Gens & Domingos, 2013)[1]. Intuitively, sum nodes represent mixture and product nodes represent independencies. SPNs can also be described recursively as follows: every SPN is either a tractable univariate distribution, a weighted sum of SPNs with identical scopes, or a product of SPNs with disjoint scopes.

To compute the probability of a complete configuration, compute the value of each node starting at the leaves. Each leaf is a univariate distribution which evaluates to the probability of one variable according to that distribution. Sum and product nodes evaluate to the weighted sum and product of their child nodes in the network, respectively. To compute the probability of a partial configuration, we need to sum out one or more variables. In an SPN, this is done by setting the values of all leaf distributions for those variables to 1. Conditional probabilities can then be computed as the ratio of two partial configurations. Thus, computing marginal and conditional probabilities can be done in linear time with respect to the size of the SPN, while these operations are often intractable in Bayesian and Markov networks with high treewidth.

2.1. Arithmetic Circuits

Arithmetic circuits (ACs) (Darwiche, 2003) are an inference representation that is closely related to SPNs. Like an SPN, an AC is a rooted, directed, acyclic graph in which interior nodes are sums and products. The representational

[1] Poon and Domingos (2011) also allow for non-decomposable product nodes, but we adopt the more restrictive definition of Gens and Domingos (2013) for simplicity.

differences are that ACs use indicator nodes and parameter nodes as leaves, while SPNs use univariate distributions as leaves and attach all parameters to the outgoing edges of sum nodes.

Arithmetic circuits are closely related to logical formulas represented in negation normal form (NNF), except that NNF formulas use disjunctions and conjunctions rather than sums and products. We adapt NNF properties defined by Darwiche (2003) to describe three important AC properties: i) An AC is *decomposable* if the children of a product node have disjoint scopes. ii) An AC is *deterministic* if the children of a sum node are mutually exclusive, meaning that at most one is non-zero for any complete configuration. iii) An AC is *smooth* if the children of a sum node have identical scopes.

An AC represents a valid probability distribution if it is decomposable and smooth. ACs generated by compiling graphical models are typically deterministic as well. This is different from most previous work with SPNs, where sum nodes represent mixtures of distributions and are not deterministic. SPNs can be made deterministic by explicitly defining random variables to represent the mixtures, making the different children of each sum node deterministically associated with different values of a new hidden variable.

We further demonstrate the equivalence of the AC and SPN representations with the following two propositions.

Proposition 1. *For discrete domains, every decomposable and smooth AC can be represented as an equivalent SPN with fewer or equal nodes and edges.*

We can also show the other direction:

Proposition 2. *For discrete domains, every SPN can be represented as an AC with at most a linear increase in the number of edges.*

The proofs of both propositions can be found in the supplementary materials.

2.2. Learning SPNs and ACs

SPNs and ACs are very attractive due to their ability to represent a wide variety of tractable structures. This flexibility also makes it especially challenging to learn their structure, since the space of possible structures is very large.

Several different methods have recently been proposed for learning SPNs. Dennis and Ventura (2012) construct a region graph by first clustering the training instances and then repeatedly clustering the variables within each cluster to find smaller scopes. When creating new regions, if a region with that scope already exists, it is reused. Given the region graph, Dennis and Ventura convert this to an SPN by introducing sum nodes to represent mixtures within each region and product nodes to connect regions to sub-regions. Gens and Domingos (2013) also perform a top-down clustering, but they create the SPN directly through recursive partitioning of variables and instances rather than building a region graph first. The advantage of their approach is that it greedily optimizes log-likelihood; however, the resulting SPN always has a tree structure and does not reuse model components. Peharz et al. (2013) propose a greedy bottom-up clustering approach for learning SPNs that merges small regions into larger regions.

Two AC learning methods have been proposed as well. Lowd and Domingos (2008) adapt a greedy Bayesian network structure learning algorithm by maintaining an equivalent AC representation and penalizing structures by the number of edges in the AC. This biases the search towards models where exact inference is tractable without placing any a priori constraints on network structure. Lowd and Rooshenas (2013) extend this idea to learning Markov networks with conjunctive features, and find that the additional flexibility of the undirected representation leads to slightly better likelihoods at the cost of somewhat slower learning times.

