Learning Sum-Product Networks with Direct and Indirect Variable Interactions

Amirmohammad Rooshenas **Daniel Lowd**
Department of Computer and Information Science
University of Oregon
Eugene, OR 97403
{pedram,lowd}@cs.uoregon.edu

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

Sum-product networks (SPNs) [2] are a deep probabilistic representation with efficient, exact inference. This makes SPNs a compelling alternative to deep belief networks [15] and deep Boltzmann machines [22], offering fast weight learning and inference while obtaining state-of-the-art results in a number of image analysis tasks [5]. However, learning the structure of an SPN remains a challenge: because SPNs are a lower-level representation than traditional graphical models, the space of possible structures is much higher.

Early SPN structures were specified by hand and were tailored to image data. For example, the model used by Poon and Domingos [2] consists of a hierarchy of regions and subregions, each representing a rectangle of contiguous pixels. Since then, several attempts have been made to learn the model structure directly [13, 3, 19]. Structure learning allows SPNs to fit the data much better than a fixed structure, and is essential for applying SPNs in domains that lack an obvious two-dimensional structure. All previous SPN learning algorithms perform some type of top-down or bottom-up hierarchical clustering in order to build the structure. Dennis and Ventura [13] use k-means clustering over the instances and the variables in order to find good candidate regions, and then arrange them into an SPN structure similar to the one used by Poon and Domingos. Gens and Domingos [3] maximize the likelihood directly by repeatedly clustering instances to create sum nodes or variables to create product nodes. Peharz et al. [19] build an SPN through bottom-up clustering, finding larger and larger groups of related variables. All three approaches rely on clustering and use search operations that make local changes to the SPN. This leads to SPN structures that represent the interactions among variables *indirectly* through mixtures.

This approach is good at discovering 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 (MN) with binary-valued variables. This MN 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, representing 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. In the Gens and Domingos [3] algorithm, 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, most Markov network structure learning algorithms can easily learn a simple grid structure. These algorithms focus on identifying direct interactions or independencies among the observed variables rather than clusters or latent variables. By searching through a different space, they can find different kinds of structure. Recently, there has been increased interest in learning *tractable* graphical models, such as thin junction trees [8, 20, 12, 14] and arithmetic circuits [4, 6]. These structures can be compactly represented as SPNs, but previous SPN learning algorithms are very unlikely to recover such structures, even when they are the best fit for the domain.

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 [13, 3] with previous work on learning tractable Markov networks through greedy search [6].

On a set of 20 discrete density estimation benchmarks, our ID-SPN method consistently outperforms state-of-the-art methods for learning both SPNs and other tractable probabilistic models. ID-SPN has better test set log-likelihood than LearnSPN [3] on every single dataset, and a majority of these differences are statistically significant. ID-SPN is more accurate than the tractable Markov network learner ACMN [6] on 17 datasets. Furthermore, ID-SPN was always more accurate than two other algorithms for learning tractable models, mixtures of trees [9] and latent tree models [7]. To fully demonstrate the effectiveness of ID-SPN at learning accurate models, we compared it to intractable Bayesian networks learned by the WinMine Toolkit [21], 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) [2] 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 [2, 5].

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 [3]. Intuitively, sum nodes represent mixtures 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.

For example, SPNs can easily represent a naive Bayes mixture model as a sum of products of univariate distributions. SPNs can also represent thin junction trees by introducing sum and product nodes for the different states of the cliques and separators sets; see Poon and Domingos [2] for a simple example. Since SPNs can be composed as described by the recursive definition, SPNs can

also represent mixtures of thin junction trees, such as the mixture of trees model [9], and hierarchical mixtures of mixtures, such as latent tree models.

An SPN can be used to answer probabilistic queries as follows. To compute the probability of a complete variable assignment, replace each of the leaf distributions with the probability of its individual variable assignment. Then evaluate all sum and product nodes starting at the bottom. The probability of the configuration is denoted by the value of the root node. To compute marginal probabilities, repeat the same process but replace the leaf distributions of all marginalized variables with 1. Conditional probabilities can similarly be computed as a ratio of marginal probabilities. All of these operations are linear in the size of the SPN (number of nodes and edges), while marginalization often requires exponential time in Bayesian and Markov networks with high treewidth.

Arithmetic circuits (ACs) [17] are an inference representation closely related to SPNs. Like an SPN, an AC is a rooted, directed, acyclic graph in which interior nodes are sums and products. The main differences are that ACs use unweighted sum nodes instead of weighted sum nodes and use indicator and parameter nodes as leaves instead of univariate distributions.

Arithmetic circuits are closely related to logical formulas represented in Negation Normal Form (NNF). We adapt NNF properties defined by Darwiche [17] to describe three important AC properties:

- An AC is *decomposable* if the children of a product node have disjoint scopes.
- 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.
- An AC is *smooth* if the children of a sum node have identical scopes.