While both classes of learning algorithms work with equivalent representations and learn rich, tractable models, the types of relationships they discover are very different. The SPN learning algorithms emphasize mixtures and only consider local modifications when searching for better models. This results in models that use implicit latent variables to capture all of the interactions among the observable variables. In contrast, the AC learning methods are based on learning algorithms for graphical models without hidden variables. Instead of searching within the sum-product network structure, they search within the space of graphical model structures. Some of these operations could lead to large, global changes in the corresponding AC representation. For example, a single operation that increases the model treewidth by one could require doubling the size of the circuit. (Methods for learning thin junction trees are similar in their search space, except that ACs can also exploit context-specific independence to obtain more compact models.)

Each of these directions has its own strengths. Mixture models are good at representing domains that have natural clusters, since each cluster can be represented by a relatively simple mixture component. Thin junction trees and arithmetic circuits are better when the domain has conditional or context-specific independencies, since they can represent direct relationships among the observed variables. Neither is more expressive than the other: converting a mixture model to a model with no latent variables typically results in a clique over all variables, and converting a thin junction tree to a mixture of trees or other latent vari-

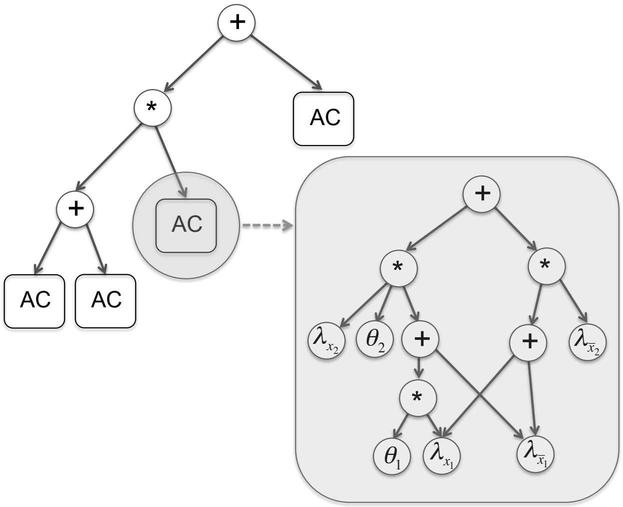

Figure 1. Example of an ID-SPN model. The upper layers are shown explicitly as sum and product nodes, while the lower layers are abbreviated with the nodes labeled "AC." The AC components encode graphical models over the observed variables represented as arithmetic circuits (AC), and may involve many nodes and edges.

able model could require an exponential number of clusters. In the following section, we propose a method that combines both approaches in a unified algorithm.

3. ID-SPN

In this section, we describe ID-SPN, our approach to learning sum-product networks. ID-SPN combines top-down clustering with methods for learning tractable Markov networks to obtain the best of both worlds: indirect interactions through latent cluster variables in the upper levels of the SPN as well as direct interactions through the tractable Markov networks at the lower levels of the SPN. ID-SPN learns tractable Markov networks represented by ACs using the ACMN algorithm (Lowd & Rooshenas, 2013); however, this could be replaced with any algorithm that learns a tractable multivariate probability distribution that can be represented as an AC or SPN, including any thin junction tree learner.

For learning an SPN structure, LearnSPN (Gens & Domingos, 2013) recursively performs two operations to create a tree-structured SPN: partitioning the training data to create a sum node, representing a mixture over different clusters, or partitioning the variables to create a product node, representing groups of independent variables within a cluster. The partitioning is chosen greedily to maximize the (regularized) likelihood of the training data. This process continues recursively, operating on fewer variables and examples until it reaches univariate distributions which become the leaves.

ID-SPN performs a similar top-down search, clustering instance and variables to create sum and product nodes, but it may choose to stop this process before reaching univariate distributions and instead learn an AC to represent a tractable multivariate distribution with no latent variables. Thus, LearnSPN remains a special case of ID-SPN when the recursive clustering proceeds all the way to univariate distributions. ACMN is also a special case, when a tractable distribution is learned at the root and no clustering is performed. ID-SPN uses the likelihood of the training data to choose among these different operations.

Another way to view ID-SPN is that it learns SPNs where the leaves are tractable *multivariate* distributions rather than univariate distributions. As long as these leaf distributions can be represented as valid SPNs, the overall structure can be represented as a valid SPN as well. For ease of description, we refer to the structure learned by ID-SPN as a sum-product of arithmetic circuits (SPAC). A SPAC model consists of sum nodes, product nodes, and AC nodes. Every AC node is an encapsulation of an arithmetic circuit which itself includes many nodes and edges. Figure 1 shows an example of a SPAC model.

We can also relax the condition of learning a valid SPN structure and allow any tractable probabilistic models at the leaves. In this case, the SPAC model generalizes to a sum-product of tractable models. Such models may not be valid SPNs, but they retain the efficient inference properties that are the key advantage of SPNs. Nevertheless, we leave this part for future exploration and continue with SPAC, which has a valid SPN structure.

3.1. Learning

Algorithm 1 illustrates the pseudocode of ID-SPN. The ID-SPN algorithm begins with a SPAC structure that only contains one AC node, learned using ACMN on the complete data. In each iteration, ID-SPN attempts to *extend* the model by replacing one of the AC leaf nodes with a new SPAC subtree over the same variables. Figure 2 depicts the effect of such an extension on the SPAC model. If the extension increases the log-likelihood of the SPAC model on the training data, then ID-SPN updates the working model and adds any newly created AC leaves to the queue.

As described in Algorithm 2, the extend operation first attempts to partition the variables into independent sets to create a new product node. If a good partition exists, the subroutine learns a new sum node for each child of the product node. If no good variable partition exists, the subroutine learns a single sum node. Each sum node is learned by clustering instances and learning a new AC leaf node for

Algorithm 1 Algorithm for learning a SPAC.

```
function ID-SPN($T$)
input: Set of training examples, $T$; set of variables $V$
output: Learned SPAC model
$n \leftarrow$ LearnAC($T, V$).
SPAC $\leftarrow n$
$N \leftarrow$ leaves of SPAC // which includes only $n$
while $N \neq \emptyset$ do
  $n \leftarrow$ remove the node with the largest number of samples in
      its footprint from $N$
  $(T_{ji}, V_j) \leftarrow$ set of samples and variables used for learning $n$
          // footprint and scope of $n$
  subtree $\leftarrow$ extend($n$, $T_{ji}$, $V_j$)
  SPAC' $\leftarrow$ SPAC with $n$ replaced by subtree
  if SPAC' has a better log-likelihood on $T$ than SPAC then
    SPAC $\leftarrow$ SPAC'
    $N \leftarrow N \cup$ leaves of subtree
  end if
end while
return SPAC
```

Algorithm 2 Algorithm for extending SPAC structure.

```
function extend($n$, $T$, $V$)
input: An AC node $n$, set of instances $T$ and variables $V$ used
for learning $n$
output: A SPAC subtree representing a distribution over $V$
learned from $T$
p-success $\leftarrow$ Using $T$, partition $V$ into approximately
          independent subsets $V_j$
if p-success then
  // Learn a product node
  for each $V_j$ do
    let $T_j$ be the data samples projected into $V_j$
    s-success $\leftarrow$ partition $T_j$ into subsets of similar
              instances $T_{ji}$
    if s-success then
      // Learn a sum node
      for each $T_{ji}$ do
        $n_{ji} \leftarrow$ LearnAC($T_{ji}, V_j$)
      end for
      $n_j \leftarrow \sum_i \frac{|T_{ji}|}{|T_j|} \cdot n_{ji}$
    else
      $n_j \leftarrow$ LearnAC($T_j, V_j$)
    end if
  end for
  subtree $\leftarrow \prod_j n_j$
else
  s-success $\leftarrow$ partition $T$ into subsets of similar instances $T_i$
  if s-success then
    // Learn a sum node
    for each $T_i$ do
      $n_i \leftarrow$ LearnAC($T_i, V$)
    end for
    subtree $\leftarrow \sum_i \frac{|T_i|}{|T|} \cdot n_i$
  else
    // Failed to extend $n$
    subtree $\leftarrow n$
  end if
end if
return subtree
```

each data cluster. These AC nodes may be extended further in future iterations.

As a practical matter, in preliminary experiments we found that the root was always extended to create a mixture of AC nodes. Since learning an AC over all variables from all examples can be relatively slow, in our experiments we start by learning a sum node over AC nodes rather than a single AC node.

Every node n_{ji} in a SPAC (including sum and product nodes) represents a probability distribution over a subset of variables V_j learned using a subset of the training instances T_i. We use T_{ji} to denote the projection of T_i into V_j, and refer to it as the *footprint* of node n_{ji} on the training set, or to be more concise, the footprint of n_{ji}.