To represent a valid probability distribution, an AC must be 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, proven in the supplementary materials.

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

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

2.1 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 [13] 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 [3] 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. [19] propose a greedy bottom-up clustering approach for learning SPNs that merges small regions into larger regions.

AC learning methods have been proposed as well. Lowd and Domingos [4] 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

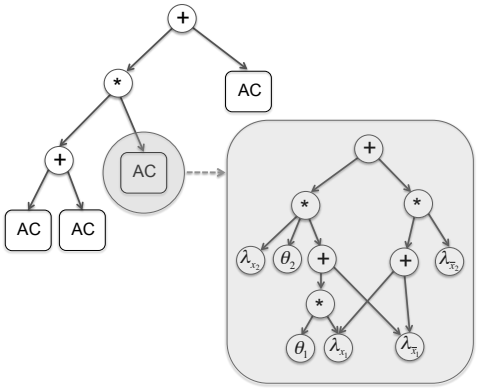

Figure 1: Example of an ID-SPN model. The upper layers are shown explicitly as sum and product nodes, while the lower layers are abbreviated with the nodes labeled "AC."

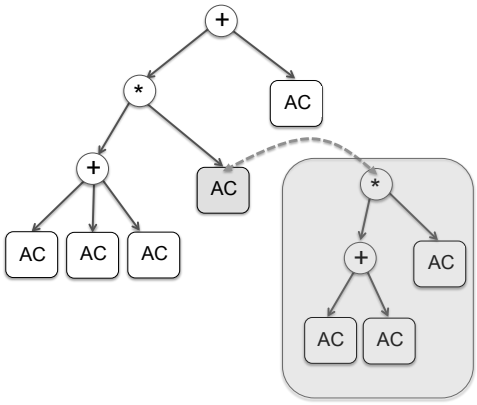

Figure 2: One iteration of ID-SPN: it tries to extends the SPAC model by replacing an AC node with a SPAC subtree over the same scope

exact inference is tractable without placing any a priori constraints on network structure. Lowd and Rooshenas [6] 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, an operation that increases the treewidth by one could double the size of the circuit.

3 ID-SPN

We now 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 MNs at the lower levels of the SPN.

ID-SPN performs a top-down search similar to LearnSPN [3], clustering instance and variables to create sum and product nodes, but it may stop this process before reaching univariate distributions and instead use ACMN [6] to learn a tractable multivariate distribution, represented as an AC. 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.

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 a node from N
  $(T_{ji},\ V_j) \leftarrow$ set of samples and variables used for learning n
  subtree $\leftarrow$ extend(n, Tji, V_j)
  SPAC' $\leftarrow$ replace n by subtree in SPAC
  if SPAC' has a better log-likelihood on T than SPAC **then**
    $N \leftarrow N \cup$ leaves of SPAC'
    SPAC $\leftarrow$ SPAC'
  end if
end while
return SPAC

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

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 1, 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 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 SPAC (including sum and product nodes) represents a valid probablity distribution P_{ji} over a subset of variables V_j. To learn n_{ji}, ID-SPN uses a subset of the training set T_{ji}, which is a subset of training samples T_i projected into V_j. We call T_{ji} 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 scopes of its children. Therefore, to create a product node, we must partition variables into sets that are approximately independent. We use pairwise mutual information, $\sum_{X_k,X_l \in j} \frac{C(X_k,X_l)}{|T_{ji}|} \log \frac{C(X_k,X_l)\cdot|T_{ji}|}{C(X_k)C(X_l)}$ where $C(.)$ counts the occurences of the configuration in the footprint of the product node, to approximately measure the dependence among different sets of variables. 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 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 [13], 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 [6]. ACMN learns a Markov network using a greedy, score-based search, but it uses the size of the corresponding AC 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 AC 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. We chose ACMN because its representation is more flexible than thin junction trees and it learns very accurate models on a range of structure learning benchmarks [6].

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 [10, 1, 16, 6, 3]. In order to show the accuracy of ID-SPN, we compared it with the state-of-the-art learning method for SPNs (LearnSPN) [3], mixtures of trees (MT) [9], ACMN, and latent tree models (LTM) [7]. To evaluate these methods, we selected their hyper-parameters according to their accuracy on held-out validation data.

For LearnSPN, we used the same learned SPN models presented in the original paper [3]. We used the WinMine toolkit (WM) [21] for learning intractable Bayesian networks. For LTM, we ran the authors' code[1] with its default EM configuration to create the models with different provided algorithms: 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.

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

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

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 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: $P^{ji} = max\{P\frac{|T_j i|}{|T|} \cdot \frac{|V_j}{|V|}, P^{min}\}$ where parameter P can be C, SP or ME. When SPAC becomes deep, C^{min} and SP^{min} help avoid overfiting and ME^{min} permits ID-SPN to learn useful leaf distributions. Since ID-SPN has a huge parameter space, we used random search [18] instead of grid search. We sampled configurations uniformly from the space of discrete parameter values.