A product node estimates the probability distribution over its scope as the product of approximately independent probability distributions over smaller scopes: $P(V) = \prod_j P(V_j)$ where V is the scope of the product node and V_js are the scope of its children. Therefore, to create a product node, we must partition variables into some sets that are approximately independent. We measure the dependence of two variables with pairwise mutual information, $\sum_{x_1 \in X_1} \sum_{x_2 \in X_2} \frac{C(x_1,x_2)}{|T_i|} \log \frac{C(x_1,x_2) \cdot |T_i|}{C(x_1)C(x_2)}$ where X_1 and X_2 are a pair of variables, x_1 and x_2 range over their respective states, and $C(.)$ counts the occurrences of the configuration in the footprint of the product node. A good variable partition is one where the variables within a partition have high mutual information, and the variables in different partitions have low mutual information. Thus, we create an adjacency matrix using pairwise mutual information such that two variables are connected if their empirical mutual information over the footprint of the product node (not the whole training set) is larger than a predefined threshold. Then, we find the set of connected variables in the adjacency matrix as the independent set of variables. This operation fails if it only finds a single connected component or if the number of variables is less than a threshold.

A sum node represents a mixture of probability distributions with identical scopes: $P(V) = \sum_i w_i P^i(V)$ where the weights w_i sum to one. A sum node can also be interpreted as a latent variable that should be summed out: $\sum_c P(C = c)P(V|C = c)$. To create a sum node, we partition instances by using the expectation maximization (EM) algorithm to learn a simple naive Bayes mixture model: $P(V) = \sum_i P(C_i) \prod_j P(X_j|C_i)$ where X_j is a random variable. To select the appropriate number of clusters, we rerun EM with different numbers of clusters and select the model that maximizes the penalized log-likelihood over the footprint of the sum node. To avoid overfitting, we penalize the log-likelihood with an exponential prior, $P(S) \propto e^{-\lambda C|V|}$ where C is the the number

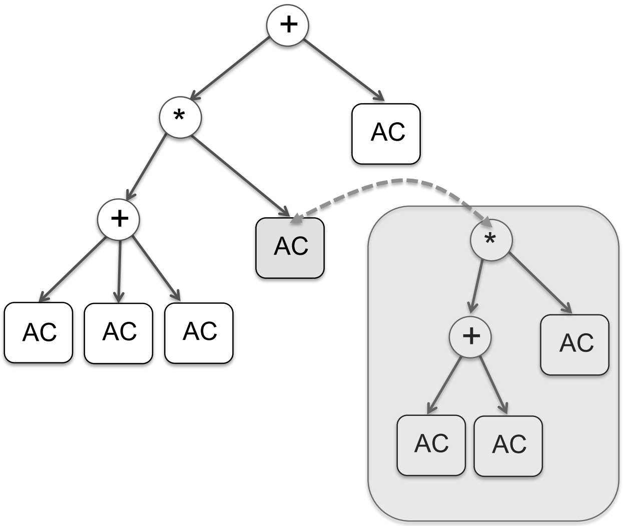

Figure 2. One iteration of ID-SPN: it tries to extends the SPAC model by replacing an AC node with a SPAC subtree over the same scope

of clusters and λ is a tunable parameter. We use the clusters of this simple mixture model to partition the footprint, assigning each instance to its most likely cluster. We also tried using k-means, as done by Dennis and Ventura (Dennis & Ventura, 2012), and obtained results of similar quality. We fail learning a sum node if the number of samples in the footprint of the sum node is less than a threshold.

To learn leaf distributions, AC nodes, we use the ACMN algorithm (Lowd & Rooshenas, 2013). ACMN learns a Markov network using a greedy, score-based search, but it uses the size of the corresponding arithmetic circuit as a learning bias. ACMN can exploit the context-specific independencies that naturally arise from sparse feature functions to compactly learn many high-treewidth distributions. The learned arithmetic circuit is a special case of an SPN where sum nodes always sum over mutually exclusive sets of variable states. Thus, the learned leaf distribution can be trivially incorporated into the overall SPN model. Other possible choices of leaf distributions include thin junction trees and the Markov networks learned by LEM (Gogate et al., 2010). We chose ACMN because it offers a particularly flexible representation (unlike LEM), exploits context-specific independence (unlike thin junction trees), and learns very accurate models on a range of structure learning benchmarks (Lowd & Rooshenas, 2013).

The specific methods for partitioning instances, partitioning variables, and learning a leaf distribution are all flexible and can be adjusted to meet the characteristics of a particular application domain.

4. Experiments

We evaluated ID-SPN on 20 datasets illustrated in Table 1.a with 16 to 1556 binary-valued variables. These datasets or a subset of them also have been used in (Davis & Domingos, 2010; Lowd & Davis, 2010; Haaren & Davis, 2012; Lowd & Rooshenas, 2013; Gens & Domingos, 2013). In order to show the accuracy of ID-SPN, we compared it with the state-of-the-art learning method for SPNs (LearnSPN) (Gens & Domingos, 2013), mixtures of trees (MT) (Meila & Jordan, 2000), ACMN, and latent tree models (LTM) (Choi et al., 2011). To evaluate these methods, we selected their hyper-parameters according to their accuracy on the validation sets.

For LearnSPN, we used the same learned SPN models presented in the original paper (Gens & Domingos, 2013). We used the WinMine toolkit (WM) (Chickering, 2002) for learning intractable Bayesian networks.

We slightly modified the original ACMN codes to incorporate both Gaussian and L_1 priors into the algorithm. The L_1 prior forces more weights to become zero, so the learning algorithm can prune more features from the split candidate list. The modified version is as accurate as the original version, but it is considerably faster. For ACMN, we used an L_1 prior of 0.1, 1, and 5, and a Gaussian prior with a standard deviation of 0.1, 0.5, and 1. We also used a split penalty of 2, 5, 10 and maximum edge number of 2 million.

For LTM, we ran the authors' code[2] with its default EM configuration to create the models with each provided algorithm: CLRG, CLNJ, regCLRG, and regCLNJ. For MT, we used our own implementation and the number of components ranged from 2 to 30 with a step size of 2. To reduce variance, we re-ran each learning configuration 5 times.

For ID-SPN, we need specific parameters for learning each AC leaf node using ACMN. To avoid exponential growth in the parameter space, we selected the L_1 prior C^{ji}, split penalty SP^{ji}, and maximum edges ME^{ji} of each AC node to be proportional to the size of its footprint: $P^{ji} = \max\{P\frac{|T_{ji}|}{|T|} \cdot \frac{|V_j|}{|V|}, P^{min}\}$ where parameter P can be C, SP or ME. When SPAC becomes deep, C^{min} and SP^{min} help avoid overfitting and ME^{min} permits ID-SPN to learn useful leaf distributions. Since ID-SPN has a huge parameter space, we used random search (Bergstra & Bengio, 2013) instead of grid search. In order to create a configuration, we sampled each parameter uniformly from a predefined set of parameter values.

For learning AC nodes, we selected C from 1.0, 2.0, 5.0, 8.0, 10.0, 15.0 and 20.0, and SP from 5, 8, 10, 15, 20, 25, and 30. We selected the pair setting of (C^{min}, SP^{min}) between (0.01, 1) and (1, 2), and ME and ME^{min} are 2M

[2] http://people.csail.mit.edu/myungjin/latentTree.html

Table 1. Dataset characteristic and Log-likelihood comparison. ● shows significantly better log-likelihood than ID-SPN, and ○ indicates significantly worse log-likelihood than ID-SPN. † means that we do not have value for that entry.

	a) Datasets characteristics				b) Log-likelihood comparison					
Dataset	Var#	Train	Valid	Test	ID-SPN	ACMN	MT	WM	LearnSPN	LTM
NLTCS	16	16181	2157	3236	-6.02	● -6.00	-6.01	-6.02	-6.11	○ -6.49
MSNBC	17	291326	38843	58265	-6.04	○ -6.04	○ -6.07	-6.04	○ -6.11	○ -6.52
KDDCup 2000	64	180092	19907	34955	-2.13	○ -2.17	-2.13	○ -2.16	-2.18	○ -2.18
Plants	69	17412	2321	3482	-12.54	○ -12.80	○ -12.95	○ -12.65	○ -12.98	○ -16.39
Audio	100	15000	2000	3000	-39.79	○ -40.32	○ -40.08	○ -40.50	○ -40.50	○ -41.90
Jester	100	9000	1000	4116	-52.86	○ -53.31	○ -53.08	○ -53.85	○ -53.48	○ -55.17
Netflix	100	15000	2000	3000	-56.36	○ -57.22	○ -56.74	○ -57.03	○ -57.33	○ -58.53
Accidents	111	12758	1700	2551	-26.98	○ -27.11	○ -29.63	● -26.32	○ -30.04	○ -33.05
Retail	135	22041	2938	4408	-10.85	○ -10.88	-10.83	○ -10.87	-11.04	○ -10.92
Pumsb-star	163	12262	1635	2452	-22.40	○ -23.55	○ -23.71	● -21.72	○ -24.78	○ -31.32
DNA	180	1600	400	1186	-81.21	● -80.03	○ -85.14	● -80.65	○ -82.52	○ -87.60
Kosarek	190	33375	4450	6675	-10.60	○ -10.84	-10.62	○ -10.83	○ -10.99	○ -10.87
MSWeb	294	29441	32750	5000	-9.73	○ -9.77	○ -9.85	● -9.70	○ -10.25	○ -10.21
Book	500	8700	1159	1739	-34.14	○ -35.56	○ -34.63	○ -36.41	-35.89	-34.22
EachMovie	500	4524	1002	591	-51.51	○ -55.80	○ -54.60	○ -54.37	-52.49	†
WebKB	839	2803	558	838	-151.84	○ -159.13	○ -156.86	○ -157.43	-158.20	○ -156.84
Reuters-52	889	6532	1028	1540	-83.35	○ -90.23	○ -85.90	○ -87.55	-85.07	○ -91.23
20 Newsgroup	910	11293	3764	3764	151.47	○ -161.13	○ -154.24	○ -158.95	○ -155.93	○ -156.77
BBC	1058	1670	225	330	-248.93	○ -257.10	○ -261.84	○ -257.86	-250.69	○ -255.76
Ad	1556	2461	327	491	-19.00	● -16.53	● -16.02	-18.35	-19.73	†