Table 1: Dataset characteristics and log-likelihood comparison. • shows significantly better log-likelihood than ID-SPN, and ○ indicates significantly worse log-likelihood than ID-SPN. † indicates datasets where LTM failed to learn a model.

Dataset	Var#	ID-SPN	ACMN	MT	WM	LearnSPN	LTM
NLTCS	16	-6.02	• -6.00	-6.01	-6.02	-6.11	○ -6.49
MSNBC	17	-6.04	○ -6.04	○ -6.07	-6.04	○ -6.11	○ -6.52
KDDCup 2000	64	-2.13	○ -2.17	-2.13	○ -2.16	-2.18	○ -2.18
Plants	69	-12.54	○ -12.80	○ -12.95	○ -12.65	○ -12.98	○ -16.39
Audio	100	-39.79	○ -40.32	○ -40.08	○ -40.50	○ -40.50	○ -41.90
Jester	100	-52.86	○ -53.31	○ -53.08	○ -53.85	○ -53.48	○ -55.17
Netflix	100	-56.36	○ -57.22	○ -56.74	○ -57.03	○ -57.33	○ -58.53
Accidents	111	-26.98	○ -27.11	○ -29.63	• -26.32	○ -30.04	○ -33.05
Retail	135	-10.85	○ -10.88	-10.83	○ -10.87	-11.04	○ -10.92
Pumsb-star	163	-22.40	○ -23.55	○ -23.71	• -21.72	○ -24.78	○ -31.32
DNA	180	-81.21	• -80.03	○ -85.14	• -80.65	○ -82.52	○ -87.60
Kosarek	190	-10.60	○ -10.84	-10.62	○ -10.83	○ -10.99	○ -10.87
MSWeb	294	-9.73	○ -9.77	○ -9.85	• -9.70	○ -10.25	○ -10.21
Book	500	-34.14	○ -35.56	○ -34.63	○ -36.41	-35.89	-34.22
EachMovie	500	-51.51	○ -55.80	○ -54.60	○ -54.37	-52.49	†
WebKB	839	-151.84	○ -159.13	○ -156.86	○ -157.43	-158.20	○ -156.84
Reuters-52	889	-83.35	○ -90.23	○ -85.90	○ -87.55	-85.07	○ -91.23
20 Newsgroup	910	151.47	○ -161.13	○ -154.24	○ -158.95	○ -155.93	○ -156.77
BBC	1058	-248.93	○ -257.10	○ -261.84	○ -257.86	-250.69	○ -255.76
Ad	1556	-19.00	• -16.53	• -16.02	-18.35	-19.73	†

Table 2: Statistically significance comparision. 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	–

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 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 iteration of ID-SPNs to 5, 10, or 15, which helps avoid overfitting and controls the learning time. Learning sum nodes and AC nodes 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 every methods. We could not learn a model using LTM for EachMovie and Ad datasets, so we excluded these two datasets for all comparisons involving LTM. We use $\bullet$ to indicate that on a dataset, the corresponding method has significantly better test set log-likelihood than ID-SPN, and $\circ$ 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 signficantly better than LearnSPN, ACMN, MT and LTM on 11, 17, 15, 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[2] with 100 burn-in and 1000 sampling iterations. Since the Gibbs sampler can approximate marginal probablities better then joint probablities, we also computed 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. We computed C(M)LL values for 20 benchmarks with the number of query variables of 10%, 30%, 50%, and 70% (80 different settings). We computed statistically significance using a paired t-test with p=0.05. ID-SPN is significantly more accurate on 75 out of 80 settings and is significantly worse on only 1.[3]

For each fraction of query variables, we computed the average inference time per query. In order to weight the queries from all datasets equally, we first computed averages for each dataset and then computed the average and maximum among all datasets. For WinMine, the average time per query ranges from 203ms to 1555ms, depending on the number of query variables. ID-SPN is faster and shows less dependence on the number of query variables, ranging from 159ms to 168ms. Over all datasets, the maximum time per query is 1554ms for ID-SPN and 8682ms for WinMine. Longer runs of Gibbs sampling might lead to more accurate results but would make WinMine even slower.

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) [9], 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.

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

[3] Full results appears in the supplementary document.

References

[1] D. Lowd and J. Davis. 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.

[2] H. Poon and P. Domingos. 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.

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

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

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

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

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

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

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

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

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

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

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

[14] D. Shahaf and C. Guestrin. Learning thin junction trees via graph cuts. In *Proceedings of the Twelfth International Conference on Artificial Intelligence and Statistics (AISTATS 2009)*, pages 113–120, 2009.

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

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

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

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

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

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

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

[22] R. Salakhutdinov and G. Hinton. Deep boltzmann machines. In *Proceedings of the Twelfth International Conference on Artificial Intelligence and Statistics (AISTATS 2009)*, pages 448–455, 2009.