Table 3. Conditional (marginal) log-likelihood comparison, the bold numbers show significantly better values

	10% Query		30% Query		50% Query		70% Query		90% Query	
Dataset	ID-SPN	WM	ID-SPN	WM	ID-SPN	WM	ID-SPN	WM	ID-SPN	WM
NLTCS	**-0.3082**	-0.3157	**-0.3138**	-0.3193	**-0.2881**	-0.2940	**-0.3393**	-0.3563	**-0.3622**	-0.4093
MSNBC	**-0.2727**	-0.2733	**-0.3130**	-0.3149	**-0.3285**	-0.3333	**-0.3403**	-0.3522	**-0.3528**	-0.3906
KDDCup 2000	**-0.0322**	-0.0329	**-0.0318**	-0.0325	**-0.0309**	-0.0316	**-0.0323**	-0.0337	**-0.0329**	-0.0361
Plants	**-0.1297**	-0.1361	**-0.1348**	-0.1444	**-0.1418**	-0.1590	**-0.1500**	-0.1876	**-0.1656**	-0.2797
Audio	**-0.3471**	-0.3630	**-0.3699**	-0.3808	**-0.3766**	-0.3854	**-0.3801**	-0.3957	**-0.3870**	-0.4309
Jester	**-0.4651**	-0.4869	**-0.4964**	-0.5096	**-0.5042**	-0.5128	**-0.5090**	-0.5231	**-0.5158**	-0.5589
Netflix	**-0.4893**	-0.5079	**-0.5254**	-0.5366	**-0.5349**	-0.5451	**-0.5416**	-0.5637	**-0.5507**	-0.6012
Accidents	**-0.1316**	-0.1368	**-0.1532**	-0.2226	**-0.1748**	-0.3165	**-0.1993**	-0.4523	**-0.2274**	-0.5834
Retail	-0.0794	-0.0800	**-0.0783**	-0.0794	**-0.0788**	-0.0807	**-0.0796**	-0.0823	**-0.0803**	-0.0841
Pumsb-star	-0.0727	**-0.0675**	**-0.0809**	-0.1077	**-0.0906**	-0.1584	**-0.1023**	-0.2920	**-0.1193**	-0.7654
DNA	-0.3371	-0.3373	**-0.3815**	-0.3957	**-0.3942**	-0.4450	**-0.4156**	-0.4898	**-0.4374**	-0.5335
Kosarek	**-0.0482**	-0.0514	**-0.0507**	-0.0533	**-0.0520**	-0.0547	**-0.0529**	-0.0568	**-0.0544**	-0.0623
MSWeb	**-0.0294**	-0.0300	**-0.0305**	-0.0318	**-0.0307**	-0.0328	**-0.0317**	-0.0350	**-0.0326**	-0.0375
Book	**-0.0684**	-0.0771	**-0.0674**	-0.0736	**-0.0673**	-0.0720	**-0.0673**	-0.0711	**-0.0675**	-0.0734
EachMovie	**-0.0958**	-0.1078	**-0.0976**	-0.1046	**-0.0985**	-0.1008	**-0.0986**	-0.1012	**-0.1005**	-0.1140
WebKB	**-0.1744**	-0.1881	**-0.1766**	-0.1858	**-0.1766**	-0.1834	**-0.1778**	-0.1844	**-0.1795**	-0.1929
Reuters-52	**-0.0911**	-0.1015	**-0.0889**	-0.0963	**-0.0896**	-0.0954	**-0.0905**	-0.0968	**-0.0920**	-0.1054
20 Newsgroup	**-0.1633**	-0.1807	**-0.1641**	-0.1753	**-0.1644**	-0.1708	**-0.1646**	-0.1688	**-0.1653**	-0.1724
BBC	**-0.2353**	-0.2503	**-0.2360**	-0.2459	**-0.2340**	-0.2406	**-0.2338**	-0.2406	**-0.2337**	-0.2462
Ad	-0.0080	-0.0078	-0.0086	-0.0087	**-0.0087**	-0.0180	**-0.0090**	-0.0767	**-0.0104**	-0.2849
Avg. Time (ms)	159	203	164	621	162	1091	168	1554	171	2021
Max. Time (ms)	286	1219	310	3641	312	6255	319	8682	319	11102

and 200k edges, respectively. We used similar Gaussian priors with a standard deviation of 0.1, 0.3, 0.5, 0.8, 1.0, or 2.0 for learning all AC nodes.

For learning sum nodes, the cluster penalty λ is selected from 0.1, 0.2, 0.4, 0.6 and 0.8, the number of clusters from 5, 10, and 20, and we restarted EM 5 times. When there were fewer than 50 samples, we did not learn additional sum nodes, and when there were fewer than 10 variables, we did not learn additional product nodes.

Finally, we limited the number of main iterations of ID-SPN to 5, 10, or 15, which helps avoid overfitting and controls the learning time. Learning sum nodes and AC nodes

Table 2. Statistically significance comparison. Each table cell lists the number of datasets where the row's algorithm obtains significantly better log-likelihoods than the column's algorithm

	ID-SPN	LearnSPN	WM	ACMN	MT	LTM
ID-SPN	–	11	13	17	15	17
LearnSPN	0	–	0	1	2	10
WM	4	6	–	10	7	13
ACMN	3	7	7	–	9	13
MT	1	7	11	11	–	15
LTM	0	0	3	4	2	–

is parallelized as much as it was possible using up to 6 cores simultaneously.

We bounded the learning time of all methods to 24 hours, and we ran our experiments, including learning, tuning, and testing, on an Intel(R) Xeon(R) CPU X5650@2.67GHz.

Table 1.b shows the average test set log-likelihood of each method. We could not learn a model using LTM for the EachMovie and Ad datasets, so we excluded these two datasets for all comparisons involving LTM. We use • to indicate that the corresponding method has significantly better test set log-likelihood than ID-SPN on a given dataset, and ○ for the reverse. For significance testing, we performed a paired t-test with p=0.05. ID-SPN has better average log-likelihood on every single dataset than LearnSPN, and better average log-likelihood on 17 out of 20 datasets (with 1 tie) than ACMN. Thus, ID-SPN consistently outperforms the two methods it integrates. Table 2 shows the number of datasets for which a method has significantly better test set log-likelihood than another. ID-SPN is significantly better than WinMine on 13 datasets and significantly worse than it on only 4 datasets, which means that ID-SPN is achieving efficient exact inference without sacrificing accuracy. ID-SPN is significantly better than LearnSPN, ACMN, MT and LTM on 11, 17, 15, and 17 datasets, respectively.

We also evaluated the accuracy of ID-SPN and WinMine for answering queries by computing conditional log-likelihood (CLL): $\log P(X = x|E = e)$. For WinMine, we used the Gibbs sampler from the Libra toolkit[3] with 100 burn-in and 1000 sampling iterations. Since the Gibbs sampler can approximate marginal probabilities better than joint probabilities, we also computed conditional marginal log-likelihood (CMLL): $\sum_i \log P(X_i = x_i|E = e)$. For WinMine, we reported the greater of CLL and CMLL; however, CMLL was higher for all but 12 settings. The reported values have been normalized by the number of query variables.

We generated queries from test sets by randomly selecting the query variables, using the rest of variables as evidence. Table 3 shows the C(M)LL values ranging from 10% query and 90% evidence variables to 90% query and 10% evidence variables. The bold numbers indicate statistical significance using a paired t-test with p=0.05. ID-SPN is significantly more accurate on 95 out of 100 settings and is significantly worse on only 1.

We also report the per-query average time and per-query maximum time, both in milliseconds. We computed the per-query average for each dataset, and then report the average and the maximum of the per-query averages. For WinMine, the per-query average time significantly varies with the number of variables. The per-query maximum time shows that even with only 1000 iterations, Gibbs sampling is still slower than exact inference in our ID-SPN models.

5. Conclusion

Most previous methods for learning tractable probabilistic models have focused on representing all interactions directly or indirectly. ID-SPN demonstrates that a combination of these two techniques is extremely effective, and can even be more accurate than intractable models. Interestingly, the second most accurate model in our experiments was often the mixture of trees model (MT) (Meila & Jordan, 2000), which also contains indirect interactions (through a single mixture) and direct interactions (through a Chow-Liu tree in each component). After ID-SPN, MT achieved the second-largest number of significant wins against other algorithms. ID-SPN goes well beyond MT by learning multiple layers of mixtures and using much richer leaf distributions, but the underlying principles are similar. Therefore, rather than focusing exclusively on mixtures or direct interaction terms, this suggests that the most effective probabilistic models need to use a combination of both techniques.

Acknowledgments

This research was partly funded by ARO grant W911NF-08-1-0242, NSF grant OCI-0960354, and a Google Faculty Research Award. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of ARO, NSF, or the U.S. Government.

[3] http://libra.cs.uoregon.edu

References

Bach, F.R. and Jordan, M.I. Thin junction trees. *Advances in Neural Information Processing Systems*, 14:569–576, 2001.

Bergstra, J. and Bengio, Y. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13:281–305, 2013.

Chechetka, A. and Guestrin, C. Efficient principled learning of thin junction trees. In Platt, J.C., Koller, D., Singer, Y., and Roweis, S. (eds.), *Advances in Neural Information Processing Systems 20*. MIT Press, Cambridge, MA, 2008.

Chickering, D. M. The WinMine toolkit. Technical Report MSR-TR-2002-103, Microsoft, Redmond, WA, 2002.

Choi, M. J., Tan, V., Anandkumar, A., and Willsky, A. Learning latent tree graphical models. *Journal of Machine Learning Research*, 12:1771–1812, May 2011.

Darwiche, A. A differential approach to inference in Bayesian networks. *Journal of the ACM*, 50(3):280–305, 2003.

Davis, J. and Domingos, P. Bottom-up learning of Markov network structure. In *Proceedings of the Twenty-Seventh International Conference on Machine Learning*, Haifa, Israel, 2010. ACM Press.

Dennis, A. and Ventura, D. Learning the architecture of sum-product networks using clustering on variables. In *Advances in Neural Information Processing Systems 25*, 2012.

Elidan, G. and Gould, S. Learning bounded treewidth Bayesian networks. *Journal of Machine Learning Research*, 9(2699-2731):122, 2008.

Gens, R. and Domingos, P. Discriminative learning of sum-product networks. In *Advances in Neural Information Processing Systems*, pp. 3248–3256, 2012.

Gens, R. and Domingos, P. Learning the structure of sum-product networks. In *Proceedings of the Thirtieth International Conference on Machine Learning*, 2013.

Gogate, V., Webb, W., and Domingos, P. Learning efficient Markov networks. In *Proceedings of the 24th conference on Neural Information Processing Systems (NIPS'10)*, 2010.

Haaren, J. Van and Davis, J. Markov network structure learning: A randomized feature generation approach. In *Proceedings of the Twenty-Sixth National Conference on Artificial Intelligence*. AAAI Press, 2012.

Lowd, D. and Davis, J. Learning Markov network structure with decision trees. In *Proceedings of the 10th IEEE International Conference on Data Mining (ICDM)*, Sydney, Australia, 2010. IEEE Computer Society Press.

Lowd, D. and Domingos, P. Naive Bayes models for probability estimation. In *Proceedings of the Twenty-Second International Conference on Machine Learning*, pp. 529–536, Bonn, Germany, 2005.

Lowd, D. and Domingos, P. Learning arithmetic circuits. In *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence*, Helsinki, Finland, 2008. AUAI Press.

Lowd, D. and Rooshenas, A. Learning Markov networks with arithmetic circuits. In *Proceedings of the Sixteenth International Conference on Artificial Intelligence and Statistics (AISTATS 2013)*, Scottsdale, AZ, 2013.

Meila, M. and Jordan, M. Learning with mixtures of trees. *Journal of Machine Learning Research*, 1:1–48, 2000.

Peharz, R., Geiger, B., and Pernkopf, F. Greedy part-wise learning of sum-product networks. In *Machine Learning and Knowledge Discovery in Databases*, volume 8189 of *Lecture Notes in Computer Science*, pp. 612–627. Springer Berlin Heidelberg, 2013.

Poon, H. and Domingos, P. Sum-product networks: A new deep architecture. In *Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence (UAI-11)*, Barcelona, Spain, 2011. AUAI Press.

Roth, D. On the hardness of approximate reasoning. *Artificial Intelligence*, 82:273–302, 1996.